1991

A Survey and analysis of algorithms for the detection of termination in a distributed system

Lois Rixner

Rochester Institute of Technology
Computer Science Department

A Survey and Analysis of Algorithms for the Detection of Termination in a Distributed System

by
Lois R. Rixner

A thesis, submitted to
The Faculty of the Computer Science Department,
in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science.

Approved by:

Dr. Andrew Kitchen	7/22/91 Date
Dr. James Heliotis	7/22/91 Date
Dr. Peter Anderson	22 July 91 Date

Title of Thesis: A Survey and Analysis of Algorithms for the Detection of Termination in a Distributed System

I, Lois R. Rixner hereby deny permission to the Wallace Memorial Library, of RIT; to reproduce my thesis in whole or in part.

Date: July 22, 1991

Abstract - This paper looks at algorithms for the detection of termination in a distributed system and analyzes them for effectiveness and efficiency. A survey is done of the published algorithms for distributed termination and each is evaluated. Both centralized distributed systems and fully distributed systems are reviewed. The algorithms are analyzed for the overhead and conclusions are made about the situations in which they can be used, i.e. an operating system, a real-time system, or a user application. An original algorithm is presented for the asynchronous case with first-in-first-out message ordering. It allows any process to initiate detection of termination and makes use of multiple tokens.

TABLE OF CONTENTS

1. Introduction 1

1.1. Purpose 1
1.2. Definitions 2
1.3. Approach 4
1.4. Correctness of a Solution 7
1.5. Properties of an Asynchronous Solution 8
1.6. Underlying Computation 9
1.7. Tools for Distributed Algorithms 10
1.8. Network Definitions 13
1.9. Assumptions 14
1.10. Restrictions 14
1.11. Formal Definition of Termination 15
1.12. Phases of Termination Problem 15
1.13. Computation of Overhead 16
1.14. Perspectives 17
1.15. Historical Perspectives 17
1.16. Summary 21

2. Synchronous Centralized Solutions 22

2.1. Introduction 22
2.2. Issues 22
2.3. Assumptions 23
2.4. Solutions Using a Tree Topology 23
2.5. Solutions Using a Hamiltonian Cycle 32
2.6. Summary 36

3. Synchronous Distributed Solutions 38

3.1. Assumptions 38
3.2. Issues 38
3.3. Solution Using Tokens 40
3.4. Solutions Using a Logical Clock 42
3.5. Summary 45

4. Asynchronous Centralized Solutions 47

4.1. Assumptions 47
4.2. Issues 47
4.3. Overview 51
4.4. Arora's Solution to the Problem 52

4.4.1. Control Messages Used in the Algorithm 52
4.4.2. Overhead for the Algorithm 54

4.5. Chandrasekaran's Dual Approach to the Problem 55
4.6. Rozoy's Algorithm 57

4.6.1. Boundaries on the Number of Control Messages 59

4.7. Dijkstra's Solution to the Problem 59
4.8. Arora's Solution Using a Bidirectional Ring 61
4.9. Summary 63

5. Asynchronous Distributed Solutions 66

5.1. Assumptions 66
5.2. Approaches 66
5.3. Solutions for a Hamiltonian Cycle Topology 68

5.3.1. Arora's Solution to the Problem 68

5.3.1.1. Solution Two 70
5.3.1.2. Solution Three 72

5.3.2. Haldar's Solution to the Problem 76

5.4. Solutions for an Arbitrary Network Topology 78

5.4.1. Misra's Solution to the Problem 78
5.4.2. Skyum's Solution to the Problem 82
5.4.3. Huang's Solution to the Problem 84
5.4.4. Eriksen's Solution to the Problem 87

5.5. Solution for an Application which Requires Synchroneity 88
5.6. Summary 90

6. Solutions for Unreliable Channels 94

6.1. Dijkstra's Solution to the Problem 95

6.1.1. Definitions for Dijkstra's Solution 95
6.1.2. Approach 95
6.1.3. Advantages 98

6.2. Kumar's Solution 98
6.3. Mattern's Solutions 105

6.3.1. The Four Counter Method 105
6.3.2. The Sceptic Algorithm 106
6.3.3. A Time Algorithm 107
6.3.4. Vector Counters 108
6.3.5. Channel Counting 109

6.4. Summary 110

7. A New Algorithm 112

8. Conclusions 120

8.1. Synchronous Communication 122
8.2. Asynchronous Communication 124

8.2.1. FIFO Ordering for Messages 124
8.2.2. Non-FIFO Ordering for Messages 128

8.3. Summary 131
8.4. Future Directions 133

1. INTRODUCTION

The detection of termination for a sequential program is trivial. The process knows its own state and it may terminate when its task is complete. The detection of termination is also easy for concurrent processes executing on one machine and sharing memory. The information required to allow the processes to decide if they can terminate can be stored at some shared memory location. However, when several sequential processes must work together in a distributed system, shared memory is non-existent and other methods of detection of termination must be found. The problem arises because a process is aware only of its own local state and is not aware of the states of other processes at that same instant in time. Francez [FRAN80] calls the problem of designing control communication to detect termination, and superimposing it on an application, the *distributed termination problem.* The more abstract and general term *global quiescence* is sometimes used to represent the situation where an application is globally deadlocked and it is merely a matter of interpretation as to whether the system is deadlocked or terminated. Chandy and Misra [CHAN86] and Shavit and Francez [FRAN86] introduced this term.

1.1. Purpose

The purpose of this paper is to survey the literature for termination algorithms beginning with Francez [FRAN80] and to analyze them for appropriateness and correctness. As each algorithm is considered, this paper will consider the efficiency of the algorithm in solving the problem. We will also look at any peculiarities of the solutions that illustrate the problems that must be solved in order to create a correct solution to the distributed termination problem. Even incorrect algorithms can be useful in illustrating conditions that must be taken into account to solve the problem correctly.

In addition, this author will propose an algorithm for an asynchronous, distributed system with first-in-first-out message ordering that does not require clocks.

1.2. Definitions

Francez [FRAN80] identifies two different classes of termination problem. *Endoterminating* processes may terminate on reaching some final state determined only by the fulfillment of a condition. For example a sequential process might terminate after executing a specific number of statements, or when the exit procedure is called. On the other hand *exoterminating* processes are those that depend on other processes in the group for information to do their own computation. The set of all processes on which a process depends is called its *termination dependency set (TDS).* An exoterminating process may not terminate until all processes within its TDS are ready to terminate, i.e. have finished all of their work. Processes working together to solve a problem in a distributed system are exoterminating. Therefore it will be necessary that no process within the group terminates before all processes are ready to terminate.

In the context of distributed systems, several terms should be understood at the outset. All processes running in parallel and working together to find a solution for a computation P, are called *communicating sequential processes.* By definition they have no shared memory and they share information only by sending and receiving messages.

For the sake of consistency, the computation will always be referred to as P and the processes which do the work of the computation will be referred to as p_i. This author assumes that P and p_i contain both the application and the superimposed control code used by the detection algorithm. The number of processes in the system will always be represented by n.

A computation is ready to terminate if it meets its *Global Terminating Condition (GTC)*. The GTC refers to the terminating condition for the computation as a whole. For consistency's sake, I will name the boolean predicate, representing the GTC, B throughout this paper. Each p_i of the computation will have a local predicate, always named b_i, which will represent some local boolean condition for termination. If each p_i was running on its own, this boolean condition, b_i, would be sufficient for termination. However, since the p_i are running together, the conjunction of all the b_i's is necessary to meet B. Once B has been met, it cannot change - it is monotonic.

The processes p_i exchange communications of two different types: basic communication and control communication. *Basic communication, bc*, refers to any exchange of messages relative to the computation. *Control communication, cc*, refers to the exchange of messages relative to the termination algorithm.

In the single processor, concurrent environment, a process is said to be active if it is executing its text and passive if it is blocked. In the distributed environment this is not the case. A process is considered *active* if its local predicate b_i is not met. This could mean the process is actively executing its text or waiting for a bc. A process is considered *passive* only if its local predicate b_i has been met.

If a process is passive it may send/receive cc; it may also receive bc although it may not initiate (send) bc. Upon receipt of a bc, a passive process immediately becomes active (and its b_i becomes false). Notice that the b_i are not monotonic.

If a process is active it may send and receive bc. It may become passive at any time, if its local predicate b_i becomes true. Active processes may or may not be able to send/receive cc depending upon the algorithm under consideration.

The above definitions describe a *transaction oriented model* of distributed computation. All of the algorithms presented assume a transaction oriented model except those presented by Mattern [MAT87B].

Mattern assumes an *atomic model* of distributed computation. In the atomic model all local actions are performed in zero time and therefore there is no need for the two states passive and active - no messages are received unless a process is passive. A process may at any time take any message from one of its incoming communication channels, immediately change its internal state and at the same instant send out any number of messages, possibly none at all. For convenience Mattern assumes that all atomic actions are totally globally ordered. If every local computation initiated by the receipt of a message terminates, the transaction oriented model is equivalent to the atomic model of distributed computation. For Mattern, the advantage of the atomic model is that there is no concurrent activity of processes. He believes this gives him a better insight into the problem of distributed termination.

1.3. Approach

This author believes that the algorithms can be best understood by grouping them. Initially many restrictions are put on the communication channels which make developing and understanding algorithms easier. In addition, communication may block a process (*synchronous*)[1] or allow the process to continue to execute (*asynchronous*).[2] Furthermore, initial efforts allowed one process to control the system and are referred to as *centralized.* Centralized solutions have the inherent property that the control code contained in each

1 The sender does not send the message until the receiver is ready to receive.
2 The sender does not wait for the reception of the message to occur.

process cannot be the same.

The centralized solutions to the distributed termination problem make use of a single process, the *detector*, to detect termination. The detector initiates the detection algorithm and is the only process with sufficient information to determine if termination has actually occurred. The messages the detector sends out are called *probes*. Chandy and Lamport [CHAN85] introduced the term *distributed snapshot* to refer to the method by which processes record their own states and the states of communication channels so that the set of process and channel states recorded form a global system state.

The earliest centralized algorithms used the CSP notation described by Hoare [HOAR78] to pseudo-code their algorithms. These algorithms are especially vulnerable to deadlock due to Hoare's definition for communication. The author assumes the reader has a basic knowledge of CSP; however a few points are of special interest. It is important to remember that the communication is synchronized in CSP. In addition there is no buffering; therefore, if process p_i wishes to send to process p_j, p_i must wait until p_j is ready to receive from p_i. When both the sender and the receiver are waiting for each other, the two processes are said to *correspond* and a *simultaneous communication* occurs. This property of CSP can lead to deadlock if there is insufficient nondeterminism in the control code. If two processes are each waiting for input from the other, deadlock could easily occur. An unusually large number of messages will increase the possibilities for deadlock. The CSP requirement of simultaneous communication led to the description of these algorithms as synchronous.

Francez distinguishes between "the instance at which communication is enabled and the instance at which it actually takes place" [FRAN82,282]. An *insistant* communication

is a communication "which, once enabled, prevents any other activity until it occurs (even if there are alternative enabled communications)" [FRAN82,289]. An *indulgent* communication is a communication which occurs only whenever a process is not doing anything else, i.e., "the enabling of a communication is intentionally separated from its actual occurrence" [FRAN82,289]. The earliest algorithm presented by Francez [FRAN80] uses insistant communication - once the cc are enabled, no bc may occur until the cc occurs, i.e., the underlying computation is frozen. In his later algorithm [FRAN82], Francez uses indulgent communication, i.e., although a cc may be enabled, alternative bc may occur, and the basic computation is not frozen.

Later efforts were fully distributed in an attempt to make the systems more fault tolerant. The *fully distributed* solutions to the distributed termination problem make use of *symmetric* code - the code for the control algorithm is the same on all processors. In addition, a distributed solution allows any node to initiate termination detection and any node, not necessarily the initiator, to actually detect termination.

The reader should be aware that solutions are classified as centralized or distributed based on the mechanism used to detect termination. This means that a either a centralized or a distributed termination algorithm can be built on a distributed application. The key issue is that, in centralized solutions, the initiator is determined **before** the application begins execution and **can not change**. In the fully distributed solutions the initiator is determined **during** execution of the application. Either all of the processes can take turns being the initiator or all of the processes may have the capability to be the initiator at any time (or even the same time) during the execution of the application.

In several of the distributed algorithms, instantaneous communication is assumed. An

instantaneous communication is defined as when a send/receive occurs, the message is at the sender or at the receiver - not on the communication line itself.

As knowledge of the distributed termination problem grew, the researchers realized that an algorithm could best be developed through the use of invariants. An *invariant* is a condition which must hold true at all times during the solution of a problem. The use of invariants has the advantage of making it easier to prove the algorithms correct.

There are several other conditions to be considered which are beyond the scope of this paper. This paper will consider only *static* distributed systems - the process set is fixed for the lifetime of the system. But it is also necessary to find efficient and correct algorithms for *dynamic* distributed systems - processes can be created and/or destroyed during the lifetime of the system. The entire issue of faulty processors will be avoided - techniques for discovering that a processor is faulty and for correcting the results of its behavior need to be developed and applied to the distributed termination problem but will not be discussed here. Also the effects of message loss due to a faulty channel will not be explored, although the literature indicates that termination cannot be verified if messages are lost.

1.4. Correctness of a Solution

In order for a distributed termination algorithm to be correct, the algorithm must be guaranteed not to cause a *false termination.* False termination is the term used when the control code declares a termination which has not actually occurred - some process is still active. Any correct termination algorithm will have to be able to detect all possible cases of *behind the back communication* in order to prevent false termination. Behind the back communication can occur whenever the detector has visited a process, finds it passive, and assumes it will remain passive. An active process which has not yet been visited by a

detector may send messages to any process in the system - including processes already visited and assumed passive by the detector - and cause them to become active. If a detector does not learn of this behind the back communication, a false termination will be found.

Apt and Richier [APTR85,480] suggest the following criteria as a guide to judging if a solution to the distributed termination problem is correct.

Property 1: Whenever P properly terminates, then all processes are passive.

Property 2: There is no deadlock in P.

Property 3: If all processes become passive then eventually P will properly terminate.

Property 4: If not all processes are passive then eventually a statement from the original program will be executed.

These properties are especially important for the synchronous solutions. Because of the nature of CSP it is very easy to write control code which will deadlock and violate property 2. Property 1 is really saying that a false termination will not be detected, i.e., all behind the back communication will be caught. Property 3 says that if termination occurs, it will be detected. Property 4 says that if termination has not occurred, then the basic computation will not be prevented from executing.

1.5. Properties of an Asynchronous Solution

Ferment and Rozoy [FERM87] introduce a discussion of the problems involved in asynchronous communication. Most of the algorithms for the asynchronous case to be presented in this paper assume that either all communication is instantaneous or a process

can test for the emptiness of its buffers. When the network is not a local one, neither one of these properties would hold and other information would be needed.

If a process can neither test for the emptiness of its buffer nor perform instantaneous communication, Ferment and Rozoy suggest that there are three critical parameters. If one of these parameters can be met then finite termination algorithms can be found for the system. The parameters are:

1. Transmission Order - All messages that have the same priority and are sent over the same communication channel obey the *fifo* rule.[3]

2. Priority of Detection Messages - All cc have a lower priority than bc.

3. Communication synchronicity - There is a uniform bound during which messages are treated, i.e., the number of messages processed before any given message is received and processed is bounded. This is a bound on actions, not on time and is clearer within Rozoy's definition of the distributed system as a finite state machine.

All of the algorithms presented in chapters 4 and 5 assume either instantaneous communication or the ability to test for an empty channel, as well as a fifo ordering of messages. Chapter 6 will look at algorithms that have the ability to test for an empty channel.

1.6. Underlying Computation

Several of the algorithms assume a *diffusing computation* for the underlying computation. A diffusing computation is based on a directed graph, in which one of the nodes has no incoming edges - this node is called the environment;[4] all of the other nodes are called

3 The ordering of the messages is First-In-First-Out.
4 It acts as such to the rest of the graph.

the 'internal nodes'. The environment may send a message to one or more of its successors in the graph. This message is sent only once. The internal nodes may send messages to their successors only after their first message has been received.

An *engaged node* is a node not in its neutral state, i.e., it has received at least one message and has not returned signals for every message received.

The edge from p_i to p_j is an *engagement edge* if p_j is a neutral node and p_i is the first process to send p_j a message.[5]

1.7. Tools for Distributed Algorithms

Raynal [RAY88b] suggests three tools which can be used to implement fully distributed algorithms:

1. the use of markers or tokens
2. logical clocks
3. sequence numbers for messages.

Although there is no designated process as detector, when a marker or token is used, the detection of termination is restricted to the process which holds the token. The token is initially placed by the compiler. In effect, the only process that can initiate a probe is the process holding the token. The token is passed along until the probe is falsified - i.e., an active process is found. A new probe is then issued by the process holding the token when its b_i becomes true. When a process receives the token containing its own probe unfalsified, then termination is detected. There are several ways of guaranteeing that this will happen;

5 p_j may go from neutral to engaged and back to neutral. When p_j goes neutral it has no engagement but it may become engaged again with the same (or a different) process and have the same (or different) engagement edge.

Arora and Sharma [AROR83] offer one solution.

Events occurring in a distributed system are sometimes difficult to order. Lamport [LAMP78,558] discusses "the happened before relation" and gives a distributed algorithm for extending it to a consistent total ordering of all of the events in the system. The result of this algorithm is the concept of a distributed logical clock. Abstractly, "a clock is just a way of assigning a number to an event, where the number is thought of as the time at which the event occurred" [LAMP78,559]. A clock for each process may be implemented by a counter which is used as a timestamp for events. The clocks may be synchronized within the system whenever a message is sent. Each message contains a timestamp. The receiving process increments its clock (counter) to be whichever is greater - its own time (represented by its counter) or the time (represented by the timestamp) of the message.

This method of synchronization can also be used when real clocks are used. Since no two clocks will run at exactly the same rate, they will tend to drift further and further apart. A resetting of clocks through comparison of times whenever a message is received is sufficient to order the events in a system. Apt and Richier [APTR85] offer several solutions for virtual clocks derived from solutions which assume real clocks.

The third tool, sequence numbers on messages has none of the problems of the previous tools. With the use of sequence numbers, each process manages a counter. The counter is initialized to zero and increases monotonically. Whenever a message is sent out, the current sequence number is incremented and the new sequence number is attached to the message. The counter at the site of each process is used only for messages which it sends.

When a control message is returned to the sender, the sender can verify that no cc

have been sent more recently by comparing the current sequence number with the sequence number attached to the incoming message. A more recent cc has not been sent if they match.

Processes receiving a message from another process can verify that it is a new message (rather than a duplicate) if the sequence number is not a duplicate of the sequence number attached to other messages received from that process.

If sequence numbers are used, any process may initiate a probe at any time. In addition, the overhead of logical clocks is non-existent. However, as a result, it is possible that all processes may have a probe in the system at the same time. Some method must be used to eliminate unnecessary probes to reduce the number of cc. None of the synchronous distributed solutions make use of sequence numbers; however, we will see them in Arora's [AROR87] asynchronous solution.

Arora and Sharma [AROR83] offer a construct which illustrates how b_i may be set. The programmer is given two new statements: set and reset. Both of these statements take a variable number of boolean parameters. If all of the boolean expressions of the set statement are true, the process is passive[6] and b_i is set equal to true. If all of the boolean expressions of the reset statement are true, the process is active and b_i is set equal to false. It is not clear if the assignment to b_i is implicit to these statements or must be done explicitly by the programmer. In either case, the programmer is expected to add these statements to his distributed program in the appropriate places.

6 Arora uses the term *stable* to mean passive.

1.8. Network Definitions

A *spanning tree* contains all the nodes in the network and some collection of edges so that there is exactly one path between each pair of nodes; it is inherently acyclic.

A *Hamiltonian ring* is a cycle in which each node is visited exactly once. Thus each process has a successor on the cycle to which it can forward a message and a predecessor from which a message can be received. A Hamiltonian ring for the cc is sometimes superimposed on the system. Sometimes the term *circular directed graph CDG* is used instead.

In a *broadcast network* there is a single communication channel shared by all of the nodes in the network. Inherent in broadcast systems is that all messages sent by any node are received by all other nodes. Something in the message itself must specify for whom it is intended. After receiving a message not intended for itself, a node just ignores it.

In this paper, the term *arbitrary network* will be used to indicate that the network structure is not specified.

Two nodes are *neighbors* if they are directly adjacent, i.e., they have a direct communication line.

The *diameter* of a network, D, is the greatest distance between any two nodes in the network and the *distance* between any two nodes is the shortest path between them. The authors of the presented distributed termination algorithms do not define the method used for computing the distance between two nodes; however, this author has found that the number of hops between nodes covers most cases. When computing the time and message complexities, this author uses the number of hops.

In a *weakly connected network* with directed edges, there exists a cycle, not necessarily a simple cycle, which includes every edge of the network at least once.[7] In a *strongly*

7 For any p_i, p_j there is either a path $p_i \rightarrow p_j$ or $p_j \rightarrow p_i$, but not necessarily both paths.

connected network with directed edges, removal of any one edge between any pair of nodes maintains connectivity.

1.9. Assumptions

There is one assumption made for all of the algorithms: The underlying computation P is written in such a way that it will always eventually terminate and therefore, each p_i will also terminate[8] - deadlock will not occur in the computation itself. Initially, the algorithms we consider will assume that all channels of the network are reliable: they do not alter messages, lose them, duplicate them or desequence them, later we will relax this restriction to some extent.

1.10. Restrictions

Francez suggests the following criteria for an ideal solution for the detection of termination in P [FRAN80,44]:

1. The required solution is independent of the specific problem P is trying to solve.
2 The control communication is inserted into the solution for P with the smallest possible additions to the basic communication.
3. The solution is independent of the number of processes, n, into which P is broken up.
4. The solution is independent of the specific neighborhood relationships holding among the p_i's.

8 This assumption must be made because whether or not a process will terminate is generally an undecidable problem.

5. The solution does not require new communication channels, i.e. the cc use the same channels as the bc.

1.11. Formal Definition of Termination

We can summarize our discussion as follows. The application P consists of a group of communicating sequential processes, p_i, $i = 1...n$, which execute in parallel in a distributed environment. Control code is added to the p_i so that termination may be detected. These processes communicate with each other only through exchange of messages. Each process has a local predicate, b_i, $i = 1...n$, to represent its local condition for termination. When the conjunction of all of the b_i's is true at the same time instant, the global predicate, B, representing the GTC is also true and all of the processes may terminate. Note, distributed termination "is actually a deadlock situation where all processes are ready to accept but none is willing to initiate communication" [FRAN82,288]. Therefore the control code is used to recognize this deadlock situation as a readiness to terminate.

1.12. Phases of the Termination Problem

The distributed termination problem may be broken into two phases: the detection phase and the termination phase. The termination phase[9] is trivial, once detection has occurred, and is not dealt with in this paper nor in the majority of the literature. Although, strictly speaking, termination occurs only when the termination phase is complete, this paper will call the deadlock situation, where all processes are ready to terminate, termination.

9 After B has been met, the termination phase consists of the explicit communication to each p_i to terminate.

1.13. Computation of Overhead

Overhead can be computed for the termination detection algorithms with respect to the number of cc. When computation of the overhead requires knowing the total number of bc, this number will be represented by m. There are two cases to consider:

1. The best case is when the initiator of the probe is the last process to meet its b_i and therefore, as it visits each process, p_i, the detector finds that p_i has also met its b_i.

2. The worst case is when each process sends only one bc (or no bc) before meeting its b_i. Then every subsequent bc is sent to a process, p_i, whose b_i has already been met. When p_i receives the bc it becomes active again.

The overhead includes both control messages needed to enable the detection of termination and control messages required for detecting termination after it has actually occurred.

The actual overhead of each algorithm must also take several other things into consideration. The most important of these is the amount of additional code which must be executed whenever a bc is performed. Also of importance is the amount of extra memory needed for each process to implement the control code. In a few cases the size of the detection message will also be of importance. In most of the algorithms presented in this paper the size of the message will remain constant independent of the size of the system. However, in the case where messages are not received in the order sent, the size of the detection message may be dependent on the number of processes in the system [KUMA85] [MAT87A] [MAT87B]. The algorithm presented by Rozoy [ROZO86] also uses a message dependent upon the size of the network.

The requirements for a real-time system, an operating system, or a user application

dictate that no one algorithm is the best solution for all situations. For example a real-time system might require an algorithm which is fastest most of the time, even if it uses a lot of memory. A user application might be concerned with the control algorithm's effect on the bc and an operating system might need an algorithm which uses the least amount of memory.

1.14. Perspectives

In this paper, solutions for which the channels have the most restrictions will be looked at first. Synchronous communication is also more restrictive and therefore will be considered first. Synchronous solutions can be either centralized or fully distributed. The centralized solutions more closely resemble a single-processor system and are useful to the understanding of termination. The fully distributed synchronous solutions offer an opportunity to look at the additional issues involved when the control algorithm is distributed. Yet, the environment is still very much controlled through the synchronous communication.

Next asynchrony will be introduced. Again there are several centralized solutions as well as several fully distributed solutions.

Finally some restrictions on the channels will be lifted, e.g., the order of the messages will not be required to be first-in-first-out. This presents additional problems in the asynchronous environment, but cannot happen in the synchronous environment. Both centralized and distributed asynchronous distributed solutions will be considered when restrictions on the ordering of messages are removed.

1.15. Historical Perspectives

Unfortunately, from an historical perspective these groupings overlap and researchers borrowed ideas from each other. For example, in a chapter where only algorithms using synchronous communication are considered, the reader will notice that ideas are incorporated that may have been developed by another researcher for an algorithm using asynchronous communication that will be introduced in a later chapter. In an attempt to make the actual ordering clearer, a chronological history is given here.

The earliest algorithms were written in 1980. Francez's algorithm [FRAN80] used synchronous communication and was based on a spanning tree model, which the programmer was required to derive. Dijkstra's algorithm [DIJK80] was also based on a spanning tree model; however, the application itself - a diffusing computation - built the tree and the communication was asynchronous.

Francez provides further background information in his article on interval assertions [FRAN81] and an improved version of his first algorithm [FRAN82]. Misra [MISR82] then merges the idea of a diffusing computation [DIJK80] and Francez's use of CSP [FRAN80].

The literature of 1983 presents four algorithms. Arora [AROR83] and Rana [RANA83] offer synchronous distributed solutions. Arora's solution uses tokens and a ring network topology. Rana uses clocks and his results are incorrect. Dijkstra [DIJK83] offers a solution that uses tokens and a ring topology. Since he does not specify what kind of communication occurs, only that the communication facility exists, the algorithm will be included with the asynchronous centralized solutions. However, the importance of this algorithm is that Dijkstra also introduces the idea of using invariants to develop the algorithm and *colors* the tokens to represent whether or not a bc has occurred. Misra [MISR83]

applies this idea of colored tokens to the asynchronous distributed situation.

In 1984, Topor [TOPO84] merged the idea of a colored token [DIJK83] with that of a spanning tree [FRAN82].

1985 brought a wide variety of solutions to the problem. Apt and Richier [APTR85] offered a solution, based on the use of clocks, for the synchronous distributed case that uses a ring topology. Richier [RICH85] also offers a very complicated solution for the same case, also using a ring. There is a minor error in this algorithm: one of the variables never "gets set". It does not use clocks; however, a boolean variable replaces the idea of colored tokens. Szymanski [SZYM85] offers an algorithm for the asynchronous distributed case. However, it is specific for his application which must be synchronized. Strictly speaking, he does not solve the distributed termination problem as defined by Francez [FRAN80]. Finally Kumar [KUMA85] offers several algorithms for the asynchronous distributed case which lift the restriction of maintaining the ordering of messages in the send/receive process.

1986 brought many possible solutions; however, most of the researchers presented them in conjunction with an idea they considered more important, i.e. the algorithms were an afterthought used to illustrate their point. Apt [APT86] offers a synchronous centralized solution for a ring topology. However, his real purpose is to introduce correctness proofs for termination algorithms that use CSP. He uses his algorithm to show how a proof may be done. Rozoy [ROZO86] offers an asynchronous centralized solution based on a finite state machine model. However, her real prupose is to introduce a theorem on the bounds for the cc in any asynchronous algorithm. Skyum [SKYU86] introduces an algorithm for the asynchronous distributed case that assumes a strongly connected broadcast

network. However, his application also requires synchrony and Francez's termination problem is not solved. Arora [AROR86] offers an asynchronous distributed algorithm based on information a process has about its neighbors. However, it is incorrect according to Tan et al [TAN86].

In 1987, we have several incorrect algorithms. Chandy [CHAN87] offers a mathematical proof for the verification of termination detection. He assumes the existence of a distributed snapshot and proves that termination has/has not occurred based on the snapshot. Arora [AROR87] presents an algorithm for the asynchronous distributed case that uses sequence numbers and the distance function. This author finds the algorithm to be incorrect due to the fact that it does not take into consideration all of the possibilities for behind the back communication. Hazari [HAZA87] presents an incorrect synchronous distributed solution because the algorithm does not take "into account that passive processes may be reactivated by communications from processes that are still active" [TEL87]. Ferment and Rozoy [FERM87] present ideas relative to the problems which must be surmounted in the asynchronous case. Mattern [MAT87A] [MAT87B] presents several algorithms in the Kumar strain for the asynchronous distributed case where the ordering of messages is not maintained.

1988 brings several solutions for the asynchronous case. Arora [AROR88] offers a centralized solution for a ring. The solution is quite complicated due to the fact that he has previously presented incorrect solutions and been called on it in the literature. Haldar's solution [HALD88] is based on Arora's [AROR87] [AROR83] and is simpler although it uses the same concepts of a distance function and sequence numbers. Tel and Mattern [TELM89] prove that this solution is incorrect. Arora and Gupta [AROG88] dispute Tan et al's criticism [TAN86] of their 1986 algorithm [AROR86] and present an augmentation

of the algorithm which this author believes is still incorrect. Eriksen [ERIK88] presents a solution for the distributed case on a ring topology. The only information each process needs to know is the upper boundary on the number of processors in the network. Huang [HUAN88] offers a distributed solution which uses logical clocks. The algorithm is described in a general fashion for use with any topology. The reader would have to adapt the algorithm to the particular network topology used.

This author was only able to find two articles in the literature for 1989 - both by Arora and Gupta. In the first article [AROG89], they present a framework for deriving ring-based termination algorithms. It appears to this author to simply be a generalization of algorithms previously presented by these authors and as such will not be discussed in this paper. The second paper [AROR89] introduces three algorithms which employ bi-directional control communication around a ring. One is for the asynchronous centralized case and the other two are for the asynchronous distributed case.

