

---

Dissertations, Theses, and Masters Projects

Theses, Dissertations, & Master Projects

---

1997

## Global state predicates in rough real-time

Jean Ann Mayo

*College of William & Mary - Arts & Sciences*

Follow this and additional works at: <https://scholarworks.wm.edu/etd>



Part of the [Computer Sciences Commons](#)

---

### Recommended Citation

Mayo, Jean Ann, "Global state predicates in rough real-time" (1997). *Dissertations, Theses, and Masters Projects*. Paper 1539623907.

<https://dx.doi.org/doi:10.21220/s2-hst4-gb04>

This Dissertation is brought to you for free and open access by the Theses, Dissertations, & Master Projects at W&M ScholarWorks. It has been accepted for inclusion in Dissertations, Theses, and Masters Projects by an authorized administrator of W&M ScholarWorks. For more information, please contact [scholarworks@wm.edu](mailto:scholarworks@wm.edu).

## **INFORMATION TO USERS**

**This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.**

**The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.**

**In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.**

**Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.**

**Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.**

# **UMI**

**A Bell & Howell Information Company  
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA  
313/761-4700 800/521-0600**



**GLOBAL STATE PREDICATES  
IN ROUGH REAL TIME**

**A Dissertation**

**Presented to**

**The Faculty of the Department of Computer Science**

**The College of William and Mary in Virginia**

**In Partial Fulfillment**

**Of the Requirements for the Degree of**

**Doctor of Philosophy**

**by**

**Jean Mayo**

**1997**

**UMI Number: 9815241**

**Copyright 1998 by  
Mayo, Jean Ann**

**All rights reserved.**

---

**UMI Microform 9815241  
Copyright 1998, by UMI Company. All rights reserved.**

**This microform edition is protected against unauthorized  
copying under Title 17, United States Code.**

---

**UMI**  
**300 North Zeeb Road**  
**Ann Arbor, MI 48103**

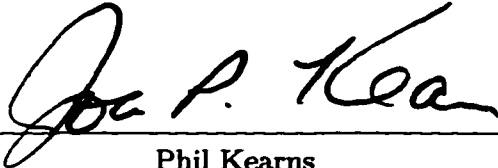
## APPROVAL SHEET

This dissertation is submitted in partial fulfillment  
of the requirements for the degree of

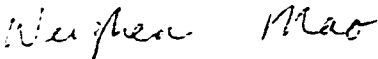
Doctor of Philosophy

  
\_\_\_\_\_  
Jean Mayo

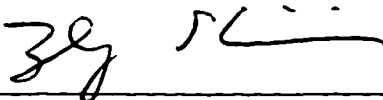
Approved August, 1997

  
\_\_\_\_\_  
Phil Kearns  
Thesis Advisor

  
\_\_\_\_\_  
William Bynum

  
\_\_\_\_\_  
Weizhen Mao

  
\_\_\_\_\_  
Paul Stockmeyer

  
\_\_\_\_\_  
Shiwei Zhang  
Department of Applied Science

## DEDICATION

I dedicate this dissertation to husband, Kurt Joachim, and to my parents, Alton and Ruth Mayo.

# Contents

<b>List of Figures</b>	<b>vii</b>
<b>Acknowledgements</b>	<b>x</b>
<b>Abstract</b>	<b>xi</b>
<b>1 Introduction</b>	<b>2</b>
1.1 Introduction . . . . .	2
1.2 Event Ordering . . . . .	4
1.3 Global Predicate Evaluation . . . . .	7
1.4 Thesis Outline . . . . .	10
<b>2 Applying Roughly Synchronized Clocks</b>	<b>13</b>
2.1 System Model . . . . .	14
2.2 Clock Synchronization . . . . .	15
2.3 Causal Ordering . . . . .	21
2.4 Related Work . . . . .	24
2.5 Synchronous and Globalized Local Properties . . . . .	27
2.5.1 Definitions . . . . .	27



2.5.2	Properties of SLPs and GLPs . . . . .	29
<b>3</b>	<b>Stable Predicates</b>	<b>34</b>
3.1	Distributed Termination . . . . .	35
3.1.1	Problem . . . . .	35
3.1.2	A Simple System . . . . .	36
3.1.2.1	Solution Without Clock Synchronization . . . . .	37
3.1.2.2	Solution with Perfect Clock Synchronization . . . . .	38
3.1.2.3	Solution With Globally Timestamped Properties . . . . .	43
3.1.2.4	Efficiency Considerations . . . . .	50
3.1.3	General Termination Detection . . . . .	55
3.1.3.1	Previous Work . . . . .	55
3.1.3.2	Asynchronous Communication . . . . .	60
3.1.3.3	Token Broadcasts . . . . .	70
3.1.3.4	Tree-Based Routing . . . . .	73
3.1.3.5	Reduced Active Process Interruption . . . . .	76
3.1.4	Conclusions . . . . .	83
3.2	Distributed Deadlock . . . . .	83
3.2.1	System Model . . . . .	84
3.2.2	Problem . . . . .	87
3.2.3	Previous Work . . . . .	88
3.2.4	Algorithm . . . . .	92
3.2.5	Discussion . . . . .	102
3.3	Global Snapshots . . . . .	103

3.3.1	Problem . . . . .	103
3.3.2	Previous Work . . . . .	105
3.3.3	Discussion . . . . .	106
3.4	Conclusions . . . . .	107
<b>4</b>	<b>Unstable Predicates</b>	<b>108</b>
4.1	Problem . . . . .	109
4.2	Previous Work . . . . .	115
4.3	Scheduled Evaluation . . . . .	120
4.3.1	Conjunctive Process State Predicates . . . . .	121
4.3.2	General Predicates . . . . .	126
4.3.3	Discussion . . . . .	129
4.4	Centralized Evaluation . . . . .	130
4.5	Conclusions . . . . .	145
<b>5</b>	<b>Conclusions</b>	<b>146</b>
5.1	Summary . . . . .	146
5.2	Directions for Further Research . . . . .	150
<b>A</b>	<b>Unstable Predicate Evaluation Using GLPs</b>	<b>153</b>

# List of Figures

1.1	Causal Ordering . . . . .	5
1.2	Consistent Global States . . . . .	7
1.3	Simultaneous Snapshots . . . . .	9
2.1	Lamport Logical Clocks . . . . .	21
2.2	Mattern's Vector Clocks . . . . .	23
2.3	Local Significance of SLP Timestamps . . . . .	28
2.4	Process State and Protocol for Globalizing a Time-stamped Local Property	30
2.5	Overlap of SLPs with Unequal Timestamps . . . . .	31
3.1	Protocol ( <i>TD-Rana</i> ): Rana's Termination Detection Protocol . . . . .	39
3.2	Failure of Rana's Termination Detection Protocol . . . . .	43
3.3	Protocol ( <i>TD-GLP</i> ): TD with Rough Clock Synchronization . . . . .	44
3.4	Protocol ( <i>TD-SLP</i> ): Efficiency Modification to Protocol <i>TD-GLP</i> . . . . .	51
3.5	Successive Control Waves for a Four Process System . . . . .	57
3.6	Mattern's Logical Clock Protocol for Termination Detection . . . . .	59
3.7	Protocol ( <i>TD-Asynch</i> ): TD Under Asynchronous Communication . . . . .	61
3.8	Protocol ( <i>TD-Bnd</i> ): TD With Bounded Message Transmission Time . . . . .	65

3.9 Protocol ( <i>TD-Bcast</i> ): TD with Broadcasts . . . . .	69
3.10 Tree Token Routing . . . . .	73
3.11 Protocol ( <i>TD-Tree</i> ): TD using Tree Routing . . . . .	74
3.12 Protocol ( <i>TD-Quiet</i> ): TD with Reduced Control Communication . . . . .	77
3.13 Transaction Wait For Graph (WFG) . . . . .	87
3.14 Mensace and Muntz Protocol Example - Initial Resource Allocation . . . . .	89
3.15 Mensace and Muntz Protocol Example - Requests, Local Blocking Pairs . . . . .	89
3.16 Mensace and Muntz Protocol Example - Final TWF Graphs . . . . .	90
3.17 Protocol ( <i>DD-Oneres</i> ): Transaction Manager State and Protocol . . . . .	93
3.18 Protocol ( <i>DD-Oneres-Dm</i> ): Data Manager Process State and Protocol . . . . .	94
3.19 Execution with Time Driven Global State Predicate Changes . . . . .	104
4.1 Varying Timelines for Two Process Execution History . . . . .	110
4.2 Consistent Global State Lattice . . . . .	114
4.3 Examples of lcmvectors . . . . .	117
4.4 Protocol ( <i>Sched-Conj</i> ): Scheduled Conjunctive Predicate Evaluation . . . . .	121
4.5 Real Time $\epsilon$ Interval Overlap . . . . .	127
4.6 Protocol ( <i>Sched-Gen</i> ): Scheduled Predicate Evaluation . . . . .	129
4.7 Protocol ( <i>Unstable-App</i> ): Global Predicate Detection - Application Process . . . . .	130
4.8 Protocol ( <i>Unstable-Mon</i> ): Global Predicate Detection - Monitor Process . . . . .	132
4.9 Exemplary Execution for Protocols <i>Unstable-Mon</i> and <i>Unstable-App</i> . . . . .	139
4.10 Execution of Protocol <i>Unstable-Mon</i> . . . . .	140
A.1 Protocol ( <i>UnstableGLP-App</i> ): Modification to Protocol <i>Unstable-App</i> . . . . .	154

**A.2 Protocol (*UnstableGLP-Mon*): Modification to Protocol *Unstable-Mon* . . . 155**

## ACKNOWLEDGMENTS

The author wishes to thank her advisor, Dr. John P. Kearns, for his guidance and support. She also thanks her friend, Mary Holup, for her patience.

# ABSTRACT

Distributed systems are characterized by the fact that the constituent processes have neither common memory nor a common system clock. These processes communicate solely via message passing. While providing a number of benefits such as increased reliability, increased computational power, and geographic dispersion, this architecture significantly complicates many of the tasks of software development and verification, including evaluation of the program state. In the case of distributed systems, the program state is comprised of the local states of the constituent processes, as well as the state of the channels between processes, and is called the *global state*.

With no common system clock, many distributed system protocols rely on the global ordering of local process events imposed by the message passing that occurs between processes. This leads to a partial global ordering of local process events, which can then be used to determine which process states could (or could not) have occurred simultaneously, e.g. determine which global states are *consistent*.

Traditional predicate evaluation protocols evaluate predicates on the global state of a distributed computation using consistent global states. This evaluation is complicated by the fact that the event ordering imposed by message passing is only partial. A complete history of the global states that occurred during an execution cannot always be constructed. This introduces inefficiency into predicate detection protocols and prohibits detection of certain predicates.

The assumption that no global time base exists because no global system clock exists is overly restrictive in certain cases. A number of fault-tolerant clock synchronization protocols have appeared in the literature. These protocols keep the difference in process clock readings at any instant within some known bound, creating a rough global time base for the distributed system.

This dissertation explores the use of this rough global time base for global state predicate evaluation within distributed systems. By structuring the evaluation on the assumption that a global time base exists, we can develop simple and efficient protocols for both stable and unstable predicate evaluation. Further, we can evaluate certain predicates which are not easily evaluated using consistent global states. We demonstrate these advantages by developing protocols for detection of distributed termination, distributed deadlock detection, and detection of certain unstable predicates as they occur. As the global time base is rough, we can only detect unstable predicates which remain true for a sufficient duration. We additionally develop several formalizations which assist the protocol developer in dealing with the fact that the global time base is not perfect. We demonstrate the application of these formalizations within the protocols that we develop.

**GLOBAL STATE PREDICATES  
IN ROUGH REAL TIME**



# Chapter 1

## Introduction

### 1.1 Introduction

Distributed systems are characterized by the fact that the constituent processes have neither a common memory nor a common system clock. Processes are connected via some communication medium, or *channel*. The processes then communicate solely via message passing. This system architecture provides a number of benefits.

- Distributed systems can perform computations redundantly. This increases overall system reliability. This type of distributed system is often used for life critical control system applications. Fly-by-wire aircraft control systems are an example. In this type of control system, several microprocessors perform identical computations using redundant inputs. The processors then vote on the output control signal, voting out any processor which has strayed too greatly from the others [2].
- Distributed systems can be geographically dispersed. This allows remote access to centralized services, such as access to a file system maintained by a library. These are

often client-server applications. It also allows a number of geographically dispersed resources to be connected into a single application, such as banking transaction management.

- Finally, distributed systems can provide greater computational power than that provided by a single computer. An example of this type of system is message-passing multiprocessors. These are computers which are comprised of a number of connected, but autonomous, microprocessors, which are used concurrently for parallel computations.

While providing a number of benefits, this system architecture significantly complicates the task of software development and verification. One of the fundamental problems is that no global state exists naturally in these systems. It must be constructed via message passing.