1.16. Summary

This chapter gave the reader an overview of the contents of this paper. It contained a formal description of the termination problem and an overview of its development. All terms that are used in this paper were defined here. Finally the logic behind the ordering of topics and a historical perspective were presented in order to clarify some of the overlapping topics.

2. SYNCHRONOUS CENTRALIZED SOLUTIONS

2.1. Introduction

This chapter will begin the analysis of synchronous centralized solutions by looking at the issues involved in declaring termination for this case. It will also make explicit the assumptions used in each of the algorithms presented. It will then look at the algorithms written to solve this version of the problem and finally analyze the overhead involved in each case.

2.2. Issues

Centralized solutions have the advantage that they make the issues in termination clear, because they are not clouded by the issues in a truly distributed system.

One of the most basic questions is what conditions are sufficient to determine that a computation P is completed and therefore can terminate. The obvious answer is that all p_i have satisfied their local b_i. However it turns out this is not a sufficient condition.

For instance, consider a computation P, which consists of three processes p_1, p_2, and p_3, connected in a ring. Suppose processes p_1 and p_2 are passive, the detector has visited them, and knows that they are passive. Suppose also that p_3 is active. Before going passive p_3 sends one final bc to p_2, causing p_2 to become active. When the detector visits p_3, it is passive and the detector assumes termination - all processes are passive. However, this is incorrect because p_2 is currently active. This is an example of behind the back communication. Any correct termination algorithm will have to be able to detect all possible cases of behind the back communication.

2.3. Assumptions

At this point, the assumption that all channels of the network are reliable holds. In addition all messages are received in a relatively short, finite amount of time, and both send and receive are blocking operations.

2.4. Solutions Using a Tree Topology

Many of the early synchronous solutions assume a network with a tree topology superimposed upon it for the control communication. In the earliest solutions the tree has to be derived by the programmer who knows the TDS for each process. There are three solutions of this type. The first published algorithm is written by Francez [FRAN80]. This algorithm is extremely cumbersome and freezes the bc of the computation; however, it illustrates the issues involved. The second solution, also by Francez [FRAN82], eliminates freezing of the computation and is somewhat more efficient. The third solution, written by Topor [TOPO84] borrows from Francez [FRAN82] and Dijkstra [DIJK83]. The result is clearer than Francez but efficiency and overhead remain the same as Francez's algorithm [FRAN82]. There is also a solution, which uses a tree topology, that does not require the programmer to derive the tree. Misra [MISR82] uses the diffusing computation, as introduced by Dijkstra [DIJK80] for his underlying application. The diffusing computation builds the tree for him.

Francez assumes the network is represented by a weakly connected, directed communication graph, G_P, containing one node for each p_i. A second graph, T_P is derived from G_P and reflects the termination dependencies within P. T_P is also a weakly connected, directed graph. The strategy is to arrange the dependencies among the p_i such that T_P will be acyclic (a spanning tree). (If T_P contains a cycle, deadlock situations

could occur.) After the spanning tree is derived, one process, the root, is assigned the job of detector. The algorithm works as follows:

1. When the root of the tree reaches its final state, i.e., its b_i is met, it initiates a control wave to its descendants in the tree.

2. When the wave reaches a node, it does the following:

   If the node has satisfied its local predicate, bc is frozen (this prevents behind the back communication) and the wave is passed down to all of its descendants.

   If the node has not satisfied its local predicate, the wave is stopped and a negative response is passed back up.

3. When the wave reaches the leaves it is passed back up to the root as follows:

   If all of the nodes in p_i's subtree have satisfied their local predicate, pass a positive response up to p_i's parent.

   If p_i, or any node in its subtree, has not satisfied its local predicate, pass a negative response up to p_i's parent.

4. If the root receives a positive response, it initiates termination. If the root receives any negative response, it propagates an 'unfreezing wave' so that basic communication may resume. When this wave is received by the leaves they initiate a fourth wave. This wave may pass a node which has not met its local predicate only after that node performs one basic communication. This prevents the possibility of an endless control loop. Only when the root receives this fourth wave may it initiate another control wave to detect termination.

Notice that a cc may be received only when the process is waiting for a bc; however,

once enabled, cc is insistant.

The overhead in the best case is $2n$ where n is the number of processes in the system. In the worst case it is $(4n * m) + 2n$[1] where m is the number of basic communications. In this algorithm the control code interferes with the efficiency of the execution of the application itself through the freezing of bc.

The resulting algorithm is not efficient and it would be better if the programmer did not have to derive the spanning tree himself. However, Francez made a large contribution to the field by defining the problem and exploring the dependencies between the processes into which the application must be partitioned.

A later algorithm offered by Francez [FRAN82] has a smaller overhead with respect to the control strategy and does not delay the computation by freezing the bc. The control communication is indulgent instead of insistant to further reduce the time consumed by it and increase efficiency. For this algorithm each node has two additional local variables:

advance	initially true, which is set to true whenever a basic communication takes place
send-w2[*#of children*]	also initially true, which is set to false to indicate wave w2 may be propagated down to this child.

The initial idea was to have three waves; however, waves one and three can be combined to form the single wave w1-3. The algorithm works as follows:

1 Nissim claims the overhead is $4n*m$ because he counts the extra cc only until the moment when termination occurs, but does not count the cc to detect that termination has actually occurred.

1. Wave w1-3 is initiated by the leaves as their local predicates, b_i, become true. The current value of *advance* is passed up to the parent, the variable *advance* of the leaf is then set to false, and the node sits and waits for a message from any other process, although it may not initiate any communication unless it receives a communication first.

2. When wave w1-3 is received by an internal node, it may propagate it up to its parents when the following conditions are met:

   1. It has received the wave from all of its children

   2. Its local predicate b_i has been met.

   It sends the 'or' of all of the values of *advance* of its children 'or'ed with the local value of *advance*. It then resets its own *advance* to false.

3. When wave w1-3 is received by the root from all of its children, it checks the accumulated value of *advance*. If the value is false it initiates termination. If the value is true, some process has had a communication since the last w1-3 (and is therefore active) and the computation may not terminate.

4. Wave w2 is then propagated down. The root first sets *send-w2*[j] to true for all of its children, j. Then indulgently, it sends w2 to each child j, resetting *send-w2*[j] to false as w2 is sent. Upon receiving w2, each internal node, indulgently, also sets *send-w2*[j] to true, sends the wave to all children j and resets *send-w2*[j] to false.

5. Upon receipt of w2 by a leaf, it may re-initiate wave w1-3 as soon as its local predicate b_i becomes true.

Francez removes the delay of bc in favor of cc through the allowance of indulgent cc and the addition of the variable *advance*. The number of control communications for this algorithm, according to Francez, is reduced to $2n*m+n$ in the worst case[2] (from $4n*m+2n$) where m is the number of basic communications and n is the number of processes. The best case[3] is now $3n$. However, the programmer is still responsible for deriving the spanning tree.

The third algorithm using a spanning tree is Topor's. Dijkstra [DIJK83] introduces the idea of colored nodes and colored tokens. Topor borrows this concept to replace the accumulated values of the variable *advance* in Francez's algorithm. The algorithm works the same as Francez's [FRAN82]; wave w1-3 passes a node under the same conditions; however, the 'color' of the node determines the 'response'.

The algorithm is developed by defining an invariant composed of the following predicates:

R0: All nodes which have passed the token on are passive.

R1: Some node which has passed the token on is black.

R2: The token is black.

The resulting invariant, which must hold true at all times during the computation is: R0 or R1 or R2.

The algorithm makes use of the following rules [TOPO84,34-35] to keep the invariant while passing the token up through the nodes to the root:

---

2 actually $2(n-1)*m+(n-1)$

3 Wave w1-3 will never cause a termination the first time.

Rule 0: A passive leaf that has a token transmits a token to its parent; a passive internal node that has received a token from each of its children transmits a token to its parent; an active node does not transmit a token. When a node transmits a token it is left without any tokens.

Rule 1: A node sending a message becomes black.

Rule 2: A node that is black or has a black token transmits a black token, otherwise it transmits a white token.

Rule 3: A node transmitting a token becomes white.

Rule 4: If node 0 (the root) has received a token from each of its children, and it is active or black or has a black token, it becomes white, loses its tokens, and sends a repeat signal to each of its children.

Rule 5: An internal node receiving a repeat signal transmits the signal to each of its children.

Rule 6: A leaf receiving a repeat signal is given a white token.

This algorithm produces an overhead which is the same as the overhead for the algorithm of [FRAN82]. The use of a token to replace the disjunction of the variable *advance* is easier to understand; however, it appears nothing has been gained, except to combine the ideas of the two algorithms.

Misra's algorithm [MISR82] is an adaptation of Dijkstra's diffusing computation [DIJK80] using asynchronous communication to the synchronous case. The reader is advised to read and understand Dijkstra's algorithm[4] before trying to understand Misra's.

4 See section 6.1 for an explanation of Dijkstra's algorithm.

Misra's problem is different from Dijkstra's because in the CSP implementation "the sender is unable to determine locally whether it can ever send a message, because that depends on the state of the receiver as well" [MISR82,38].

Each process, p_i, keeps an activity graph that contains a list of all the processes, p_j, that have sent p_i messages and a list of all the processes p_k to whom p_i has sent messages. The p_j are the arc predecessors of p_i and the p_k are the arc successors of p_i. Whenever a bc is sent from p_i to p_j an activity arc is created and maintained in the activity graph. If prior to receiving a message from p_i, p_j is disengaged, then two activity arcs are created. Arc (i,j) is called a tree arc and arc (j,i) is a non-tree arc. The tree arcs build the spanning tree which Francez required the programmer to derive. If a process is already engaged, it is unclear if one or two non-tree arcs are created.

In addition each process, p_i, also maintains several variables:

$r_j(i)$	a boolean variable for each neighbor p_j $r_j(i)$ true indicates that p_j thinks p_i is waiting to receive from p_j
$s_j(i)$	a boolean variable for each neighbor p_j $s_j(i)$ true indicates that p_j thinks p_i is waiting to send to p_j
thinkblocked(i)	true denotes that: 1. p_i is not executing 2. for every process p_j that p_i is waiting to receive from, $s_i(j)$ is false

3. for every process p_k that p_i is waiting to send to, $r_i(k)$ is false.

Note: *thinkblocked(i)* denotes only whether or not p_i **thinks** it is blocked; the $r_i()$s and $s_i()$s may be inconsistent with the true waiting status of the neighbors of p_i. Misra contends that this inconsistency does not affect the correctness of the algorithm.

Two methods of signalling control messages are now superimposed on the computation. The *A-signals* delete a single activity arc by updating the activity graph. The *B-signals* are used by the process, p_i, to inform a neighbor, p_j, that p_i "has changed its waiting status (from waiting-to-send/receive to not-waiting-to-send/receive or vice versa)" [MISR82,39] for p_j.

Non-executing processes are always waiting for A- and B-signals. Therefore, there is no blocking for signal transmission. The authors do not mention what happens in the case where a signal is sent to an executing process. Because of the definition of CSP, this is sufficient cause for the algorithm to fail. The authors offer the following rules to govern the transmission of signals:

Rule 1: (waiting condition for transmission of B-signal). Process p_i waits to send a B-signal to process p_j if and only if $r_j(i)$ or $s_j(i)$ is inconsistent with process i's true waiting status. (Note that i can deduce $r_j(i)$ and $s_j(i)$ from the B-signals that it has already sent.)

Rule 2: (waiting condition for transmission of an A-signal to delete a non-tree arc) Process p_i waits to send an A-signal to process p_j, where (j,i) is a nontree arc, if *thinkblocked(i)* is **true**.

Rule 3: (waiting condition for transmission of an A-signal to delete a tree arc) Process p_j waits to send an A-signal to process p_i, where (i,j) is a tree arc, if and only if:

1. *thinkblocked(j)* is **true**
2. there is no other incident (i.e. outgoing or incoming) activity arc on process j
3. process p_j has ensured (by sending appropriate B-signals) that, for every neighbor p_k, $r_k(j)$ and $s_k(j)$ truly reflect the waiting status of process p_j.

It is not explicitly stated in the paper, but this author assumes that when a bc is sent two arcs are **always** created. Then when one of the processes sends the A-signal (and it should be the receiver if this algorithm is to mimic Dijkstra's) then both "arcs" are deleted because the activity graph is updated. In the case where one tree arc and one non-tree arc are created, the A-signal can only be sent by the receiver of the bc because that process is the only one which can delete the tree arc. This is a real fuzzy area in Misra's presentation of the algorithm.

The computation is started by the environment; A-signals are sent for the same reasons as in Dijkstra's algorithm [DIJK80] - to help the receiving process keep track of the number of messages received. B-signals are sent as needed, by process p_i to process p_j to inform p_j of p_i's wish to rendezvous or not.

The control overhead for this algorithm consists of the total sum of the A- and B-signals. In the worst case, B-signals are sent by one process to all of its neighbors every time a message is sent. This could be as many as $m * n$ cc if all nodes are connected to

every other node in the network. In addition m A-signals are sent - one for each bc sent. This gives a total of $m + m * n$ cc. The additional memory overhead would be quite substantial in this case. Each process would have $2 * n + 1$ boolean variables, for a total of $2 * n^2 + n$. In addition each process would have two arrays - one for its predecessors and one for its successors. Since it is possible for a process to be on these lists multiple times, the total amount of memory used in the system for the lists would be $2m$. Each bc would have additional overhead as well - both the A-signal and one or more B-signals, as well as some method of keeping track of whether or not the receiver of the bc is already engaged. The amount of overhead for Misra's algorithm is substantially higher than the overhead for the other tree topologies that we've looked at. In chapter 6, where Dijkstra's algorithm is introduced, the reader will find that Dijkstra's algorithm also has a much lower overhead.

2.5. Solutions Using a Hamiltonian Cycle

When a Hamiltonian ring is used for the cc, the restriction not to add new channels for the control communication is released, i.e., a process, p_i which may not send a bc to p_j may send a cc to p_j. There are two centralized solutions using a Hamiltonian cycle which we will look at. Essentially they are the same, although they use different methods to keep track of communication between two probes.

Francez [FRAN81] introduces the idea of an interval assertion implemented with logical clocks. An interval assertion (T^K, T^L, B) where $T^K < T^L$ is defined as "true if B holds during the entire time interval [T^K, T^L] and false otherwise" [FRAN81,283]. A sequence of sets of time instances T^1, T^2,... are chosen dynamically such that $T^k = \{t_i^k \mid 0 \leq i < n\}$ and $\max_{0 \leq i < n} t_i^k < \min_{0 \leq i < n} t_i^{k+1}$. At time t_i^{k+1} the i^{th} process, p_i, makes a local test in order to verify the interval assertion $\alpha_i^k = (t_i^k, t_i^{k+1}, b_i)$. The truth value of the interval

assertions for all the p_i is collected by one process. If all the α_i^k are true, then $\{\alpha_i^k \mid 0 \leq i < n\}$ is a set of true local interval assertions which covers $(\max_{0 \leq i < n} t_i^k, \min_{0 \leq i < n} t_i^{k+1}, B)$. If any are false then the process can start a new set of tests.

Francez uses this idea of interval assertion along with the following three additional variables for this control algorithm:

bc_i: true if b_i is true and no bc has occurred since b_i first became true in this time interval, otherwise false

$count_i$: counter for the control messages which acts as the upper end in the time interval assertion

db_i: a copy of $count_i$ at the time bc_i last became true - it serves as the lower end of the time interval in the interval assertion

The algorithm works as follows:

1. Process p_0 is chosen to be the detector. It initiates the detection wave when its bc_i becomes true. Each receiving process passes the wave on when its bc_i is also true. The first wave around the ring will always fail.

2. Now all processes have the start 'time' for the interval assertion, $db_i = 0$, and the current 'time' $count_i = 1$. Any process which is activated will not be able to pass the wave on until b_i becomes true again; however, then $db_i < count_i$ will not hold and the wave will contain the value false.

3. When p_0 receives the wave it may terminate if it is true or initiate a new wave if it is false.

The overhead for this algorithm in the best case will be $2n$.[5] In the worst case it is $n * m + n$.[6] However, there is additional overhead to keep track of the 'time'. The point of interest is that although few researchers reference this work, the concept of interval assertions, as introduced here, is the basis for all methods of detecting termination. Some attempt must be made to keep track of any communication which occurs between probes - whether it be an explicit interval assertion or the colored token used by Dijkstra [DIJK83] or any of the more complicated methods to be described later.

The main purpose of the paper presented by Apt [APT86] is to present a method for doing correctness proofs on CSP programs that are in normal form.[7] He introduces the criteria for a correct solution (presented in chapter 1) and presents a very simple algorithm for the sole reason of showing how to do a correctness proof. This author does not believe that an analysis of Apt's proof method will aid the reader in gaining understanding of the distributed termination problem because it is useful for such a restrictive set of CSP programs. Therefore, we will look at the algorithm that is offered, but not the correctness proof for it.

Apt claims his algorithm is based on Francez's interval assertions [FRAN81] and Dijkstra's colored tokens [DIJK83]. He has taken the idea that the algorithm must keep track of any bc taking place during any given time span. The method he uses to do this is

---

5 The token must traverse the ring at least twice. The first time it finds the b_i of all of the processes to be true. The second time it guarantees that the b_i of all processes remains true - no behind the back communication has occurred.

6 In the worst case every bc causes a falsification of the probe. Then the probe must continue its traversal of the ring to reach the detector. One final probe must be sent to guarantee that termination has occurred.

7 A CSP program in normal form is defined as follows:

P = $[p_1 \,||\, ... \,||\, p_n]$ where for every i, $1 \leq i \leq n$, $p_i :: init_i : *[S_i]$ and S_i is of the form $\underset{j \in \Gamma_i}{\Box} g_{i,j} \rightarrow s_{i,j}$ and

1. each $g_{i,j}$ contains an I/O command addressing p_j
2. none of the statements $init_i$ or $s_{i,j}$ contain an I/O command

a boolean variable that is just an implementation of the colored token idea.

Each process maintains the following variables:

send$_i$ true when process, p_i, holds the probe and has not passed it on yet

false when process, p_i, does not hold the probe

moved$_i$ true when a bc has taken place

set to false whenever the probe has been passed to the right-hand side neighbor, it remains false as long as no bc occurs

The algorithm works as follows: p_1 sends a probe with a value of true to its right-hand side neighbor when its b_1 becomes true. Each process maintains its copy of *moved*$_i$ based on whether or not a bc has occurred. When the probe arrives, if *moved*$_i$ is still false, then the probe is passed along unchanged. If *moved*$_i$ is true, the probe is falsified and then passed along. Once the probe has become false, it does not become true again. When it returns to the detector, p_1, termination will be initiated if the probe is true. If the probe is false, p_1 will initiate another probe whenever p_1's b_1 is true.

Notice from the code for the algorithm, contained in the appendix, that p_1, the initiator, does not contain the variable *moved*. This is because p_1 does not initiate the probe until its b_1 has been met; therefore, any bc reaching p_1 from p_i will cause the probe to fail when it reaches p_i.

The control overhead for this algorithm will be $2n$ in the best case and $n * m + n$ in the worst case for the same reasons as in Francez's algorithm [FRAN81]. The memory overhead will be 2 booleans per process for a system-wide total of $2n - 1$ boolean variables. The only addition to the bc is to set a boolean variable to true.

2.6. Summary

In this section we looked at algorithms which designate one process as the detector for termination. Some of the algorithms make use of a tree topology, derived by the programmer, for both bc and cc. Others use a Hamiltonian ring as the topology for the cc, although the communication channels for the application could be quite different.

In table 2-1 it is easy to see that the algorithms based on a Hamiltonian cycle have half of the cc control overhead of the spanning tree topologies in the worst case. In the best case, the cc overhead for a ring is the same as that for a tree.

Algorithm	Additions to bc	Best Case Overhead	Worst Case Overhead	Additional Memory[1]
TREE TOPOLOGIES				
[FRAN80]	no	$2n$	$4n * m + 2n$	$< 4n - 3$ booleans
[FRAN82]	set advance = T	$3n$	$2n * m + n$	$< n$ int constants $< n$ integers $< 4n - 3$ booleans
[TOPO84]	make node black	$3n$	$2n * m + n$	$< 5n$ booleans[2]
[MISR82]	send A-signal send multiple B-signals	$mn + m$	no upper bound	$< 2n^2 + n$ booleans $2m$ integers
RING TOPOLOGIES				
[FRAN81]	set bc_i = F	$2n$	$n * m + n$	$2n$ booleans $2n$ integers
[APT86]	set $moved_i$ = T	$2n$	$n * m + n$	$2n$ booleans

1 These are totals for the system, rather than for individual nodes.

2 This author assumes that *color* can be implemented as a boolean.

Table 2-1

Except for Francez's first algorithm [FRAN80], all of the algorithms require at least

one additional statement any time a bc is sent. This statement is an assignment to indicate a bc has taken place; it prevents behind the back communication. Except for Misra's algorithm, the algorithms are equal in terms of additional overhead for the bc.

The table shows that the tree topologies require much more memory than the ring topologies. The main reason for this is the boolean arrays which contain information to keep track of whether or not the wave has been propagated to a child and the children's responses. Apt's algorithm requires the least amount of additional memory.

Overall it would appear that Apt's algorithm has the lowest overhead. The spanning trees, by their very nature will always have communication bottlenecks getting to the root. The ring topology does not have this problem; however, it also has only one process which may initiate and detect termination. Much of the substantial overhead for Misra's algorithm is a result of attempting to convert an algorithm which is simple for the asynchronous case, to the synchronous case. Given a choice, this author believes it would be better for the programmer to derive his own spanning tree than to use a diffusing computation in the synchronous case.

When a specific process is designated as the detector, the control code which is added to each process cannot be symmetric. In the case of the spanning tree there is control code for the root, internal nodes, and leaves. In the case of the ring there is code for the detector and code which can be used by all of the other nodes. We would like to have a termination detection algorithm that eliminates these bottlenecks and has code which is symmetric, i.e., the same for every process. The distributed solutions can solve this problem.

3. SYNCHRONOUS DISTRIBUTED SOLUTIONS

A distributed system provides an environment within which the user is able to spread the processing needs of a program over several processors. If it is to be effective, it cannot be controlled by a single processor. - This would reduce it to a centralized system, with all of the bottlenecks contained therein. However there are situations in which individual processors are required to know the global state of the system, e.g., the distributed termination detection problem. Methods must be derived to enable **each** processor to learn the state of the entire system. This chapter introduces the earliest attempts to do this.

3.1. Assumptions

The assumption that all channels are reliable continues to hold. In addition messages are received in a relatively short, finite amount of time, although both send and receive block. The restriction that no new channels may be added for the control communication is lifted. All of the synchronous distributed solutions assume the existence of a Hamiltonian ring, or circular directed graph (CDG), for the control code.

3.2. Issues

There are three published synchronous distributed solutions which are incorrect. These are of interest because they illustrate the kinds of situations which must be guaranteed not to occur for an algorithm to be correct. Rana [RANA83] proposes an early solution that assumes the existence of synchronized local real clocks. He describes the change in method from a centralized solution as follows:

> Although an arbitrary process may initiate a detection wave, eventually the last processes to satisfy their local predicate would succeed in detecting the termination and thus be responsible for initiating the termination wave. [RANA83,46]

There are two errors in the algorithm that are due to code which does not take all of the properties of CSP into account. The biggest problem is the possibility of deadlock if all processes satisfy their b_i at the same time instant. It is possible that everyone will be trying to send, and no one will be waiting to receive. In addition, the code is written such that, once b_i is satisfied, more than one probe will be sent out. This just adds to the deadlock possibilities as well as adding needlessly to the control overhead.

The algorithm of Hazari and Zedan [HAZA87] is an example of a mistake easily made. They present code in the programming language Occam[1] for both phases of termination. In order to guarantee that communication will not be attempted with a terminated process, they guarantee that initiation of a termination wave will only occur once. However as Tel et al [TEL87] point out, they also allow initiation of a detection wave to occur only once by each process. - This is not the definition of the distributed termination problem. Once a process p_i has initiated a probe, if it becomes active, via a behind the back communication, it may not initiate another probe once its b_i is met again. If all processes are reactivated after initiating a probe, when the GTC is finally met, there will be no process which can detect termination. This algorithm is also incorrect because as Tel et al [TEL87] point out, it detects false termination. When a ring is used, to receive your own probe back is not a sufficient condition to ensure termination. The algorithm by Francez [FRAN81] from the previous chapter is an illustration that two complete cycles are needed to guarantee proof of termination.

The algorithm of Richier [RICH85] is an example of 2 careless errors. The actual detection of termination depends upon the knowledge that the variable 'known' has the

1 Jones, Geraint, *Programming in Occam*, Englewood Cliffs N.J.: Prentice Hall International, 1987.

value true. The author neglects to set this variable to true; however, the reader can assume where this statement should be put. In addition all probes verify that the count last seen is the same as the current count carried by the probe. However, processes that initiate a probe never set this value. The variable *kcount* should be set to 0 when variables are initialized. The author allows each process to initiate one detection wave only. This generates a maximum of n tokens. As a token is received by process p_i, the information it contains is stored until p_i's b_i becomes true. At that time, the accumulated values of all of the tokens received will be passed on as one token. It is possible to end up with only one token in the system.

This algorithm has considerable overhead. The algorithm requires memory for five booleans and one integer at each node. Whenever a bc occurs, three booleans must be set to false. In the best case[2] , there will be n^2 cc. In the worst case, $n-1+2nm$ cc could be sent out. This algorithm is unnecessarily complicated in this author's opinion.

3.3. Solution Using Tokens

Arora and Sharma [AROR83] introduce a distance function in addition to the token. This function gets the distance between two nodes from a pre-defined table which resides at each node and is transparent to the programmer. There are two additional variables:

T a local boolean which represents whether or not the token is at this process

FP a local variable that represents a process that has had a communication from this process and is the farthest away (initially set to nil)

2 After the first p_i becomes true, each of its successors meet their b_i just after the probe is received.

The token has the name *PN*. The token is given to some process, p_i, at random by the compiler and *PN* is initialized to p_i's predecessor.

The algorithm works as follows:

1. Whenever any process does a send as part of its bc, if its FP = nil then FP is set to the id of the recipient of the bc. If FP is not nil the distance function is invoked and FP is set equal to the process farther away - the recipient of the message or FP.

2. When a process is passive and holds the token, the token is passed on as follows:

```
if passive then
    if FP = nil then
        if PN = this process then
            initiate termination
        else
            pass the token to the next process in the CDG
    else
        begin
        if dist (FP) > dist (PN) then
            PN := FP;
        FP := nil;
        pass token to next process in the CDG;
        end
```

By changing the process name associated with the token if the inequality holds between FP and PN, behind the back communication is detected. The control code for this algorithm does not interfere with the bc; however, the distance function must be called each time a send is executed. This will add time to the computation. Although the authors do not compute the control overhead, this author believes the overhead for control messages in the best case[3] will be n or $2n-1$. The overhead in the worst case is $m * n$ where m is

3 Each node will be stable when the token reaches it; however, the token will have to make one complete loop before FP = nil for all of the processes. If the FP of each node is its predecessor, the token will have to make a second loop and will detect termination when it reaches the initiator's predecessor. If the FP of each node is its successor, one loop suffices and FP equals PN when it reaches the initiator and best case is then n.

the number of messages in the computation. This is about the same overhead in the best case as in the algorithms by Francez [FRAN81] and Apt [APT83] from the previous chapter and in the worst case it is better by n messages. The overhead for calling the distance function is not considered in these numbers.

3.4. Solutions Using a Logical Clock

Apt and Richier develop an algorithm using a ring topology and logical clocks. We have already seen the use of logical clocks as a tool by Francez [FRAN81]. Apt and Richier [APTR85] offer a much fuller explanation and several algorithms for a distributed environment.

Termination detection would be easy if a real global clock could be used. Apt and Richier describe the mechanism on a ring topology as follows [APTR85,475]:

> Whenever a machine turns passive it notes down the current time instant t and sends to its right-hand side neighbor a detection message with the time stamp t. The aim of this message is to verify whether other machines were also passive at the time instant t. If this is the case then termination is detected: at the time instant t all machines were passive.

This clock is read by each process only through message-passing. The end result is a bottleneck at the process which controls the clock. Essentially this is a centralized, rather than a fully distributed, solution.

A better solution would be to use local real time clocks. However, local clocks cannot be physically synchronized. Lamport [LAMP78] has given us a standard clock synchronization procedure which can be integrated into a termination detection algorithm as follows [APTR85,490]:

> We advance the value $clock\text{-}time_i$ of the local clock of p_i to $\max(clock\text{-}time_i, time)$ upon reception by p_i of a detection message with the time stamp, $time$.

With the introduction of local clocks, determining whether all processes are passive at the same time instant becomes less "transparent". However, the same effect may be achieved with a virtual clock as follows:

> In each process replace the local time clock by an integer variable T representing a virtual local clock. Initialize T to zero. Increment T by one, whenever the clock is consulted. Synchronization with other clocks is done as described previously.

The resulting solution makes use of the following variables:

ok True if the clock has been read and the process is passive

set to false when a bc occurs

ensures reading of the clock value occurs only once when the process becomes passive

sent true if a probe has been sent

set to false when a bc occurs

ensures that sending of a detection message takes place only once during each period when the process is passive

T the counter used as the virtual clock

count the number of processes which have received this detection message

time the timestamp of this message

fait set to true when $count = 2n$ to enable a process to exit the main loop and send a termination message.

In this solution the probe must go around the ring twice before termination is detected. The first cycle synchronizes the clocks and the second cycle checks that no process has become active since then. Initially *ok* and *sent* are false and T is 0.

The algorithm works as follows:

1. Whenever a bc occurs set *ok* and *sent* to false.

2. When b_i becomes true, set *ok* to true and increment T.

3. If *ok* is true and *sent* is false, then initiate a probe with $time = T$ and $count = 1$ and set *sent* to true.

4. If b_i and *ok* are true then a detection message may be received.

   If its $count = 2n$ then set *fait* to true and initiate the termination phase.

   If its $count \leq 2n$ then

   if b_i is not met then

   $T = \max(time, T)$

   purge the message

   else if b_i is true then

   if the timestamp, *time*, of the message $< T$ then

   purge the message

   else if $time \geq T$ then

   increment the *count*

   pass on the detection message

   set *sent* = true

Notice that the algorithm contains a statement which purges the detection message if b_i is not met. The value of b_i can change because b_i is dependent not only on variables of p_i but also some auxiliary variables.

This author computes the best case cc overhead[4] for this algorithm to be $2n + n - 1$,

4 All processes go passive in order around the ring in the direction of the cc. Each probe goes only to its successor before being purged because of time. The last process to go

and the worst case overhead[5] to be $m * n + 2n$. The bc has the overhead of setting the booleans *ok* and *sent* to false. And each process requires memory for three boolean and three integer variables.