Knowledge of the system state, and the ability to evaluate predicates on that state, are required for application development on any system architecture. For instance, production computations are often monitored through the use of assert macros to assure continued operation in accordance with the specification [22]. The ability to set breakpoints and examine the current program state is fundamental to program debugging. Application control often requires the ability to detect certain system states, such as termination, deadlock, or the loss of a token.

In a sequential program, construction of the global state is trivial. It is simply a snapshot of all program variables' values, including implicit variables, such as a program counter and a stack pointer, at some instant in time. In a distributed system, however, the program state is not so readily available due to the lack of a common time base and the physical

dispersion of program components.

For example, consider a system of  $N$  processes. If the process clocks are perfectly synchronized, we can easily construct a global state. Each process just takes a snapshot of its state at some agreed-upon clock value  $T$ . These local states can then be accumulated, via message passing, into a snapshot of the global system state at the instant all process clocks read  $T$ . However, if the clocks are not synchronized, a global state constructed in this manner is meaningless. Each process clock may have read  $T$  at different instants in real time.

## 1.2 Event Ordering

In the absence of a global time base, message passing is the primary mechanism for ordering local events globally. We know that a message takes non-zero time to travel from one process to another. We then know that any event prior to, and including, a message send in one process occurred before any event after, and including, the corresponding message receipt in another process. If it weren't for the ordering imposed by message passing, there would be no way to determine when the events of one process occurred in relation to the events of another process (unless the execution were artificially controlled in some way).

Lamport formalized this notion as the "happens before" relation " $\rightarrow$ ". He defined it as the smallest relation satisfying the following three conditions [34].

1. If  $a$  and  $b$  are events in the same process, and  $a$  comes before  $b$ , then  $a \rightarrow b$ .
2. If  $a$  is the sending of a message by process  $P_i$  and  $b$  is the receipt of that message by process  $P_j$  then  $a \rightarrow b$ .

3. If  $a \rightarrow b$  and  $b \rightarrow c$ , then  $a \rightarrow c$ .

If  $a \not\rightarrow b$  and  $b \not\rightarrow a$ , then  $a$  and  $b$  are said to be *concurrent*,  $a \parallel b$ . Concurrent events *could* have occurred at the same time instant during the execution. This type of ordering is sometimes referred to as the *causal order*, since we can determine when one event  $a$  might have an impact on (caused) another event  $b$  based on this ordering.

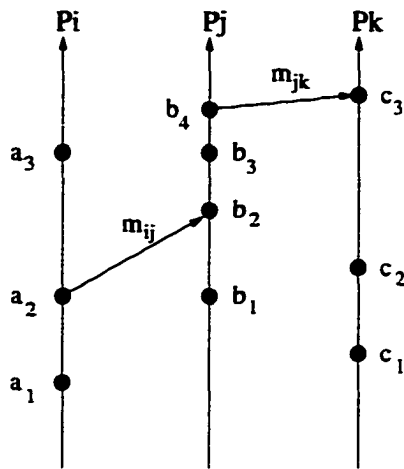


Figure 1.1: Causal Ordering

An example is shown in figure 1.1. Here the events executed in processes  $P_i$ ,  $P_j$ , and  $P_k$  are shown in their order of occurrence. We know that real time increases along each axis of the figure. However, in the absence of a global time base, we don't know the when the events in one process occurred relative to the events in other processes, other than the fact that we know messages won't travel backwards in time. Each axis may be scaled on a different and varying *real* time scale. The time scale is irrelevant to this event ordering. We could expand or shrink any process' time line, so long as the local event order was maintained and messages didn't travel backward in time, and the causal event ordering would be unchanged.

We then know that event  $a_2$  must occur before event  $b_2$ , as message transmission requires

nonzero real time. Thus  $a_1 \rightarrow b_2$ . Since we know that events can only occur sequentially in any single process, we also know that  $a_1$  occurs before  $b_3$ . Finally, we can say nothing definitive about the order of occurrence of event  $c_2$  in relation to any of the events in processes  $P_i$  and  $P_j$ . Event  $c_2$  is then *concurrent* with each of the events in processes  $P_i$  and  $P_j$ .

This ordering underlies the notion of a *consistent global state* [43]. A consistent global state is a set of local process states, one from each process, and channel states, one from each channel, which *could* have occurred at some instant in real time during the execution. Most predicate evaluation algorithms evaluate predicates over consistent global states.

A consistent global state can be constructed by taking a *cut* through the ordered sets of events, one from each process and channel, comprising an execution. All events leading to the states contained in the cut are part of the cut. The cut is then consistent if, for any event  $b$  in the cut, if  $a \rightarrow b$  then  $a$  is also in the cut. This is illustrated in figure 1.2. Cut  $A$  is inconsistent because a state in process  $P_k$  in which message  $m$  has been received cannot have occurred at the same time as a state in process  $P_i$  in which message  $m$  has not yet been sent.

Henceforth we use the term “instantaneous” global state synonymously with global state to refer to a set of process states, one from each process, which occurred at the same time instant. This emphasizes its difference from a consistent global state, which may or may not have ever occurred during the execution. Note that an instantaneous global state is consistent, but a consistent global state is not necessarily an instantaneous global state.

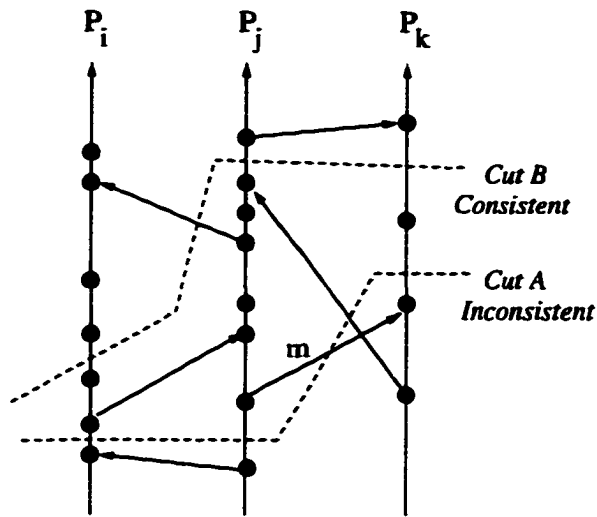


Figure 1.2: Consistent Global States

### 1.3 Global Predicate Evaluation

Since a consistent global state may, or may not, have ever occurred during the computation, it is impossible to detect certain predicates using only consistent global states. For example, suppose the following code segments run on two processors  $P_i$  and  $P_j$  simultaneously. Further suppose that we want to detect the predicate "Variable  $y$  has the value 3 in all processes".

$P_i$ : $y=3$ input( $x$ ) if ( $x = \text{TRUE}$ ) $y=5$	$P_j$ : input( $y$ ) $z=y+3$ $y=3$
---	---

Now suppose we observe the following states of each process for some execution.

$P_i$ : $y=3$ $x=\text{TRUE}$ $y=5$	$P_j$ : $y=2$ $z=5$ $y=3$
--	------------------------------------

In the absence of communication, each state in  $P_i$  is concurrent with every state in  $P_j$ ,

and vice-versa. Based on this knowledge alone, we cannot determine whether or not the predicate ever held at some real time instant, only that it could have.

If we had a perfect global time base then we would know the time at which each of these events occurred. We could then determine whether or not the predicate held at some point. For example, suppose that each process clock reads integer values and we observed the following states.

$C_i = 1 : y=3$	$C_j = 1 : y=2$
$C_i = 5 : x=TRUE$	$C_j = 2 : z=5$
$C_i = 6 : y=5$	$C_j = 3 : y=3$

With the additional information about the system's behavior in an absolute time frame, we can determine that the predicate was never true.

Construction of a consistent global state can also be complex. For example, consider Lai and Yang's algorithm for global snapshots [62]. According to their protocol, each process takes a snapshot at its convenience, but adheres to the following rules.

- Every process is initially white and turns red when it takes a snapshot.
- Every message sent by a red(white) process is colored red(white).
- A white process must take a snapshot before it receives a red message. (Thus, the arrival of a red message will cause a white process to take a snapshot.)

This scheme is more complex than the method we outlined earlier, by which a global snapshot is taken by having each process take its state at an agreed upon time instant.

Further, it would be more complex to have multiple snapshots in progress. For example, consider the execution depicted in figure 1.3. Here, process colors, as indicated by line type, are shown as a function of real time for a two process system. In order to have multiple

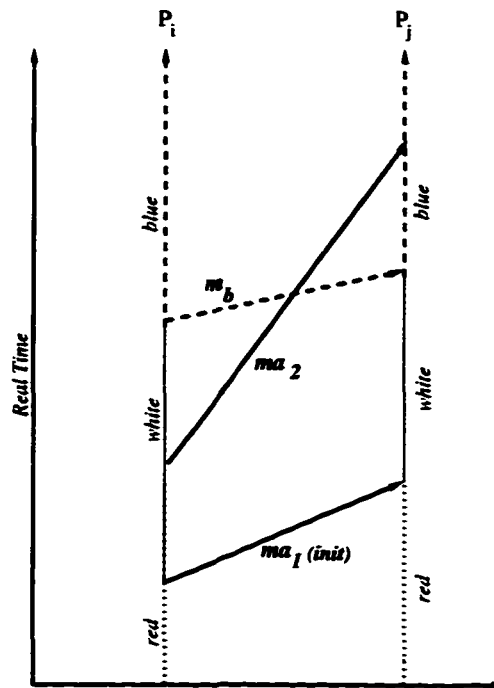


Figure 1.3: Simultaneous Snapshots

snapshots in progress, each process would have to keep track of what colors it had already been so that, upon receipt of a colored message, the process knows whether or not it has already taken its state. For example, in the figure, process  $P_j$  receives two messages which are colored white. Further,  $P_j$  is a color other than white at the time the messages are received. In the absence of other information,  $P_j$  would have to assume that it should take its state when it receives each of these messages. A naive approach to remedy this problem would be to suggest that we ensure that each process receives a unique snapshot “initiation” message. Then processes only take their state on receipt of an “initiation” message. This is shown in figure 1.3. When  $P_j$  receives message  $ma_1(init)$ , it knows, by virtue of the fact that the message contains the *init* field, that it should take its state. Similarly, the absence of the *init* field in message  $ma_2$ , lets  $P_j$  know that it need not take its state, and should forward the received message to the collector. In this case, though, messages colored



with a color the process has not yet been, would have to be buffered until the snapshot initiation message is received. For example, if  $P_j$  received  $ma_2$  before it received message  $ma_1$ , message  $ma_2$  would have to be buffered until  $P_j$  takes its state, to avoid an inconsistent snapshot. Thus, the process would again have to know what color it had been so that it knows whether or not the message should be buffered. If snapshots are “instantaneous”, these ordering problems can be avoided.

## 1.4 Thesis Outline

The assumption that no global time base exists is, in certain cases, more stringent than is necessary. Although it is impossible to achieve perfect clock synchronization in a distributed system [67], clocks are commonly roughly synchronized. By roughly synchronized, we mean that the difference between any two system clocks at some instant in time is always within some known fixed bound. If we denote the reading of  $P_i$ 's process clock at real time instant  $t$  by  $C_i(t)$ , this is more formally stated as follows:

$$|C_i(t) - C_j(t)| \leq \epsilon \text{ for all } t.$$

Although roughly synchronized clocks have been applied toward a number of distributed system problems [36, 54], their application toward global predicate evaluation has received little attention to date [68].

Our thesis investigates the use of a rough global time base in global predicate evaluation. We examine both how a global time base can be used to advantage in predicate evaluation and propose ways to deal with the fact that the clock synchrony is rough.

We will show that algorithms structured on the use of a rough global time base can provide several advantages over algorithms structured on the use of consistent states.

- Using roughly synchronized clocks, we can develop simple and efficient stable predicate evaluation algorithms. Evaluating predicates over a consistent state is sufficient for stable predicates. However, the use of a rough common time base can lead to simple, efficient, and flexible algorithms for evaluation of stable predicates in certain types of distributed systems.
- Using roughly synchronized clocks we can readily evaluate certain time-based predicates. Algorithms structured on the use of consistent states cannot easily answer questions like “Were all the valves closed by two o’clock?” or “At what time did the system terminate?” Evaluation of these types of predicates is important for distributed monitoring systems.
- Using roughly synchronized clocks, we can reduce the amount of computation required for post mortem analysis of a given execution. When clocks are roughly synchronized, we can significantly reduce the number of consistent states. Any local process states which occurred at clock readings that differ by more than  $\epsilon$ , the maximum difference in clock readings at a real time instant, cannot have occurred simultaneously during the computation.
- Using roughly synchronized clocks we can detect, with certainty, the truth of certain unstable predicates which are undetectable using consistent global states. Clearly, the use of roughly synchronized clocks in detecting unstable predicates is limited. We cannot detect all unstable predicates. However, we can detect unstable predicates

which remain true for a sufficient duration.

In chapter 2, we discuss clock synchronization and related work. We then develop some general techniques for applying roughly synchronized clocks in global predicate evaluation. In chapter 3, we demonstrate the use of these techniques by applying them to the evaluation of stable predicates. In chapter 4, we discuss both runtime and post mortem evaluation of unstable predicates. Finally, in chapter 5, we give our conclusions.

## Chapter 2

# Applying Roughly Synchronized Clocks

In this chapter we present a foundation for applying roughly synchronized clocks to predicate evaluation. First, we develop the system model that we will use throughout the remainder of the thesis. We then discuss the basis for structuring predicate evaluation around a rough global time-base: clock synchronization. We address the immediate questions of reliability and achievable clock skews. In section 2.3, we discuss the traditional asynchronous foundation for structuring predicate evaluation protocols: causal event ordering. A thorough understanding of causal event ordering is required to put our work into context. In section 2.4, we then discuss related work. Finally, in section 2.5, we develop constructs to facilitate development of protocols which assume the existence of a rough global time base.

## 2.1 System Model

We consider a distributed system to be a set of reliable processes  $\{P_0, \dots, P_{n-1}\}$ . These processes share no memory and communicate solely via message passing. As a notational convenience we denote the set of system process indices by  $SYS$ . Each process is assumed to have access to a local clock.

Each process has a local state which changes as a result of the actions which the process performs. Processes perform two kinds of actions: internal actions, which change the local state, and message send or receive actions. Both a message send and receive are performed as atomic actions, along with any associated change to the process state. The term *action* is used synonymously with *event*.

Each process is also assumed to have access to a local clock. The clock for  $P_i$  is represented by the nondecreasing real valued function  $C_i$ , where  $C_i(t) = T$  is the time on  $P_i$ 's clock at real time instant  $t$ . Throughout this work we adopt the notational convention that real times are denoted by lowercase letters and process clock values are denoted by uppercase letters.  $C_i^{-1}(T)$  represents the interval of real time instants at which  $C_i$  reads the value  $T$ .  $C_i(a)$  denotes the value of  $C_i$  when event  $a$  is executed by  $P_i$ .

$C_i(t)$  is assumed to be a nondecreasing function of real time with sufficient resolution to distinguish between any two actions by  $P_i$ . The clocks are assumed to be roughly synchronized within some known bound  $\epsilon$ . System clocks also have an associated *drift rate*  $\rho$  which is the rate of drift of clock time from real time. The drift rate of  $C_i$  is given by  $\rho_i$ . We assume that these drift rates are bounded by some maximum  $\rho_M$ .

We state the clock assumptions formally in our *clock axioms*.

- C1. For all  $i, j \in SYS$  and for all  $t \geq 0$ ,  $|C_i(t) - C_j(t)| < \epsilon$ .
- C2. For all  $i$  in  $SYS$ , if  $C_i(t) > C_i(t')$  then  $t > t'$  and if  $t \geq t'$  then  $C_i(t) \geq C_i(t')$ .
- C3. For all  $i$  in  $SYS$ , there exists  $\rho_M \ll 1$  such that  $(1 - \rho_M)(t' - t) \leq C_i(t') - C_i(t) \leq (1 + \rho_M)(t' - t)$ .
- C4. If  $a$  and  $b$  are events within process  $P_i$  then  $C_i(a) \neq C_i(b)$ .

In axiom C3, we assume that the error caused by the discrete clock granularity is negligible compared to that due to drift. Typical values of the constant  $\rho$  for quartz clocks are on the order of  $10^{-6}$ . Thus, throughout the paper, we ignore terms on the order of  $\rho^2$  or smaller.

A *process execution sequence*  $S_i$  is the ordered set of events that occur within process  $P_i$  during a given execution. An execution sequence  $S$  is the set of process execution sequences which comprise a particular execution; i.e.,  $S = \{S_i, i \text{ in } SYS\}$ . A *process time history* maps a process execution sequence into an absolute real time frame. It is the ordered set of pairs  $(a, C_i(a))$ , where one component of each pair corresponds to an event  $a \in S_i$ . Similarly, an execution time history is comprised of the set of process time histories, one from each process in the system, for a particular execution. Thus, two execution sequences can be identical; however, if the actions comprising the sequences occur at different instants of real time, the associated time histories are different.

## 2.2 Clock Synchronization

Clock synchronization is done in hardware [28, 57, 64], software [21, 38, 37, 61], and hybrid combinations of hardware and software [53]. The bound on clock skew depends on which

of these methods is employed.

Hardware solutions operate in a phase-locked loop. The hardware clock at each node is the output from a voltage-controlled oscillator. The voltage applied to the oscillator is proportional to the phase error between its own clock and a reference signal generated from the other system clocks. Since all clocks adjust to this reference signal, the clocks are kept in synchrony. Hardware solutions are typically able to provide skews on the order of tens of nanoseconds [64]. However, hardware schemes are often prohibitively expensive, especially in large distributed systems. Further, these schemes are impractical in systems which are physically dispersed.

Software solutions work by maintaining a logical clock in addition to the hardware clock. The logical clock is synchronized to the other logical clocks and provides the time base for activities at that node. A synchronization algorithm runs periodically at each node and is responsible for updating the logical clock.

As an example, we consider a software synchronization scheme proposed by Cristian [10]. The problem, obviously, in software clock synchronization is the variability in message delay. His scheme is based on the observation that the error in  $P_0$ 's reading of  $P_1$ 's clock at some real time instant is a function of the round trip delay of the message used by  $P_0$  to obtain  $P_1$ 's clock value. Since the value of any single observed delay lies on some distribution of all possible delay times,  $P_0$  can make repeated readings of  $P_1$ 's clock to attempt to get a reading closer to the minimum message delay, thus reducing the reading error.

For example, suppose that process  $P_0$  reads the clock of process  $P_1$  by sending a message requesting the current time. When  $P_1$  receives the message, it responds with its current clock value. Let  $D$  be the round trip delay between the sending of the initial message and

the reception of the reply as measured by the hardware clock at  $P_0$ . Here the value on the hardware clock is represented by the function  $H_i(t)$ . This hardware clock function is assumed to obey clock axiom C3. Cristian observed that  $H_1(t)$ , where  $t$  is the instant that  $P_0$  received  $P_1$ 's reply, is somewhere on the interval

$$[T + \mathbf{min} * (1 - \rho_M), T + 2D(1 + 2\rho_M) - \mathbf{min} * (1 + \rho_M)],$$

where  $\mathbf{min}$  is the minimum message transmission time, and  $T$  is the clock value contained in  $P_1$ 's reply message. Thus,  $P_0$  can determine the interval which contains  $P_1$ 's clock value by *measuring* the round trip delay  $2D$ . If  $P_0$  then assumes that  $C_1(t)$  is the midpoint of this interval, the maximum error  $e$  of  $P_0$ 's estimation of  $C_1$ 's value is

$$e = D(1 + 2\rho) - \mathbf{min}.$$

Then the smaller the round trip delay is, the smaller  $P_0$ 's error in reading  $P_1$ 's clock. Thus, if  $P_0$  wants a minimum reading error of  $\varepsilon$  then it must *discard* any reading attempt for which it measures a round trip delay greater than  $2U$ , where

$$U = (1 - 2\rho_M)(\varepsilon + \mathbf{min}).$$

Any round trip delay smaller than  $2U$  is successful, and  $2U$  is referred to as the *timeout delay* necessary for achieving reading precision  $\varepsilon$ . When  $P_0$  observes a successful round trip, it has reached *rapport* with  $P_1$ .

Let  $p$  is the probability that  $P_0$  observes a round trip delay greater than the timeout



delay  $2U$ . In order to avoid  $P_0$  attempting to read  $P_1$ 's clock ad infinitum, a maximum value  $k$  for number of successive attempts must be chosen. Then the probability of success within  $k$  attempts is

$$1 - p^k$$

and the average number of messages  $\bar{n}$  for achieving rapport is

$$\bar{n} = \frac{2}{1 - \rho_M}.$$

Cristian's protocol assumes the existence of a unique, continuously available, master time source. This might be implemented using a radio receiver, which receives Universal Time Coordinated (UTC) broadcasts from the National Bureau of Standards. The receivers can be attached to processors via dedicated busses. *Masters* have access to this external time and synchronize to it. Slaves then synchronize with the masters, in a similar fashion.

A slave synchronizes with the master by periodically attempting to reach rapport. Each attempt at rapport is comprised of at most  $k$  attempts to read the master's clock. Successive read attempts are separated by  $W$  clock time units, where  $W > 2U$ . If all  $k$  attempts fail, then the slave must leave the group of synchronized slaves. When rapport is reached, the speed of the slave's logical clock  $C_0$  is set according to the relation

$$C_0(t) \equiv H_0(t) + A(t).$$

The adjustment function must avoid logical discontinuities. He considers the linear adjust-

ment function

$$A(t) = m * H_0(t) + N,$$

where  $m$  and  $N$  are computed periodically. If, at local rapport time  $\mathcal{L}$ , a slave estimates that the master clock displays time  $\mathcal{M}$ , the speed of the slave clock must be adjusted so that it shows time  $\mathcal{M} + \alpha$ , instead of  $\mathcal{L} + \alpha$ ,  $\alpha$  time units after the rapport. Here  $\alpha$  is the *amortization* parameter. Then, since the slave clock shows the value  $\mathcal{L} = H_0(t)(1 + m) + N$  at the beginning of the amortization interval and the value  $(H_0(t) + \alpha)(1 + m) + N$  at the end of the amortization interval, then

$$m = (\mathcal{M} - \mathcal{L})/\alpha, \text{ and } N = \mathcal{L} - (1 + m)H_0(t)$$

for the  $\alpha$  time units after rapport. The slave clock can be allowed to run at the speed of the local clock between the end of the amortization interval and the time of the next rapport. The maximum difference  $ms$  between  $C_0$ , the slave logical clock, and  $C_1$ , the master logical clock, is given by the relation

$$ms \geq U - \mathbf{min} + \rho_M * k(1 + \rho_M)W.$$

Thus, for some choice of  $U$ ,  $k$ , and  $W$ , the smallest master slave maximum deviation that can be achieved is

$$ms_{\min} \geq U - \mathbf{min} + \rho_M * k(1 + \rho_M)W.$$

Thus, for aggressive algorithms for which  $U$  is close to  $\mathbf{min}$ , deviations close to  $\rho_M * k(1 + \rho_M)W$  can be achieved. If  $U$  is chosen to be close to the maximum message delay, thus

ensuring that rapport is always reached, then the deviation is close to  $(\max - \min)$ , where  $\max$  is the maximum message delay.

There are many methods for synchronizing clocks in software, and the characteristics of the clock skew bound varies accordingly. The algorithms can largely be classified into those which guarantee synchronization in the presence of arbitrary failures, and probabilistic algorithms, such as Cristian's protocol, by which there is a non-zero probability that the clocks will lose synchronization. The clock skew achievable by each approach varies greatly. The algorithms which guarantee that synchronization is maintained have larger skews than probabilistic algorithms. However, several of the algorithms which guarantee the synchronization [61, 21] are able to provide clock skews which are at least on the order of the message passing delay, i.e., tens of milliseconds [55]. (This would be close to  $(\max - \min)$ , the "safe" delay for Cristian's protocol.) However, these algorithms require that message delay be bounded and that this bound be known. Probabilistic schemes do not require bounded message delay. Further, they are able to provide even tighter skews, significantly less than the median message delay, as we showed above.

Ramanathan proposes using a combination of software and inexpensive hardware [53]. His protocol is based on the observation that the variability in message passing delay can be significantly reduced by timestamping synchronization messages in hardware prior to their use by the synchronization software. His algorithm guarantees that synchronization is maintained and yet is able to provide worst case skews two to three orders of magnitude tighter than software schemes which guarantee synchronization maintenance. For example, on a 512 node hypercube, allowing two faults, they were able to achieve a worst case skew of 200 microseconds, even when the maximum message transit delays were as large as 50

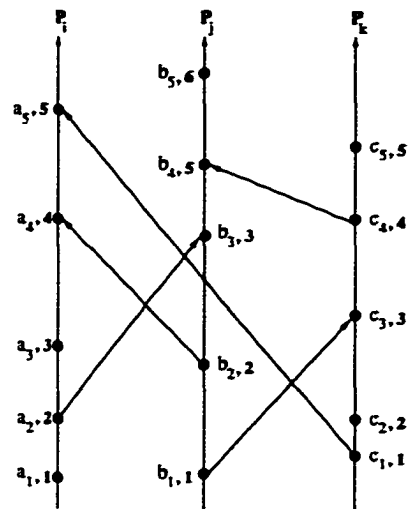


Figure 2.1: Lamport Logical Clocks

milliseconds.

Thus, we can assume that fault-tolerant clock synchronization is available and that clock skews at least on the order of message passing delays can be practically obtained.