Apt and Richier develop two other solutions: one allows detection messages to overtake and destroy each other. The other does not purge messages received by an active process. This author is not convinced that either is a step forward.

3.5. Summary

This section looked at algorithms whose code is symmetric. The algorithms used synchronous communication and ring topologies for the cc. Of the three tools for solving distributed problems suggested by Raynal [RAY88B], Arora and Sharma [AROR83] and Richier [RICH85] made use of tokens, Apt and Richier [APTR85] made use of logical clocks, and there are no published algorithms using sequence numbers for synchronous communication.

Table 3-1 illustrates the overhead for each solution. Arora's token solution has a smaller cc overhead, smaller memory requirements, and a bc overhead which is about the same as the logical clock solution. Richier's token solution allows n tokens to be generated, but attempts to limit the number of detection messages in the system at any given time. The end result is that Richier's solution has a very heavy overhead.

---

passive initiates a probe which detects termination.

5 Each process initiating a probe finds its predecessor active. The probe causes $n - 1$ cc. The actual overhead is $m * (n - 1) + 2n$.

Algorithm	Additions to bc	Best Case Overhead	Worst Case Overhead	Additional Memory
RING TOPOLOGY				
[AROR83]	call dist function	n	$m * n$	n booleans n integers
[APTR85]	set ok = F set sent = F	$3n - 1$	$m * n + 2n$	$3n$ booleans $3n$ integers
[RICH85]	set known = F set turn2 = F set color = F	n^2	$n + 2nm$	$5n$ booleans n integers

Table 3-1

Three algorithms are too few to allow one to draw solid conclusions; however, it would seem that it is possible for a token solution to require a smaller number of cc than a logical clock solution. When only one token is circulating through the system [AROR83], the algorithm is closer to a centralized solution - a process may initiate a probe only if it is holding the token. If a process, p_i, falsifies the probe, the token is held by p_i until p_i is ready to initiate a probe. The algorithm offered by Richier is more fully distributed because any process may initiate a probe at any time. One of the problems later researchers have had to face is that of reducing this overhead when tokens are not used.

4. ASYNCHRONOUS CENTRALIZED SOLUTIONS

In previous chapters, solutions using synchronous communication have been discussed. However, in the real world asynchronous communication is the norm. This chapter looks at the earliest solutions to the termination problem with asynchronous communication - centralized solutions. Rana [RANA83] suggests two approaches to the asynchronous solution for termination. We will look at solutions using each of these approaches and also at a theorem which attempts to describe the conditions for termination in the asynchronous case.

4.1. Assumptions

The assumptions that all channels are reliable continues to hold. We will continue to assume that termination will actually occur. Send is no longer a blocking operation. In addition either all bc are acknowledged or else message transmission is instantaneous. Instantaneous communication allows the researchers to avoid the problems of behind the back communication that would occur if message transmission could take an indefinite amount of time. A channel is considered empty if all messages have been received along that channel.

4.2. Issues

Again we ask the question: what conditions are sufficient to determine that a computation, P, is complete and therefore may terminate. In the synchronous case, the sending process blocked until the receiver was ready to receive. In the asynchronous case, we have no idea when a message will be received. Yet, we must still guarantee that behind the back communication will not occur. It will be sufficient if we guarantee that the following two

conditions are met:

1. For all p_i, b_i must be satisfied.

2. All channels between the p_i must be empty.

In order to guarantee that these two conditions have been met, we are saying that all processes must be passive at some time instant and that all of the channels must be empty of bc at that same time instant. In the asynchronous case, we know that if all of the b_i are true, behind the back communication can not happen when all the channels are clear, since an empty channel implies that all bc have been received. In the centralized case, again one process is responsible for declaring termination.

K. Mani Chandy [CHAN87] presents a theorem which is meant to illustrate the principles involved in the asynchronous distributed termination situation. Chandy describes a distributed system, presents two invariants, proves them, and finally states a theorem which follows from them. His description of the system agrees with everything we've seen so far. Chandy's theorem requires that each process record its state at most once. This distributed snapshot is the equivalent of sending out a single probe that visits each process and returns with the state of all of the processes in the system. Chandy assumes that distributed snapshots may be taken at any time during processing. If the application is not ready to terminate, a new snapshot may be taken at a later time and the results of previous snapshots are discarded.

Chandy describes a distributed system making use of the following variables:

p the name of a process, $p \in \{p_1, p_2, ..., p_n\}$

p.fini has this process recorded its state?

p.idle is this process idle?

p.IDLE the recorded value of *p.idle*

c, b the names of channels

c.s the number of bc messages sent along this channel

c.r the number of bc messages received on this channel

c.S the recorded value of *c.s*

c.R the recorded value of *c.r*

e an event in the system

d the set of processes that have recorded their states, i.e. the set of processes that have become passive at least one time.

The system works as follows:

Rule 1: An idle process with empty incoming channels remains idle.

Rule 2: An idle process does not send messages and any messages it receives were sent at an earlier time.

Rule 3: The number of messages sent along a channel is greater than or equal to the number of messages received from that same channel,[1] i.e. $c.s \geq c.r$.

Rule 4: If the number of messages sent is greater than or equal to the number of messages received, then no matter what event occurs next in the system, that relationship will continue to hold, i.e. for integers k and j

$\{(c.s = k)$ and $(c.r = j)$ and $(k \geq j)\}$ e $\{k \geq c.r \geq j\}$

1 Some messages may still be in transit at any given point in time.

Rule 5: The number of messages sent along a channel is monotonically nondecreasing, i.e. $\{c.s = k\}$ e $\{c.s \geq k\}$

Chandy requires that, at some arbitrary time, each process records its state at most once as follows:

if $\neg p.fini$ then
begin
$p.fini$:= true;
$p.IDLE := p.idle$;
[for all input channels c of $p :: c.R := c.r$]
[for all output channels c of $p :: c.S := c.s$]
end;

This results in a distributed snapshot, which is probably taken by means of sending out a probe. Chandy then claims the invariants of the system to be:

invariant [for all output channels, c, of processes in $d :: c.s \geq c.S$] and

[for all input channels, c, of processes in $d :: c.r \geq c.R$]

invariant [for all p in $d :: p.IDLE$] and [for all channels, c, between processes in $d ::$

$c.S = c.R$] and

[for all channels, b, from processes not in d to processes in $d :: b.s = b.R$]

==>

[for all p in $d :: p.idle = p.IDLE$] and

[for all output channels, c, of processes in $d :: c.s = c.S$] and

[for all input channels, c, of processes in $d :: c.r = c.R$]

Finally he states this theorem:

[for all $p :: p.fini$ and $p.IDLE$] and [for all $c :: c.S = c.R$]

==> [for all $p :: p.idle$] and [for all $c :: c.s = c.r$]

Notice that when termination has actually occurred, the recorded values satisfy the antecedent of the theorem. If termination has not occurred, the antecedent is not satisfied and a new snapshot must be taken, i.e., a new probe must be initiated, always being careful to discard the results of the previous probe. Chandy's theorem could be used to develop a detection algorithm and/or to prove that if termination is declared, termination has actually occurred.

4.3. Overview

Rana [RANA83,46] suggests two potential approaches to deal with the asynchronous communications case:

(i) Modify the global termination condition: ensure that all processes satisfy their local predicates at a particular time and that no message is pending to be delivered.

(ii) Modify the local predicates: modify all processes such that an ack is expected (sent) for each basic communication message sent (received) by a process. Let β_i denote the modified local predicate b_i; then β_i becomes true only when b_i is true and all expected acks have been received.

The algorithms of Dijkstra [DIJK83], Rozoy [ROZO86] and Arora [AROR89] take the first approach. Arora's algorithm [AROR88] takes the second approach. Chandrasekaran [CHAN90] uses a combination of both approaches. Arora [AROR88] superimposes his solution to the termination problem on a spanning tree model for the underlying computation. Chandrasekaran assumes a spanning tree for the algorithm and superimposes a diffusing computation upon it. Rozoy uses the diffusing computation model for the underlying computation and essentially follows messages around the system. Dijkstra uses a Hamiltonian ring for the network topology and Arora [AROR89] uses a bidirectional ring for the network topology.

4.4. Arora's Solution to the Problem

The algorithm presented by Arora et al [AROR88] is the result of two erroneous tries by the authors to solve the distributed termination problem in the fully distributed case. They have learned from their experiences and this author believes they have written a correct algorithm for the asynchronous centralized case.

When comparing Arora's model to Francez's [FRAN80], [FRAN82] or Topor's [TOPO84] models, the reader should remember that in the latter three cases, no additional channels were allowed for the cc. Arora's use of additional channels and the asynchronous communication provide additional possibilities for behind the back communication to occur.

4.4.1. Control Messages Used in the Algorithm

Arora requires an ack to be sent for every bc received. It is not clear from the paper how these acks influence the algorithm. According to Rana [RANA83], a correct algorithm should not allow a process to satisfy its b_i unless all outstanding acks have been received. However, Arora never makes this specification.

Arora allows four kinds of cc messages to be sent: 'I-am-passive', 'I-am-through', 'I-am-up-again', and the detection message initiated by the root. Each process knows who its children are in the control structure and who its neighbors (those to whom it may send a bc) are in the underlying network.

The 'I-am-passive' message is sent to p_i's neighbors whenever p_i meets its b_i. This is recorded in the variable $state_j(p_i)$ for each neighbor p_j receiving this message. The only time $state_j(p_i)$ is recorded as active is when p_j sends a message to p_i. Therefore it is possible for p_j to think p_i is passive when in reality p_i is active becuase p_k has sent a message to

p_i. This should not be a problem, as we shall see, because there are other protections. This incorrect knowledge may cause an 'I-am-through' message to be sent which could have been avoided, but it will never cause false termination to occur.

The 'I-am-through' message is sent when a process is passive, it has received the 'I-am-through' message from all its children (if it has any), and it thinks all of its neighbors are passive. Notice that if a process, p_i, communicates with its neighbor, p_j, and then goes passive, this message will not be sent until p_i receives the 'I-am-passive' message from p_j.

The 'I-am-up-again' message is sent to a parent whenever a process becomes active again or if the 'I-am-up-again' message is received from a child. This is protection against behind the back communication. The sender of the bc may not be in this portion of the tree, yet detection of a false termination will not be able to occur. As a matter of fact, the use of this message precludes the need for knowledge about neighbors at all. However, the use of neighbors prevents some detection messages from being initiated.

The final message to be considered is the detection message. This message is initiated by the root when

1) it is passive

2) all of its children have sent the 'I-am-through' message

3) and it thinks all of its neighbors are passive.

This message is passed down through the tree by each node p_i, only if the same conditions are met as for the root. When the message reaches the leaves, it is passed back up and reaches the root only if every node is indeed passive. This guarantees that a false termination will not be detected.

Using the criteria presented by Apt and Richier (ch 1) it can be shown that this algorithm is correct. In the description it has been shown that false termination cannot occur. In addition termination will be detected if it occurs. As each process becomes passive it will send an 'I-am-through' message. Eventually each process will know that its children and neighbors are passive and be able to pass the message on to its parent until the message is reached by the root. If a process is still active, it will eventually execute. The control code does not block the bc. In addition the cc will not cause deadlock. If the root initiates a detection message which becomes falsified, the active process will eventually cause an 'I-am-through' message to be sent up to the root and a new detection message will be initiated.

4.4.2. Overhead for the Algorithm

There is quite a bit of overhead for this algorithm. Whenever a bc is sent to p_j, the receiver's state must be marked as active in $state_i(p_j)$. When a bc is received, an 'I-am-up-message' must be sent if p_i was passive when the bc was received and an ack must be returned.

The memory overhead is also substantial. If all processes are directly connected to every other process, each process would require $3n$ booleans and $2n$ integers. The minimum, if each process has only 1 child and 2 neighbors, would be 5 integers and 5 booleans.

Whenever p_i goes passive, it sends 'I-am-passive' messages to its neighbors. If no process is reactivated after it goes passive and all processes have a neighbors then $a * n$ messages are sent. In addition each child sends an 'I-am-through' message to its parent for a total of $n - 1$ additional 'I-am-through' messages. If no process is reactivated, no 'I-am-up-

again' messages are sent. Finally $2*(n-1)$ detection messages are sent. Therefore, the total number of control messages is $m + an + 3n$ in the best case.

In the worst case[2] , there will be 'I-am-up-again' messages also. If all processes have a neighbors and d is the depth of the tree, then ma 'I-am-passive' messages, a maximum of md 'I-am-through' messages, m acknowledgements, a maximum of m 'I-am-up-again' messages, and a maximum of $2nm$ detection messages will be sent for a total of $m(a+d+2+2n)$ control messages.

4.5. Chandrasekaran's Dual Approach to the Problem

The main purpose of this paper [CHAN90] is to present a new lower bound for the overhead due to cc in the worst case situation. In addition the authors present an algorithm which meets this lower bound.

The existence of a root process and a spanning tree are assumed. All processes except the root run the same version of the algorithm. Topor's algorithm [TOPO84] is the starting point for the algorithm. In Topor's algorithm essentially the root initiated the detection wave and any of the internal processes could falsify it if they were active. However, responses were required to be sent back to the root when the wave was falsified so that a new wave could be initiated. Chandrasekaran and Venkatesan describe a method whereby the wave is never falsified and therefore only one wave need occur, reducing the number of cc substantially. Essentially a diffusing computation is superimposed on the application at the point where any process becomes active after receiving the wave. From that point on in the computation, each process, p_i, keeps track of all bc it sends/receives to/from process

2 In the worst case every bc reactivates a process, p_i, whose neighbors and children are passive - p_i has already sent an 'I-am-through' message.

p_j by placing either to/p_j or from/p_j on a stack in local memory. When p_j becomes passive, it sends acknowledgments to every process p_k on the stack as from/p_k. Any stack element to/p_k is removed from the stack when an ack is received from p_k.

A process p_i passes the wave back up to the root only when the following conditions have been met:

1. p_i is passive
2. p_i has received the detection wave on all of its incoming links.
3. p_i's local stack is empty.
4. p_i has received the terminate wave from each of its children - obviously each leaf, p_j, initiates this wave when conditions 1-3 are true for p_j.

Chandrasekaran refers to the sending of the initial detection wave as coloring the links. When the wave is received, the incoming link on which it was received is colored and outgoing links from that node are also colored. Acknowledgments must be sent only when messages are received from a colored link. Acknowledgments are expected only when messages are sent out on colored links.

The result is that the worst case overhead occurs in the situation where the root is ready to terminate but all other processes are active. Then a max of $2n$ cc and m acknowledgments would be required. In the best case all processes are passive before the root initiates the termination detection wave and the overhead is $2n$.

This worst case overhead is lower than most we've seen so far. However, there is a tradeoff - the cost of memory space can be quite high. For every bc sent after the algorithm is initiated, memory requirements will be 2 booleans (to/from) and 2 integers. There is also need for 2|c| booleans, where |c| is the number of communication channels in the

system, to indicate if the link is colored or not.

In addition, whenever a bc is sent/received each process must verify its state (DT or NDT - have I received the detection wave from my parent?). If the state is DT (detecting termination) the sender/receiver must be added to the local stack.

4.6. Rozoy's Algorithm

The purpose of Rozoy's paper [ROZO86] is to develop two theorems which describe the bounds on the number of cc required to detect termination in an asynchronous system. However, the sketch of a proof, which she provides, is based on a description of the distributed system as a finite state machine with an unbounded buffer for receiving messages, a transition function for changing states, and a function to send messages. All communication is considered instantaneous. Although a brief description of the model[3] is given, there are missing pieces of information.

In the process of developing the proofs for her theorems, Rozoy does offer a termination algorithm. The assumption that all messages are received in a short finite amount of time is critical for the correctness of the algorithm. The algorithm works as follows:

Rule 1: Local memory $mem(p_i)$ contains the names of all processes, p_j, to whom p_i has sent a bc.

Rule 2: When the initiator, p_i, goes passive a token is sent out containing a list of the names of all processes, p_j, to whom p_i has sent a bc (- the contents of $mem(p_i)$.)

---

3 A complete description is given in
Rozoy, B., "Detection de la terminaison dans les reseaux distribues." Tech. Rept. 1986, Universite de Paris, VII LITP.
which is written in french.

Rule 3: Upon receipt of the token by p_j:

a. p_j adds the contents of $mem(p_j)$ to the token list.

b. p_j deletes its name from the token list.

c. p_j clears $mem(p_j)$.

d. When b_j becomes true, the token is passed to any process, p_k, in the token list.

Rozoy claims that the following invariant remains true throughout the computation as her proof of the algorithm.

P0: A $\vee$ B $\vee$ C

where

A is true if for all p_i, if p_i's name is not in the token list then p_i has remained idle since the last visit.[4]

B is true if for all p_i, if p_i's name is in the token list, then either p_i has been active or will be active since the last visit by the token.

C is true if the token contains more than one process name or a single name - of a process which is still active.

---

4 The English translation is difficult to understand here. The definition given above for A is what I think she is trying to say. However, I don't think it is what she means. The terminology she uses is "p has not to be visited by the token". I assume she means p's name is not in the token list, since the token only visits those processes on its list. However, consider the following scenario:

> The initiator puts the names of five processes in the token list. The token visits the first process on the list and adds its local memory to the token list. In the meantime any process activated by the remaining four processes in the token list are not on the list.

Eventually all of these process names will be put in the token list. However, there is a point in time when they are not on the list although Rozoy implies that the token list will always be correct.

Notice that the token follows no specified path, nor does it verify its findings for termination before claiming termination. For this algorithm to work correctly, the send of a bc **must** be immediate - there can be no lapse of time until it arrives at its destination. If there is a lapse of time, it is possible for false termination to be claimed. It should also be noted that the size of the probe varies for each cc and has a maximum size of n.

4.6.1. Boundaries on the Number of Control Messages

Rozoy suggests that a lower bound for the number of cc can never be less than m - all bc must be acknowledged or followed.[5] In a fully connected network where additional channels have been added to the network for the control algorithm, the upper bound is dependent on the size of the system and requires $m+n$ cc messages. When the termination algorithm is well suited to the network - no additional channels are added - then the upper bound is dependent on the size of the problem and requires $2(m+1)$ cc messages.

4.7. Dijkstra's Solution to the Problem

Dijkstra et al [DIJK83] introduced an algorithm for the detection of termination of distributed processes in a ring topology where more than one process can reside on each machine. The signalling facilities are assumed available and specifics are not mentioned, except that all communication is instantaneous. The algorithm is general enough to handle either asynchronous or synchronous communication. This algorithm checks each machine for whether or not its set of processes have terminated by passing a token around the ring. The token is either white (no bc has been seen) or it is black (some process has performed a bc). In addition to the local predicate, b_i, for each process, all machines have the variable

5 If the detection tokens are distant from the traffic, they cannot guess whether the traffic is finished or very slow. Therefore they have to follow the message.

colour which is white (no process on this machine is active i.e. has performed a bc) or black (some process is active). Then for the following propositions [DIJK83,218], where t is the unique id of the machine which has the token and N is the number of machines in the network:

P0: for all $i : t < i < N$: machine $Nr.i$ is passive.

P1: there exists $j : 0 \leq j \leq t$: machine $Nr.j$ is black.

P2: the token is black.

The invariant P0 $\vee$ P1 $\vee$ P2 must be true at all times.

This is accomplished through the following rules [DIJK83,218-9]:

Rule 0: When machine $Nr.i+1$ is active it keeps the token; when it is passive it hands over the token to machine $Nr.i$.

Rule 1: A machine sending a message makes itself black.

Rule 2: When a machine $Nr.i+1$ propagates the probe it hands over a black token to machine $Nr.i$ if it is black itself, whereas if it is white it leaves the colour of the token unchanged.

Rule 3: After the completion of an unsuccessful probe, machine $Nr.0$ initiates a next probe.

Rule 4: machine $Nr.0$ initiates a probe by making itself white and sending a white token to machine $Nr.N-1$.

Rule 5: Upon transmission of the token to machine $Nr.i$, machine $Nr.i+1$ becomes white.

The number of control messages required is $2N$ in the best case because the probe will

never succeed the first time around. In the worst case it is $N * m + N$, where m is the number of bc sent and N is the number of machines in the system.

4.8. Arora's Solution Using a Bidirectional Ring

Arora and Gupta [AROR89] present several algorithms that assume a bidirectional ring as the control topology for the cc. The first of these algorithms assumes a predesignated process to initiate the probes for the detection algorithm.

The algorithm uses the following variables at each node p_i:

$bcm(p_i)$	a boolean flag which is set whenever p_i sends/receives a bc and reset whenever p_i receives the probe, initially 0
$procflg(p_i)$	a boolean flag which is set whenever p_i changes state from active to passive and reset whenever p_i receives the probe, initially 0
$seq(p_i)$	for $i = 1$, the sequence number of the current set of probes for all other i, the sequence number for the currently forwarded probe message
prevpm	a boolean flag which is reset to 0 when p_1 initiates the first set of probe messages, it remains 0 afterwards a boolean flag which records the status of the forwarded probe message for the remaining p_i
distance1	the distance clockwise around the ring from this process to the initiator

distance2 the distance anticlockwise around the ring from this process to the initiator

pas a boolean flag for p_1 which is initially false and is set to true when b_1 becomes true the first time and remains true

There are two kinds of cc messages: the probe message which gathers information concerning the states of the processes, and the repeat-probe message which is a signal that a probe has been falsified and a new probe should be initiated.

The algorithm works as follows:

1. The initiator sends probes out in both directions.
2. Upon receipt of a probe, each process waits until its b_i has been met before verifying the probe and passing it on. If the probe is already falsified the flags of the process are reset and the probe is passed on. Otherwise the flags are checked - if either one is set then the probe is falsified else it remains unfalsified. In either case the flags are reset and the probe is passed on.
3. Whenever a process receives a probe that carries a sequence number that has already been seen, the flags of the process are checked and the termination phase is initiated if both probes were unfalsified and a repeat-probe signal is sent back to the initiator if either probe is falsified. (The initiator acts on only one of the repeat-probe signals.)

In previous algorithms Arora and Gupta did not verify that no bc have been received and the process has not been active since the last probe. They seem to have solved that problem here. Processes may be reactivated after the probe passes them; however, the initiating process of any string of bc will always be caught. Eventually a probe will travel around the ring from both directions and find all processes passive and initiate the

termination phase correctly.

The overhead for bc for this algorithm is comparable to others we've looked at. However, the memory requirements are quite high: 3 booleans at each node, 4 at the initiator and 3 integers at each node, 1 at the initiator. The best case overhead for the cc is less than $3n$ - the $2n$ cc required to set and check flags normally found in ring algorithms[6] plus the repeat-probe signal which is at most $.5n$. The worst case overhead is $mn + 2n$ plus $.5mn$ for the repeat-probe signal which results in a worst case cc of less than $2mn + 2n$. This is slightly higher than normal for ring algorithms because both send and receive set flags. The reader should notice that although the cc is worse than for most ring algorithms, it is possible that it will take less time for the algorithm to actually run, because there are 2 probes circulating at the same time.

4.9. Summary

This section looked at algorithms which assumed asynchronous communication. Table 4-1 illustrates the overhead for each solution. Notice that all of the algorithms have additional work to be done before a bc can be sent. As in previous chapters the spanning tree model has a much higher overhead than the the ring topology - both in terms of additional memory and the number of cc. The spanning tree model is the least efficient method of implementing termination algorithms; however, it does help to make the issues clear. Notice that all algorithms that use trees as the network topology for the cc keep a list of processes with whom they've had communication. Dijkstra's algorithm [DIJK83] is the most efficient of all of the algorithms and also one of the easiest to understand. It requires a very simple addition to the bc, and very little additional memory. The cc

6 Add one to this value to guarantee that both probes are seen twice by one process.

Algorithm	Additions to bc	Best Case Overhead	Worst Case Overhead	Additional Memory
DIFFUSING COMPUTATION MODEL				
[ROZO86]	add receiver to local mem	m	$m+n$ [1] $2(m+1)$ [2]	$<n^2$ integers
SPANNING TREE TOPOLOGY				
[AROR88]	mark receiver active return ack send cc msg as necessary	$<n^2+3n+m$ [3]	$<m(3n+d+2)$ [3]	$<3n^2$ booleans $<2n^2$ integers
[CHAN90]	if state = DT save recvr's name save sndr's name	$2n$	$<2n+m$	$<2m$ chars $<2m+2n^2$ booleans
RING TOPOLOGY				
[DIJK83]	make machine black	$2N$ [4]	$N*m+N$	n booleans
BIDIRECTIONAL RING				
[AROR89]	set flag at send/recv	$<3n$	$<2mn+2n$	$3n+1$ booleans $3n-2$ integers

1 A fully connected system is assumed.

2 No new channels are added.

3 This assumes each process has a maximum of n neighbors.

4 N represents the number of machines in the system. If there is only one process on each machine this is the same as 2n.

Table 4-1

overhead in the best case is the best for all of the algorithms presented in this chapter, although the worst case is not as good as the algorithm which assumes a diffusing computation. As in previous chapters, it would seem that the ring topology allows algorithms to require a lower overhead. Arora's bidirectional ring is comparable to Dijkstra's algorithm in the best case cc overhead and double in the worst case. However, the actual detection of

termination may take the same amount of time since it must be remembered that Arora sends cc in opposite directions at the same time. However, Arora's algorithm requires substantially more memory and may not be a good choice if minimal use of memory is important.

It is interesting to notice that Dijkstra's centralized algorithm requires the same overhead as the centralized algorithms for the ring topologies in the synchronous case. However, Arora's algorithm [AROR88] has a much higher overhead than the algorithms based on a tree topology presented in chapter two. This author does not believe that it is necessary to have this high an overhead; however, Arora was looking for an algorithm that would always be correct, after his two unsuccessful attempts. In an attempt to limit cc, probes are initiated only after sufficient supporting evidence for the GTC is obtained; however, the end result is a substantially larger number of cc. Notice that while Chandrasekaran's algorithm has a very low cc overhead, the cost of memory and additions to the bc offset the gain when m is high. They are in normal range when m is low.

5. ASYNCHRONOUS DISTRIBUTED SOLUTIONS

Any valid asynchronous distributed solution requires that any process be able to detect termination and that all behind the back communication will be caught despite the fact that message communication is not synchronous. This is complicated by the fact that no one process is given the job of declaring termination - a fully distributed solution requires that any process be able to initiate a probe and/or detect that the GTC has been met.

In chapters two thru four, we have only looked at tree topologies and ring topologies for the underlying network. In this chapter we will broaden our scope to include algorithms for broadcast networks and arbitrary network topologies.

5.1. Assumptions

As in previous chapters, these algorithms continue to assume that the application P will terminate and that the channels for communication are reliable. Communication does not block the sending process and all messages are received in a finite amount of time. Most algorithms assume that message transmission is instantaneous.

5.2. Approaches

We will look at three approaches to the solution of this problem in the literature. There are several solutions that assume a ring topology for the underlying network. Arora, Rana, and Gupta present three solutions [AROR86], [AROR87], [AROR89] and Haldar [HALD88] also offers a solution based on a ring topology. Only one of these solutions is correct. The literature supports the error in Arora's first solution [TAN86] - the detection of false termination. This author believes that the second solution does not solve the prob-

lem encountered in the first algorithm. Haldar's solution is based on this second solution and contains the same error. Tel and Mattern [TELM89] give a counterexample to prove Haldar's solution is incorrect. Arora's third solution is actually two algorithms. Although the first is incorrect, it is easily fixed. The other is similar to his first two solutions and seems to be correct. It is important to look at these algorithms and understand the situation which leads to the detection of false termination. They illustrate very clearly the kinds of problems which a correct distributed asynchronous solution must take into consideration.

There are several solutions which assume an arbitrary network topology. Misra's [MISR83] solution, using colored tokens, is the asynchronous distributed version of Dijkstra's [DIJK83] synchronous centralized solution. Skyum [SKYU86] presents an algorithm which assumes knowledge of the diameter of the network. Huang [HUAN88] offers a solution that requires the use of logical clocks. Eriksen's algorithm [ERIK88] requires only that all processes know the size of the network. The network is represented as an undirected graph, although the communication lines are directed.

A third approach toward solving the termination problem differs from the accepted definition of distributed termination. Szymanski [SZYM85] offers an algorithm whose purpose is to detect termination in applications which require some degree of synchroneity even though asynchronous communication is used. The algorithm will be presented; however, this author does not believe it will lead to a working solution to the generic distributed termination problem. Szymanski's solution assumes that all messages are received within an arbitrary finite time.

5.3. Solutions for a Hamiltonian Cycle Topology

In this section we will look at several solutions offered by Arora and Gupta. The solution offered in 1986 [AROR86] is incorrect and countered by Tan et al [TAN86]. Arora et al present another algorithm in 1987 [AROR87] that takes into consideration some of the criticisms offered by Tan. In 1988, Arora [AROG88] counters Tan's criticisms of the 1986 algorithm. All discussion of the algorithm presented in 1986 will be considered before discussing the later algorithms, even though this is not the chronological order of events. In 1989, Arora et al [AROR89] present three similar algorithms - one that has been discussed in the previous chapter, and two for the distributed case.

5.3.1. Arora's Solution to the Problem

The algorithm introduced by Arora et al [AROR86] was written to improve on the previous algorithms by [RANA83] and [AROR83] by eliminating the need for distance functions, logical clocks, or counters. However, Tan, Tel and Van Leeuwen [TAN86] found that it does not work correctly and is not easily fixed. The restriction for this algorithm is that bc messages may be sent only to neighbors of a process. A Hamiltonian ring topology is assumed for the control communication. A local predicate, b_i, is maintained as before. In addition records of each neighbor's state are kept (passive/active). A neighbor is marked active by p_i whenever p_i sends a message to it. A process is marked passive when an 'I-am-passive' message is received.[1] A probe may be initiated by process p_i whenever p_i and all of its neighbors are passive.[2] Each probe message contains the process id of its initiator. The probe is passed on only when the process receiving it (and all of its neighbors)

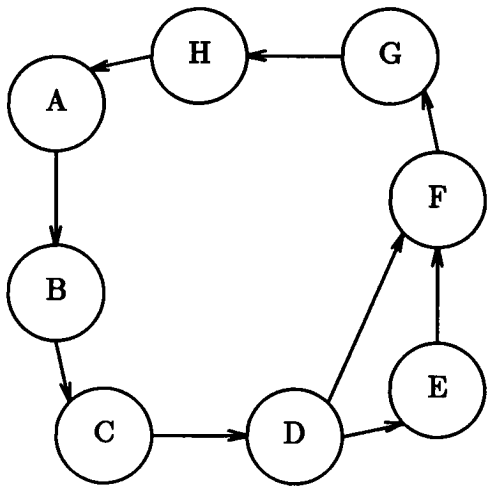
1 A process sends this message each time it changes state from active to passive.
2 Notice p_i does not know that p_j is active if p_j was activated by any process other than p_i.

are passive. When a process receives its own probe back, it may begin the termination phase. Because behind the back communication can occur and is not caught, this algorithm allows false detection of termination to occur.