### 2.3 Causal Ordering

We have already formally defined Lamport's "happens before" relation, which is the basis for ordering events so that we can partially determine the real time ordering of system events. In this section, we look at two well-known protocols for ordering the events within a given execution according this relation. We also show how this event ordering can be used to determine whether a set of channel and local process states, one from each process and channel, could have occurred simultaneously during the execution, i.e., establish whether a set of local process and channel states is consistent.

There are two well-known approaches to establishing the "happens before" relation on the events of a particular execution: Lamport's logical clocks [34] and Mattern's vector time

[43].

Lamport's logical clocks satisfy the following condition.

For any events  $a, b$  : if  $a \rightarrow b$  then  $L(a) < L(b)$ , where  $L_i$  is a clock function which assigns a number  $L_i(a)$  to event  $a$ ; if  $a$  is an event at  $P_i$  then  $L_i(a) = L(a)$ .

In order to satisfy this condition, the clocks at each process must obey the following implementation rules:

LIR1. Every  $P_i$ ,  $i$  in  $SYS$ , increments  $L_i$  between any two successive local events.

LIR2. (a) If event  $a$  is the sending of a message  $m$  by process  $P_i$ , then  $P_i$  appends timestamp  $L_i(a)$  to the message.

(b) Upon receiving the message  $m$ ,  $P_j$  sets  $L_j$  greater than or equal to the maximum of the current value of  $L_j$  and the message timestamp.

An example execution is given in figure 2.1. Here we show a three process execution. Event, logical clock value pairs are shown along each axis. It is important to note that Lamport's clock conditions only ensure that if  $a$  happens before  $b$  then  $C(b)$  is greater than  $C(a)$ , but that the converse is not true. For example,  $a_2$  happens before  $b_5$  and  $C(a_2)$  is greater than  $C(b_5)$ . However,  $C(b_5)$  is greater than  $C(c_5)$ , but  $b_5$  is concurrent with  $c_5$ .

Unlike Lamport's logical clocks, Mattern's implementation provides a necessary and sufficient condition to determine the causal relation between any two events. His clock values are vectors with one component for each process. Each local process clock must satisfy the following conditions.

MIR1. A process  $P_i$  increments its component of the local time vector  $V_i$  prior to each local action; i.e.,  $V_i[i] \leftarrow V_i[i] + 1$ .

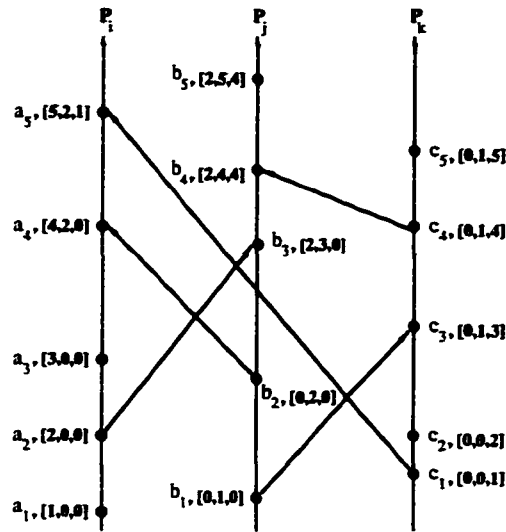


Figure 2.2: Mattern's Vector Clocks

MIR1. If event  $a$  is the sending of a message by process  $P_i$ , then  $P_i$  appends its time vector  $V_i$  to the message. Upon receiving this message, and after incrementing  $V_j[j]$ ,  $P_j$  updates its vector to be the component-wise maximum of its current vector  $V_j$  and the vector timestamp  $V_{msg}$  on the message; i.e.,  $V_j \leftarrow \text{sup}(V_j, V_{msg})$ .

He defines the following operators on the vector clock values.

For any two time vectors  $u$  and  $v$

$$u \leq v \text{ iff } \forall i : u[i] \leq v[i],$$

$$u < v \text{ iff } (\forall i : u[i] \leq v[i]) \wedge (u \neq v), \text{ and}$$

$$u \parallel v \text{ iff } \neg(u < v) \wedge \neg(v < u).$$

Vector clocks implemented according to these rules have the following property.

For any two distinct events  $a$  and  $b$ :  $a \rightarrow b$  iff  $V(a) < V(b)$ , and  $a \parallel b$  iff  $V(a) \parallel V(b)$ .

Figure 2.2 shows the vector clock values for the same execution given in figure 2.1. Here, the clock values reflect the fact that  $a_2 \rightarrow b_5$  and that  $b_5 \parallel c_5$ .

Our goal for predicate evaluation is to determine which sets of channel and process states could have occurred simultaneously during an execution. This can be determined by partitioning the the set of system events into a *consistent cut*. If  $E$  is the set of all system events, then a consistent cut  $\kappa$  is a finite subset of  $E$ , containing at least one event from each process, such that if  $a \in \kappa$  and  $b$  happens before  $a$ , then  $b \in \kappa$  [43]. In other words, any message received before the cut was sent before the cut. A consistent global state is then the state of all channels (those messages which cross the cut line) and the set of local states, one from each process, at the moment the cut event occurs. Cut events do not change the state of a process. Both a consistent and and inconsistent cut are depicted in figure 1.2 in chapter 1.

## 2.4 Related Work

Some groundwork for the development of clock based predicate evaluation is available in the work of Neiger and Toueg [49]. They developed results for a restricted class of predicates called *internal specifications*. Briefly, a predicate is internally specified if it makes no reference to real time. For example, the specification

$$\forall t > 0 \forall i, j \in SYS [ | C_i(t) - C_j(t) | < \epsilon ],$$

which specifies that all system clocks are roughly synchronized, is not an internal specification, because it cannot be stated without reference to real time. Whereas, the specification

$$\forall i \in SYS [a \in \mathcal{S}_i],$$

which specifies that every process eventually executes action  $a$  is an internal specification.

Their result for systems with rough clock synchronization additionally requires that message delivery time be bounded, and that each message be timestamped, by the sender, with the sender's clock value at the time of the send, and that it is held in the receiver's buffer until the receiver's clock has exceeded the message timestamp. They showed that in such systems, if a problem has an internal specification then an algorithm to solve the problem can be developed under the assumption that clocks are perfectly synchronized.

Kopetz studied the application of roughly synchronized clocks to distributed real-time systems. He proposed that significant system events be restricted to the lattice points of a globally synchronized space/time lattice [23].

The goal of this restriction is to ensure the following properties.

P1 All nodes act on different observations in the same order (*consistent order property*).

P2 All nodes act on the same observation at about the same time (*simultaneity property*).

P3 All nodes act on different observations in the temporal order of their occurrence (*temporal order property*).

He defines the precision of the local clocks relative to the number of ticks each one has



gone through. The precision of the clocks is then

$$\Pi = \text{MAX}(\forall j, k \in \text{SYS}, \forall n |z(k_n) - z(j_n)|)$$

where the  $k_n$  denotes the  $n$ th tick of the clock for process  $P_k$  and  $z(k_n)$  is the value of absolute real time when the clock at process  $P_k$  ticks for the  $n$ th time.

A notion of “global time” can be approximated by incorporating only certain ticks of the local clocks. The spacing of these local ticks constitutes the granularity  $g_g$  of the “global clock”. Then, if  $g_g > \Pi$ , all local clocks will tick for the  $i$ th time within the same interval. Suppose then that  $g_g$  is equal to  $\Pi + K$ . If all events are restricted to the  $K$  time interval, they will all be timestamped with the same tick value by every clock.

He then shows that all three properties can be guaranteed by confining the significant events in the system to a sparse time base. If the granularity of this time base is greater than  $2g_g + \delta$ , where  $\delta$  is the maximum message passing delay, then all three properties given above can be guaranteed.

Neiger and Toueg’s is a powerful result, but is restrictive in that it is only applicable to problems with internal specifications. This would certainly seem to exclude unstable predicates, whose truth may oscillate arbitrarily, as well as predicates based on attainment of a certain system state at a real time instant. Kopetz’ result requires that either all events be restricted to the sparse time base, or that the events that should be restricted are known in advance.

In the following section, we develop several constructs that facilitate the application of roughly synchronized clocks to the specific problem of predicate evaluation. Unlike Neiger

and Toueg's result, the approaches we develop are applicable to predicates which are not internally specified. Further, unlike Kopetz' sparse time, we do not require the ability to determine in advance which actions should be confined to a sparse time grid.

## 2.5 Synchronous and Globalized Local Properties

### 2.5.1 Definitions

A *local property*  $A_i$  is an assertion about the local state of process  $P_i$ . It is simply a boolean expression evaluated over the set of variables defined by a single process (including implicit system variables such as the process' program counter if necessary).

A *time-stamped local property* (TLP) is a triple, denoted  $\mathcal{L}(T, i, A_i)$ , where  $T$  is a process clock value,  $i$  is a process index, and  $A_i$  is a local property of process  $P_i$ .  $A_i$  was evaluated on the state of  $P_i$  at some instant when  $C_i$  read  $T$ . We call  $T$  the property's *timestamp*. More formally,

$\mathcal{L}(T, i, A_i)$  holds if and only if  $A_i$  holds for  $P_i$  at some real time instant  $t$  such  
that  $C_i(t) = T$ .

A TLP is then a statement about a given process which was known when that process' clock read the timestamp value  $T$ . Since the clocks in the distributed system are only roughly synchronized,  $A_i$  need not hold when any other processor clock reads  $T$ . The local significance of the timestamp on a TLP makes it somewhat useless. The roughness in the synchronization of clocks introduces a degree of uncertainty into the notion of time provided by the collection of physical clocks. That uncertainty manifests itself when trying to combine TLPs from different processors to make an assertion about the joint states of

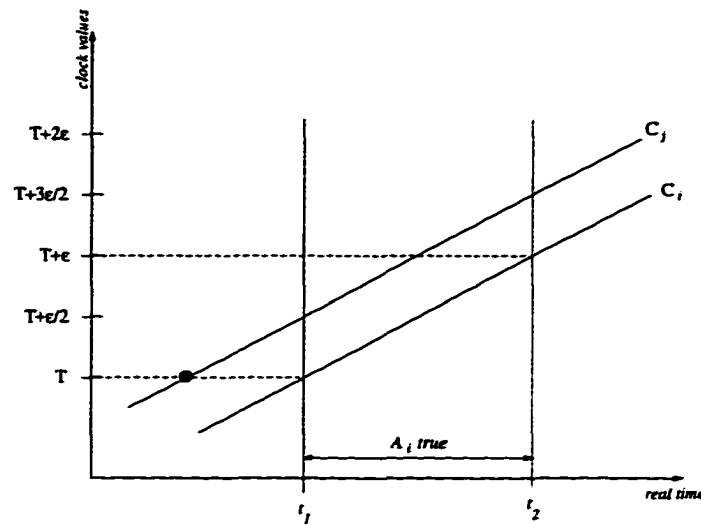


Figure 2.3: Local Significance of SLP Timestamps

those processors. For example, let  $L_1$  be  $\mathcal{L}(T, i, \text{“ } x=3 \text{”})$  and  $L_2$  be  $\mathcal{L}(T, j, \text{“ } y=3 \text{”})$ . We cannot be sure that the values of  $x$  and  $y$  were ever 3 at the same instant of real time, as each process clock could have read  $T$  at a different real time instant.  $L_1$  is meaningful only in the context of  $P_i$  and  $L_2$  is meaningful only in the context of  $P_j$ .

A *synchronous local property* (SLP), denoted  $S(T, i, A_i)$ , is a local property that remains true for at least the maximum clock skew  $\epsilon$ , as read by the local process clock. More formally,

$S(T, i, A_i)$  if and only if  $A_i$  holds over the interval  $[t_1, t_2]$ , inclusive, where

$$C_i(t_1) = T \text{ and } C_i(t_2) = T + \epsilon.$$

Note that we do not require that the property be true for all real time instants  $t_1$  at which  $C_i(t_1) = T$  or, similarly, all  $t_2$  such that  $C_i(t_2) = T + \epsilon$ .

If a property remains true for at least  $\epsilon$  then the timestamps on the local assertions can, under certain restrictions, be used to detect an instant in time at which some property is true for all processes in the system. We discuss this in more detail in the following section. However, the timestamp on such a property is not necessarily meaningful on any other

process' clock. If a process asserts  $S(T, i, A_i)$ , then  $A_i$  was not necessarily true when the clock of any other process read  $T$ . This is depicted in figure 2.3. Here we show the process clock values of  $P_i$  and  $P_j$  versus absolute real time, The real time interval over which  $A_i$  is true if  $P_i$  asserts  $S(T, i, A_i)$  is marked. (For clarity, we depict systems with infinite clock resolution, although this is not required of the system clocks.) Note that  $A_i$  was not true when  $C_j$  read  $T$  although  $C_j$ 's clock clearly obeys the clock axioms.