Arora and Gupta [AROG88] counter the assertion of Tan et al [TAN86] that this algorithm is incorrect. Arora claims that there is no delay in handling the cc. However, he verifies that in certain situations a delay may result in the declaration of a false termination as per the example of Tan et al. In an attempt to remedy this problem Arora and Gupta augment the original algorithm with additional cc to update the states of the neighbors of a process. This requires an additional boolean variable at each node, the construction of a list of the id of each of p_i's neighbors every time an 'I-am-passive' message is sent, and additional cc. As a result, when a process becomes passive it does not initiate a probe, but rather the last process in the list of the "I-am-passive" message initiates the probe when it becomes passive.

This still does not solve the problem because processes do not send "I-am-active" messages which allows behind the back communication to occur and not be caught! Let's assume all processes passive A - G, except C in Tan's setup of a Hamiltonian ring with the additional link DF.[3] Let's look at the probe initiated by E. Just after the probe leaves E, C sends a bc to D and then goes passive (sending its "I-am-passive" message). D sends a bc to F, and F sends a bc to G after the probe has passed G. D and F go passive (sending "I-am-passive" messages) leaving G active. A thinks G is passive and passes the probe on. The probe reaches E and false termination is declared because G is still active.

3 See the figure on the next page.



5.3.1.1. Solution Two

The algorithm written in 1987 by Arora et al [AROR87] attempts to take into consideration the problems pointed out by Tan et al [TAN86] which were not solved in Arora's earlier algorithm [AROR86]. However, in this author's opinion, a false termination can still be declared.

Arora's algorithm assumes a Hamiltonian ring in which control messages travel in one direction only. Local variables are as follows:

$SeqNum_i$ initially 0, it gets incremented each time a message is sent out

$IdList_i$ a list of all processes p_j to whom p_i has sent messages for the computation

When a process becomes passive it may initiate a detection message which consists of its id, its current sequence number, $SeqNum_i$, and 2 bit flags, *F1* and *F2* (initially set to 0). This detection message is never purged; it always goes completely around the ring.

Upon receipt of the probe, each process p_j deletes the id of the initiator of the message from $IdList_j$. If this list is not nil (empty) now or if p_j is active, p_j "falsifies" the probe (if it has not already been done) by setting *F1* equal to 1. If p_j is active, it also sets

F2 to 1. The probe is then passed on.

When the initiator, p_i, receives the message back, p_i purges the probe if $F2 = 1$, or if $F1 = 1$ and $SeqNbr_i$ is not equal to the sequence number contained in the probe or if p_i is active. If $F1 = 1$ and the sequence numbers match, a new probe is sent and if $F1 = 0$ and sequence numbers match p_i initiates the termination phase. Once termination has occurred, it may be detected simultaneously by more than one process.

Now consider the following scenario: three processes - A, C, and E - are connected in a Hamiltonian ring. The cc travel in a clockwise direction from A to E. Process A sends bc to process E. Processes E and C do not send any bc. A goes passive and immediately afterward E goes passive. In the next step, each probe moves forward one process and finds the following:

Process E's probe finds process A passive, removes its name from $IdList_A$, leaving an empty $IdList_A$.[3]

When process A's probe reaches process C, C has just sent a bc to process A and gone passive. A will remove its name from $IdList_C$ and because C is passive and $IdList_C$ is now empty, A's probe will continue on unfalsified.

Process C's probe will remain unfalsified when it reaches process E because E is passive and $IdList_E$ is empty.

In the next step, we observe the following:

Process E's probe finds process C passive, and $IdList_C$ empty; therefore, it continues on unfalsified.

Process A's probe will find process E passive and $IdList_E$ empty and it will also

3 The bc that C sends at this step of the computation - to be described in the next sentence of the main body of the paper - has not been received by A yet.

continue on unfalsified.

Process C's probe will find process A active and be purged by A.

In the final step of the detection phase, we observe the following:

When A's probe returns, it finds A active and the probe is falsified.

C's probe has already been purged.

When E's probe returns to E, it will declare termination since it has never been falsified; however, process A is still active and it is a false termination.

The problem becomes evident when we consider Francez's concept of interval assertions [FRAN81]. Termination may only be declared when all processes have been continuously idle over some time interval. Arora et al [AROR87] do not even consider the time intervals during which the b_i are met! In addition, when p_i receives its probe back, there is no verification that $IdList_i$ is empty. It would be possible for p_i to be reactivated, become passive and send out a second probe during the circuit of the first probe.

Even if this algorithm were correct, its control overhead is substantial. In the best case, it is n^2 since all processes must initiate a detection message on becoming passive if for no other reason than to eliminate their id from everyone else's control block. In the worst case, the control overhead is $m * n + n^2$. This algorithm is unwieldy - much of the code is repeated needlessly because of its placement and could be written more concisely. In addition it contains too many parameters to the probe. Haldar's algorithm [HALD88] is based on this algorithm and contains the same problem.

5.3.1.2. Solution Three

Arora and Gupta [AROR89] present two algorithms for the distributed case. They assume a bidirectional Hamiltonian ring for the control topology and are based on an

algorithm for the centralized case found in the same article and discussed previously in section 4.8 of this paper. The variables and code for the first distributed version are similar to those of the previously presented algorithm.[5] The differences are as follows:

1. The code when p_i becomes passive in this version is the same as for the centralized version.

2. The code for receivng a repeat-probe-signal is not needed.

3. The code for receiving a dectection message is similar to the previous code except that a new probe (instead of a repeat-probe-signal) is issued by the process half way around the ring.

This first version is called a "shifting process control algorithm." This author finds the "shifting process control algorithm" to be incorrect. Arora is attempting to guarantee that the probe messages will not double each time a message is falsified. (Remember a repeat-probe-signal is not sent out, two of these were not a problem - the initiator simply ignored the second one.) However, in the process he has made it possible for the probe message to disappear and termination never to be detected. Suppose that four processes are connected in a ring such that p_1 is the predecessor of p_2, p_2 is the predecessor of p_3, etc. Process p_3 is the last active process. Process p_1 initiates the algorithm and passes the detection message to p_2 and p_4. Process p_2 passes an unfalsified detection message on to p_3 and p_4 also passes an unfalsified detection message to p_3. If p_3 receives the detection message from p_4 first, p_3 falsifies it and passes it to p_2 who purges it and does not reissue a detection message because p_3 is not the predecessor of p_2. When p_3 receives the

5 The discussion of the centralized version is found in section 4.8. The pseudocode for the current version can be found in the appendix pp. 185-186.

detection message from p_2 it falsifies it and purges it; since a probe is not reissued, termination will never be detected.

This author believes the problem is easily fixed. Once the sequence number $pseq(p_i)$, matches the sequence number carried by the detection message, the detection message is purged if it is unfalsified and p_i is active. At this point the code for initiating a new probe should be inserted if the sender of the probe was the predecessor of p_i. The original code can be found in the appendix, the altered code is as follows:

```
if pseq(p_i) = msg.seqdm then
    if msg.F1 = 0 then
        if prevpm(p_i) = 0 and (bcm(p_i) = 0 and procflg(p_i) = 0) then
            enter the termination phase
        else
            purge the detection message
            if msg.source = pred(p_i) then          {the additional code begins here}
                seq(p_i) = seq(p_i) + 1
                msg.seqdm := seq(p_i)
                pseq(p_i) := msg.seqdm
                msg.type := probe
                msg.F1 := 0
                send (succ(p_i), msg, p_i)
                send (pred(p_i), msg, p_i)
                bcm(p_i) := 0
                procflg(p_i) := 0
                prevpm(p_i) := 0                    {the additional code ends here}
    else
        purge the detection message
        if msg.source = pred(p_i) then
            seq(p_i) = seq(p_i) + 1
            msg.seqdm := seq(p_i)
            pseq(p_i) := msg.seqdm
            msg.type := probe
            msg.F1 := 0
            send (succ(p_i), msg, p_i)
            send (pred(p_i), msg, p_i)
            bcm(p_i) := 0
            procflg(p_i) := 0
            prevpm(p_i) := 0
```

This guarantees that only one probe will be initiated and that in fact it will be initiated.

The memory required for this algorithm is 4 booleans and 2 integers at each node. Whenever a bc is sent/received a boolean flag is set. The cc overhead in the best case is $2n+2$. In the worst case[6] it is $mn+m+2n+2$.

The second distributed version of the algorithm is called a "multiple process algorithm" because any process may initiate a probe at any time. The concept of updating neighbor processes is used as in previous algorithms [AROR86] [AROR87]. Each process keeps a copy of its own state, the state of its neighbors, the sequence number and the validity of the last probe seen from every process, and its own sequence number. When a probe is received by p_i from p_j, the algorithm works as follows:

```
if p_i and all of its neighbors are passive then
    if the last sequence number seen from this process matches the sequence number
    on the probe then
        if the probe was unfalsified the last time then
            enter the termination phase
        else
            purge the probe
    else
        remember this sequence number
        set the status of the probe to be remembered as unfalsified
        pass the probe to the next process
else
    remember this sequence number
    set status to be remembered as falsified
    purge the probe
```

In this algorithm Arora has finally verified that all processes have been passive since the last visit of the probe. However, the cost is considerable in terms of memory overhead. Each process requires n booleans and n integers to remember the status of previous probes

6 The first probe will always be falsified because all of the flags will be set when processes become passive the first time. To complete a cycle around the ring takes $n+1$ cc. Then for every message that falsifies the probe $mn+m$ cc will be passed. Finally an unfalsified probe will be verified with $n+1$ cc. Therefore, the total number of cc is $m(n+1)+2(n+1)$.

from each process. They also require up to n integers to keep track of who their neighbors are and up to n booleans to keep track of their states. In addition an integer is required to keep track of the sequence number for probes initiated by this process. This gives a possible total of $2n$ booleans and $2n+1$ integers at each node. The addition to the bc is to set a flag at both send and receive. Every time a process goes passive, as many as n cc are required to update its state for its neighbors. A count of the cc is difficult to determine because it depends on the number of neighbors each process has and the timing of when they go passive. However, if all processes have only 1 neighbor and go passive at the same time for the first time there will be $n^2 + n$ cc. If after every bc the sending process goes passive there will be mn update messages and there could be as many as $2mn^2$ probe messages.

5.3.2. Haldar's Solution to the Problem

Haldar's algorithm takes the concept of a distance function from Arora's earlier algorithm [AROR83] and the list of processes with which it has communicated from the later algorithm [AROR87]. Probes, *msg*, contain the id of the initiator, the *FP* of the initiator (as in [AROR83]) and a 1 bit key to falsify the message (the variable *F1* of [AROR87]). A process may initiate a probe if it is passive and the probe has a higher priority than any other message waiting for service. This guarantees that an active process will not hold it up.

Haldar's algorithm uses the following variables:

FP	The id of the farthest process down the ring with which p_i has communicated when p_i was active

IdList$_i$ This is a list of processes with whom this process initiates communication

Haldar gives the following rules for the actions to be taken upon receipt of a detection message:

Rule 1: As soon as the detection message gets falsified, the bit flag *F1* is set to 1 and remains 1 until the message is purged.

Rule 2: The unique identification carried with the detection message is removed from the *IdList* of the visited process if present.

Rule 3: If *F1* becomes 1 before the control message generated by p_i reaches *msg.FP* the message is purged by *msg.FP*.

Rule 4: As soon as the control message generated by a process p_i crosses *msg.FP*, the message is purged if it is received by an active process or by a process having a nonempty *IdList*.

When these rules govern the action of the algorithm, if a process ever receives its own probe back the authors assume the process may begin termination. However, if you consider the scenario that we looked at to prove Arora's algorithm [AROR87] incorrect, it is obvious that the same situation is present here. Again all processes passive at the same time has not been considered. The comments of Tel and Mattern [TELM89] present another counterexample to illustrate this point.

This author believes Haldar has an additional problem - it is possible for the control algorithm to cause deadlock! Notice that a process puts the name of the recipient of all bc in its *IdList* whenever bc are sent. Notice also that a probe is initiated only once each time a process becomes passive. Finally notice that a probe is purged as soon as it is

falsified and has passed *msg.FP*. However, one of the processes between *msg.FP* and the initiator of the probe, p_i, may have sent p_i a bc while p_i was still active. Even if the probe is not purged, p_i's name is not removed from any *IdList* beyond *msg.FP*. If p_i's probe is falsified, p_i will not ever send out another probe and no other process will be able to declare termination because some *IdList* will contain p_i's name.

In addition no check is made to verify that p_i is passive and $IdList_i$ is empty if its probe is returned unfalsified - this is potentially dangerous.

The authors believe they have solved the distributed termination problem without the use of timestamps and clock synchronization as in Rana's algorithm [RANA83] and without the use of sequence numbers as in Arora's algorithm [AROR87]. If this algorithm were correct, in the best case, it would require only $n + n - 1$ ($2n$ as an upper bound) cc. In the worst case, the authors contend both algorithms use about the same number of control messages - $m * n + n^2$. This algorithm does not feel cluttered by additional information; however, it does require the use of the farthest function which adds time on to the basic communications of the computation. Although it may be an improvement over Arora's algorithm, Haldar's algorithm does not solve the distributed termination problem correctly.

5.4. Solutions for an Arbitrary Network Topology

5.4.1. Misra's Solution to the Problem

Misra's algorithm [MISR83] makes use of a marker to determine if a process has been continuously idle over an interval of time. Processes are colored black initially and whenever they receive a bc. Processes become white when the marker is passed on. The marker

is passed on only if a process is idle.

The idea behind the algorithm is similar to Dijkstra's [DIJK83] and Topor's [TOPO84] basic premise. In each of these algorithms the marker was returned to the initiating process which decided whether or not to terminate the computation. Therefore, the easiest approach was to color the marker as well as the processes. However, in Misra's algorithm, any process is capable of detecting the termination. Therefore, rather than carrying a color, the marker carries a count of the number of processes visited which were white - continuously idle since the last visit of the marker - at the time of the arrival of the marker. Any process which is black sets the marker to 0, white processes increment the marker. Any process which increments the marker to n - the number of processes in the system - may declare termination.

Using Apt's criteria, [APT86] we can show that this algorithm is correct.

1. If the computation terminates, the marker will find each process passive and be passed on immediately. It will eventually find n white processes in a row and and termination will be declared.

2. If termination is declared, it will actually have occurred. If the marker counts n processes as continually idle, then it found all processes passive and because the order of all messages is FIFO, all messages will have been flushed from the network and it will be impossible for any process to be reactivated.

3. The algorithm does not cause deadlock of the computation, because it does not interfere with it in any way.

4. The superimposed detection scheme does not delay the computation proper indefinitely. There are only two additional control statements - an assignment when

a bc is received and the passing on of the marker which only occurs when a process is idle.

The algorithm is general enough to be used in an arbitrary network. Looking at a Hamiltonian ring for the cc, we can see that the algorithm carries very little overhead. The additional memory is a single boolean variable, *color*, which holds the color of the process. Upon receipt of a bc, assignment of the value black is made to *color*. The marker consists only of an integer. The cc overhead is minimal in the best case[6] - $2n$. This represents the initial round of visits where all processes have become passive and the second round of visits to check that they have been continuously idle. In the worst case[7], the cc overhead is $mn + n$ if all messages are sent to p_i's predecessor.

This algorithm is very simple to understand and requires a minimal amount of overhead. Notice that the only additional information required is the number of processes involved in the application.

Misra's article is of special interest because he also presents a method of implementing his algorithm in an arbitrary network in which no global information is available to any process. However, the marker must be able to traverse all edges in both directions.

Again the marker is placed at any process initially. Misra sketches out a depth first search although any method of search may be used. Whenever a black process is found, a new depth first search is begun. Each new search is called a *round*. Each process maintains the following variables:

6 No process is reactivated by any bc.
7 Every bc reactivates a process.

color a value of black indicates that a bc has been performed since the last visit by the marker

a value of white indicates that no bc have been received since the last visit by the marker

roundnbr this is the round number attached to the last token seen, probably initialized to 0

p(i) the index of p_i's parent

Γ_i the indices of all 'children' of p_i - the processes to whom p_i has passed on this round of the marker

When the marker encounters a black process, p_i, it colors the process white, increments *msg.roundnbr*, sets *roundnbr* equal to *msg.roundnbr*, and whenever p_i becomes passive, departs to any process connected to p_i. Although not specifically mentioned in the pseudo code, it is obvious that Misra intends for each process to keep track of from whom it receives the token and to whom it passes the token. Whenever a token with *msg.roundnbr* not equal to *roundnbr* appears, the list of children should be set to nil again.

This algorithm is best described by looking at the actual code. When a process, p_i, is white when it receives the marker the following code is executed:

```
if msg.roundnbr > roundnbr then
    p(i) := sender of token
    roundnbr := msg.roundnbr;
    propagate the marker
else if msg.roundnbr = roundnbr then
    if sender ∈ Γ_i then
        propagate the marker
    else
        return the marker to the last sender
```

The propagation of the marker is done as follows:

```
Choose an 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
     if there is no father then
          initiate termination
     else
          send the marker to the father
```

The overhead for this version of the algorithm is higher. The addition to the bc is the same for this network topology; however, each node now requires four variables. In addition the count of the cc overhead is substantially higher. Because there is not a Hamiltonian cycle, the marker must visit all possible edges. Let's assume there are c edges, where $c \geq n$. Then the best case overhead will be $3c$, because the marker will first encounter all of the processes as black and start as many as c rounds. The next round will detect termination but will require $2c$ cc. In the worst case, all processes except for one will go passive about the same time and all messages will reactivate some process. If the reactivated process is the last process to be visited in the round the overhead will be $m * 2c + 3c$ cc. This is even higher than the overhead for the tree topologies which we looked at because **all** edges must be traversed to guarantee no behind the back communication. If time is important and the complexity of the problem is not high, this is a high price to pay versus setting up a tree before the application begins and assigning one process the job of the root. However, if the complexity of the application is high, this would not be a problem.

5.4.2. Skyum's Solution to the Problem

Skyum's solution [SKYU86] requires that messages be transmitted instantaneously. There are two additional statements required: *disable* does not allow bc to cause an inter-

rupt, and *enable* allows interrupts to occur again. All of the code between the disable and enable statements is indivisible. It appears that the control cycle does not begin for each p_i until p_i has gone passive for the first time. The control cycle is then interrupted whenever a bc reactivates p_i and when p_i becomes passive again the control cycle continues from where it left off. The protocol has the following form:

broadcast a message

cycle

await that no inbuffers are empty;

read one message from each inbuffer;

broadcast a message

endcycle

This splitting up of the computation into parts separated by a broadcast to all outbuffers followed by a reading from each inbuffer introduces a local time concept. This time concept is continuous enough for the p_i to 'know' that if they have not heard from any active process[8] for some 'time', then there are no more active processes in the configuration.

The overhead for this algorithm is quite high. The memory requirements for each process are 1 boolean, 2 integers, and 1 constant. The addition to the bc is comparable to other algorithms - set active to true. However, the count of cc is substantial. If we assume that the network is a Hamiltonian ring, each process will have only one inbuffer and one outbuffer. The choice of the constant, L, will greatly effect the number of cc. By definition L must be greater than $2d$; however in a hamiltonian ring the diameter is equal to $n-1$. Therefore L will be equal to $2n$. Because each process must accumulate L

8 An active process is indicated by a warning carrying an age of 1.

warning messages, the cc overhead is as follows. In the best case $n+2n^2$ cc will be required and in the worst case $n+m*2n^2$ cc will be required. If we assume a broadcast network with connections to all processes, the count of cc is even higher. In the best case, it will be n^2+2n^3; in the worst case it will be n^2+m*2n^3. This is a very expensive algorithm in terms of the cc overhead!

5.4.3. Huang's Solution to the Problem

Huang [HUAN88] builds his algorithm on a system which has logical clocks, and no specific topology for either the bc or the cc. The basic approach is as follows: When a process, p_i, becomes passive it makes an announcement to the rest of the processes that it is ready to terminate at its own (p_i's) clock time. If the other processes do not know of a later time at which a process has gone passive, they return an agreement. Any process that can obtain agreements from all processes in the system may initiate termination.

Each process is given a logical clock,[9] represented by the variable $time_i$ which consists of two parts: a time value, $time_i.value$ and the name of the process that owns the clock, $time_i.name$. Each process also remembers the latest idleness time received in an announcement, and the name of the process announcing it,[10] in the variable $Btime_i$. Two clock times are related as follows:

$(time_i.value, p_i) > (time_j.value, p_j)$ if

(a) $time_i.value > time_j.value$, or

(b) $time_i.value = time_j.value$, and $p_i > p_j$

Note: we interpret greater as later

9 In Huang's algorithm the clock is represented by x_i.
10 In Huang's algorithm the latest idleness time is represented by z_i.

All clocks have the properties of being totally ordered and monotonically increasing. To guarantee the monotonic increasing property of the clock, p_i adjusts its clock each time a cc is received to be the max of p_i's clock and the clock time carried by the cc. In addition, p_i increments its clock by one each time it becomes passive.

There are three types of cc: announcements, *Ann*, agreements, *Agr*, and acknowledgments, *Ack*. Each carries the logical clock of the process sending the cc. Specifics of the algorithm can be found in the appendix. There is a special case that should be noticed here. A process can not send a bc and then go passive immediately; it must wait for an acknowledgment before going passive. During this time it cannot send any other bc, although it may perform its own computations, and receive and acknowledge other bc. By the same token, any process receiving a bc, must immediately respond with an acknowledgment.

The reader will note that the agreements are not counted in the algorithm printed in the appendix. The reason for this is that the method will be different for differing topologies.

If the topology is a tree, there will not be a specific count. There will always be m acknowledgments. In addition, the root will send an announcement to all of its children. If all children are passive and this is a later time, the announcement will be passed down to all of their children and so on. If at any time an active process, or a process which has a later idleness time, is encountered then the announcement will be purged. Then each node, p_j, needs to keep a variable that holds the information that an announcement has been received back from all of its children. When all announcements have been received by p_j, it passes the announcement back up to its parent. When the root receives the

announcement from all of its children, it may conclude termination. There is no mention of how the root knows to send a second announcement if it does not get a return announcement from all of its children. Although a tree topology is specifically mentioned in the article, this author believes the use of a tree would imply a centralized algorithm and this is clearly not what Huang intended.

If the topology is a Hamiltonian ring, again a specific variable to hold the count will not be required. Any node will be able to initiate an announcement along the ring. As each node receives it, the announcement will be passed on only if this process is passive and the clock time of the message is greater than the latest idle time known. Otherwise the message is purged. If a process ever gets its own announcement back, it will know to initiate termination. In the best case[12] this could mean $2n$ cc. In the worst case[13] the count of cc is $< m * n + n^2$. In both cases there will also be m acknowledgments.

If the topology is totally arbitrary and a broadcast network is used then each process, p_i, must keep a count of agreements returned from an announcement. In the best case all processes go passive at about the same time and the network is deluged with announcements. The code for each process could be written in such a way that all announcements are received before any agreements are returned. In this case there would be n announcements and n agreements to the process with the latest idle time, along with the m acknowledgments for a total of $2n + m$ cc. In the worst case,[14] each bc reactivates some

12 All processes become passive at about the same time. Each announcement is sent to its successor on the ring and finds a later idleness time and is purged. The process with the latest idleness time receives its announcement back.

13 Initially all processes except one go passive. Then all further bc from any process, p_i, reactivate the predecessor of the sender. Therefore the announcement from p_i will visit n−1 processes before it becomes purged. The process with the latest idleness time will eventually receive its probe back.

14 All processes except one return agreements after some process goes passive and each bc reactivates some process.

cess. There will be as many as $2mn + 2n + m$ cc.

The memory requirements are two clocks for each process and, in the case of the arbitrary broadcast network, an additional integer. The addition to a bc sent by p_i, is an acknowledgment from the receiver that contains a clock and a possible resetting of p_i's clock. The reader will notice that the clock overhead is not cheap.

5.4.4. Eriksen's Solution for an Arbitrary Network

This algorithm requires that each process, p_i, contains a local counter, ctr_i, and access to two auxiliary functions $\min_i$, and $\max_i$. These functions return the minimal/maximal value of the counters of all of p_i's neighbors. A constant, L, is predefined to be greater than or equal to the size of the network plus the diameter of the network minus 1.

Initially all counters are set to 0 and whenever any process becomes reactivated, after having been passive, its counter is restored to 0. The behavior of a passive process, p_i, is described by the following rules:

Rule 1: if $ctr_i = \mathrm{L}$ then terminate

Rule 2: if $\min_i + 1 < ctr_i$ then $ctr_i := 0$

Rule 3: if $\max_i - 1 \leq ctr_i \leq \min_i$ then $ctr_i := ctr_i + 1$

The updating of counters is considered an indivisible operation. Rule 2 guarantees that whenever any process goes from passive to active, eventually the counter for all processes will be restored to zero. Rule 3 guarantees that all bc will be caught. If the counters are incremented slowly in this fashion, a false termination will not be declared. Every time a process changes its counter, it shares this information with each of its neighbors.

This algorithm is really simple to understand; however its proof of correctness is based

on a formalization of the rules using graph theory and is beyond the scope of this paper. The modification of bc is minor - a counter is set to 0. The addition of memory is also minor - a single integer. However, the count of cc could be quite high. Notice that the counter must be incremented to $n+d-1$ by **each** process - after it becomes passive. In the best case, all processes become passive at the same time and the count is $n^2+n*(d-1)$ assuming each process has only 1 neighbor - the topology is really a Hamiltonian ring. In the worst case, every message reactivates some process and the count is at least $mn+n^2+n*(d-1)$, again assuming each process has only 1 neighbor. This is a higher overhead than the centralized algorithms for tree topologies. It will be even higher for a truly arbitrary network that allows each process to have more than 1 neighbor for the control topology. In the worst case, the network is completely connected and the count of cc for the best case is $n^3+n^2*(d-1)$. For a completely connected network the worst case count of cc is $mn+n^3+n^2*(d-1)$.

5.5. Solution for an Application Which Requires Synchroneity

Szymanski et al [SZYM85] present an algorithm for a specific class of computation. The definition of the computation is motivated by attempts to use a network of distributed processes to solve a large set of simultaneous equations. This problem requires some degree of synchroneity and therefore the termination algorithm is actually a variation on the distributed termination problem described in this paper. The computation is described as follows:

> The main computation satisfies the following property: a process outputs its $(j+1)$th output along every outgoing communication channel only after receiving the jth input along every incoming channel [SZYM85,1136].

Each process counts the number of times it has received input from all of its predecessors. This value is used as a local counter of the main computation steps and as an index of process activities. To synchronize termination of the main computation, each process has to stop with the same value of the main computation step. The termination detection algorithm involves sending tokens through the network and evaluating node states. Tokens are labeled by integer values ranging from 0 to $D + 1$ where D is the diameter of the network.

There are two additional variables for each process used in this algorithm:

I_i The minimum label of all tokens received at node i in the jth input.

S_i The integer label to be put on the jth output token.

$S_i(j)$ is defined for process p_i and step j as follows:

0	if b_i is false
$I_i(j)+1$	if b_i is true and $I_i(j) < S_i(j-1)$
$S_i(j-1)+1$	otherwise

The algorithm is simple:

```
for all p_i do
    begin
    S_i = 0
    while S_i < D do
        send out tokens labeled by S_i
        read input-tokens
        evaluate b_i
        if not b_i then
```

$S_i = 0$

else

$S_i = \min\ (I_i, S_i) + 1$

end while

stop the process

end

The overhead for this algorithm is very low. The state is evaluated and added on to the outgoing bc. There are two additional integer variables for each process and no additional control communication. However, the algorithm is of use only for very specialized problems.

5.6. Summary

This chapter looked at three approaches to solving the distributed termination problem:

1. Hamiltonian ring network topologies
2. synchronization within an asynchronous environment
3. arbitrary network topologies - including trees where the root is arbitrarily self-chosen

In the first approach, only one of the solutions offered was correct. However, knowledge was gained from looking at these algorithms - they made the possibilities for additional occurrences of behind the back communication clear. One of Arora's algorithms was fixable yielding results that are typical for a ring. The other carries a very high overhead in terms of both memory and the count of cc.

The second approach deviated from the accepted definition of termination. This solution illustrated the possibilities for reducing the overhead of termination algorithms if the algorithm makes use of the unique characteristics of the particular application.

The arbitrary network approach is actually a broader look at the problem than has been used in previous chapters. The reader will notice that in the algorithms presented by Misra [MISR83] and Huang [HUAN88] the worst case overhead for a ring is approximately half as much as the overhead for an arbitrary network in the former case or broadcast network in the latter case. In the best case overhead, Misra's ring algorithm requires two thirds the number of cc as does his arbitrary network algorithm and Huang's overhead for the best case is the same in both situations. In the case of Skyum [SKYU86] and Eriksen [ERIK88] the count of cc overhead in both cases is less by a factor of n for the algorithms based on ring topologies than for the algorithms based on arbitrary network topologies. Mindful of this, the reader should be careful to compare overhead for algorithms presented in previous chapters to the algorithms in this chapter which are based on ring topologies.

In comparing the total overhead of all the algorithms which assume a ring topology, we notice that the additions to the bc are all essentially the same except for Huang's algorithm which requires a clock update as well. Misra requires the least amount of additional memory closely followed by Eriksen. Skyum's algorithm requires more than three times as much memory as Misra's and Huang's algorithm requires four times as much memory as Eriksen's. If the amount of memory used were the only criterion for choosing an algorithm, either Misra's or Eriksen's algorithm would be a good choice. However, if message volume (and therefore speed) is also a criteria then Misra's algorithm would be the best choice.

If the number of basic messages required for the application is small, a clock is built into the system, and memory space is not an issue, Huang's solution would be almost as good as Misra's solution. This author would not choose Skyum's solution as the best for any situation.

When for one reason or another it is not possible to superimpose a Hamiltonian ring upon a network, the solutions based on an arbitrary network topology are the only choice.

The reader should be aware that all values presented are worst case scenarios for the network and some networks may require fewer cc. Again the additions to the bc are the same in all cases with the exception of Huang's algorithm which requires clock overhead. In terms of memory overhead, Misra's algorithm now requires a substantially greater amount of memory than any of the other algorithms. The memory requirements for Skyum's and Eriksen's algorithm remain the same and the additional memory for Huang's algorithm is minimal. Therefore if memory requirements are the most important, Eriksen's algorithm for arbitrary networks would be the best choice.

If keeping the volume of message traffic low is of prime importance, Misra's algorithm would be the best choice. If the volume of basic communications is relatively low Huang's algorithm would be a good second choice. If the size of the network was sufficiently small relative to the basic communications, Eriksen's solution would also be a possible choice. However, the cc overhead for Skyum's solution does not ever make it a good choice.