In order to make the timestamp on a local property meaningful globally, we introduce the concept of a *globalized local property* (GLP). A GLP is denoted  $\mathcal{G}(T, A_i)$ , where  $T$  is a timestamp and  $A_i$  is a local property of process  $P_i$ . A globalized local property is just like a TLP with the additional requirement that for all  $j \in \text{SYS}$ ,  $A_i$  is known about the state of  $P_i$  when  $C_j$  reads  $T$ . We formally define a GLP as follows:

$\mathcal{G}(T, A_i)$  if and only if  $A_i$  holds for all  $\{t : C_i(t) = T\}, i$  in  $\text{SYS}$ .

Obviously protocol must be developed in order to globalize a TLP. In the next section we give a simple protocol for globalizing the timestamp on a local property and develop several properties of SLPs and GLPs which facilitate simple and efficient predicate detection.

### 2.5.2 Properties of SLPs and GLPs

We now develop several useful properties of SLPs and GLPs. The first theorem underlies a protocol for globalizing a TLP. It shows that if the truth of a local predicate remains constant over an interval of  $2\epsilon$ , as read by the local process clock, then a time-stamped assertion can be made with a timestamp that is meaningful by any process clock.

**Theorem 1** *If  $A_i$  holds over the interval  $[t_1, t_2]$ , where  $C_i(t_1) = TS - \epsilon$  and  $C_i(t_2) = TS + \epsilon$ , then  $\mathcal{G}(TS, A_i)$ .*

$state_j$	$P_j$ 's state, initially false
$T_j$	value of $C_j(t)$ when $\mathcal{L}(T_j, j, A)$ can first be asserted
$alarm_j(T)$	alarm which signals $P_j$ when $C_j(t) = T$ ; $alarm_j(0)$ cancels the alarm
$TS_j$	timestamp of the globalized local predicate

$state_j$	Event	Action
<i>unsatisfied</i>	$A_j$	$T_j = C_j(t)$ assert $\mathcal{L}(T_j, j, A_j)$ $alarm_j(T_j + 2\epsilon)$ $state_j = transition$
<i>transition</i>	$\neg A_j$	$timer_j(0)$ $state_j = unsatisfied$
	$timer_j$ expires	$TS_j = T_j + \epsilon$ assert $\mathcal{G}(TS_j, A_j)$

Figure 2.4: Process State and Protocol for Globalizing a Time-stamped Local Property

PROOF: Let  $C_j(ts_j) = TS$ . By clock axiom C1,  $TS - \epsilon < C_i(ts_j) < TS + \epsilon$ . By our assumption,  $A_i$  holds over the interval  $[TS - \epsilon, TS + \epsilon]$ , inclusive, as read by  $C_i$ . Thus,  $A_i$  holds at  $C_j(ts_j) = TS$  for all  $j \in SYS$  and, by definition,  $\mathcal{G}(TS, A_i)$ . ■

Thus if  $P_i$  wants to be able to tell its peers in the system some important fact about its state at a certain point in its computation, it simply ensures  $\mathcal{G}(T, A_i)$  using theorem 1 and it can then convey this information to its peers.

A simple protocol for making such an assertion is given in figure 2.4. The protocol is specified as a set of actions that process  $P_j$  takes in response to events when it is in a given state. Here process  $P_j$  is asserting the truth of a predicate  $A_j$  over its local process state.  $P_j$  begins in state *unsatisfied*. As soon as the local predicate  $A_j$  becomes true,  $P_j$  can assert the TLP  $\mathcal{L}(T_j, j, A_j)$ .  $P_j$  then sets its timer for  $T_j + 2\epsilon$  and enters the *transition* state. In the *transition* state,  $P_j$  is waiting for the timer to expire. If  $A_j$  becomes false before the timer expires, the GLP cannot be asserted, so  $P_j$  resets the timer and re-enters the *unsatisfied* state. When the timer expires, the local predicate  $A_j$  has remained true for  $2\epsilon$  and the GLP  $\mathcal{G}(TS_j, A_j)$  can be asserted. At this point,  $A_j$  was true when the clock of any

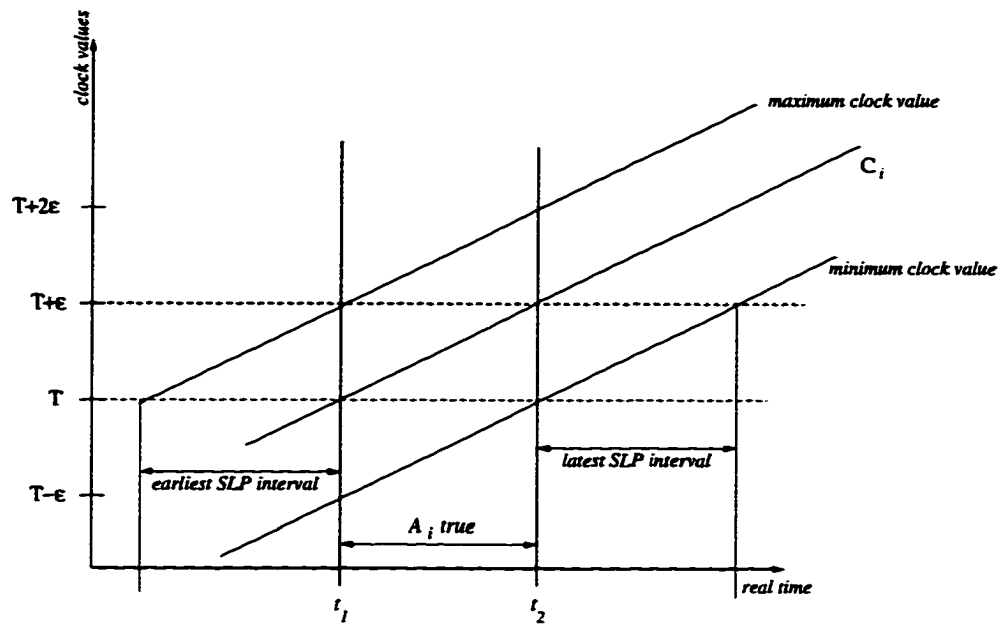


Figure 2.5: Overlap of SLPs with Unequal Timestamps

other processor read  $TS_j$ .

It should be clear that, if each process  $P_j$ ,  $j$  in  $SYS$ , asserts a GLP  $\mathcal{G}(TS, A_j)$  with the same timestamp  $TS$ , then there was an instant in real time at which all of the local predicates  $A_j$  were true. By definition, when a process asserts  $\mathcal{G}(TS, A_j)$ , the local predicate  $A_j$  was true when the clock of any other process read  $TS$ . If all processes have asserted GLPs with this same timestamp, then all the local predicates were true when the clock of process  $P_i$  read  $TS$ . In fact, all the local predicates were true when *any* process clock read  $TS$ . Clearly, then, there was an interval of real time values at which all the predicates were true.

If each process  $P_j$ ,  $j$  in  $SYS$ , instead asserts an SLP  $\mathcal{S}(TS, j, A_j)$  with the same timestamp, there was an instant in time at which all of the local predicates were true, but we know only that the clock of some processor read  $TS$  at this instant. This is shown in the following theorem.

**Theorem 2** *If  $\mathcal{S}(TS, i, A_i)$  holds for all  $i$  in  $SYS$  then there exists a real time instant  $t$  such that  $\mathcal{L}(C_i(t), i, A_i)$  for all  $i$  in  $SYS$ . Further,  $C_i(t) = TS$  for some  $P_i, i$  in  $SYS$ .*

PROOF: Suppose  $\mathcal{S}(TS, i, A_i)$  for every  $i$  in  $SYS$ . Now consider the last instant at which any process clock reads  $TS$ . Let this instant be  $tf$  and let  $C_j(tf) = TS$ . Then, by clock axiom C1 and our assumption that  $tf$  is the last instant some process reads  $TS$ ,

$$TS \leq C_i(tf) < TS + \epsilon \quad \text{for every } i \text{ in } SYS.$$

By assumption,  $\mathcal{S}(TS, i, A_i)$  holds, and thus  $A_i$  holds over the interval  $[TS, TS + \epsilon]$ , as read by  $C_i$  for every  $i$  in  $SYS$ . Thus, at real time instant  $tf$ , the predicate  $(A_0 \wedge A_1 \wedge \dots \wedge A_{N-1})$  holds and  $C_j(tf) = TS$ . ■

If we assume that the bound  $\epsilon$  on the clock skew is tight, that is,  $|C_i(t) - C_j(t)| \leq \epsilon$ , rather than  $|C_i(t) - C_j(t)| < \epsilon$ , then it is simple to show that if the timestamps on SLPs are not equal, then the real time intervals over which the local predicates are true may or may not overlap. This is illustrated in figure 2.5. Here we show the value of local clock  $C_i$  versus real time. We assume that  $P_i$  has asserted the SLP  $\mathcal{S}(T, i, A_i)$ . We also show the minimum and maximum values any other process clock can read given the value shown for  $C_i$ . Using these clock values, we show the latest and earliest real time intervals over which some other process' SLP with this same timestamp can occur. These intervals overlap only at real time instants  $t_1$  and  $t_2$ . Clearly, then, if the timestamp on the other process' SLP is either greater than, or less than,  $T$ , then the intervals do not necessarily overlap.

These theorems form a foundation from which we can develop protocols that exploit the advantages of assuming a global time base for predicate evaluation. In the next two chapters

we will look, in turn, at the evaluation of stable and unstable predicates. In each case, the assumption of a global time base allows development of simple and efficient protocols for predicate evaluation. In certain cases, these protocols detect predicates which would be difficult, if not impossible, to detect without the assumption of a global time base. As we will see, the use of SLPs and GLPs greatly facilitates development of these protocols.



## Chapter 3

# Stable Predicates

In this chapter, we demonstrate the use of GLPs and SLPs in evaluation of certain stable predicates. After a stable predicate becomes true in any consistent global state, it remains true indefinitely. Common stable predicates are termination, deadlock, and token loss.

We begin, in section 3.1, by demonstrating the application of GLPs and SLPs by solving the well known termination detection problem. We present several algorithms which, in addition to demonstrating the application of GLPs and SLPs, demonstrate the advantages of structuring certain predicate evaluation algorithms on the use of a global time base. In section 3.2, we show that the approach taken in detection of distributed termination can also be used to develop simple and efficient solutions to the more complex problem of distributed deadlock detection and resolution. Finally, in section 3.3, we discuss evaluation of general stable predicates.

### 3.1 Distributed Termination

Detection of termination, a stable global predicate, is a well-known problem for distributed systems. It contains many of the challenges characteristic of distributed processing. It has been studied extensively, and solutions abound in the literature [12, 18, 19, 11, 56, 63, 39, 32, 50, 33]. In this section, we use this problem to demonstrate both the advantages of structuring predicate evaluation algorithms on the use of a global time base and the application of GLPs and SLPs.

We begin by describing the problem of distributed termination detection. We then present protocols which solve the problem. The first four protocols, given in section 3.1.2, solve the problem for the same simple system with one exception, the system clocks have varying precisions. We contrast the assumptions that clocks are not synchronized, that they are perfectly synchronized, and that they are roughly synchronized. These protocols demonstrate the advantages of assuming a global time base and the use of GLPs and SLPs.

In the remaining algorithms, given in section 3.1.3 we demonstrate the ease with which the approach used in the earlier algorithms is extended to varying system architectures and performance goals. We present solutions tailored for various system characteristics including bounded message delay, asynchronous communication, and a broadcast network. We also present an algorithm which ensures that processes receive at most one control related message during any period of application processing.

#### 3.1.1 Problem

The general problem of distributed termination detection can be stated as follows. We consider a system comprised of  $N$  processes. These processes communicate solely via message

passing. Each process is considered always to be in one of two states, either *active* or *passive*. When active, a process is performing application-related processing. When passive, a process performs only control-related processing, that is, processing used to determine whether or not the application is terminated. A process may transition from active to passive spontaneously upon satisfaction of some local predicate. A process may transition from passive to active only upon receipt of an application message from an active process. The problem is to determine when the application is terminated.

If communication is synchronous then the application terminates at the first instant all processes become simultaneously passive. As all processes are passive at this instant, the only way the system could reactivate is if some process receives an application message which is currently in the channels. However, synchronous communication is “instantaneous,” so the channels must be empty if all processes are passive. If communication is asynchronous, we must deal with the possibility of outstanding messages. In this case, the system is terminated at the first instant all processes are passive and there are no outstanding messages.

### 3.1.2 A Simple System

In this section we present several algorithms that focus on termination detection for a specific distributed system architecture. We consider a system in which communication is synchronous and reliable, processes are reliable, and control messages travel unidirectionally through a ring of the system processes. Application message routing is unrestricted. This system was chosen because it is simpler than many other distributed systems and because solutions already exist in the literature for both systems with no clock synchronization and systems with perfect clock synchronization.

The first of these algorithms, by Dijkstra, Feigen, and van Gasteren [11], assumes no clock synchronization. The second, by Rana [56], assumes perfect clock synchronization, i.e.,  $C_i(t) = C_j(t)$  for every real time instant  $t$  and for every  $i, j$  in  $SYS$ . In the third algorithm [44], we use the techniques presented in the previous chapter to adapt Rana's algorithm to systems in which the process clocks are only roughly synchronized. These algorithms demonstrate the advantages of structuring predicate evaluation algorithms on the assumption of a global time base and the application of GLPs. With the fourth algorithm [45], we address efficiency considerations in using GLPs. This algorithm solves the problem for the same system as the previous three protocols; however, we make use of SLPs rather than GLPs, contrasting the application of each.

### 3.1.2.1 Solution Without Clock Synchronization

Dijkstra's algorithm is token-based. One process,  $P_0$ , initiates all tokens and eventually detects the termination. All tokens  $P_0$  initiates are colored *white*. A process receiving the token waits until it is passive to propagate it. Thus, all processes propagating the token are passive. This would be sufficient, except a process which has already propagated the token can be activated by some process which has not yet propagated the token, but will eventually. A coloring scheme is used to detect any such activations.

All processes are initially white. A process sending an application message turns itself *black*. A white process propagates a white token. Black processes turn a received token black prior to propagating it. A process becomes white again once it propagates the token. Thus, a token only remains white if no application messages were transmitted during its traversal. Termination is declared when  $P_0$  receives a white token. Thus, if  $P_0$  declares termination, all

processes were passive when they propagated the token and they will remain so indefinitely, as no process could have been reactivated by one of its successors in the control cycle after having propagated the token.

**Algorithm** The algorithm is given below. (Here “message” refers to an application message; the token is the sole control message.)

- (1) A process sending a message makes itself black.
- (2)  $P_0$  initiates a token by making itself white and sending a white token to  $P_1$ .
- (3) When active,  $P_i$  keeps the token; when passive, it sends the token to  $P_{(i+1) \bmod N}$ .
- (4) When  $P_i$  propagates the token, it sends a black token to  $P_{(i+1) \bmod N}$  if it is black itself; if  $P_i$  is white, the token color is unchanged.
- (5) After completion of an unsuccessful (black) token  $P_0$  initiates a next token.
- (6) Upon transmission of the token to machine  $P_{(i+1) \bmod N}$ ,  $P_i$  becomes white.

**Performance** Dijkstra’s algorithm is  $O(MN)$  in the number of messages passed to detect the termination, where  $M$  is the number of application messages transmitted during the execution. It detects termination in at most two cycles of the first token initiated once the system is terminated. It does not require that the application freeze in order to detect the termination.

### 3.1.2.2 Solution with Perfect Clock Synchronization

Rana’s algorithm is based on the same assumptions as Dijkstra’s, with the exception that he assumes that each process has an independent clock and that all these clocks are perfectly

$state_j$  indicates  $P_j$ 's state, initially *active*  
 $TS_j$   $P_j$ 's timestamp, initially 0  
 $Token(TS_i, i)$  token initiated by  $P_i$  with timestamp  $TS_i$   
 $clock_j$  current value of  $P_j$ 's clock

$state_j$	Event	Action
<i>active</i>	satisfy local predicate	$TS_j \leftarrow clock_j$ initiate $Token(TS_j, j)$ $state_j \leftarrow idle$
	$Token(TS_i, i)$ received	discard $Token(TS_i, i)$
<i>idle</i>	application msg received	$state_j \leftarrow active$
	$Token(TS_i, i)$ received	if $(TS_j \leq TS_i) \wedge (i = j)$ then declare termination if $(TS_j \leq TS_i) \wedge (i \neq j)$ then propagate $Token(TS_i, i)$ if $(TS_j > TS_i)$ then discard $Token(TS_i, i)$

Figure 3.1: Protocol (*TD-Rana*): Rana's Termination Detection Protocol

synchronized. His algorithm is also token based. However, all processes initiate tokens and any process may detect termination. Unlike Dijkstra's protocol, the solution is fully distributed and symmetric.

**Algorithm** The protocol and associated process state are shown in figure 3.1. The algorithm is presented as a set of rules for the way that process  $P_j$  reacts to events when it is in a given state.

Processes can be in one of two states with respect to the protocol, either *active* or *idle*. In the *active* state, processes are performing application related processing. In the *idle* state, processes are passive with respect to the application; they are performing only control-related processing. Each process initiates a token upon transitioning from the *active* to the *idle* state. The token is timestamped with the time at which the process became passive. It also contains the initiating process' index. The index is used to determine whether or not the token has circulated completely. (In Rana's algorithm the token contained a counter,

rather than the initiator's index. We use the process index here to be consistent with algorithms presented later in the chapter.)

An active process receiving the token discards it. A passive process receiving another process' token compares the timestamp on the token with its own local timestamp. If its local timestamp is less than or equal to the timestamp on the token then the token is propagated. If its own timestamp is greater than the timestamp on the token then the token is discarded. If a process receives its own token, with its current timestamp value, then termination is declared; otherwise, the token is discarded.

**Correctness** Rana proved the correctness of his protocol [56]. Here, we present similar arguments for reference in the discussion of the next protocol. The original protocol presentation did not address the clocks' granularity. In order to simplify the discussion, we assume the system clocks have infinite resolution.

In order to prove the algorithm correct, we must establish that: if termination is declared then the system is terminated (safety), and if the system is terminated then some process will declare the termination (liveness).

Throughout the remainder of this chapter, we let  $C_i^{-1}(T)$  denote *any* real time instant  $t$  at which  $C_i$  read  $T$ . If clocks have infinite resolution, then  $C_i^{-1}(T)$  denotes a unique time instant. Otherwise,  $C_i^{-1}(T)$  represents any instant  $t$  in the interval of real time at which  $C_i$  read  $T$ .

**Theorem 3 (Safety)** *If a process  $P_j$  declares termination then the application is terminated.*

**PROOF:** By the protocol,  $P_j$  only declares termination if it has received its own token. If

$P_j$  has received its own token, then every  $P_i$ ,  $i$  in  $SYS$ ,  $i \neq j$ , has propagated  $P_j$ 's token.

Let  $tt_i$  be the real time instant at which  $P_i$  propagates  $Token(TS_j, j)$  and let  $TS_i$  be  $P_i$ 's timestamp at that time. By the protocol,  $P_i$  only propagates  $Token(TS_j, j)$  if its timestamp  $TS_i$  is less than or equal to  $TS_j$ . Then, given perfect clock synchronization, all processes were passive at  $C_j^{-1}(TS_i)$ . Since process clocks are non-decreasing,

$$C_j^{-1}(TS_i) \leq C_j^{-1}(TS_j)$$

Clearly, the instant  $tt_i$  at which  $P_i$  propagates the token is greater than or equal to the instant  $C_j^{-1}(TS_j)$  at which  $P_j$ 's clock reads the timestamp value. Thus

$$C_j^{-1}(TS_i) \leq C_j^{-1}(TS_j) \leq tt_i.$$

By the protocol, any process propagating  $P_j$ 's token is continually passive over the interval  $[C_j^{-1}(TS_i), tt_i]$ . Thus, all processes are passive at real time instant  $C_j^{-1}(TS_j)$  and, under synchronous communication, the computation must be terminated. ■

**Lemma 1** *Let  $TS_{max}$  be the highest valued timestamp generated during the computation, and let  $P_j$  be a process which generated timestamp  $TS_{max}$ . Then the system is terminated at  $C_j^{-1}(TS_{max})$ .*

**PROOF:** Let  $TS_{final}$  be the final timestamp generated by any process  $P_i$ ,  $i \in SYS$ . Since  $TS_{max}$  is the highest timestamp generated during the computation,  $TS_{final} \leq TS_{max}$ . Since clocks are perfectly synchronized, all processes were then passive at  $C_j^{-1}(TS_{final})$ . Since the clocks are non-decreasing,  $C_j^{-1}(TS_{final}) \leq C_j^{-1}(TS_{max})$ . Thus, all processes became finally



passive at or before  $C_j^{-1}(TS_{max})$ , and the computation must be terminated at that time. ■

**Theorem 4 (Liveness)** *If the system is terminated then some  $P_j$  will eventually declare termination.*

**PROOF:** Consider the highest valued timestamp  $TS_{max}$  generated during the computation. Let  $P_j$  be a process which generated timestamp  $TS_{max}$ . By lemma 1, the system is terminated at  $C_j^{-1}(TS_{max})$ .

According to the protocol, upon its transition to the *idle* state,  $P_j$  will initiate a token  $Token(TS_{max}, j)$ . Then, since the system is terminated at  $C_j^{-1}(TS_{max})$ , all processes will receive  $P_j$ 's token in the *idle* state. By our assumption, all processes have timestamps less than or equal to  $TS_{max}$ . By the protocol, all processes will then propagate the token and  $P_j$  will detect the termination. ■

**Performance** Like Dijkstra's algorithm, Rana's does not require that a process freeze application processing in order to detect termination. Also like Dijkstra's algorithm, it requires  $O(MN)$  messages to detect termination. However, it requires only a single token circulation once the system is terminated. Further, it does not require selection of a "leader" to initiate tokens and detect the termination. The algorithm is fully distributed and symmetric.

Thus, the assumption of perfect clock synchronization, though unrealistic, facilitates development of a more efficient and simpler algorithm. Further, if we assume that the system clocks are accurate then this algorithm can provide, via the message timestamp, additional information about the exact time at which the system became terminated.

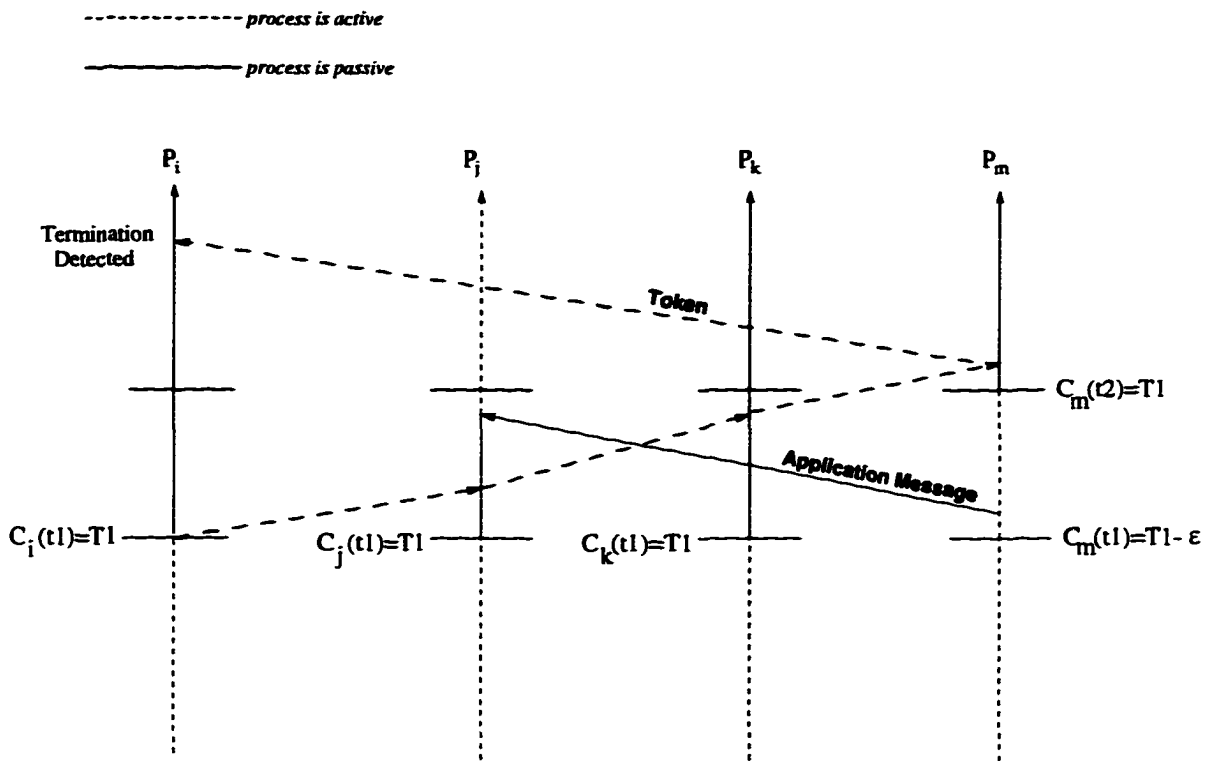


Figure 3.2: Failure of Rana's Termination Detection Protocol

### 3.1.2.3 Solution With Globally Timestamped Properties

Although Rana's algorithm is fully distributed and symmetric and detects termination faster than Dijkstra's, the algorithm only works if clocks are perfectly synchronized. Figure 3.2 illustrates a situation in which the algorithm fails when clocks are only roughly synchronized. Each axis in the figure represents execution for a single process and is scaled by absolute real time. Points of interest are labeled with the corresponding process clock values. Here, all processes went passive when their clocks read  $T1$ . The figure illustrates circulation of a single token, initiated by  $P_i$ , although by the protocol all processes will initiate a token.  $P_m$ 's clock runs slower than the other process clocks, which are all synchronized perfectly; its clock reads the token timestamp value  $T1$  later than the other process clocks, but before the token has reached it. Thus, when the token reaches  $P_m$ , it will be propagated, and  $P_i$