In conclusion, we can say that in most cases, Misra's algorithm is the best solution to the distributed termination problem in the asynchronous distributed case. Huang and Eriksen share the second choice position dependent upon the situation. Skyum's solution is not the best solution in any situation.

Algorithm	Additions to bc	Best Case Overhead	Worst Case Overhead[1]	Additional Memory
BIDIRECTIONAL RING TOPOLOGY				
[AROR89]				
single probe	set flag at snd/rcv	$2n$	$mn + 2n$	$4n$ booleans $2n$ integers
multiple probes	set flag at snd/rcv	$n^2 + n$	$< mn + 2mn^2$	$< 2n^2$ booleans $< 2n^2 + n$ integers
ARBITRARY NETWORK TOPOLOGY				
[MISR83]				
ring	set color to black	$2n$	$mn + n$	n booleans
arbitrary	set color to black	$\geq 3n$	$\geq 2mn + 3n$	n booleans $< n^2 + 2n$ integers
[SKYU86]				
ring	set active to true	$n + 2n^2$	$n + 2mn^2$	n booleans $2n$ integers 1 int constant
broadcast	set active to true	$n^2 + 2n^3$	$n^2 + 2mn^3$	n booleans $2n$ integers 1 int constant
[HUAN88]				
ring	send ack update clock	$2n + m$	$< mn + n^2 + m$	$4n$ integers[2]
arbitrary	send ack update clock	$2n + m$	$2mn + 2n + m$	$5n$ integers
[ERIK88]				
ring	at recv ctr := 0	$< 2n^2$	$< mn + 2n^2$	n integers
arbitrary	at recv ctr := 0	$< 2n^3$	$< mn + 2n^3$	n integers
SYNCHRONIZED SOLUTION				
[SZYM85]	add state to bc	none	none	$2n$ integers

1 In the case where a cycle length or the diameter of the network is used to compute the overhead, n and a relation ($>$ or $<$) are substituted for easier comparison.

2 Notice the agreements are not included in this count. See page 85 of the text for an explanation.

Table 5-1

6. SOLUTIONS FOR UNRELIABLE CHANNELS

In chapters two thru five we have assumed that the communication channels are reliable, i.e., messages are not altered, lost, duplicated, or desequenced. The purpose of this chapter is to look at the results of lifting some of these restrictions from the communication channel. In particular we will release the restriction that all messages must be delivered in the order in which they are sent. Messages are received in an arbitrary but finite amount of time - they are not instantaneous.

In chapters 4 and 5 we have claimed that all processes being passive at the same time instant and all channels being empty of messages are sufficient conditions to declare termination in a distributed system. The FIFO property of message delivery enabled the algorithms to guarantee that the channels were empty. With the lifting of this requirement, other methods must be found to guarantee that all channels are empty. There are two approaches:

1. Require an acknowledgment to be sent for every bc received.
2. Keep a count of messages sent/received.

Dijkstra's algorithm [DIJK80], is one of the earliest algorithms found in the literature and follows the first approach. There are two researchers, of whom this author is aware, that follow the second approach: Devendra Kumar [KUMA85] and Friedemann Mattern [MATT87A] [MATT87B].

Initially one might assume that what is needed is to count all messages sent, $S()$, and all messages received, $R()$. Then when $S() = R()$ becomes true for the system, it may be assumed the system has terminated. However, in the discussion that follows the reader will notice that the method used to count the messages affects the outcome - additional

conditions may also be required.

6.1. Dijkstra's Solution to the Problem

Dijkstra and Scholten [DIJK80] solve the distributed termination problem through the design of a signalling scheme to be superimposed on a diffusing computation proper. This scheme allows the diffusing computation proper to signal the environment when completion of the computation has occurred. The algorithm requires that an acknowledgment is sent for every bc received.

6.1.1. Definitions for Dijkstra's Solution

Each process, p_i, needs a method to keep track of how many messages it has received from its predecessors, p_j, and therefore how many signals (acks) must be returned and to whom. Each process, p_i, may be considered to have a *bag* into which the name of process p_j is placed each time a message is received from p_j. A signal may be returned by p_i, to each p_j contained in its bag.

A *coronet* differs from a bag in that a process, p_i, holding a coronet remembers the name, p_k, of the very first process sending a message and placed in the coronet. When it is time to return the signals, the very last signal to be sent is the signal to p_k.

The term *deficit* is defined to be the number of messages sent along an edge minus the number of signals returned along it. C is the sum of the deficits of the incoming edges for any node. D is the sum of the deficits of the outgoing edges for any node.

6.1.2. Approach

Dijkstra and Scholten approach the solution to the distributed termination problem by designing a signalling scheme to be imposed upon a diffusing computation and requiring

an invariant to hold during the entire computation. They structure the problem in the following way:

1. No process, except the environment, may send a message until it has first received a message - definition of diffusing computation.
2. At all times, each edge may never have carried more signals than messages - definition of signalling, a response to a message.

Each edge is assumed to be able to accommodate two-way traffic, but only messages of the computation proper in the one direction and signals in the opposite direction.

In order to satisfy the second premise in the structure of the problem Dijkstra and Scholten impose the invariant P0:

P0: Each edge has a non-negative deficit.

P0 can be kept invariant by keeping P1 and P2 invariant for each node.

P1: $C \geq 0$

P2: $D \geq 0$

Notice that initially $C = 0$ and $D = 0$ for all nodes. Only the environment node will have the right to send a signal and it will always have $C = 0$ and $D \geq 0$. For all of the internal nodes a message may be sent only if $C > 0$ i.e., $C \geq 1$. By the same token, the last signal may be sent only if $D = 0$, i.e., signals have been received for all messages sent. These two conditions can be guaranteed by keeping

P3: $C > 0$ or $D = 0$

invariant. If all nodes keep P3 invariant for themselves, then P3 will be kept invariant over the computation. Since all edges must carry equal numbers of messages and signals

when the environment node has $D = 0$, the computation will be complete and termination will have occurred. At termination the internal nodes have $C = 0$ or ($C = 1$ and $D > 0$) and the environment has $C = 0$ and $D \geq 0$.

If process p_i holds a bag to determine which processes, p_k, need to receive a signal the only restriction on p_i is that the last signal may not be sent unless $D = 0$ for p_i. If a coronet is used instead of a bag then

P4: all engaged nodes are reachable from the environment via directed paths, all edges of which have positive deficits.

may also be held invariant. The result is that the diffusing computation builds the tree which in Francez's algorithms had to be derived by the programmer.[1] Since the environment "has control", when it returns to its neutral state the diffusing computation has terminated.

Dijkstra and Scholten demonstrated the truth of this as follows:

(a) each engagement edge connects two engaged nodes (because it has a positive deficit and, hence, leads from a node with $D > 0$ to a node with $C > 0$);

(b) engagement edges do not form cycles (because, when the edge from p_i to p_j became an engagement edge, p_j was initially neutral and, hence, had no outgoing engagement edge);

(c) each engaged internal node has one incoming engagement edge (on account of P3 and because its bag has been replaced by a coronet).

From (a), (b), and (c) we conclude that the engagement edges form a rooted tree -

1 See chapter 2 for a description of Francez's algorithms.

with the environment at its root - to which each engaged node, but no neutral node belongs. Hence its edges provide the paths whose existence implies the truth of P4...[DIJK80,3]

6.1.3. Advantages

The advantage of this algorithm is its simplicity. Although it is not explicitly stated, for this algorithm to work correctly the final signal from each process, p_i, should occur only when the equivalent of b_i becoming true has been met.

Although the algorithm is simple, it does carry a high cc overhead. The number of cc is m and can never be any lower. In addition there are two integer variables to be maintained for each process. The bc is modified to include a check to see if it is legal to send a message (i.e., that the node sending is engaged) and a counter (the deficit for that edge) is incremented. Finally additional code must be added to return the signals: check on conditions, send the signal and decrement a counter (the deficit for that edge).

Despite its disadvantages this algorithm is important for several reasons:

1. It is one of the earliest algorithms to be published (second only to Francez).
2. It is the first algorithm developed for asynchronous communication, and it is also correct for message order that is not fifo.
3. It is a good example of the use of invariants to develop an algorithm.

6.2. Kumar's Solution

Kumar's purpose in writing this article [KUMA85] is to instruct the reader in the method used to develop his algorithms. Three initial algorithms are introduced as well as improvements and variations to them, some of which are incorrect. This is done for the

purpose of aiding the reader in gaining insight into the problem.

Kumar looks at three classes of algorithms:

1. algorithms which assume a single cycle in the network,[2] not necessarily a simple cycle, and count the bc sent/received on **each** line
2. algorithms which assume a single cycle in the network, again not necessarily a simple cycle, and count the total number of bc sent/received in the system
3. algorithms which assume multiple cycles (these do not have to be disjoint)

This third class takes specific properties of the network into consideration and cannot be considered a generic solution. As the number of cycles increase, it degenerates to an algorithm for the centralized case. Therefore, this author chooses not to discuss it in this paper.

The variables used for class 1 of this algorithm are all arrays of integers with dimensions sufficiently large to include each edge of the network. They have the following meaning for the class 1 algorithms:

sntm an array carried by the probe to indicate all messages sent in the system

recm an array carried by the probe to indicate all messages received in the system

$sntp_i$ an array which contains a count of all messages sent out on each edge by p_i

$recp_i$ an array which contains a count of all messages received by p_i along each edge

2 Kumar defines a cycle to include every process of the network at least once - all edges do not need to be traversed.

srm used in the improvement to the algorithm as the sole contents of the probe

$srm = sntm - recm$

Note: $sntp_i$ and $recp_i$ are large enough to contain a count of messages for **every** edge in the system even though p_i may not be a vertex on that edge. Initially Kumar suggests all arrays are initialized to 0. However, in an attempt to eliminate the need to keep track of whether or not a complete cycle has been made at least once, initializations for each edge, e, are made as follows:

$sntm(e) := 1$

$recm(e) := 1$

$sntp_i(e) := 0$

$recp_i(e) := 2$

For both classes of algorithms based on a single cycle, Kumar uses an algorithm skeleton. Although Kumar does not use the statements *disable* and *enable*, this author borrows them from Skyum and Eriksen [SKYU86] to indicate the atomicity of a visit by the probe. The skeleton is as follows:

After the probe arrives at p_i, it waits until b_i is met.

disable

$sntm := sntm + sntp_i$

$sntp_i := 0$

$recm := recm + recp_i$

$recp_i := 0$

enable

Declare termination or depart from p_i

The algorithm is simple to understand; however, it is quite inefficient. Whenever a bc is sent/received $sntp_i/recp_i$ must be incremented for the edge that carried the bc. There is a single probe token placed randomly at some process in the system. Each node contains a list of the next node for the probe to visit. If the cycle is simple there will only be one name in the list, otherwise the node must also remember to which node the probe is to be sent at its next visit. The probe carrys two arrays with dimensions $|c|$ – the length of the cycle. Each node also has two arrays of dimension $|c|$. When the probe reaches each node it adds $sntp_i/recp_i$ to the appropriate values in the probe and resets $sntp_i/recp_i$ to 0. The major question to be asked concerning this algorithm is: 'When may termination be declared?' The condition $sntm = recm$ is sufficient if the probe has made at least one complete cycle of visits. The second set of suggested initializations guarantees that all processes have been visited at least once. Since the probe is not passed on until a process is passive, if the probe ever finds $sntm = recm$ then all processes are passive and all channels are empty.

Additional memory required is n counters for each edge of a fixed communication graph, or, in the case of a Hamiltonian cycle, a total of n^2 counters. In the best case the count of cc will be $|c|$ - if the network is a Hamiltonian cycle it is n. In the worst case the count of cc will be $m(|c|-1)$. The cost of this algorithm in terms of memory, and communication overhead - both bc and cc - is quite high. Kumar suggests several possible improvements:

1. use only one array, srm, in the probe
2. reduce the size of the arrays at individual nodes - so that they contain only the edges connected to p_i - but add a mapping function from names for the edges

connected to p_i to the global names for the edges

Neither of these improvements solve the most serious problems - messages on each edge are counted separately and the size of the probe is dependent on the number of edges in the network.

The algorithms for class two are an attempt to reduce the size of the message carried by the probe. The meanings for the variables used in class 2 algorithms are as follows:

sntm2 the probe carries an integer to indicate the total number of messages sent in the system

recm2 the probe carries an integer to indicate the total number of messages received in the system

$sntp2_i$ integer to indicate the total number of messages sent by p_i

$recp2_i$ integer to indicate the total number of messages received by p_i

srm2 used in the improvement of the algorithm as the sole contents of the probe

$srm2 = sntm2 - recm2$

Note: each of these variables is initialized to 0.

The algorithm for class 2 counts total messages in the system. The algorithm works in the same way as for class 1. The difference in the use of the variables requires a new answer to the question: 'When will the probe be able to declare termination?' This method allows false detection of termination to occur. Kumar [KUMA85,89] presents an example "to show that there exist computations where in an infinite sequence of visits, the probe continuously finds that $sntm2 = recm2$, i.e. termination has occurred, and yet the primary computation never terminates." Consider the following case [KUMA85,89]:

Consider a network of 10 processes. The cycle C is the elementary cycle 1, 2, ..., 10, 1. Initially the probe 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 probe 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 probe visits 10, and departs.
5. The above steps 1-4 are repeated indefinitely.

Obviously, after every visit the probe will find that $sntm2 = recm2$. But the primary computation would never terminate!

A correct algorithm must not detect termination in this example. This situation proves that $sntm2 = recm2$ at the end of a visit is not sufficient to detect termination. However, Kumar suggests that if at the beginning of the visit $sntp2_i = recp2_i = 0$ and $sntm2 = recm2$ after the visit for a full cycle of $|c|$ visits, then termination may be declared. Notice this means that p_i has neither sent nor received any bc since the last visit by the probe.

Kumar offers a more efficient algorithm[3] by making the following changes:

1. check that $recp2_i = 0$ before each visit
2. check $sentm2 = recm2$ only at the end of the visit to the last process in the cycle.

Every time the probe finds $recp2_i \neq 0$ before a visit, a new cycle is started. In addition memory may be conserved by replacing *sntm2* and *recm2* by $srm2 = sntm2 - recm2$ and replacing $sntp2_i$ and $recp2_i$ by $srp2_i = sntp2_i - recp2_i$. In order to maintain the check $recp2_i = 0$ before each visit, $recp2_i$ becomes a boolean flag to indicate if a bc has been

3 the number of cc required to detect termination after termination has occurred is reduced.

received since the last visit by the probe.

Kumar offers three methods of detecting the end of a cycle:

1. Sequence Length Counter - the probe carries a counter, *ctr*, initially set to 0. If $recp2_i = 0$, the counter is incremented, otherwise it is reset to 1. At the end of each visit, check the counter. If $ctr \geq |c|$ then check the condition $sntm2 = recm2$. If this condition is false, then at the beginning of the next visit, since $ctr \geq |c|$, *ctr* is reset to 1.

2. Round Number - each process remembers the round number which the probe carried at the last visit. Whenever the probe visits a process and finds $recp2_i \neq 0$, the round number is incremented. When the probe encounters a process with the same round number, if $recp2_i = 0$ at the start of the visit and $sntm2 = recp2$ after the visit, then termination may be declared. The round number of the probe is initialized to 1 and the round number of the processes is initialized to 0.

3. Initial Process Id - the probe keeps the name of the initial process in the current cycle of visits. Whenever $recp2_i \neq 0$, the process id is changed to be the current process. When the probe returns to the process carried in initial process id, finds $recp2_i = 0$ at the start of the visit and $sntm2 = recp2$ then termination may be declared.

This algorithm reduces the communication length of the cc, as well as the additional memory requirements. The number of cc used in the best case is[4] $|2c|$. In the worst case, the number of cc is $(|c| - 1) * (m) + |c|$.

4 This assumes that every process sends at least 1 bc. If no process sends any bc, the best case is $|c|$.

6.3. Mattern's Solutions

Mattern [MAT87B] offers several algorithms, which he claims are efficient, for the solution of the termination problem. They are all based on the idea of message counting, but have differing characteristics. The five groups are as follows:

1. The four counter method
2. The sceptic algorithm
3. A time algorithm
4. The vector counter method
5. Channel counting.

The first two groups require two probes to detect termination - the initial probe and a confirming probe. The latter three groups require only one probe. The vector counter method is explained in great detail in a different article by Mattern [MAT87A].

In previous chapters we have assumed a transaction oriented model of distributed computation. Mattern assumes an atomic model of distributed computation.[4]

6.3.1. The Four Counter Method

In the four counter method, each process, p_i, keeps track of all of the messages that it sends/receives. Any process, p_j, can initiate a probe. When the probe visits p_i, it adds the number of messages sent by p_i to its cumulative total of messages sent, $msg.S()$, and the number of messages received by p_i to its cumulative total of messages received, $msg.R()$. When the probe returns, a second probe is sent out, holding $msg.S'()$ and $msg.R'()$. When the second probe returns if $msg.S() = msg.R() = msg.S'() = msg.R'()$ then termination

4 See chapter 1 page 4 for the definition of an atomic model.

may be declared. The problem with this method of detecting termination is that there is no answer to the question 'how often should a probe be reinitiated?' and the network may be flooded with unnecessary probes.

This algorithm requires two additional integers in memory at each node. In addition, *S()/R()* must be incremented at each send/receive. Mattern does not specify the network topology for this algorithm; however, for a ring, the best case overhead will be $2n$, and there is no upper bound for the worst case. If there are multiple probes, the best case overhead will be $4n^2$.

6.3.2. The Sceptic Algorithm

A sceptic algorithm is an algorithm that makes use of a flag to detect the occurrence of a send/receive of bc between probes. We've seen examples of sceptic algorithms in Dijkstra's [DIJK83] and Topor's [TOPO84] colored token algorithms, in Francez's third algorithm [FRAN82], and in Kumar's class 2 algorithm [KUMA85]. Mattern's sceptic algorithm adds flags to the four counter algorithm and works as follows:

> As the first probe goes around the ring it initializes the flags of each process to true and accumulates the number of messages sent/received. If *msg.S()* = *msg.R()* then a second probe is sent out which only checks the flags, otherwise a new first probe is initiated. During the second probe, if no flag has been found set to false (because a bc has occurred) then termination is declared, otherwise a new first probe is initiated.

This algorithm is best used in a centralized or single token network. It is not used as easily in a ring because probes from different processes may interfere with each other when processes use a single flag. In a ring the best case overhead will be $2n$ if a single token is used. In a multiple token solution, the best case overhead will be $4n^2$. There is no upper

bound on the cc for the worst case. Overhead for the bc is to set the flag at either a send or a receive, and to increment either $S_i()$ or $R_i()$. Additional memory is two counters at each node for bc sent/received and the flag. When multiple probes are used, there will be n flags at each node.

6.3.3. A Time Algorithm

In the time algorithm presented by Mattern, only one probe is needed to detect termination - no confirmation probe is required. Each process, p_i, makes use of the following variables:

clock the local clock, a counter initialized to 0

count the local message counter, it is equivalent to $s_i() - r_i()$

tmax the latest send-time of all messages received by p_i, initialized to 0

tstamp the time stamp on any bc

The probe, *msg*, consists of the following information:

time the timestamp of the cc

count the accumulator for the message counters

invalid the flag that indicates that a bc has been received since the last round

init the id of the process initiating the probe

Whenever a bc is sent, *count* is incremented and bc is timestamped with *clock*. When a bc is received, *count* is decremented and *clock* is reset to the greater of *tstamp* and *tmax*. A probe is initiated by incrementing the clock, setting *msg.time* to this new value for the local clock, initializing *msg.count* to *count* and *msg.invalid* to false, and sending the probe to succ(p_i). When a probe is received, the local clock is synchronized to the max of

msg.time and *clock*. Then if this process was not the initiator, then *msg.invalid* is set to *msg.invalid* or $tmax \geq msg.time$, and *msg.count* is incremented by *count* and passed on to the next process. If this process is the initiator, then if *msg.count* is 0 and *msg.invalid* is still false, then termination may be declared. If termination is not declared, a new probe is initiated. Mattern does not suggest a method for limiting the number of times a probe is reinitiated.

The additional memory requirements for this algorithm are three integers for each process. The bc are augmented with a time stamp, and whenever a bc is sent/received a counter is incremented/decremented. In the best case [5] the cc overhead is $< n^2$, and in the worst case there is no upper bound.

6.3.4. Vector Counters

The purpose of this method is to count messages in such a way that it is not possible to mislead the accumulated counters. A ring topology is assumed and every process, p_i, maintains a vector, $count_i$, of length n. The probe, a single token, also contains a vector, *msg.count*, of length n. The system initiates the single token and any process may detect termination while it is holding the token. In a previous article [MAT87A], Mattern explicitly states that each process has an additional flag, *have_vector*, that indicates whether or not the token is currently visiting that process. The $count_i$ are used to count the number of messages each p_i sends/receives since the last visit of the probe. Whenever p_i sends a bc to p_j, $count_i[j]$ is incremented. Whenever p_i receives a bc, $count_i[i]$ is decremented. When the token visits process p_i, the contents of *msg.count* is added to $count_i$. If the

5 The first process, p_i, to become passive initiates a probe. As p_i's probe reaches p_j, the successor of p_i, p_j goes passive. No probe meets an active process. There are actually $\frac{n(n+1)}{2}$ cc.

result of this operation is that $count_i[i] > 0$ then some process has sent a bc to p_i and p_i holds the token until the bc is received and only then is the token passed on. If $count_i[i] \leq 0$ then if $msg.count = 0^*$ (the null vector) then termination may be declared. If $msg.count \neq 0^*$ then $msg.count$ is set to $count_i$, the token is passed to the next process in the ring, and $count_i$ is reset to 0^*. In order to guarantee that the probe makes at least one complete round of visits before declaring termination, $msg.count[j]$ is initialized to 1 for all j and $count_i[i]$ is initialized to -1 for each process i (for all j, $j \neq i$, $count_i[j]$ is initialized to 0). This is explicitly mentioned in the previous article [MAT87A] although it is not mentioned here [MAT87B]. This method is very similar to Kumar's improvement for class 1 algorithms.

When this algorithm is used on a ring topology, the best case for cc overhead is n and the worst case overhead is mn. However, the message carried by the probe is length n. The addition to memory is a boolean and an array of n integers at each node. The additon to the bc is an increment/decrement at every send/receive.

6.3.5. Channel Counting

In the vector counter method each message was counted twice; by its sender and by its receiver. The sender had individual counters indexed by the recipient of the messages, whereas the receiver did not differentiate between the senders. Mattern regards the channel counting method as a refinement of the vector counter method; the receiver takes note of the sender and keeps track of the number of messages received by each node, using the appropriate counter. This principle is similar to the class 1 algorithm presented by Kumar [KUMA85] which is based on counting messages on every communication channel. Because the algorithms are so similar, Mattern's version will not be described here.

6.4. Summary

At first glance, the total amount of overhead for all of the algorithms looks similar. However, it must be remembered that the length of the probe for Kumar's class 1 and Mattern's vector and channel counting algorithms is dependent on the size of the network, and therefore, the communication overhead is substantially raised. In addition these three algorithms use substantially more memory than the others, although the additions to the bc are comparable.

Mattern's Four Counter, Sceptic, and Time algorithms may also have a heavy communication overhead because there is no restriction on how often probes are initiated, although the use of memory and the additions to the bc are within normal range. The communication overhead is even higher for the Four Counter and Sceptic algorithms when there are multiple initiators of tokens. By eliminating these six algorithms, we are left with two algorithms to choose from to detect termination in the asynchronous distributed case with non-FIFO message passing:

Dijkstra's algorithm

Kumar's algorithm for class 2

Kumar's algorithm requires less memory than Dijkstra's algorithm. In addition Kumar's algorithm requires fewer cc than Dijkstra's algorithm in the best case and is of the same order of magnitude in the worst case. This author's choice of detection algorithm is Kumar's; however, Dijkstra's algorithm is extremely simple to understand and could be used in an arbitrary network if memory space is not a problem.

Algorithm	Additions to bc	Best Case Overhead	Worst Case Overhead	Additional Memory
DIFFUSING COMPUTATION MODEL				
[DIJK80]	incr deficit ok to send msg? add receiver to coronet	m	m	$2n + m$ integers
RING NETWORK TOPOLOGY				
[KUMA85][1]				
class 1[2]	at send/recv incr count	c	mc	c^2 integers
class 2	at send: incr count at recv: decr count set flag	$2c$	$mc + c$	n integers n booleans
[MAT87B]				
4 ctrs	at send/recv incr count	$2n$ [3] $4n^2$ [4]	no upper bound	$2n$ integers
sceptic	set flags at send/recv incr count	$2n$ [3] $4n^2$ [4]	no upper bound	$2n$ integers n booleans [5]
time	at send incr count timestamp at recv decr count reset clock	$< n^2$	no upper bound	$3n$ integers
vectors[2]	at send incr count at recv decr count	n	mn	n^2 integers n booleans

1 A cycle is used for both of Kumar's algorithms, if it is a Hamiltonian cycle n edges are required, if not $c \geq n$ edges are required.

2 The size of the probe is dependent upon the number of processes in the system.

3 A single token is used.

4 Multiple tokens are used.

5 If multiple tokens are used, then n^2 booleans are required.

Table 6-1

7. A NEW ALGORITHM

In chapter 5, three algorithms were presented that allow any process to initiate a detection probe at any time, Eriksen [ERIK88], Arora and Gupta [AROR89], and Huang [HUAN88]. Both Arora and Eriksen have best case communication overheads of $O(n^2)$ and Huang has a best case overhead of $2n + m$. All three algorithms have worst case communication overheads of $O(n^2 + mn)$. Arora's algorithm keeps track of the state of the neighbors of each process, Eriksen's algorithm makes use of counters, and Huang's algorithm makes use of a virtual clock. This author presents an algorithm for the asynchronous FIFO message ordering case that has a comparable overhead and uses tokens.

In the belief that the simpler the algorithm is to understand, the greater the chance that it will be efficient, we start with the concept of keeping track of whether or not a process has been passive for the entire time since the last visit of the probe. This requires that each process maintain a single variable, bc_i, to indicate when a bc has been received. Initially bc_i is assigned the value false. It is also assigned the value false whenever a bc is received. When a probe arrives at an active process, it is immediately purged. If the process is passive, the value of bc_i is checked. A value of false falsifies the probe. Before leaving p_i, a value of true is assigned to bc_i. If a probe can return to its initiator unfalsified, all bc_i were true, then termination can be declared. This will take one round of visits to initialize the probe and a second round of visits to guarantee that the process has not been active.

This is essentially Dijkstra's algorithm [DIJK83] using boolean values instead of colored tokens. The problem is that since every process can initiate a probe, the probes may interfere with each other by resetting bc_i. Therefore, we need a way for each probe to

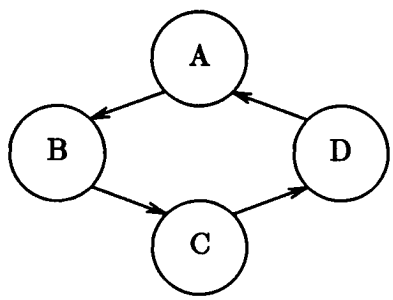
check the status of a process in the current time interval without interference from others. One way of doing this is for each process to maintain an array of bc_i, one for each process in the system. Then when a probe from p_j visits process p_i, it checks the value of $bc_i[j]$ and then reassigns it the value true when it leaves. In this way, we have eliminated interference by probes from other processes. The idea of maintaining an array is borrowed from Mattern's vector algorithm [MATT87B].

Now there will be no interference between probes; however, the system will be flooded with cc. We need a means of controlling the number of probes sent out. Answers for the following questions are required:

1. Must a process send out a probe every time it becomes passive?
2. If a probe returns falsified after the second round should a third round be initiated?
3. May a single process initiate multiple probes?

When the last process in the system, p_i, becomes passive, it does not send out any bc. If this were not the case, some other process would be guaranteed to be active after p_i. This leads us to believe that if a process does not send out any bc during this activation, it might be the last active process in the system. If this is the case, we could require that every process maintain an additional variable, $sent_i$, that is assigned the value true each time a process sends a bc. Initially $sent_i$ is assigned the value false. After verification of its value, $sent_i$ is also assigned the value false just before p_i becomes passive. Then we can require that a process may not initiate a probe unless $sent_i$ is equal to false. In reality this is not the case. Consider the following scenario: Four processes, A, B, C, and D are connected in a ring as shown in the figure below. Process A sends bc to B, C, and D and

remains active. Process B sends bc to C and C sends bc to B and then both become passive. After receiving the message from A, D returns a message to A and then goes passive. After process A receives the message from D, A goes passive. Clearly A is the last process in the system to go passive; however, because A has sent messages during this activation, it will not initiate a probe. Neither will B, C, nor D. Therefore, no probes will be initiated and termination will not be detected.



Next we consider the possibility of changing the meaning of the variable, $sent_i$, to mean that a bc has been sent out after the most recent bc was received. The variable $sent_i$ is initially set to false. After every receipt of a bc, $sent_i$ is assigned the value false and after every send of a bc, $sent_i$ is assigned the value true. When a process becomes passive, a probe is initialized only if $sent_i$ is false. Then only processes that *could* be the last active process in the system can initialize probes. The problem here is to guarantee that a probe is initialized every time termination occurs. There are two cases: 1) p_i does not send any bc after receipt of the last bc and 2) p_i does send a bc after receipt of the last bc. In the first case $sent_i$ remains false and a probe is initiated. In the scond case, $sent_i$ is assigned the value true and a probe is not initiated. We must be able to guarantee that there will always be a process that does initiate the final probe. There are several possibilities:

1. After receiving a message from p_j, p_i is required to send a message to p_k as a result of the message from p_j and then p_i goes passive immediately.

2. The work of p_i is interrupted by a bc from p_j. No work is required as a result of the bc from p_j; p_i sends a message to p_k and then immediately goes passive.

3. Same scenario as in case one except that p_i remains active indefinitely.

4. Same scenario as in case two except that p_i remains active indefinitely.

In cases one and two, p_i does not initiate a probe and there is still an active process in the system. If p_k becomes passive without sending a bc, p_k will initiate a probe and termination will be detected. If p_k does send a bc, some other process will be the last active process and will initiate a probe.

Cases three and four present a problem. Because p_i has sent out a bc after receipt of the last bc, p_i will not initiate a probe when it becomes passive. Assume that after receipt of the bc, p_k goes passive and initiates a probe. The probe finds p_i active and is purged. Process p_i eventually goes passive; the application is terminated but it will never be detected.