$state_j$	indicates $P_j$ 's state, initially <i>active</i>
$TP_j$	$P_j$ 's clock value at the instant it satisfied its local predicate
$TS_j$	$P_j$ 's timestamp, initially 0
$alarm_j(T)$	$P_j$ 's alarm which signals $P_j$ when $C_j$ reaches $T$ ; $alarm_j(0)$ cancels the alarm
$\Gamma_j$	set of all tokens received by $P_j$ during the current <i>transition</i> state, initially empty; the operator $\text{tsmax}$ applied to $\Gamma_j$ returns the timestamp and process index from the token with the largest timestamp value
$Token(TS_i, i)$	token initiated by $P_i$ with timestamp $TS_i$
$clock_j$	current value of $P_j$ 's clock

$state_j$	Event	Action
<i>active</i>	satisfy local predicate	$TP_j \leftarrow clock_j$ $alarm_j(TP_j + 2\epsilon)$ $\Gamma_j \leftarrow \emptyset$ $state_j \leftarrow transition$
	$Token(TS_i, i)$ received	discard $Token(TS_i, i)$
<i>transition</i>	application msg received	$alarm_j(0)$ $state_j \leftarrow active$
	$Token(TS_i, i)$ received	$\Gamma_j \leftarrow \Gamma_j \cup \{Token(TS_i, i)\}$
	$alarm_j$ expires	$TS_j \leftarrow TP_j + \epsilon$ $TS_i, i \leftarrow \text{tsmax}(\Gamma_j)$ if $(TS_i \geq TS_j)$ then propagate $Token(TS_i, i)$ else initiate $Token(TS_j, j)$ $state_j \leftarrow idle$
<i>idle</i>	application msg received	$state_j \leftarrow active$
	$Token(TS_i, i)$ received	if $(TS_j \leq TS_i) \wedge (j = i)$ then declare termination if $(TS_j \leq TS_i) \wedge (j \neq i)$ then propagate $Token(TS_i, i)$ if $(TS_j > TS_i)$ then discard $Token(TS_i, i)$

Figure 3.3: Protocol (*TD-GLP*): TD with Rough Clock Synchronization

will declare termination. However, during the lag time between the real time instant  $P_i$ 's clock read  $T_1$  and the instant  $P_m$ 's clock read  $T_1$ ,  $P_m$  reactivated  $P_j$ , which had already propagated  $P_i$ 's token, and the application is not terminated.

In Rana's algorithm, each process asserts the time that it went passive relative to the time the token initiator went passive, by comparing its local timestamp with the token timestamp value. This comparison guarantees that all processes propagating the token were passive at the instant the token was initiated, the token timestamp value. The algorithm

fails if clocks are only roughly synchronized because the token timestamp value and the local clock value cannot be meaningfully compared. Thus, it would seem that if each process globally timestamps its passivity, i.e.,  $P_i$  asserts the GLP  $\mathcal{G}(TS_i, P_i \text{ passive})$ , then Rana's algorithm would work with no additional modifications and, in fact, this is the case. We describe the modified algorithm more formally below.

**Algorithm** The protocol and associated process state are shown in figure 3.3. Like Rana's, our algorithm is token-based. Processes can be in one of three states, *active*, *transition*, or *idle*. A process is active with respect to the application in the *active* state. It is passive with respect to the application in both the *transition* and *idle* states. The *transition* state is entered upon satisfaction of the local predicate. It is essentially a pause of duration  $2\epsilon$  in order for  $P_i$  to globally timestamp the local property " $P_i$  is passive." The *idle* state is entered from the *transition* state once the local property has been globally timestamped. In the *idle* state, a process is waiting for an application message, receipt of its own token, or declaration of termination.

Rather than initiating a token immediately upon satisfaction of the local predicate, as done in Rana's algorithm, each process  $P_i$  waits until the local property " $P_i$  is passive" has been globally timestamped in accordance with theorem 1, given in chapter 2. Then, if it has not received a token with an equal or higher timestamp during the period it was globalizing the timestamp, it initiates a token and enters the *idle* state. If it has received a token with an equal or higher timestamp, then it propagates the highest-valued token it has received and enters the *idle* state. (This is done for efficiency. The algorithm works equally well if all tokens with equal or higher valued timestamps are propagated, as is done in Rana's algorithm.) The criterion for token propagation from the *idle* state is the same as

that for Rana's algorithm; tokens with equal or greater timestamps are propagated. As in Rana's algorithm, termination is declared when a process receives its own token containing its current timestamp value.

**Correctness** In order to prove the algorithm correct, we must establish safety and liveness, as we did for Rana's algorithm.

The safety proof for Rana's algorithm essentially showed that, for every  $P_i$  propagating  $P_j$ 's *winning* token  $Token(TS_j, j)$ , that is, a token which circulates completely,  $P_i$  was passive at  $C_j^{-1}(TS_i)$ , where  $TS_i$  is  $P_i$ 's timestamp when it propagates  $P_j$ 's token. Then

$$C_j^{-1}(TS_i) \leq C_j^{-1}(TS_j) \leq tp_i,$$

where  $tp_i$  is the real time instant at which  $P_i$  propagated the token. Rana's algorithm fails when clocks are only roughly synchronized because the timestamp comparison alone does not ensure that all processes were passive when  $C_j$  read their timestamp values. However, globalizing timestamps assures this; i.e.,  $P_i$  was passive at  $C_j^{-1}(TS_i)$ . We do not repeat the safety argument here.

The liveness proof is also similar to that given for Rana's protocol. We showed the system was terminated at  $C_j^{-1}(TS_{max})$ , where  $TS_{max}$  is the highest valued timestamp generated during the computation and  $P_j$  is a process which generated timestamp  $TS_{max}$ . By using GLPs, it is easy to establish that this is true in a system with rough clock synchronization. The argument is virtually identical to that given for lemma 1, with one exception. If  $TS_{final}$  is the final timestamp generated by any  $P_i$ ,  $i$  in  $SYS$ , then we know that all processes were passive at  $C_j^{-1}(TS_{final})$  by theorem 1, given in chapter 2, rather than relying on perfect clock

synchronization. Once we have established that the system is terminated at  $C_j^{-1}(TS_{max})$ , we only need argue that all processes will propagate a token with this timestamp. Such an argument would be identical to that made for Rana's protocol if processes in the *transition* state propagated *all* tokens with equal or higher valued timestamps. However, for increased efficiency, only a single token leaves any process when it changes from the *transition* state to the *idle* state. We give the modified proofs below.

**Theorem 5 (Liveness)** *If the system is terminated, then some  $P_j$  will eventually declare termination.*

PROOF: Consider the highest valued timestamp  $TS_{max}$  generated during the computation, and let  $P_j$  be a process which generated timestamp  $TS_{max}$ . If more than one process generates timestamp  $TS_{max}$ , let  $P_j$  be the process whose alarm expires first. By lemma 1, the system is terminated at  $C_j^{-1}(TS_{max})$ .

By assumption,  $P_j$  is the first process with timestamp  $TS_{max}$  whose alarm expires. Thus,  $P_j$  could not have received a token with a greater timestamp than its own,  $TS_{max}$ , while in the *transition* state and, by the protocol, will initiate  $Token(TS_{max}, j)$ .

Any process receiving  $Token(TS_{max}, j)$  must be in either the *transition* or the *idle* state. By the protocol, any process which receives a token with timestamp  $TS_{max}$  in the *idle* state will propagate it. A process in the *transition* state may discard a token with this timestamp. However, all processes in the *transition* state will propagate some token with timestamp  $TS_{max}$ , and, by the nature of the control cycle routing, some token with timestamp  $TS_{max}$  will circulate completely, and termination will be detected. ■

**Performance** Assuming that processes are never swapped out and that the time required to execute the instructions given in the protocol is insignificant in relation to message transmission delay, this algorithm detects termination within an interval of  $N\delta + 3\epsilon(1 + \rho_M)$  from the instant at which the final process becomes passive, where  $\delta$  is the maximum message transmission time. The interval between the instant at which the application first terminates and the instant at which the termination is detected is called the *detection delay*. We prove this bound on detection delay below.

**Lemma 2** *Assume that instruction execution time is insignificant in relation to message transmission time, and that processes are never swapped out. Then the detection delay of the algorithm given in figure 3.3 is  $N\delta + 3\epsilon(1 + \rho_M)$ .*

**PROOF:** Let  $tt$  be the instant at which the system terminates and let  $th$  be the latest instant at which a token leaves a process in the *transition* state. Then, ignoring instruction execution time, the maximum detection delay  $\beta$  is  $th - tt + N\delta$ .

Let the highest valued timestamp generated during the computation be  $TS_{max}$ . We have already shown that some token with timestamp  $TS_{max}$  will circulate and detect termination. Let  $P_j$  be any process with timestamp  $TS_{max}$  whose token circulates completely. We have already shown that the system is terminated at  $C_j^{-1}(TS_{max})$ . Let  $tp_j$  be the earliest instant for which  $C_j(tp_j) = TS_{max} - \epsilon$ , then

$$tp_j \leq tt.$$

Now consider the real time instant  $tr_j$  at which  $P_j$  initiates  $Token(TS_{max}, j)$ . (This is based on our assumption that processes are not swapped out and that instruction execution

time is negligible.) By the protocol,  $C_j(tr_j) = TS_{max} + \epsilon$ . Then, by clock axiom C3,

$$tr_j - tp_j \leq 2(1 + \rho_M)\epsilon.$$

By clock axiom C1,  $C_i(tr_j) \geq TS_{max}$ ,  $i$  in  $SYS$ . By the protocol, any process  $P_i$  which propagates  $Token(TS_{max}, j)$  must have a timestamp  $TS_i$  less than or equal to  $TS_{max}$ . Also by the protocol,  $P_i$  must leave the *transition* state and propagate  $Token(TS_{max}, j)$  by the time  $C_i$  reads  $TS_{max} + \epsilon$ . Then

$$th \leq tr_j + (1 + \rho_M)\epsilon.$$

Thus,

$$\begin{aligned} \beta &\leq th - tt + N\delta \\ &\leq tr_j + (1 + \rho_M)\epsilon - tt + N\delta \\ &\leq tr_j + (1 + \rho_M)\epsilon - tp_j + N\delta \\ &\leq 3(1 + \rho_M)\epsilon + N\delta. \end{aligned}$$

■

Like Dijkstra's and Rana's algorithms, ours requires  $O(MN)$  messages in order to detect termination. Also like Rana's, this algorithm can detect termination in a single token circulation once the system is terminated. However, it delays the token prior to initiation. As we have shown, this delay is at most  $3(1 + \rho_M)\epsilon$ , neglecting instruction execution time and assuming that processes are never swapped out. The efficiency gain in detecting termination, once the system is terminated, by using this algorithm, rather than Dijkstra's, will then depend on how close  $\epsilon$  is to the average message delivery time and how many processes



comprise the system.

Also like Rana's, this algorithm provides information about when the system became terminated. If we assume that all system clocks are accurate, then we know that the system was terminated at the token timestamp value in real time. This is not the first instant at which the system was terminated, as in Rana's algorithm; however, it is within  $\epsilon$  of the initial instant at which the system became terminated.

Finally, like Rana's, our algorithm is fully distributed and symmetric. Thus, using rough clock synchronization, we can construct an algorithm that provides the benefits of using perfect clock synchronization, with only a small performance degradation.

#### 3.1.2.4 Efficiency Considerations

Rana's protocol uses circulation of a timestamped token to detect the truth of the global predicate. Complete circulation of the timestamped token indicates that the global predicate is true when the token initiator's clock reads the token timestamp value. By using GLPs, it is easy to see that our modification to Rana's protocol works, even though clocks are only roughly synchronized. The correctness arguments are essentially the same.

Although not quite so obvious, the algorithm works equally well if SLPs, rather than GLPs, are used. In this case, all processes are passive at or before the instant that  $P_j$  initiates the winning token, rather than when  $C_j$  reads the token timestamp value. We present the modified algorithm and correctness proofs below [45].

**Algorithm** The protocol is identical to protocol *TD-GLP* (figure 3.3) with the exception that processes wait only  $\epsilon$  prior to initiating or propagating a token. Each process timestamps the token with its SLP timestamp, the clock value at the beginning of the  $\epsilon$  interval.