A solution to this problem is to maintain another variable at each node, $purged_i$. The purpose of this variable is to remember if a probe has been purged by p_i. In cases three and four described above, termination will not be detected if a probe is purged. Note that a purged probe is **only** important if this action has occurred since the receipt of the last bc. Therefore, p_i should initiate a probe if it has not sent out any bc since receipt of the last bc **or** if it has purged a probe since receipt of the last bc. This will cause a few more probes to be initiated, but will also guarantee the detection of termination.

Now consider the second question, should a third round of the probe be initiated if the second round returns falsified? Ideally the answer to this question should be no. We want to limit the number of unnecessary cc. A probe is falsified if between its first and second round, any process has received a bc. In this situation, it is possible for a probe from p_j to never actually encounter an active process and yet to be falsified on the second round. If this is the case, then some process will become passive after p_j and another probe will be initiated. Therefore the probe from p_j may be eliminated after the second round.

Finally, consider the third question, may a single process initiate multiple probes? The answer has to be yes. Consider the following scenario: The probe of p_i is purged by the active process p_k. The chain of events is such that no further probes are initiated and p_i is reactivated. If p_i does not send any bc and it is the last active process in the system, it must be able to initiate a probe when it becomes passive.

Furthermore, only one probe per process may be circulating in the ring at any given time. If p_i has two probes in the system at the same time, they can interfere with each other and cause the detection of a false termination. This may be remedied by requiring each probe to carry a sequence number. If a process ever receives a probe of its own which carries a sequence number less than the current sequence number, the probe is purged. In an effort to reduce the communication overhead, the sequence number and the round number can be represented in a single integer - odd numbers represent round one and even numbers represent round two.

If we are to use this method, the probe must contain the following information:

SeqNbr	An integer, initially 0, that represents both the sequence number and the round number - an odd integer represents round one and an even integer represents round two
flag	This is a boolean which is initialized to true to indicate that no process has been active since the last visit
init	The name of the initiator of the probe

Additional memory at each node is:

bc_i	An array holding n boolean values, all initially false, to indicate whether or not a bc has been received since the last probe received from the process indicated by the index into the array
$sent_i$	A boolean, initially false, that indicates whether or not p_i has *sent* any bc after the receipt of the last bc
$purged_i$	A boolean, initially false, that inidicates whether or not p_i has *purged* any probes after the receipt of the last bc
$seqdm_i$	An integer, initially 0, that indicates the value of the sequence number used on the most recent detection message

When a bc is sent by p_i to p_j the following code is executed:

$sent_i$:= true;

When a bc is received by p_i from p_j the following code is executed:

for $k := 1$ to n do
    $bc_i[k]$:= false;
$sent_i$:= false;
$purged_i$:= false;

When process p_i becomes passive the following piece of code is executed:

if $purged_i$ or $\neg$ $sent_i$ then
    $msg.SeqNbr := seqdm_i + 1$;
    $seqdm_i := seqdm_i + 2$;
    $msg.flag$:= true;
    $msg.init := p_i$;
    send (succ(p_i), *msg*, p_i);

When a process p_j receives a probe initiated by p_i, the following piece of code is executed:

```
if ¬ b_j then
      purge the detection message
      purged_j := true;
else if msg.init ≠ p_j then
      if bc_j[i] ≠ true then
            msg.flag := false;
            send (succ(p_j), msg, p_j);
            bc_j[i] := true;
      else
            send (succ(p_j), msg, p_j);
else if msg.init = p_j then
      if msg.SeqNbr + 2 ≤ seqdm_i then
            purge detection message
      else if Odd(msg.SeqNbr) then
            msg.SeqNbr := msg.SeqNbr + 1;
            msg.flag := true;
            send (succ(p_j), msg, p_j);
      else {Even(msg.SeqNbr)}
            if msg.flag = true then
                  initiate termination
            else
                  purge detection message
```

Using Apt's criteria [APT86], we can show that this algorithm is correct.

1. When termination is declared, all processes will be passive. This is true because any probe that returns to its initiator unfalsified after the second round will have guaranteed that no process is active.

2. The algorithm does not cause deadlock of the computation because it does not interfere with it in any way.

3. When all processes become passive, termination will eventually be declared. The development of the algorithm shows that this is true.

4. If not all processes are passive, then eventually a statement from the original program will execute. The algorithm does not interrupt execution of the applica-

tion and therefore statements from the application itself are free to execute.

The memory overhead for this algorithm is $n+2$ booleans and one integer at each node. This is substantially less than the memory requirements for Arora's algorithm and comparable to Huang's memory overhead if the $n+2$ booleans can be stored in the same amount of memory space as a single integer.

At the send/receive of a bc, this algorithm requires updating of 1 or more booleans as does Arora's algorithm. Eriksen's algorithm requires updating an integer value and Huang's algorithm requires an update to the clock and an acknowledgement of the bc.

The communication overhead for this algorithm in the best case[1] is $2n$. The best case overhead is $2n+m$ for Huang's and $O(n^2)$ for Arora's and Eriksen's algorithms; therefore, there is some savings in the communication overhead. In the worst case[2] the communication overhead has an upperbound of $2n^2+2mn$. This communication overhead is significantly less than the overhead for Arora's algorithm $(2mn^2+mn)$, about equal to the overhead for Eriksen's algorithm $(2n^2+mn)$, and approximately double the overhead for Huang's algorithm (n^2+mn+m).

Overall, the overhead for this algorithm is comparable to the three other algorithms of its type. It has a better best case communication overhead than the other algorithms and the potential for a better worst case overhead as well.

---

1 The best case for this algorithm is when every process sends a bc for every bc received. Then only the last process to become passive will initiate a probe because it does not send out a final bc.

2 Initially all processes but one go passive without sending bc, all remaining messages cause a probe to be initiated and all probes never encounter a process while it is active. Therefore, all probes will go around the ring twice before being purged. The last message will eventually cause the initiation of a probe that detects the termination.

8. CONCLUSIONS

In this paper we have looked first at centralized algorithms and then at distributed algorithms for each type of communication. Initially, it had been this author's assumption that centralized algorithms would be easier to understand and control, but that bottlenecks would occur; therefore distributed algorithms would be more efficient. However, we have seen that distributed algorithms that allow any process to initiate detection of termination have the largest communication overhead and researchers have not been able to reduce this overhead to that of a single token algorithm or a centralized algorithm.

Chapter 3 considered the synchronous distributed case and found three correct algorithms in the literature. All assumed a Hamiltonian ring topology. Only one allowed any process to initiate probes, but it required a larger communication overhead, larger memory, and more additions to the basic communication component. Chapter 5 considered the asynchronous distributed case with a FIFO ordering for message passing. The algorithms of six researchers were reviewed. Three of these algorithms allowed any process to initiate a probe. These three have a communication overhead greater than the overhead of the other algorithms by an order of magnitude. Furthermore, two of these algorithms also had a significantly larger memory overhead. Chapter 6 considered the asynchronous case with a non-FIFO ordering for message passing. In this group there are only three algorithms which allow any process to initiate probes. Again, the communication amd memory overheads are significantly higher. In chapter 7 this author presented a new algorithm that allows any process to initiate a probe. It has an overhead equivalent to the overhead for the chapter 5 algorithms that allow multiple probes. The reader should note that, despite the additional overhead, these algorithms are not able to detect termination after it occurs any more quickly than other algorithms.

The reason for this problem is rooted in Francez's idea of time intervals [FRAN81]. Each probe must verify that no bc have occurred since its last visit. However, if there are multiple probes circulating in the network, the algorithm must guarantee that no one probe changes the information another probe needs to read. This results in the larger memory requirements. A larger problem is that in most algorithms,[1] if all processes initiate a probe whenever they go passive, then there are n probes all gathering the same information, and the result is redundancy.

The goal of researchers in general seems to have been a decentralized algorithm. However, experience has shown that a fully decentralized algorithm is not efficient. Single token algorithms, in which the token is positioned at random by the system initially and termination is declared by any process, seem to be the most efficient distributed algorithms.

In this chapter the author will reach some conclusions as to which algorithm is the best under different conditions. Consider three situations:

1. Synchronous communication is used.
2. Asynchronous communication is used and FIFO ordering of messages is guaranteed.
3. Asynchronous communication is used and FIFO ordering of messages is not guaranteed.

For each of these situations the following cases will be considered:

1 Exceptions are the algorithm of Apt and Richier [APTR85] that kills probes when they are falsified and Arora [AROR89] that allows any process to initiate a probe but only under certain conditions and also allows a process to purge a probe if it is falsified.

1. termination must be detected quickly after it occurs

2. memory overhead must be kept low

3. communication overhead must be kept low

Tables containing the communication and memory overhead for the algorithms may be found at the end of each chapter. In addition we will also consider the size of the probe and the amount of time, t, it takes after termination actually occurs to detect that termination has occurred. We will use the time it takes to send one cc as a measurement for the basic unit of time. In situations where more than one cc may be sent at the same time, the time for these multiple cc will be measured as one unit of time. The size of the probe also affects the time, t; however, this author will not include the size of the probe in the time measurement but will indicate where the probe size adversely affects the time measurement.

8.1. Synchronous Communication

Tables for algorithms using synchronous communication can be found in Section 2.6 and Section 3.5. Additional information can be found in Table 8-1.

In a real-time system, the most important factor is that termination be detected quickly so that the next step may be initiated. If synchronous communication is used, the three best algorithms based only on the time it takes for termination to be detected after it actually occurs, would be those of Francez [FRAN80], Misra and Chandy [MISR82], and Arora and Sharma [AROR83]. However, the reader should remember that Francez actually stops the computation each time a check for termination is made and Misra must send multiple signals for every bc sent. In addition the memory requirements and cc overhead for Misra's algorithm are much higher than those for Arora's algorithm. Although Arora

requires the use of a distance function, the cc and memory overhead for this algorithm are much smaller than for Misra's and Francez's algorithms. Arora's probe must carry an integer and the probes of the other two are empty; however, this author would recommend Arora's algorithm to be the best choice in this situation.

In the case where memory is at a premium, the best algorithms are those that require memory only for booleans: Francez [FRAN80], Topor [TOPO84], and Apt [APT86]. Of the three, Apt has the smallest requirement - only 16 bits of memory for every 8 processes in the system. Again, Francez's algorithm is not a good solution due to its slowdown of the basic computation. Apt's algorithm makes the same addition to the bc overhead and has the same probe size as does Topor's algorithm. However, Apt's cc overhead is smaller than Topor's. For these reasons, Apt's algorithm is recommended by this author when memory requirements are stringent.

When the communication overhead must be low so that an application can complete its work quickly, as in an operating system, the best algorithms are: Francez and Sintzoff [FRAN81], Apt [APT86], and Arora and Sharma [AROR83]. In these algorithms the cc overhead is smaller than all of the other algorithms using synchronous communication and the probes themselves are also small. Arora sends fewer cc messages than Apt; however, each of Arora's messages contains an integer and Apt's only contain a boolean value. In a system where messages have no minimum size restriction, Apt's algorithm actually has a lower cc overhead because of this difference in size. Although, their overheads are the same in every other respect, Francez has a larger memory requirement than Apt. For this reason, this author recommends Apt's algorithm for this situation.

Overall both Apt's [APT86] and Arora and Sharma's [AROR83] algorithms would

make good choices in the synchronous case.

Algorithm	Probe Size	Time after Termination
SPANNING TREE TOPOLOGY		
[FRAN80]	empty	$< n$
[FRAN82]	boolean flag	$< 2n$
[TOPO84]	boolean flag	$< 2n$
DIFFUSING COMPUTATION MODEL		
[MISR82]	empty	$< n$
RING NETWORK TOPOLOGY		
[FRAN81] (centralized)	boolean flag	n
[AROR83] (single probe)	1 integer	$< n$
[APTR85] (multiple probes) (initiated)	2 integers	$< 2n$
[RICH85] (multiple probes) (initiated)	boolean flag 1 integer	n^2
[APT86] (centralized)	boolean flag	n

Table 8-1

8.2. Asynchronous Communication

8.2.1. FIFO Ordering for Messages

Tables for algorithms using asynchronous communication can be found in Section 4.9 and Section 5.6. Additional information may be found in Table 8-2. The reader should notice that the algorithm presented by Skyum et al [SKYU86] is not considered because it slows down the application by causing it to execute synchronously.

Algorithm	Probe Size	Time after Termination †
SPANNING TREE TOPOLOGY		
[AROR88]	1 integer 2 bits	$< n$
[CHAN90]	2 bits	$\leq l$ ‡
DIFFUSING COMPUTATION MODEL		
[ROZO86]	< n integers	$[0, n)$
RING NETWORK TOPOLOGY		
[DIJK83] (centralized)	boolean flag	$[n, 2n)$
[MISR83] (single probe)	1 integer	$[n, 2n)$
[HUAN88] (broadcast) (multiple probes)	2 integers 2 bits	n
[ERIK88] (share info) (multiple probes)	1 integer	$< 2n^2$
[NEWALG] (tokens) (multiple probes)	2 integers 1 bit	$(n, 2n]$
ARBITRARY NETWORK TOPOLOGY		
[MISR83] (single probe)	1 integer	$[c, 2c)$ *
[HUAN88] (broadcast) (multiple probes)	2 integers 2 bits	$[2,2n)$
[ERIK88] (share info) (multiple probes)	1 integer	$< 2n^2$
BIDIRECTIONAL RING TOPOLOGY		
[AROR89] (centralized)	1 integer 2 bits	$(.5n, 1.5n)$
[AROR89] (shifting initiator) (of single probes)	1 integer 1 bit	$(.5n, n+2)$
[AROR89] (multiple probes)	1 integer 3 bits	$[n, 2n]$

Table 8-2

† Time is measured by computing the number of cc messages sent since hops take 1 unit of time. In situations where multiple messages can be sent at the same time instant, these are counted as 1 unit of time.

‡ l represents the number of levels in the tree and is less than n.

* c represents the length of the cycle in the network.

Looking just at the time to detect termination after it occurs, the best algorithms for detecting termination quickly are presented by Arora and Gupta [AROR88], Chandrasekaran and Venkatesan [CHAN90], Rozoy [ROZO86], and Huang [HUAN88]. Arora, Huang, and Rozoy all take about the same amount of time to detect termination after it occurs. However, it should also be noted that the algorithm presented by Rozoy uses a probe which can carry as many as n integers. The size of this probe slows down the communication and therefore, this could be the slowest of these three algorithms. In addition the reader should notice from Table 4-1 that the overhead is tied to the number of bc sent during the application.

Arora's algorithm slows down the computation at the receipt of bc by requiring the send of at least one cc (an ack and possibly 'I-am-up-again' messages). Huang's algorithm also requires an ack for every bc sent. This leaves Chandrasekaran's algorithm as the best choice: it takes a minimal amount of time to detect termination after it occurs, the probe size is very small, and there are no cc sent as a result of bc. The memory requirement is quite large, but this is the trade-off for a quick detection of termination.

In the situation where memory is at a premium the best algorithms are: Misra [MISR83], and Dijkstra and van Gasteren [DIJK83]. Both of these algorithms require only n booleans and both assume a ring network topology. These two algorithms have the same overhead in all cases but one: Dijkstra's probe size is a boolean value, Misra's is an integer value. Therefore Dijkstra's algorithm would run somewhat faster than Misra's. It is

interesting to note that Dijkstra's algorithm is a centralized version of Misra's single token algorithm.

The last situation considered is when the communication overhead must be low so that the application can complete its work quickly. The best algorithms for this case are: Chandrasekaran and Venkatesan [CHAN90], Dijkstra and van Gasteren [DIJK83], Arora and Gupta [AROR89], and Misra [MISRA83]. These four are chosen initially because their best case communication overhead is $O(n)$ and their worst case communication overhead is $O(n+m)$. The best case communication overheads of the others are either $O(n^2)$ or $O(n+m)$. Both Arora's centralized and single token algorithms require the probe to carry an integer and a bit flag. Misra's algorithm requires the probe to carry an integer. Dijkstra's and Chandrasekaran's algorithms have the smallest probes - a max of 2 bits. As the number of bc increases, Dijkstra's algorithm's worst case overhead increases at a much greater rate than Chandrasekaran's. Therefore, this author recommends Chandrasekaran's algorithm when communication overhead must be low.

It is interesting to note that in a ring topology, a centralized versus a distributed system makes no difference in the overhead - a simple algorithm may be found for each that requires little overhead in all areas: Dijkstra's centralized algorithm [DIJK83] and Misra's distributed version [MISR83]. Arora's single token algorithm [AROR89] has approximately the same overhead as well. All of the other algorithms for ring topologies are substantially more expensive in both communication overhead and memory overhead. This situation is definitely an example of 'simple is better'.

It should be noted that Chandrasekaran's algorithm assumes a spanning tree topology and yet it has a lower overhead in all areas except memory overhead. It appears that the

topology of the network in these two cases has less effect on the overhead than the quality of the algorithm itself. On the other hand, arbitrary networks appear to require a substantially higher communication overhead.

8.2.2. Non-FIFO Ordering for Messages

A table for algorithms using asynchronous communication with non-FIFO message ordering can be found in Section 6.4. Additional information can be found in Table 8-3.

When looking at a system which requires prompt detection of termination after it occurs, the best choices would be Dijkstra and Scholten's algorithm [DIJK80], Kumar's class 1 algorithm [KUMA85], Mattern's time algorithm [MATT87B], and Mattern's vector algorithm [MATT87A]. All four of these algorithms can detect termination in less than n time units, assuming a Hamiltonian ring topology. However Kumar's class 1 algorithm and Matern's vector algorithm require a probe whose length is based on the number of channels in the network and the number of processes in the network respectively. In reality these two algorithms are not good choices, since the size of the probe will significantly increase detection time. Mattern's time algorithm has the potential for a very high communication overhead since there are no limitations on the number of times a probe is reinitiated. Therefore this author believes Dijkstra's algorithm to be the best choice when prompt detection of termination is required because 1) the probe carries no information, 2) there is a short code segment at the receipt of cc, and 3) the total number of cc is reasonable.

Algorithm	Probe Size	Time after Termination
DIFFUSING COMPUTATION MODEL		
[DIJK80]	empty	$\leq l$ †
RING NETWORK TOPOLOGY		
[KUMA85]		
class 1 (single token)	E ††	$[0, \lvert c \rvert)$ ‡
class 2 (single token)	2 integers	$< 2 * \lvert c \rvert$ ‡
[MAT87B] ‡‡		
4 ctrs *	2 integers	$(n, 2n]$
sceptic *	2 integers	$(n, 2n]$
time (multiple tokens)	3 integers 1 boolean	$< n$
vectors (single token)	n integers	$[1, n]$

Table 8-3

† l represents the number of levels in the tree and is less than n.

†† E is the total number of communications lines in the network.

‡ c represents the length of the cycle in the network.

‡‡ All values are based on the existence of a Hamiltonian ring.

* There may be multiple or single tokens for the four counter and sceptic algorithms; however, the size of the probe and the time after termination are not effected by the number of tokens. The single token version has a pre-defined initiator and is therefore a centralized version of the algorithm.

In the case where memory is at a premium, several of the algorithms require storage in memory for $O(n^2)$ integers or $O(n^2)$ booleans, and Dijkstra's algorithm requires memory storage for O(n+m) integers. When these algorithms are eliminated the remaining algorithms are Kumar's class 2 algorithm [KUMA85], and Mattern's four counter algorithm, single token sceptic algorithm, and time algorithm [MATT87B]. Of these four, Mattern's four counter, sceptic, and time algorithms have no upper bound on their worst

case overhead because there is no control over how often probes are reinitiated after they are falsified. If the network has a Hamiltonian ring topology, this author would chose Kumar's class 2 algorithm as the best choice.

The final situation considered is when communication overhead must be as small as possible so that the application may not be detained more than necessary. Algorithms making use of a large probe are eliminated first. These include Kumar's class 1 algorithm [KUMA85], and Mattern's vector algorithm [MATT87A]. Mattern's four counter, sceptic, and time algorithms [MATT87B] can be eliminated because of their lack of control over the initiation of probes. This leaves Dijkstra's algorithm [DIJK80] and Kumar's class 2 algorithm [KUMA85] as candidates for the algorithm of choice when communication overhead must be low.

When the number of bc, m, is less than the number of processes in the system, Dijkstra's algorithm is clearly the best choice; however, this will not usually be the case. Therefore, this author's first choice is Kumar's algorithm.

Kumar's class 2 algorithm is a best choice when the application requries a minimum amount of memory and a low communication overhead. If an algorithm is required which would be best for a general application with no special requirements, this author would endorse Kumar's class 2 algorithm. Even the time required to detect termination after it occurs is greater than the best case only by a factor[2] of less than $2 \log_2 n$.

2 In the best case Dijkstra's algorithm requires $\log_2$ n of time. When the spanning tree is not built efficiently Dijkstra's time approaches n and the constant factor becomes 2.

8.3. Summary

In this chapter, nine cases were analyzed, three for each kind of communication. In the synchronous case a centralized algorithm is the best case in two out of three situations. In the asynchronous FIFO message ordering case, all three best choice algorithms are centralized algorithms. In the asynchronous non-FIFO message ordering case, one of the three best choices is also a centralized algorithm. In the remaining three situations a single token algorithm is the best choice. There are no multiple probe algorithms which can even come close to the efficiency of these best choice algorithms!

During the process of analyzing each algorithm, it seemed that ring topologies were the easiest to understand and also the most efficient. Yet in the recommended best choices, we find a spanning tree topology is the best choice in 2 out of 3 situations for the asynchronous FIFO message ordering case and in one out of 3 situations in the asynchronous non-FIFO message ordering case. It appears that if the spanning tree is built efficiently, an algorithm which assumes this topology can be as efficient as an algorithm based on a Hamiltonian ring topology, especially when the time to detect termination after it occurs is the most important factor.

Actually a Hamiltonian ring topology may not be as efficient as it appears to be if it is a virtual rather than an actual ring. If there is no physical Hamiltonian ring, the establishment of the virtual ring may require neighbor to neighbor communication where the time complexity may become very high. The worst case will be bounded by the size of the network itself. A best case communication overhead of 2n would actually be $2n^2$. Note, this is the same as the best case communication overhead for several of the algorithms that assume arbitrary networks and have also been written for ring networks [SKYU86]

[ERIK88]. Consider the following three network topologies: a star network,[3] a tree network, and a mesh network.[4] In the case of a star network, construction of a Hamiltonian ring on the network will result in 2 cc for every 1 cc passed along the ring. Therefore, a communication overhead of $2n$ will actually be $4n$. If an equivalent algorithm for an arbitrary network requiring $3c$ cc is used, the actual overhad is $3(2(n-1))$ or $6n$.

To impose a Hamiltonian ring on a network that is a well-built tree, the longest hop is the diameter of the tree and only occurs once. This number will be some constant less than n. Therefore a virtual Hamiltonian ring with a communication overhead of $2n$ will have an actual overhead of less than $2n^2$. The most inefficient tree is a linear graph. The diameter in this case is $n-1$, and a hop of this length will occur only once. All other hops will be one. Therefore given an overhead of $2n$, the actual overhead will be less than $3n$.

To impose a Hamiltonian ring on a mesh network, the longest hop will be the number of nodes in a row plus the number of nodes in a column, the diameter of the network. This hop will occur at most once and all other connections will be a single hop. Therefore if the communication overhead is $2n$, for the virtual ring, the actual overhead will be $2(n-1+d)$ which will always be less than $4n$. If the equivalent algorithm for an arbitrary network requires $3c$ cc, the actual overhad is $3(2(n-1+d))$ or less than $12n$.

These values indicate that unless the network is a well-built tree, it is more efficient to impose a Hamiltonian ring on any network, rather than to use an algorithm that assumes an arbitrary network.

---

3 All nodes, p_i, i = 1 to n-1, are connected **only** to one other node, p_n, and p_n is connected to every node in the system.

4 In a mesh network, the nodes are arranged in rows, all rows have the same number of nodes, and all columns have the same number of nodes. The number of columns need not equal the number of rows. Every node is connected to all of its horizontal and vertical neighbors, but not its diagonal neighbors.

8.4. Future Directions

Future research in this area should include the implementation of the best of these algorithms. Benchmark applications should be chosen and actual time comparisons made.

Furthermore, very little is written about algorithms suitable for broadcast networks. Likewise it would be interesting to know if the overhead for algorithms assuming a broadcast network could be reduced sufficiently to compete with algorithms run on tree networks or Hamiltonian ring networks.

One area in which research has already begun, but which this paper did not cover, is termination algorithms for dynamic networks. It is assumed that nodes would not be added/deleted to/from the network, but rather processes would be created/deleted at any of the existing nodes. It would be interesting to discover if a particular topology is more efficient than other topologies in this situation.

Another area in which future research should occur is the removal of all restrictions on the channels. Duplicate messages could be handled using sequence numbers. However, lost and/or altered cc could be a real problem. It would be worthwhile to determine whether or not an algorithm that is both simple and efficient can be found.

Up until this point, for the most part, termination algorithms have been generic. Although there have been a few algorithms written with a particular type of application in mind, in the future this author would expect to see many more algorithms developed for specific kinds of applications. In this way algorithms may be more efficient for their intended purpose. An example of this would be the algorithms written by Szymanski et al [SZYM85] and Skyum and Eriksen [SKYU86] for an application that requires synchroneity, but runs on an asynchronous communication network. Rozoy also hints at the possibility

of tailoring her algorithm to specific situations.

It is not possible to pick one termination algorithm that is suitable for all situations. This author believes that the best algorithms will be modified to take into consideration any idiosyncrasies of the application that would make the algorithm run more quickly. In particular, careful attention will be paid to the network topology on which the application will run and the process hierarchy imposed by the application. In time there will be a core of proven algorithms for a variety of situations, but this will require substantial benchmark testing.

BIBLIOGRAPHY

[APTR85] Apt, Krzysztof R. and Richier, Jean-Luc, "Real Time Clocks Versus Virtual Clocks." in Broy, M., ed. *Control Flow and Data Flow: Concepts of Distributed Programming*, Berlin: Springer-Verlag, 1985, pp. 475-501.

In this article an algorithm is developed for a synchronous distributed system using time clocks. A clear explanation is given for use of a real-time clock and the reasoning behind the changes leading to a local virtual clock. A ring cc topology is assumed.

[APT86] Apt, Krzysztof R., "Correctness Proofs of Distributed Termination Algorithms." *ACM TOPLAS,* Vol. 8, No. 3, July 1986, pp. 388-405.

The main purpose of this paper is to introduce a method for doing correctness proofs on CSP programs that are in normal form. The criteria for a correct solution and a very simple synchronous centralized algorithm are introduced for the sole purpose of showing how a correctness proof is done.

[AROR83] Arora, R.K. and Sharma, N.K., "A Methodology to Solve Distributed Termination Problem." *Information Systems,* Vol. 8, No. 1, 1983, pp. 37-39.

An algorithm is developed for a synchronous distributed system using a ring topology and tokens. Basic concepts in this algorithm are also used by AROR87 and HALD88

[AROR86] Arora, R.K., Rana, S.P., and Gupta, M.N., "Distributed Termination Detection Algorithm for Distributed Computations." *Information Processing Letters,* Vol. 22, No. 6, May 1986, pp. 311-314.

This is an algorithm for the asynchronous distributed case that is incorrect. Explanation is given in [TAN86]

[AROR87] Arora, R.K., Rana, S.P., and Gupta, M.N., "Ring Based Termination Detection Algorithm for Distributed Computations." *Microprocessing and Microprogramming,* Vol. 19, No. 3, June 1987, pp. 219-226.

This is an asynchronous distributed solution for a network that is a Hamiltonian ring. This author believes the algorithm to be incorrect. Haldar [HALD88] bases his algorithm on some of the concepts presented here.

[AROR88] Arora, R.K. and Gupta, M.N., "An Algorithm for Solving Distributed Termination Problem." *Microprocessing and Microprogramming,* Vol. 22, No. 4, October 1988, pp. 263-271.

This is an overly complicated algorithm for the asynchronous centralized case that uses a spanning tree topology.

[AROR89] Arora, R.K. and Gupta, M.N., "Bidirectional Ring-Based Termination Detection Algorithms for Distributed Computations." *IEE Proceedings Part E, Computers and Digital Techniques,* Vol. 136, No. 5, September 1989, pp 415-422.

The authors present several asynchronous algorithms for a bidirectional ring, one for the centralized case and two for the distributed case.

[AROG88] Arora, R.K. and Gupta, M.N., "More Comments on 'Distributed Termination Detection Algorithm for Distributed Computations'." *Information Processing Letters,* Vol. 29, No. 1, September 1988, pp. 53-55.

This is a letter to the editor which disputes the claim of Tan et al [TAN86] that the algorithm presented by Arora and Gupta [AROR86] is incorrect.

[AROG89] Arora, R.K. and Gupta, M.N., "A Generalized Framework for Deriving Ring Based Termination Detection Algorithms." *Journal of Microcomputer Applications,* Vol. 12, No. 2, April 1989, pp 147-157.

A reader new to this area of research may find this article helpful; however, a more advanced reader would probably find it simplistic.

[CHAN87] Chandy, K. Mani, "A Theorem on Termination of Distributed Systems." Tech. Rept. No. 87-09, Department of Computer Sciences, The University of Texas at Austin, 1987.

A theorem and its proof is presented to illustrate the principles involved in the asynchronous distributed termination situation. The article is interesting because Chandy makes use of distributed snapshots from which he proves his theorem.

[CHAN90] Chandrasekaran, S. and Venkatesan, S., "A Message-Optimal Algorithm for Distributed Termination Detection." *Journal of Parallel and Distributed Computing,* Vol. 8, No. 3, March 1990, pp. 245-252.

This algorithm is for an aynchronous network and is based on Topor's algorithm [TOPO84].

[DIJK80] Dijkstra, Edsger W. and Scholten, C.S., "Termination Detection for Diffusing Computations." *Information Processing Letters,* Vol. 11, No. 1, August 1980, pp. 1-4.

This is one of the earliest algorithms. It offers a simple but elegant solution for an asynchronous centralized system. Misra's algorithm [MISR82] is based on this work.

[DIJK83] Dijkstra, Edsger W., Feijen, W.H.J. and van Gasteren, A.J.M., "Derivation of a Termination Detection Algorithm for Distributed Computations." *Information Processing Letters,* Vol. 16, No. 5, June 1983, pp. 217-219.

A generic centralized version of the colored token algorithm is presented here. The algorithm may be implemented with either synchronous or asynchronous communication. Topor borrows the concept for use in his algorithm.

[ERIK88] Eriksen, Ole, "A Termination Detection Protocol and Its Formal Verification." *Journal of Parallel and Distributed Computing,* Vol. 5, No. 1, February 1988, pp. 82-91.

This paper offers an asynchronous distributed termination algorithm for an arbitrary network topology with directed communication lines, for the class of undirected graphs. The rules for the algorithm are formalized and then used to prove the theorem true using graph theory.

[FERM87] Ferment, D. and Rozoy, B., "Solutions for the Distributed Termination Problem." *Proceedings of International Workshop on Parallel Algorithms and Architectures,* (*Lecture Notes in Computer Science* 269), Suhl, GDR, May 1987, pp. 114-121.

This article discusses the criteria for the ability to find a finite distributed termination algorithm when the communication is asynchronous. The authors suggest that either communication must be instantaneous or else the process must be able to test for the emptiness of its buffers.

[FRAN80] Francez, Nissim, "Distributed Termination." *ACM TOPLAS,* Vol. 2, No. 1, January 1980, pp. 42-55.

The algorithm presented here is the earliest published attempt to solve the distributed termination problem. It is not a good algorithm; however, Francez defines the distributed termination problem along with the reasoning behind his definition. Every one else assumes or copies this definition.

[FRAN81] Francez, N., Rodeh, M., Sintzoff, M., "Distributed Termination with Interval Assertions." *Proceedings of Formalization of Programming Concepts, (Lecture Notes in Computer Science* 107), Peniscola, Spain, April 1981, 1981, pp. 280-291.

The concept of interval assertions is described here. While the term is not used by other researchers, the concept is a given in all other algorithms, especially when a clock is used.

[FRAN82] Francez, Nissim and Rodeh, Michael, "Achieving Distributed Termination without Freezing." *IEEE Transactions on Software Engineering,* Vol. SE-8, No. 3, May 1982, pp. 287-292.

This algorithm is for a synchronous centralized system. The programmer must derive a spanning tree for the topology. It is an improvement on [FRAN80], but is an indication that a spanning tree is not the best topology for efficiency.

[HALD88] Haldar, S. and Subramanian, D.K., "Ring Based Termination Detection Algorithm for Distributed Computations." *Information Processing Letters,* Vol. 29, No. 3, October 1988, pp. 149-153.

This algorithm is for the asynchronous distributed case and makes use of concepts developed by Arora [AROR87] and [AROR83]. Tel and Mattern [TELM89] find this algorithm to be incorrect.

[HAZA87] Hazari, Cyrus and Zedan, Hussein, "A Distributed Algorithm for Distributed Termination." *Information Processing Letters,* Vol. 24, No. 5, March 1987, pp. 293-297.

This is an algorithm for a synchronous distributed system that is incorrect as explained in [TEL87]

[HOAR78] Hoare, C.A.R., "Communicating Sequential Processes." *Communications of the ACM,* Vol. 21, NO. 8, August 1978, pp. 666-677.

This is an introduction to the language CSP and should be read before attempting to understand any of the algorithms which assume a synchronous system.

[HUAN88] Huang, Shing-Tsaan, "A Fully Distributed Termination Detection Scheme." *Information Processing Letters,* Vol. 29, No. 1, September 1988, pp. 13-18.

This author borrows the invariant technique used by Dijkstra [DIJK83] and Topor [TOPO84]. No assumptions are made about the topology, the communication is asynchronous, and the code is fully

distributed.

[KUMA85] Kumar, Devendra, "A Class of Termination Detection Algorithms for Distributed Computations." *Proceedings Fifth Conference Foundations of Software Technology and Computer Science, (Lecture Notes in Computer Science* 206), New Delhi, India, December 1985, pp. 73-100.

Kumar presents three classes of algorithm to solve the distributed termination problem in the asynchronous case where message order is not fifo. The article is written in such a way that the reader may understand why choices are made and why the variations presented are correct or incorrect.

[LAMP78] Lamport, L., "Time, Clocks and the Ordering of Events in a Distributed System." *Communications of the ACM,* Vol. 21, No. 7, July 1978, pp. 558-565.

This is an introduction to the use of clocks in a distributed system. It would be helpful to understand this material before attempting to understand algorithms which assume a clock.

[MAT87A] Mattern, Friedemann, "Experience with a New Distributed Termination Detection Algorithm." *Proceedings 2nd International Workshop Distributed Algorithms, (Lecture Notes in Computer Science* 312), Amsterdam, The Netherlands, July 1987, pp. 127-143.

In this article Mattern presents a single termination algorithm for the asynchronous distributed case where message order is not fifo. The size of the probe is dependent upon the system size.

[MAT87B] Mattern, Friedemann, "Algorithms for Distributed Termination Detection." *Distributed Computing,* Vol. 2, No. 3, December 1987, pp. 161-175.

Mattern presents five different algorithms for the asynchronous distributed case where message order is not fifo. This article contains an extensive bibliography.

[MISR82] Misra, Jayadev and Chandy, K.M., "Termination Detection of Diffusing Computations in Communicating Sequential Processes." *ACM TOPLAS,* Vol. 4, No. 1, January 1982, pp. 37-43.

This algorithm is an adaptation of Dijkstra's diffusing computation [DIJK80] using asynchronous communication to the synchronous case. The reader is advised to read and understand Dijkstra's algorithm before trying to understand Misra's.

[MISR83] Misra, Jayadev, "Detecting Termination of Distributed Computations Using Markers." *Proceedings Second Annual ACM Symposium on Principles of Distributed Computing,* Montreal, August 1983, pp. 290-294.

An asynchronous distributed version of the colored token algorithm is presented here for arbitrary networks.

[RANA83] Rana, S.P., "A Distributed Solution of the Distributed Termination Problem." *Information Processing Letters,* Vol. 17, No. 1, July 1983, pp. 43-46.

This algorithm is for a synchronous distributed system using logical clocks. It is incorrect. Apt and Richier [APTR85] provide information to aid in understanding where the error occurs.

[RAY88a] Raynel, Michel, "Algorithms for Detecting Termination." in *Distributed Algorithms and Protocols*, pp. 67-88, New York: John Wiley & Sons, 1988.

This book includes a chapter on distributed termination algorithms. It is clearer than many of the articles and may be a good place to start reading.

[RAY88b] Raynel, Michel, *Networks and Distributed Computation: Concepts, Tools, and Algorithms*, Cambridge Ma.: MIT Press, 1988.

This book describes many of the idiosyncrasies of a distributed system and some tools which can be used to solve problems.

[RICH85] Richier, Jean-Luc, "Distributed Termination in CSP." *2nd Annual Symposium on Theoretical Aspects of Computer Science, (Lecture Notes in Computer Science* 182), Saarbrucken, January 1985, pp. 267-278.

This difficult algorithm to understand is for the synchronous distributed case. It turns out to be incorrect.

[ROZO86] Rozoy, Brigitte, "Model and Complexity of Termination for Distributed Computations." *Proceedings Mathematical Foundations of Computer Science 1986, (Lecture Notes in Computer Science* 233), Bratislava, Czechoslovakia, August 1986, pp. 564-572.

The purpose of this paper is to develop two theorems which describe the bounds on the number of cc required to detect termination in an asynchronous communication system. The theorems are not proven rigorously. The distributed system is described as a finite state machine; however, there are pieces of missing information.

[SKYU86] Skyum, Sven and Eriksen, Ole, "Symmetric Distributed Termination." in Rozenberg, G. and Salomaa, A., eds., *The Book of L*, New York: Springer-Verlag, 1986, pp. 427-430.

This article presents an algorithm for the asynchronous distributed case. The algorithm makes the network configuration work synchronously to some extent, though not to the degree of Szymanski's [SZYM85] algorithm.

[SZYM85] Szymanski, B., Shi, Y., and Prywes, N.S., "Synchronized Distributed Termination." *IEEE Transactions on Software Engineering,* Vol. SE-11, No. 10, October 1985, pp. 1136-1140.

The algorithm presented here is motivated by the use of a network of distributed processes to solve a large set of simultaneous equations. The system is asynchronous and fully distributed; however, the main computation has properties which have some degree of synchroneity. This is actually a variation of the distributed termination problem.

[TAN86] Tan, R.B., Tel, G., and Van Leeuwen, J., "Comments on 'Distributed Termination Detection Algorithm for Distributed Computations'." *Information Processing Letters,* Vol. 23, No. 3, October 1986, pp. 163.

This is a letter to the editor which points out why Arora's algorithm [AROR86] is incorrect.

[TEL87] Tel, G. and Van Leeuwen, J., "Comments on 'A Distributed Algorithm for Distributed Termination'." *Information Processing Letters,* Vol. 25, No. 5, July 1987, pp. 349.

This is a letter to the editor which points out why Hazari's algorithm [HAZA87] is incorrect.

[TELM89] Tel, G. and Mattern, F., "Comments on 'Ring Based Termination Detection Algorithm for Distributed Computations'." *Information Processing Letters,* Vol. 31, No. 3, May 1989, pp. 127-128.

This is an article which points out why Haldar and Subramanian's algorithm [HALD88] is incorrect.

[TOPO84] Topor, Rodney W., "Termination Detection for Distributed Computations." *Information Processing Letters,* Vol. 18, No. 1, January 1984, pp. 33-36.

This is a synchronous centralized version of the colored token algorithm, which uses a spanning tree topology. It works the same as Francez's algorithm [FRAN82], but the actual implementation uses the concept of colored tokens from Dijkstra [DIJK83].

ADDITIONAL REFERENCES

This paper did not refer directly to the following references; however, the reader may find them useful for additional information about the subject area.

[AFES87] Afek, Yehuda and Saks, Michael, "Detecting Global Termination Conditions in the Face of Uncertainty." *Proceedings Sixth Annual ACM Symposium on Principals of Distributed Computing,* Vancouver, British Colombia, Canada, August 1987, pp. 109-124.

Afek and Saks present the *Token Collection Problem* for distributed networks in order to model fault-tolerant termination detection.

[CHAN85] Chandy, K.M. and Lamport, Leslie, "Distributed Snapshots: Determining Global States of Distributes Systems." *ACM Transactions on Computer Systems,* Vol. 3, No. 1, February 1985, pp. 63-75.

This paper presents a solution to the generic problem of determining the global state of a distributed system during a computation.

[COHL82] Cohen, Shimon and Lehmann, Daniel, "Dynamic Systems and Their Distributed Termination." *Proceedings ACM Sigact-Sigops Symposium on Principles of Distributed Computing,* Ottawa, Canada, August 1982, pp 29-33.

This paper describes a model for dynamic synchronous distributed systems, where new processes are added and terminated at execution time.

[HUAN89] Huang, Shing-Tsaan, "Detecting Termination of Distributed Computations by External Agents." *9th International Conference on Distributed Computing Systems* Newport Beach, California, June 1989, pp.79-83.

This paper presents an algorithm for detecting termination of distributed computations by an auxiliary controlling agent.

[KOO87] Koo, Richard and Toueg, Sam, "Effects of Message Loss on Distributed Termination." Tech. Rept. No. 87-823, Department of Computer Science, Cornell University, March 1987.

This report is a study of the problem of termination in distributed systems with faulty communication channels. It concludes that message loss precludes the guarantee of a terminating termination detection algorithm.

[LAI86A] Lai, Ten-Hwang, "A Termination Detector for Static and Dynamic Distributed Systems with Asynchronous Non-first-in-first-out Communication." *Proceedings 13th International Colloquium on Automata, Languages and Programming,* (*Lecture Notes in Computer Science* 226), Rennes, France, July 1986, pp. 196-205.

Lai presents an algorithm that may be used in static or dynamic systems, with synchronous or asynchronous communication and message order is either fifo or not.

[LAI86B] Lai, Ten-Hwang, "Termination Detection for Dynamically Distributed Systems with Non-first-in-first-out Communication." *Journal of Parallel Distributed Computing,* Vol. 3, No. 4, December 1986, pp. 577-599.

Lai presents a snapshot-based algorithm for the general asynchronous and dynamic model using time-stamped messages.

[LOZI85] Lozinskii, Elizer, "A Remark on Distributed Termination." *IEEE Fifth International Conference on Distributed Computing Systems,* Denver, Colorado, May 1985, pp 416-419.

An algorithm for the detection of termination in a dynamic distributed system is presented here.

[MATT89] Mattern, Friedemann, "An Efficient Distributed Termination Test." *Information Processing Letters,* Vol. 31, No. 4, May 1989, pp. 203-208.

Mattern presents a termination test that is based on Dijkstra's algorithm [DIJK83] and assumes a synchronous distributed system.

[ROKU88] Rokusawa, K., Ichiyoshi, N., Chikayama, T., and Nakashima, H., "An Efficient Termination Detection and Abortion Algorithm for Distributed Processing Systems." *International Conference on Parallel Processing,* University Park, Pa., August 1988, pp 18-22.

This paper describes an algorithm for termination detection and abortion in distributed processing systems, where processes may migrate. Message order is not necessarily fifo.

[ROZO87] Rozoy, Brigitte, "Termination for Distributed Systems with Asynchronous Message Passing: Model and Cost." Tech. Rept. 87.20, University of Paris, April 1987.

Rozoy offers several theorems and proofs that her algorithm will work in an asynchronous system. The algorithm assumes a diffusing computation, the possibility for lost messages, and non-fifo message order .

[SAND87] Sanders, Beverly, "A Method for the Construction of Probe-Based Termination Detection Algorithms." *Proceedings IFIP Conference on Distributed Processing,* Amsterdam, the Netherlands, October 1987, pp.249-257.

This paper describes the construction of a framework for probe-based termination detection algorithms and presents several algorithms in a general fashion.

[SHAV86] Shavit, Nir and Francez, Nissim, "A New Approach to Detection of Locally Indicative Stability." *Proceedings 13th International Colloquium on Automata, Languages and Programming,* (*Lecture Notes in Computer Science* 226), Rennes, France, July 1986, pp. 344-358.

This paper presents an algorithm to detect stable properties in a distributed system. The concept is interesting although not necessarily any more efficient than other algorithms.

APPENDIX

The algorithms contained in this appendix have been ordered chronologically. Many of the variable names have been changed from the names provided in the original algorithms. This is an attempt to make the algorithms easier to understand for the reader.

The reader will notice that the algorithms which make use of synchronous communication are coded using the CSP notation. Algorithms using asynchronous communication are coded using Pascal-like syntax wherever possible.

The syntax of send and receive is as follows:

send(dest, msg, source)

receive(dest, msg, source)

The first parameter names the process that the message is being sent to, the second parameter is a record containing all of the detection information that the algorithm wishes to pass on, and the third parameter is the name of the sender of the message.

Consider succ(p_i) to be the next process on the Hamiltonian circuit, pred(p_i) to be the previous process in the ring, and next(p_i) to be the successor of p_i in the direction the cc is traveling when the direction is not known. The symbol <: is used to define a relation among processes. In a Hamiltonian ring, $p_i <: p_j$ means that p_j is ahead of p_i. The statement 'purge detection message' means the detection message is not passed on, it is deleted.

In a network of arbitrary topology, a send to all processes connected to p_i is performed by the following statement:

broadcast (message)

where message is a record containing all information that the control code requires.

The following statements are borrowed from Skyum and Eriksen [SKYU86] and used to indicate the atomicity of the code contained between them:

enable allows interrupts to occur for the reception of bc

disable prevents interrupts from occurring for the reception of bc

BNF FOR CSP

COMMANDS

<command> ::= <simple command> | <structured command>

<simple command> ::= <null command> | <assignment command> | <input command> | <output command>

<structuredcommand> ::= <alternative command> | <repetitive command> | <parallel command>

<null command> ::= skip

<command list> ::= {<declaration>; | <command>;} <command>

PARALLEL COMMANDS

<parallel command> ::= [<process> {|| <process>}]

<process> ::= <process lable> <command list>

<process label> ::= <empty> | <identifier> :: | <identifier> (<label subscript>, {, <label subscript>})::

<label subscript> ::= <integer constant> | <range>

<integer constant> ::= <numeral> | <bound variable>

<bound variable> ::= <identifier>

<range> ::= <bound variable> : <lower bound> .. <upper bound>

<lower bound> ::= <integer constant>

<upper bound> ::= <integer constant>

ASSIGNMENT COMMANDS

<assignment command> ::= <target variable> := <expr>

<expr> ::= <simple expr> | <structured expr>

<structured expr> ::= <constructor> (<expr list>)

<constructor> ::= <identifier> | <empty>

<expr list> ::= <empty> | <expr> {, <expr>}

<target var> ::= <simple var> | <structured target>

<structured target> ::= <constructor> (<target var list>)

<target var list> ::= <empty> | <target var> {, <target var>}

INPUT AND OUTPUT COMMANDS

<input com> ::= <source> ? <target var>

<output com> ::= <destination> ! <expr>

<source> ::= <process name>

<destination> ::= <process name>

<process name> ::= <identifier> | <identifier> (<subscripts>)

<subscripts> ::= <integer expr> {, <integer expr>}

ALTERNATIVE AND REPETITIVE COMMANDS

<repetitive command> ::= *<alternative command>

<alternative command> ::= [<guarded command> {□ <guarded command>}]

<guardedcommand> ::= <guard> → <command list> | (<range> {,<range>}) <guard> → <command list>

<guard> ::= <guard list> | <guard list>; <input command> | <input command>

<guard list> ::= <guard element> {; <guard element>}

<guard element> ::= ,boolean expr> | <declaration>

The curly braces, {}, are used to denote none or more repetitions of the enclosed material.

This algorithm uses the following variables:

p(i)	the index of p_i's parent
Γ_i	the set of indices of p_i's children
cm	true implies that bc may occur false implies that bc is frozen and may not occur
advanced	true implies that a bc has occurred - not needed by the root false implies that bc has not occurred since the last control wave
newwave	a boolean to record the arrival of all *ready(j)* messages - only used by the root
ready	a single boolean value at each leaf to insure that a ready signal to the parent is sent only once per control cycle
ready(i)	true if either p_i has performed a bc since the last control cycle or b_i has been met **and** permission has been received from all of the children of p_i to initiate a new control cycle, i.e. all children, p_j, have performed at least one bc since the last control cycle or have met their b_j
a(i)	true implies that all children, p_j, of p_i have met their b_j false implies that at least one child, p_j of p_i has not met its b_j

Control code for the root:

initially newwave is true, and for all j, ready (j) is false

c_i :: b_i; newwave $\rightarrow$ $\prod_{j\in\Gamma_i} p_j$! **ok**;

$\prod_{j\in\Gamma_i} p_i$? a(j);

r : $\bigwedge_{j\in\Gamma_i} a(j)$;

[

r $\rightarrow$ halt

$\square$ $\neg$r $\rightarrow$ newwave := false;

$\prod_{j\in\Gamma_i} p_j$! resume

]

$\square_{j\in\Gamma_i} p_j$? ready (j) $\rightarrow$

[$\bigwedge_{j\in\Gamma_i}$ ready(j) $\rightarrow$ newwave := true;

$\prod_{j\in\Gamma_i} ready(j)$:= false

$\square \sim \bigwedge_{j\in\Gamma_i} ready(j) \rightarrow$ skip

]

Control code for an intermediate node:

initially, cm = true, for all j, ready(j) = false, advanced = false

c_i :: $p_{p(i)}$? **ok** → cm := false;
[$\neg b_i \rightarrow p_{p(i)}$! false
□ $b_i \rightarrow \prod_{j \in \Gamma_i} p_j$! **ok**;
$\prod_{j \in \Gamma_i} p_j$? a(j);
r : $\bigwedge_{j \in \Gamma_i} a(j)$;
$p_{p(i)}$! r
]
□
$p_{p(i)}$? **resume** → cm := true;
advanced := false;
$\prod_{j \in \Gamma} p_j$! **resume**;
□
□ $_{j \in \Gamma_i} p_j$? ready(j) → skip
□
($b_i \vee$ advanced) $\wedge \bigwedge_{j \in \Gamma_i}$ ready(j);
$p_{p(i)}$! true → $\prod_{j \in \Gamma_i}$ ready(j) := false

Control code for a leaf:

initially, cm = true, ready = false

c_i :: $p_{p(i)}$? **ok** → cm := false;
$p_{p(i)}$! b_i
□
$p_{p(i)}$? **resume** → cm := true;
advanced := false;
ready := true
□
ready $\wedge$ ($b_i \vee$ advanced);
$p_{p(i)}$! true → ready := false

The initial distributed program is as follows:

P :: $[p_0 \parallel \ldots \parallel p_{n-1}]$ where for each $0 \leq i < n$, $p_i :: *[S_i]$

P is assumed to have the following two properties:

1. $\bigwedge_{i=0}^{n-1} b_i \rightarrow B$
2. Communicating processes $p_0,\ldots,p_{n-1}$ can be found, so that by means of some finite sequence of bc, the parallel composition
   P :: $[p_0 \parallel \ldots \parallel p_{n-1}]$ reaches a globally stable state, in which each p_i is locally stable, and p_i initiates no communication, but is ready to communicate with other processes that would initiate such a communication.

Thus when b_i holds, p_i waits at the top level loop, and S_i contains guards which wait for possible communication.

This algorithm uses the following variables:

bc_i true if b_i is true and no bc has occurred since b_i first became true in this time interval else false

$count_i$ the counter for the control messages, it acts as the upper end in the time interval assertion

db_i a copy of $count_i$ at the time bc_i last became true - it serves as the lower end of the time interval in the interval assertion

C_i assures that a change in the value of b_i has not occurred since the beginning of the time interval
C_i is equivalent to the truth value of $db_i < count_i$

M_i the line number for the time instance at which the assignments on this line are performed and cause bc_i to be set to false

The modified program containing the control code for the root is as follows:

$p_0 :: bc_0 :=$ false; $send_0 =$ true; $count_0 = 0$; $db_0 = 0$;
*[S_0
□ b_0; $\neg bc_i \rightarrow bc_0 :=$ true; $db_0 = count_0$
□ p_{n-1} ? $s_0 \rightarrow$ [$s_0 \rightarrow$ halt
□ $\neg s_0 \rightarrow send_0 :=$ true]
□ $send_0$; bc_0; p_1 ! $(db_0 < count_0) \rightarrow send_0 :=$ false;
$count_0 := count_0 + 1$
]

The modified program containing the control code for the remaining nodes is as follows:

$p_i :: bc_i :=$ false; $send_i =$ false; $count_i ;= 0$;
*[S_i
□ b_i; $\neg bc_i \rightarrow bc_i :=$ true; $db_i := count_i$
□ p_{i-1} ? $s_i \rightarrow send_i :=$ true
□ M_i; $send_i$; bc_i; p_{i+1} ! $(s_i \wedge C_i) \rightarrow send_i :=$ false; $count_i := count_i + 1$
]

This algorithm uses the following variables:

p(i)	the index of the parent of node i in the spanning tree
Γ_i	the set of indexes of the children of p_i
w2()	the constant message representing wave2
advance	true implies that a bc has occurred false implies that a bc has not occurred since the last wave
m	the count of children of p_i that have already transmitted wave1-3
γ	the total number of children any one p_i has
w2-arrived	true implies that wave 2 has been received and wave1-3 has not been sent yet this variable is used only by the leaves and is initially set to true
send-w2[j]	an array containing a boolean value for each $j \in \Gamma_i$ true indicates that wave2 has been sent to p_j
wave1-3[j]	an array containing a boolean value for each $j \in \Gamma_i$ true indicates that a bc has occurred in one of p_j's children false indicates that all children of p_j are ready to terminate

Control code for the root, p_0:

initially m = 0, advance is true and send-w2[j] is false for all children of the root

c_i :: $\Box_{j \in \Gamma_0}$ p_j ? wave1-3[j] $\rightarrow$ m := m + 1

$\Box$ m = γ; b_0 $\rightarrow$ [$\neg$advance; $\bigwedge_{j \in \Gamma_0}$ $\neg$wave1-3[j] $\rightarrow$ halt

$\Box$ advance $\vee$ ($\bigvee_{j \in \Gamma_0}$ wave1-3[j] $\rightarrow$ m := 0;

advance := false; $\prod_{j \in \Gamma_0}$ send-w2[j] := true

]

$\Box_{j \in \Gamma_0}$ send-w2[j]; p_j ! w2() $\rightarrow$ send-w2[j] := false

Control code for an intermediate node:

initially advance is true, m is 0, and send-w2[j] is false for all children of each intermediate node

c_i :: $\underset{j \in \Gamma_i}{\square}\ p_j$? wave1-3[j] $\rightarrow$ m := m + 1

$\square\ \gamma$ = m; b_i;

$p_{p(i)}$! (advance $\vee$ ($\underset{j \in \Gamma}{\vee}$ wave1-3[j])) $\rightarrow$ m := 0;

advance := false;

$\square\ p_{p(i)}$? w2() $\rightarrow$ $\underset{j \in \Gamma_i}{\prod}$ send-w2[j] := true

$\underset{j \in \Gamma_i}{\square}$ send-w2[j];

p_j ! w2() $\rightarrow$ send-w2[j] := false;

Control code for a leaf:

initially w2-arrived is true, and advance is true

c_i :: $\square\ b_i$; w2-arrived; $p_{p(i)}$! advance $\rightarrow$

w2-arrived := false;

advance := false;

$\square\ p_{p(i)}$? w2() $\rightarrow$ w2-arrived := true;

This algorithm uses the following variables:

T	is the token at this node?
FP	initialized to nil FP contains the name of the node farthest away from p_i, with whom p_i has communicated

The token has the name PN - initially this is the name of the predecessor of the node where the token is placed.

The function dist (p_j) returns the distance from node p_i to p_j.

The following code is added to the application whenever a set[1] statement is reached:

```
set (BE1, BE2,...)
    begin
    if all BEi in the set statement are true then
        b_i := true/* this is done by the set statement */
    if T = true then
        if b_i then
            if FP = nil then
                if PN = this process then
                    initiate termination
                else
                    pass token to next process in CDG
            else /* FP ≠ nil */
                begin
                if dist (FP)  > dist (PN) then
                    PN = FP
                FP = nil
                pass token to next process in CDG
                end
    end
```

1 The set statement takes as its arguments all boolean expressions that must be true if the process is passive. If all of the boolean arguments are true, then b_i is assigned the value true.

The following code is added to the application whenever a reset[2] statement is reached:

```
reset (BE1, BE2,...)
    begin
    if all BEi in reset statement are true then
        b_i := false/* this is done by the reset statement */
    end
```

Whenever process p_i sends a bc to process p_j the following piece of code is executed:

```
    if FP = nil then
        FP := p_j
    else if dist (p_j) > dist (FP) then
        FP := p_j;
    p_j ! bc
```

Whenever a process receives the token it executes the following piece of code:

```
    begin
    if b_i = true then
        if FP = nil then
            if PN = this process then
                initiate termination
            else
                pass the token to the next process in the CDG
        else
             if dist (FP) > dist (PN) then
                begin
                PN = FP
                FP = nil
                pass the token to the next process in the CDG
                end
    end
```

2 The reset statement takes as its arguments all of the boolean expressions that must be true if a process is active. If all of the boolean arguments are true, b_i is assigned the value false.

This algorithm uses the following variables:

$BTime_i$	the last time at which p_i went passive
Time	the timestamp of the detection message.
Count	how many processes have seen this detection message.

The following is the code for each node in the ring:

```
p_i :: b_i := false;
    *[Application statements
     □ b_i → BTime_i := Clock-Time;
        Time := Btime_i;
        Count := 1;
        p_{i+1} ! detection-msg (Time, Count)
     □ p_{i-1} ? detection-msg (Time, Count) →
        [ Count = n →
              initiate termination phase
        □ Count ≠ n →
              [¬b_i → purge the message
              □ b_i →
                    [Time < BTime_i → purge the message
                    Time ≥ BTime_i →
                          count := count + 1;
                          p_{i+1} ! detection-msg (Time, Count)
                    ]
              ]
        ]
    ]
```

This algorithm uses the following variables at each node:

ok	true if the clock has been read and the process is passive set to false when a bc occurs ensures reading of the clock value occurs only once when the process becomes passive
sent	true if a probe has been sent set to false when a bc occurs ensures that sending of a detection message takes place only once during each period when the process is passive
T	the counter used as the virtual clock
count	the number of processes which have received this detection message
time	the timestamp of this message
fait	set to true when count = 2n to enable a process to exit the main loop and send a termination message.
Γ_i	the set of indices for all processes to whom p_i has an outgoing channel

The following is the code for the underlying application:

P = $[p_1 \,||\, ... \,||\, p_n]$ where for every i, $1 \leq i \leq n$,
$p_i :: {}^*[S_i]$ and S_i is of the form $\underset{j \in \Gamma_i}{\square} g_{i,j} \rightarrow s_{i,j}$ and

1. each $g_{i,j}$ contains an I/O command addressing p_j
2. none of the statements $init_i$ or $s_{i,j}$ contain an I/O command

The following is the code for each node in the ring:

```
p_i :: init; ok := false; sent := false; fait := false; T := 0;
        * [ □ j∈Γ_i ¬fait; g_{i,j} → ok := false; sent := false; s_{i,j}
          □ ¬fait; b_i; ¬ok → T := T+ 1; ok := true;
          □ ¬fait; ok; ¬sent; p_{i+1} ! detection-message (T,1) → sent := true
          □ ¬fait; b_i → ok; p_{i-1} ? detection-message (time, count) →
            [count = 2n → fait := true
             □ count < 2n →
                 [¬b_i → T := max (time, T) – purge the message
                  □ b_i →
                      [time < T → skip  – purge the message
                       □ time ≥ T → T := time;
                                    count := count + 1;
                                    p_{i+1} ! detection-message (time, count);
                                    sent := true
                      ]
                 ]
            ]
```

This algorithm uses the following variables:

first	true → this process has not created a detection wave for itself false → this process has initiated a probe (and created a token) this guarantees that a maximum of n tokens will be generated
known	true → no bc has occurred since the previous detection message has been received false → a bc has occurred
turn2	true → a message with count equal to kcount is received a second time and known is still true i.e., no bc has occurred
done	true → termination may be initiated false → the *GTC* has not been detected
kept	true → a detection message is being stored false → there are no detection messages to be sent
color	true → all processes visited have been passive since the preceeding message with the same count field false → some process has received a bc since the preceeding message with the same count field
count	the count of the number of processes that have passed this token on.
kcount	the count field of the first message seen after b_i becomes true
$s_{i,j}$	these are the application statements
Γ_i	the set of indices for all processes to whom p_i has an outgoing channel

The following is the code for the underlying application:

P = $[p_1 \,||\, ... \,||\, p_n]$ where for every i, $1 \leq i \leq n$,
$p_i :: {}^*[S_i]$ and S_i is of the form $\Box_{j \in \Gamma_i} g_{i,j} \rightarrow s_{i,j}$ and

1. each $g_{i,j}$ contains an I/O command addressing p_j
2. none of the statements $init_i$ or $s_{i,j}$ contain an I/O command

The following is the code for each node in the ring:

```
p_i :: first := true;
       known, turn2, done, kept, color := false;
       count := 0;
       kcount := 0                          {the author leaves out this initialization}
       *[ □ (j∈Γ_i) ¬done; g_{i,j} → known := false; turn2 := false; color := false; s_{i,j}
        □ ¬done; (first ∨ ¬kept);  p_{i-1} ? detection (color, count) →
            [count = n → [color → done := true
                          □ ¬color → color := turn2;
                                     count := 0
                          ]
             □ count ≠ n → color := color ∧ turn2
            ];
            [ ¬known → kcount := count
              □ known →
                          [count = kcount → turn2 := true
                          □ count ≠ kcount → skip
                          ]
            ];
            first := false; kept := true
        □ ¬done; b_i; (first ∨ kept); p_{i+1} ! detection (color, count+1) →
            kept := false; first := false;
            known := true                   {the author leaves out this assignment}
       ]
```

SZYMANSKI, SHI, AND PRYWES 1985

This algorithm uses the following variables:

I_i	the minimum label of all tokens received at node i in the jth input
S_i	the integer label to be put on the jth output token
NOTE:	Tokens are labeled by integer values ranging from 0 to $D + 1$ where D is the diameter of the network.

$S_i(j)$ is defined for process p_i and step j as follows:

0	if b_i is false
$I_i(j)+1$	if b_i is true and $I_i(j) < S_i(j-1)$
$S_i(j-1)+1$	otherwise

The algorithm is as follows:

```
for all p_i do
    begin
    S_i = 0
    while S_i < D do
        send (succ(p_i), S_i, p_i);
        read input-tokens
        evaluate b_i
        if not b_i then
            S_i = 0
        else
            S_i = min (I_i, S_i) + 1
        end while
    terminate
    end
```

ARORA RANA AND GUPTA 1986

This algorithm uses the following variables for each process:

$neighbors_i$	the set of indices for all p_i's neighbors
$state_i(p_k)$	an array for all neighbors, p_k, of p_i that keeps track of the state of p_k - active or passive, initially all are set to active
msg	this is the detection message - it contains the name of the process that initiates the probe

When p_i becomes passive the following code is executed:

b_i := true;
for all j $\in$ $neighbors_i$ do
    send(p_j, 'I-am-passive', p_i);

Whenever p_i sends a bc to p_j, the following code is executed:

$state_i(p_j)$:= active

Whenever p_i receives an 'I-am-passive' message from any p_j, j $\in$ neighbors(p_i), the following code is executed:

$state_i(p_j)$:= passive
if $state_i(p_k)$ = passive for all k $\in$ $neighbors_i$ and b_i then
    send(succ(p_i), msg, p_i);

Whenever p_i receives a probe that it did not initiate, the following code is executed:

if $state_i(p_j)$ = passive for all j $\in$ $neighbors_i$ and b_i then
    send(succ(p_i), msg, p_i);
else
    purge the detection message

Whenever p_i receives its own probe back, the following code is executed:

enter the termination phase

This algorithm uses the following variables for each process:

$send_i$	true when process, p_i, holds the probe and has not passed it on yet false when process, p_i, does not hold the probe
$moved_i$	true when a bc has take place set to false whenever the probe has been passed to the right-hand side neighbor, it remains false as long as no bc occur

The following is the code for the underlying application:

P = $[p_1 \,||\, ... \,||\, p_n]$ where for every i, $1 \leq i \leq n$,
$p_i :: init_i : *[S_i]$ and S_i is of the form $\underset{j \in \Gamma_i}{\square} g_{i,j} \rightarrow s_{i,j}$ and

1. each $g_{i,j}$ contains an I/O command addressing p_j
2. none of the statements $init_i$ or $s_{i,j}$ contain an I/O command

The following is the code for the detector:

$p_1 :: init_i$;
  $send_i$:= true;
  $*[\underset{j \in \Gamma_i}{\square} g_{i,j} \rightarrow s_{i,j}$
  $\square\ b_i$; $send_i$;
    p_{i+1} ! true $\rightarrow$ $send_i$:= false
  $\square\ p_{i-1}$? $s_i \rightarrow [s_i \rightarrow$ halt $\square\ \neg s_i \rightarrow send_i$:= true$]$
  ]

The following is the code for all other nodes:

$p_i :: init_i$;
  $send_i$:= false;
  $moved_i$:= false;
    $*[\underset{j \in \Gamma_i}{\square} g_{i,j} \rightarrow moved_i$: = true; $s_{i,j}$
    $\square\ p_{i-1}$? $s_i \rightarrow send_i$:= true
    $\square\ b_i$; $send_i$;
      p_{i+1} ! $(s_i \wedge \neg moved_i) \rightarrow send_i$:= false;
                $moved_i$:= false
    ]

This algorithm uses the following variables at each node, p_i:

active	used to indicate that bc have taken place since the last warning was sent
$count_i$	used to count the number of times that ready and warning messages have been received on all input buffers
m	the minimum number attached to all broadcast messages received in a given phase
L	L is a constant chosen to fulfill the following condition: $L \geq 2d$ where d is the diameter of the network
n	the number of processes in the network; it is assumed that n will always be greater than the diameter d by at least 1

The detection message, *msg*, contains the following information:

name	this may be either a warning or a ready message	
	warning	this message warns the other processes that some process might have been activated by p_i during its last phase
	ready	when p_i receives only this message for some time, p_i may terminate
age	this number tells the other processes that sender has been passive for this many phases when the age reaches L, a warning message becomes a ready message	

The following additional statements are available for use by the algorithm:

enable	allows interrupts to occur for the reception of bc
disable	prevents interrupts to occur for the reception of bc

Whenever process p_i becomes passive, the following code is executed:

```
active := true;
count_i := 0;
msg.name := warning;
msg.age := 1;
broadcast (msg);
cycle
    await that no inbuffers are empty;
    disable;
    read one message from each inbuffer;
    enable;
    if count_i = L then
        terminate;
    disable;
    if active then
        msg.name := warning;
        msg.age := 1;
        broadcast (msg);
        active := false;
        count_i := 0
    else
        if all inputs were (msg.name = warning and msg.age = n) or msg.name =
        ready then
            msg.name := ready;
            broadcast (msg);
            count_i := count_i + 1;
        else
            m := min {i | (msg.name = warning and msg.age = i) was read}
            msg.name := warning;
            msg.age := m + 1;
            broadcast (msg);
            count_i := 0;
    enable;
```

This algorithm uses the following variables at each node, p_i:

$seqnum_i$	the initial value is 0, but is incremented whenever p_i initiates a probe
$IdList_i$	this is a list of all processes to whom p_i sends a bc

The detection message, *msg*, contains the following information:

name	the unique identification of the process that issued the probe
seqdm	an integer representing the sequence number carried by the probe
F1	0 indicates that the probe has not been falsified 1 indicates that the probe has been falsified - either a process was found active or its IdList is not empty
F2	0 indicates that no process was found to be active 1 indicates that some process was found to be active

Whenever process p_i becomes passive, the following code is executed:

```
b_i := true
seqnum_i := seqnum_i + 1;
msg.seqdm := seqnum_i;
msg.F1 := 0
msg.F2 := 0;
send(succ(p_i), msg, p_i);
```

Whenever p_i receives a probe from process p_j, the following code is executed:

```
if msg.F1 = 1 then
    if ¬b_i then
        msg.F2 := 1;
    if msg.name ∈ IdList_i then
        remove msg.name from IdList_i
        send(succ(p_i), msg, p_i)
else {msg.F1 = 0}
    if b_i then
        if msg.name ∈ IdList_i then
            remove msg.name from IdList_i
        if IdList_i is null then
            send(succ(p_i), msg, p_i);
        else {IdList_i not null}
            msg.F1 := 1;
            send(succ(p_i), msg, p_i);
    else {¬b_i}
        msg.F1 := 1;
```

```
msg.F2 := 1;
if msg.name ∈ IdList_i then
    remove msg.name from IdList_i
send(succ(p_i), msg, p_i);
```

Whenever p_i receives its own probe back, the following code is executed:

```
if ¬b_i then
    purge the detection message
else {b_i}
    if msg.F1 = 1 then
        if msg.F2 = 0 then
            if seqnum_i = msg.seqdm then
                purge the detection message
                seqnum_i := seqnum_i + 1
                msg.seqdm := seqnum_i
                msg.F1 = 0;
                msg.F2 = 0;
                send(succ(p_i), msg, p_i);
            else {seqnum_i ≠ msg.seqdm}
                purge the detection message
        else  {msg.F2 = 1}
            purge the detection message
    else {msg.F1 = 0}
        if seqnum_i = msg.seqdm then
            enter termination phase
        else
            purge the detection message
```

This algorithm uses the following variables:

count	the number of processes that the probe has found passive
terminated	when all processes have been found passive, this is set to true
first_time	guarantees that p_i will only initiate one detection wave
c[i-1]	the Occam channel connecting p_i to its predecessor
c[i]	the Occam channel connecting p_i to its successor

The control code[3] for each node is as follows:

```
p_i :: VAR terminated, first_time, count:
       SEQ
            terminated := false
            first_time := true
            while not terminated
                 ALT
                      c[i-1] ? count
                           SEQ
                                IF
                                     count ≥ n
                                          terminated := true
                                TRUE
                                     SKIP
                                IF
                                     b_i
                                          c[i] ! (count + 1)
                                TRUE
                                     SKIP
                      SKIP & b_i & first_time
                           SEQ
                                first_time := false
                                c[i] ! 1
```

3 This algorithm is written in Occam which is related to CSP.

The Time Algorithm

This algorithm uses the following variables at each node, p_i:

clock	the local clock, a counter initialized to 0
count	the local message counter, it is equivalent to $s_i() - r_i()$
tmax	the latest send-time of all messages received by p_i, initialized to 0

The detection message, *msg*, contains the following information:

time	the timestamp of the cc
count	the accumulator for the message counters
invalid	the flag that indicates that a bc has been received since the last round
init	the id of the process initiating the probe

A bc, also *msg*, contains the following information:

tstamp	the time stamp on any bc
info	the actual message

When p_i sends a bc to p_j the following code is executed:

```
count := count + 1
msg.time := clock
msg.info := bc
send (p_j, msg, p_i)
```

When p_i receives a bc from p_j the following code is executed:

```
count := count − 1
tmax := max (msg.tstamp,tmax);
/* process the message */
```

When p_i receives a cc, the following code is executed:

```
clock := max (msg.time, clock)
if msg.init = p_i then
      if msg.count = 0 and not msg.invalid then
            initiate termination
      else
            try again
else
      msg.count := msg.count + count
      msg.invalid := msg.invalid or tmax ≥ msg.time
      send (succ (p_i), msg, p_i)
```

When p_i wants to initiate a probe the following code is executed:

```
clock := clock + 1
msg.time := clock
msg.count := count
msg.invalid := false
msg.init := p_i
send (succ(p_i), msg, p_i)
```

FRIEDEMANN MATTERN 1987

The Vector Algorithm

This algorithm uses the following variables at each node, p_i:

$count_i$	each process p_i has an array whose indices represent the destination/source of all messages sent/received by p_i and whose contents represents the sum of all messages sent to each destination minus the sum of all messages received from that same destination
have_vector	in the previous article [MAT87A], Mattern gives more details - this boolean represents whether or not the token is currently visiting this process

The detection message, *msg*, contains the following information:

count	an array which contains the sum of all messages sent minus the sum of all messages received between any two processes - each index represents a different process pair and Mattern refers to this as a vector count is initialized to 0* - the null vector

When a bc is sent by p_i to p_j the following code is executed:

$count_i[j] := count_i[j] + 1$

Whenever a bc is received by p_i from p_j the following code is executed:

$count_i[i] := count_i[i] - 1$
if $count_i[i] \leq 0$ then
    if have_vector and b_i then
        if $count_i = 0^*$ then
            initiate termination
        else
            msg.count := $count_i$
            send (succ(p_i), msg, p_i)
            $count_i := 0^*$

Whenever p_i receives the cc the following code is executed:

```
count_i := count_i + msg.count
have_vector := true
wait until b_i
disable
if count_i[i] ≤ 0 then
      if count_i = 0* then
            initiate termination
      else
            msg.count := count_i
            send (succ(p_i), msg, p_i)
            have_vector := false
            count_i := 0*
enable
```

This algorithm uses the following variables for each process p_i:

Γ_i	the set of indices of p_i's children
$p(i)$	the index of p_i's parent
$neighbors_i$	the set of indices of p_i's neighbors
$state_i(p_k)$	an array for all neighbors, p_k, of p_i that keeps track of the state of p_k - active or passive
$s(p_i)$	0 initially, cleared to 0 whenever an 'I-am-up' message is sent to p_i's parent 1 whenever an 'I-am-through' message is sent to p_i's parent
$child_i(p_k)$	an array for all children, p_k, of p_i that keeps track of the state of p_k - active or passive
$seq(p_i)$	the sequence number of the most recent detection message received from p_i's parent
$rcbdm_i(p_k)$	an array of booleans for each child, p_k, of p_i which indicates whether or not a valid detection message been received from this child, i.e. does it match the most recent detection message received from $p_{p(i)}$
$seqdm$	an integer representing the sequence number carried by the detection message - this is not additional memory, it is used only in the detection message

Whenever process p_i becomes passive, the following piece of code is executed:

```
b_i := true
for all j ∈ neighbors_i do
    send (p_j, 'I-am-passive', p_i);
```

Whenever a process, p_i changes state from passive to active the following code is executed:

```
b_i := false
if s(p_i) = 1 then      {conserve messages}
    if p_i is a leaf process or p_i is an internal process then
        send(p_p(i), 'I-am-up-again', p_i);
        s(p_i) := 0;
```

Whenever process p_i receives a bc from p_j the following statement is executed:

send $(p_j$, ack, $p_i)$;
$state_i(p_j)$:= active;

Whenever process p_i sends a bc to p_j the following statement is executed:

$state_i(p_j)$:= active;

Whenever process p_i receives the 'I-am-passive' message from p_k, the following code is executed:

$state_i(p_k)$:= passive
if $state_i(p_j)$ = passive for all j $\in$ $neighbors_i$ and b_i = passive then
    if p_i is a leaf then
        send $(p_{p(i)}$, 'I-am-through', $p_i)$;
        s(p_i) := 1
    else if p_i is the root then
        if $child_i(p_j)$ = passive for all j $\in \Gamma_i$
            seq(p_i) := seq(p_i) + 1
            seqdm = seq(p_i);
            for each j $\in \Gamma_i$ do
                $rcbdm_i(p_j)$:= false
                send $(p_j$, seqdm, $p_i)$;
    else if p_i is an internal node
        if $child_i(p_j)$ = passive for all j $\in \Gamma_i$
            send $(p_{p(i)}$, 'I-am-through', $p_i)$;
            s(p_i) := 1;

Whenever a process, p_i, receives the 'I-am-through' message from p_k, k $\in \Gamma_i$ the following code is executed:

$child_i(p_k)$:= passive
if $state_i(p_j)$ = passive for all j $\in$ $neighbors_i$ and b_i and
    $child_i(p_j)$ = passive for all j $\in \Gamma_i$ then
    if p_i is an internal process then
        send $(p_{p(i)}$, 'I-am-through', $p_i)$;
        s(p_i) := 1;
    else {if p_i is the root - leaves will never receive this message}
        seq(p_i) := seq(p_i) + 1;
        seqdm := seq(p_i);
        for each j $\in \Gamma_i$ do
            $rcbdm_i(p_j)$:= false;
            send $(p_j$, seqdm, $p_i)$;

Whenever a process, p_i, receives an 'I-am-up-again' message from p_k, and $k \in \Gamma_i$ the following code is executed:

```
child_i(p_k) := active;
if p_i is not the root then                    {clearly intended but not explicitly stated}
    if s(p_i) = 1 then
        send (p_p(i), 'I-am-up-again', p_i);
        s(p_i) := 0;
```

Whenever a process, p_i, receives a detection message from process p_j, the following code is executed:

```
if b_i and state_i(p_j) = passive for all j ∈ neighbors_i then
    if p_i is the root process then
        if seq(p_i) = seqdm and child_i(p_j) = passive for all j ∈ Γ_i then
            rcbdm_i(p_j) := true;
            if rcbdm_i(p_k) = true for all k ∈ Γ_i
                enter the termination phase
            else
                purge the detection message
        else
            purge the detection message
    else if p_i is a leaf process then
        return the detection message
    else if p_i is an internal process
        if p_j = p_p(i) then
            if child_i(p_k) = passive for all k ∈ Γ_i then
                seq(p_i) = seqdm;
                for each k ∈ Γ_i do
                    rcbdm_i(p_k) = false;
                    send(p_k, seqdm, p_i);
            else {some child is not passive}
                purge the detection message
        else if j ∈ Γ_i then
            if seq(p_i) = seqdm then
                if child_i(p_k) = passive for all k ∈ Γ_i then
                    rcbdm_i(p_j) = true;
                    if rcbdm_i(p_k) = true for all k ∈ Γ_i then
                        send (p_p(i), seqdm, p_i);
                    else {not all rcbdm_i(p_k) are true}
                        purge the detection message
                else {some child is active}
                    purge the detection message
            else {seq(p_i) <> seqdm}
                purge the detection message
else {either p_i or one of its neighbors is active}
    purge the detection message
```

This algorithm uses the following variables:

FP	the id of the farthest process down the ring with which p_i has communicated when p_i was active
$IdList_i$	this is a list of processes with whom this process initiates communication

The detection message, *msg*, consists of a record containing the following information:

name	the id of the initiator
FP	the id of the process farthest down the line with whom the initiator of this probe has communicated
F1	a flag to indicate verification of termination 0 - it is okay to terminate 1 - it is not okay to terminate

Whenever process p_i sends a bc to process p_j the following piece of code is executed:

```
if FP = nil then
      FP := p_j
else if dist (p_j) > dist (FP) then
      FP := p_j
send (p_j, msg, p_i)
add p_j to IdList_i
```

When p_i becomes passive, the following piece of code is executed:

```
b_i := true;
msg.F1 := 0;
msg.FP := FP;
msg.name := p_i;
send (succ(p_i), msg, p_i);
FP := nil;
```

When p_i receives a probe from p_j, the following code is executed:

```
if msg.F1 = 1 then
    if msg.name = p_i then   {The authors imply this cannot happen}
        msg.F1 := 0;
        msg.name := p_i;
        msg.FP := FP;
        send (succ (p_i), msg, p_i);
    else if msg.name <: p_i <: msg.FP then {on the way to farthest}
        if msg.name in IdList_i then
            remove msg.name from IdList_i
            send (succ(p_i), msg, p_i);
    else                    {at or beyond msg.FP}
        if msg.name in IdList_i then
            remove msg.name from IdList_i
        if not b_i or not empty(IdList_i) then
            purge (msg);
        else                {p_i is passive and IdList_i is empty}
            send (succ(p_i), msg, p_i);
else                {msg.F1 = 0}
    if msg.name = p_i then
        initiate termination
    else if msg.FP = p_i then
        if msg.name in IdList_i then
            remove msg.name from IdList_i
        if b_i and empty (IdList_i) then
            send (succ(p_i), msg, p_i)
        else                {not b_i or not empty (IdList_i)}
            purge (msg);
    else if msg.name <: p_i <: msg.FP then
        {p_i is on the path from msg.name to msg.FP}
        if msg.name in IdList_i then
            remove msg.name from IdList_i
        if not b_i or not empty (IdList_i) then
            msg.F1 := 1;
        send (succ (p_i), msg, p_i);
    else        {msg.FP <: p_i or msg.FP = nil - msg.name never sent bc to p_i}
        if b_i and empty (IdList_i) then
            send (succ (p_i), msg, p_i)
        else
            purge (msg);
```

This algorithm uses the following variables at each node p_i:

time$_i$ this is the current logical time; it has total ordering and monotonic increasing properties
the time consists of two parts:

value two 'clock times' are related as follows:
($time_i$.value, p_i) > ($time_j$.value, p_j) if

(a) $time_i$.value > $time_j$.value, or

(b) $time_i$.value = $time_j$.value, and $p_i > p_j$
Note: we interpret greater as later

name this is the name of the process that owns this 'clock time'

Btime$_i$ the latest idleness time ever known by p_i while p_i is idle - this variable consists of the same two parts as $time_i$

The detection message, *msg*, contains the following information:

type there are three types of cc that can be sent: announcements, *Ann*, agreements, *Agr*, and acknowledgments, *Ack*

clock the clock consists of 2 parts:

value the 'time' of this control communication

name the 'owner' of this time

When p_i becomes passive it executes the following piece of code:

```
b_i := true;
time_i.value := time_i.value + 1;
Btime_i.value := time_i.value;
Btime_i.name := p_i;
msg.type := Agr;
msg.clock := time_i;
broadcast (msg);
msg.type := Ann;
broadcast (msg);
```

When p_i sends a bc to p_j, p_i executes the following code:

```
Wait to receive a message from p_j with msg.type = Ack
time_i.value := max (time_i.value, msg.clock.value);
```

When p_j receives a bc from p_i, the following code is executed:

```
msg.type := Ack;
msg.clock.value := time_j.value;
msg.clock.name := p_j;
send (p_i, msg, p_j);
b_j := false;
```

When p_i receives a message of type Ann, it executes the following code:

```
if ¬b_i then
    time_i.value := max (time_i.value, msg.clock.value);
else if b_i then
    if msg.clock > Btime_i then
        msg.type := Agr;
        broadcast (msg);
        time_i.value := msg.clock.value;
        Btime_i.value := msg.clock.value;
        Btime_i := msg.clock.name;
```

When p_i receives a message of type Agr, the code to be executed is dependent on the network structure. When all Agr cc messages have been received termination is declared.

ARORA AND GUPTA 1989

Predesignated Process Algorithm

This algorithm uses the following variables at each node p_i:

bcm(p_i)	a boolean flag which is set whenever p_i sends/receives a bc, initially 0
distance1	the distance clockwise around the ring from this process to the initiator
distance2	the distance anticlockwise around the ring from this process to the initiator
pas	a boolean flag for p_1 which is initially false and is set to true when p_1 becomes true the first time and remains true
prevpm	a boolean flag which is reset to 0 when p_1 initiates the first set of probe messages, it remains 0 afterwards a boolean flag which records the status of the forwarded probe message for the remaining p_i
procflg(p_i)	a boolean flag which is set whenever p_i changes state from active to passive, initially 0
seq(p_i)	the sequence number of the current set of probes for p_1, and the sequence number for the currently forwarded probe message for the remaining p_i

The detection message, *msg*, consists of a record containing the following information:

seqdm	an integer representing the sequence number carried by the detection message
type	the kind of control message being sent: *repeat-probe-signals* — used to trigger a fresh set of probe-messages *probe messages* — used to detect termination
F1	a flag to verify termination 0 - it is okay to terminate, probe not falsified 1 - it is not okay to terminate, probe has been falsified

When p_i becomes passive, the following piece of code is executed:

```
state(p_i) := passive
procflg(p_i) := 1
if i = 1 then
    if not PAS then
        seq(p_i) := seq(p_i) + 1
        msg.seqdm := seq(p_i)
        msg.type := probe
        msg.F1 := 0
        send(succ(p_i), msg, p_i)
        send(pred(p_i), msg, p_i)
        PAS := true
        prevpm := 0
        bcm(p_i) := 0
        procflg(p_i) := 0
```

When p_i sends/receives a bc from p_j, the following code is executed:

```
bcm(p_i) := 1
```

When the repeat-probe-signal message is received by p_i, the following code is executed:

```
if ¬b_i then
    wait until b_i
if i ≠ 1 then
    send(next(p_i), msg, p_i)
else if msg.seqdm = seq(p_i) then
    seq(p_i) := seq(p_i) + 1
    msg.seqdm := seq(p_i)
    msg.type := probe
    msg.F1 := 0
    send(succ(p_i), msg, p_i)
    send(pred(p_i), msg, p_i)
    bcm(p_i) := 0
    procflg(p_i) ;= 0
else
    purge the repeat-probe-signal message
```

When p_i (i $\neq$ 1) receives a detection message from p_1 or p_1 receives back its own detection message, the following code is executed:

```
if ¬b_i then
    wait until b_i
if seq(p_i) = msg.seqdm then
    if msg.F1 = 0 then
        if prevpm(p_i) = 0 and (bcm(p_i) = 0 and procflg(p_i) = 0) then
            enter the termination phase
        else
            purge the detection message
    else
        purge the detection message
        if i = 1 then
            seq(p_i) = seq(p_i) + 1
            msg.seqdm := seq(p_i)
            msg.type := probe
            msg.F1 := 0
            send (succ(p_i), msg, p_i)
            send (pred(p_i), msg, p_i)
            bcm(p_i) := 0
            procflg(p_i) := 0
        else
            if distance1 ≥ distance2 then
                msg.type := repeat-probe-signal
                send(succ(p_i), msg, p_i)
            else
                msg.type := repeat-probe-signal
                send(pred(p_i), msg, p_i)
            prevpm(p_i) := 0
            bcm(p_i) := 0
else if i = 1 then
    purge the detection message
else
    seq(p_i) := msg.seqdm
    if msg.F1 = 0 then
        if procflg(p_i) = 0 and bcm(p_i) = 0 then
            prevpm(p_i) := 0
        else
            prevpm(p_i) := 1
            msg.F1 := 1
    else          {probe is falsified}
        prevpm(p_i) := 1
        bcm(p_i) := 0
        procflg(p_i) := 0
    send(next(p_i), msg, p_i)
```

ARORA AND GUPTA 1989

Shifting Process Control Algorithm

This algorithm uses the following variables at each node p_i:

bcm(p_i)	a boolean flag which is set whenever p_i sends/receives a bc, initially 0
pas	a boolean flag for p_1 which is initially false and is set to true when p_1 becomes true the first time and remains true
prevpm	a boolean flag which records the status of the forwarded probe message for p_i
procflg(p_i)	a boolean flag which is set whenever p_i changes state from active to passive, initially 0
seq(p_i)	the sequence number of the current set of probes for p_i
pseq(p_i	the sequence number of the probe message previously forwarded/issued by p_i

The detection message, *msg*, consists of a record containing the following information:

seqdm	an integer representing the sequence number carried by the detection message
type	the kind of control message being sent: *probe* used to detect termination
F1	a flag to verify termination 0 - it is okay to terminate, probe not falsified 1 - it is not okay to terminate, probe has been falsified

When p_1 becomes passive, the following piece of code is executed:

```
state(p_i) := passive
procflg(p_i) := 1
if not PAS then
    seq(p_i) := seq(p_i) + 1
    msg.seqdm := seq(p_i)
    msg.type := probe
    msg.F1 := 0
    send(succ(p_i), msg, p_i)
    send(pred(p_i), msg, p_i)
    PAS := true
    prevpm := 0
    bcm(p_i) := 0
    procflg(p_i) := 0
```

When p_i sends/receives a bc from p_j, the following code is executed:

```
bcm(p_i) := 1
```

When p_i receives back its own detection message, or a detection message from p_j (i ≠ j), the following code is executed:

```
if ¬b_i then
    wait until b_i
if pseq(p_i) = msg.seqdm then
    if msg.F1 = 0 then
        if prevpm(p_i) = 0 and (bcm(p_i) = 0 and procflg(p_i) = 0) then
            enter the termination phase
        else
            purge the detection message
            {add code for if msg.source = pred(p_i) here}
    else
        purge the detection message
        if msg.source = pred(p_i) then
            seq(p_i) = seq(p_i) + 1
            msg.seqdm := seq(p_i)
            pseq(p_i) := msg.seqdm
            msg.type := probe
            msg.F1 := 0
            send (succ(p_i), msg, p_i)
            send (pred(p_i), msg, p_i)
            bcm(p_i) := 0
            procflg(p_i) := 0
            prevpm(p_i) := 0
else
    pseq(p_i) := msg.seqdm
    if msg.F1 = 0 then
        if procflg(p_i) = 0 and bcm(p_i) = 0 then
            prevpm(p_i) := 0
        else
            prevpm(p_i) := 1
            msg.F1 := 1
    else          {probe is falsified}
        prevpm(p_i) := 1
        bcm(p_i) := 0
        procflg(p_i) := 0
    send(next(p_i), msg, p_i)
```

ARORA AND GUPTA 1989

Multiple Process Algorithm

This algorithm uses the following variables at each node p_i:

$seq(p_i)$	the sequence number of the current set of probes for p_i
$pmseq_i(p_j)$	an array of sequence numbers of probe messages previously forwarded by p_i from p_j
$status_i(p_j)$	an array of boolean flags which record the status of the last forwarded probe message from p_j
$state_i(p_j)$	an array for all neighbors, p_j, of p_i that keeps track of the state of p_j - active or passive
$s1_i$	a list of processes in the clockwise direction around the ring from this process
$s2_i$	a list of processes in the anticlockwise direction around the ring from this process

The detection message, *msg*, consists of a record containing the following information:

seqdm	an integer representing the sequence number carried by the detection message	
type	the kind of control message being sent:	
	repeat-probe-signals	used to trigger a fresh set of probe-messages
	probe messages	used to detect termination
	update messages	used to update the state of neighbor processes
F1	a flag to verify termination 0 - it is okay to terminate, probe not falsified 1 - it is not okay to terminate, probe has been falsified	

Whenever process p_i becomes passive, the following piece of code is executed:

```
state_i(p_i) := passive
for each p_j ∈ {neighbors of p_i} do
    if p_j ∈ s1_i then
        augment list1_i by p_j
    else
        augment list2_i by p_j
    if list1_i nonempty then
        msg.type := update
        msg.list := list1_i
```

```
            send (pred(p_i), msg, p_i)
        if list2_i nonempty then
            msg.type := update
            msg.list := list2_i
            send (succ(p_i), msg, p_i)
        clear list1_i
        clear list2_i
```

Whenever p_i receives a message of type = update from process p_j the following code is executed:

```
if ¬b_i then
    wait until b_i
if p_i ∈ msg.list then
    state_i(p_j) := passive
    remove p_i from msg.list
    if msg.list empty then
        purge the update message
        if state_i(p_k) = passive for all p_k ∈ {neighbours of p_i} and state_i(p_i) = passive then
            seq(p_i) := seq(p_i) + 1
            msg.seqdm := seq(p_i)
            msg.type := probe
            msg.F1 := 0
            send(pred(p_i), msg, p_i)
            send(succ(p_i), msg, p_i)
    else
        send(next(p_i), msg, p_i)
else
    send(next(p_i), msg, p_i)
```

Whenever p_i receives the probe from p_j the following code is executed:

```
if b_i and state_i(p_k) = passive for all p_k ∈ {neighbors of p_i} then
    if pmseq_i(p_j) = msg.seqdm then
        if status_i(p_j) = 0 then
            enter termination phase
        else
            purge the detection message
    else
        pmseq_i(p_j) := msg.seqdm
        status_i(p_j) := 0
        send (next(p_i), msg, p_i)
else
    pmseq_i(p_j) := msg.seqdm
    purge the detection message
    status_i(p_j) := 1
```