$state_j$  indicates  $P_j$ 's state, initially *active*  
 $TP_j$   $P_j$ 's clock value at the instant it satisfied its local predicate  
 $TS_j$   $P_j$ 's timestamp, initially 0  
 $alarm_j(T)$   $P_j$ 's alarm which signals  $P_j$  when  $C_j$  reaches  $T$ ;  $alarm_j(0)$  cancels the alarm  
 $\Gamma_j$  set of all tokens received by  $P_j$  during the current *transition* state, initially empty; the operator  $tsmax$  applied to  $\Gamma_j$  returns the timestamp and process index from the token with the largest timestamp value  
 $Token(TS_i, i)$  token initiated by  $P_i$  with timestamp  $TS_i$   
 $clock_j$  current value of  $P_j$ 's clock

$state_j$	Event	Action
<i>active</i>	satisfy local predicate	$TP_j \leftarrow clock_j$ $alarm_j(TP_j + \epsilon)$ $\Gamma_j \leftarrow \emptyset$ $state_j \leftarrow transition$
	$Token(TS_i, i)$ received	discard $Token(TS_i, i)$
<i>transition</i>	application msg received	$alarm_j(0)$ $state_j \leftarrow active$
	$Token(TS_i, i)$ received	$\Gamma_j \leftarrow \Gamma_j \cup \{Token(TS_i, i)\}$
	$alarm_j$ expires	$TS_j \leftarrow TP_j$ $TS_{i,i} \leftarrow tsmax(\Gamma_j)$ <b>if</b> $(TS_i \geq TS_j)$ <b>then</b> propagate $Token(TS_i, i)$ <b>else</b> initiate $Token(TS_j, j)$ $state_j \leftarrow idle$
<i>idle</i>	application msg received	$state_j \leftarrow active$
	$Token(TS_i, i)$ received	<b>if</b> $(TS_j \leq TS_i) \wedge (j = i)$ <b>then</b> declare termination <b>if</b> $(TS_j \leq TS_i) \wedge (j \neq i)$ <b>then</b> propagate $Token(TS_i, i)$ <b>if</b> $(TS_j > TS_i)$ <b>then</b> discard $Token(TS_i, i)$

Figure 3.4: Protocol (TD-SLP): Efficiency Modification to Protocol TD-GLP

Thus, if  $P_i$ 's clock value is  $TP_i$  when it satisfies its local predicate, its timestamp will be  $TP_i$ , rather than  $TP_i + \epsilon$ . The modified algorithm is shown in figure 3.4.

**Correctness** We again present safety and liveness arguments in order to verify our protocol. We begin by proving a token propagation theorem which will be useful in this correctness argument, as well as later correctness arguments. The theorem asserts that if a timestamped token is propagated in the manner of the previous algorithm, but using SLPs instead of GLPs, then the global predicate holds at the earliest instant the winning token could be initiated.

**Lemma 3** *Suppose  $P_j$  establishes  $S(TS_j, j, A_j)$  and then initiates a token with timestamp  $TS_j$ . Now suppose that  $P_i$  establishes  $S(TS_i, i, A_i)$ , that  $TS_i \leq TS_j$ , and that  $A_i$  is true continuously from the time  $P_i$  establishes its SLP until it receives  $P_j$ 's token. Then  $A_i$  is true at the earliest real time instant  $tr_j$  for which  $C_j(tr_j) = TS_j + \epsilon$*

**PROOF:** Let  $TS_i$  be  $P_i$ 's timestamp when it propagates  $P_j$ 's token  $Token(TS_j, j)$  and let  $tp_i$  be the latest instant at which  $C_i(tp_i) = TS_i$ . Let  $tr_j$  be the earliest instant at which  $C_j(tr_j) = TS_j + \epsilon$ . By clock axiom C1,  $C_i(tr_j) > TS_j$ . By our assumption  $TS_i \leq TS_j$  and then, by clock axiom C2,  $tp_i \leq tr_j$ .

Clearly, given non-zero message transmission time, the instant  $tt_i$  at which  $P_i$  propagates the token must be later than the instant at which  $P_j$  could first have released the token.

Thus

$$tp_i \leq tr_j \leq tt_i,$$

and, by the protocol,  $A_i$  must have held over this entire interval,  $[tp_i, tt_i]$ . Thus,  $A_i$  held at real time instant  $tr_j$ . ■

**Theorem 6 (Safety)** *If any process  $P_j$  declares termination, then the application is terminated.*

PROOF: If  $P_j$  declares termination, then it has received its own token  $Token(TS_j, j)$ . If  $P_j$  receives its own token, then every  $P_i$ ,  $i$  in  $SYS$ ,  $i \neq j$  has propagated  $P_j$ 's token. Clearly then, by the protocol and theorem 3, all processes are passive at the earliest real time instant  $tr_j$  for which  $C_j(tr_j) = TS_j + \epsilon$  and, under synchronous communication, the system must be terminated. ■

The next lemma is useful for establishing liveness. In the previous two protocols we showed that the system is terminated at the instant(s) the clock of the token initiator  $P_j$  reads the token timestamp value  $TS_{max}$ , where  $TS_{max}$  is the highest token timestamp value generated during the computation. Here we show that the system is terminated at the earliest instant  $tr_j$  for which  $C_j(tr_j) = TS_{max}$ .

**Lemma 4** *Let  $TS_{max}$  be the highest valued timestamp generated during the computation and let  $P_j$  be a process which generated  $TS_{max}$ . Then the computation is terminated at the earliest real time instant  $tr_j$  for which  $C_j(tr_j) = TS_{max} + \epsilon$ .*

PROOF: By clock axiom C1,  $C_i(tr_j) > TS_{max}$  for every  $i$  in  $SYS$ . Now suppose some process  $P_i$  is performing application processing at  $tr_j$ . Then, by the protocol, this process would generate a timestamp  $TS_i$  which is greater than  $TS_j$ , a contradiction. Thus, no process can be active at  $C_j^{-1}(TS_{max} + \epsilon)$ . ■

**Theorem 7 (Liveness)** *If the application is terminated, then some  $P_j$  will declare termination.*

**PROOF:** Consider the highest valued timestamp  $TS_{max}$  generated during the computation. Let  $P_j$  be a process which generated timestamp  $TS_{max}$ . If more than one process generates this same timestamp, then let  $P_j$  be the process whose alarm expires first. By lemma 4, the system must be terminated at the earliest instant  $tr_j$  for which  $C_j(tr_j) = TS_{max}$ .

By our assumptions,  $P_j$  could not have received a token with a higher valued timestamp while in the *transition* state, it will then initiate  $Token(TS_{max}, j)$  from the *transition* state immediately prior to entry into the *idle* state.

This token can be received by processes in the *transition* or *idle* states. By the protocol, any process in the *idle* state will propagate  $Token(TS_{max}, j)$ . A process in the *transition* state may discard  $Token(TS_{max}, j)$ . However, all processes will propagate some token with timestamp  $TS_{max}$  and, by the nature of the control cycle routing, one of these tokens will circulate completely and detect termination. ■

**Performance** By an argument similar to that made for lemma 2, again assuming that processes are not swapped out and that instruction execution time is insignificant in relation to  $\epsilon$  and the maximum transmission time  $\delta$ , the maximum detection delay is  $N\delta + 2(1 + \rho_M)\epsilon$ . Thus, we have reduced the maximum delay by  $(1 + \rho_M)\epsilon$  from the earlier algorithms.

However, the correctness proofs using this technique are not as straightforward. By using GLPs, we can reference all process timestamps to a single process clock. This allows us to order event occurrence in a single absolute time frame. This is not the case with SLP timestamps.

### 3.1.3 General Termination Detection

The previous algorithms all focused on detecting termination within a single simple system. These algorithms demonstrated both the benefits of assuming a global time base and the application of SLPs and GLPs.

In each of the time based algorithms given so far we have used the same basic approach, circulation of a timestamped token to establish an instant in time at which some local property is true for every process. The truth of each of these local properties at the same instant then implies the truth of the global termination predicate. This is a natural approach to the termination detection problem and to predicate detection in general.

In each of the algorithms we have presented, the token traverses a ring of the system processes. Further, we have only examined a simple case of the termination detection problem.

In the following sections we present several algorithms which demonstrate the broader application of this approach. We choose to use SLPs as the resulting algorithms are more efficient. However, we could have applied GLPs as well.

We begin by presenting an overview of previous solutions to the termination detection problem. We then present several algorithms that address the problem within more varied system architectures.

#### 3.1.3.1 Previous Work

Dijkstra presented the first solution to the termination detection problem in the context of *diffusing computations* [12]. A diffusing computation can be described as follows. Let  $A$  and  $B$  be nodes within a finite directed graph. If there is an edge between  $A$  and  $B$ , then

$B$  is a *successor* of  $A$ , and  $A$  is a *predecessor* of  $B$ . The *environment* is a single node with only incoming edges. A *diffusing computation* starts when the environment spontaneously sends (just once) a message to one or more of its successors. After reception of its first message, an internal node is free to send messages to its successors. Nodes send messages to successors and receive messages from predecessors; they send signals to predecessors and receive signals from successors. Each node's signalling obligation can be characterized by a "cornet", a bag in which one element has the special status of being "the oldest element". Whereas a stack is characterized by "last in, first out", a cornet is characterized by "very first in, very last out". Each message received on incoming edge adds an element to the bag, and a signal removes an element from the bag. When the environment receives a signal for each outgoing message, the computation is terminated. His protocol is valid for asynchronous communication and does not require FIFO channels.

Francez first proposed a solution to the problem which does not require the addition of communication channels between processes, like the system we described in the previous section [18]. However, the protocol may delay (freeze) execution of the underlying computation. (He later presented a solution which does not require delay of application processing [19].)

These early protocols focus on systems with synchronous communication; further, they are not distributed in the sense that a single process initiates the termination detection computation and eventually detects the termination [12, 18, 19, 48, 11, 63].

Rana proposed the first well-known fully distributed and symmetric solution. As we discussed, his solution is also restricted to systems with synchronous communication. Several later symmetric protocols focused on removing the restriction of perfectly synchronized

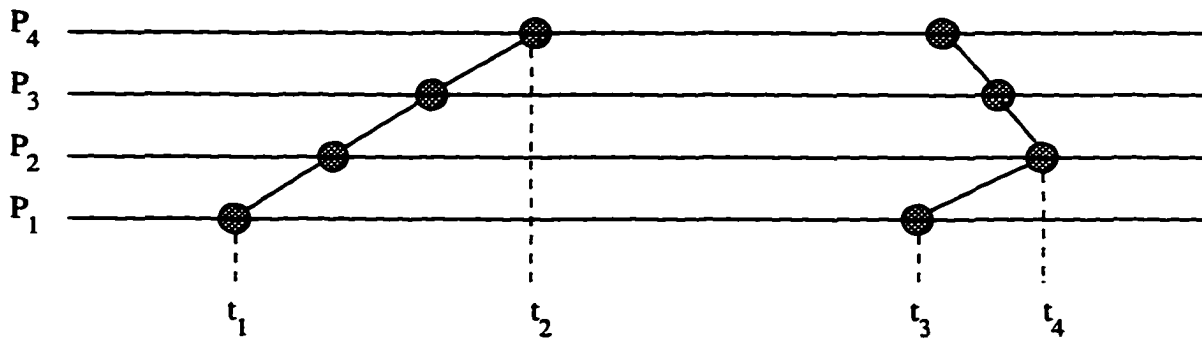


Figure 3.5: Successive Control Waves for a Four Process System

clocks, while retaining the assumption of synchronous communication [1, 8, 16].

Most later solutions focus on systems with asynchronous communication [47, 32, 40, 41, 25, 4, 50, 33]. Some of these require FIFO communication [47, 4] or take on the additional complexity required to obtain fault-tolerance or handle dynamic systems [32, 33].

Several protocols have been proposed to handle asynchronous message passing with arbitrary delivery order. Mattern proposed a simple protocol for detecting termination under this system model [40]. By his protocol, some process initiates a *control wave*, a message which causes receiving processes to report their state. By his “four counter” method, two successive control waves are initiated. This is illustrated for a four process system in figure 3.5. Each process  $P_i$  keeps monotonically increasing counters for the total number of messages sent ( $s_i$ ) and the total number of messages received ( $r_i$ ). These values are reported as part of the process state during the control wave. Let  $S(t)$  be the system-wide send count at real time instant  $t$ , and let  $R(t)$  be the system-wide receive count at real time instant  $t$ ; i.e.,  $S(t) = \sum_{i \in SYS} s_i(t)$  and  $R(t) = \sum_{i \in SYS} r_i(t)$ , respectively. Let  $S^*$  and  $R^*$  be the system wide send and receive counts reported by the first control wave. Similarly, let  $S'^*$  and  $R'^*$  be the counts reported by the second wave. It is easy to see that the following properties hold:



- $t \leq t'$  implies that  $s_i(t) \leq s_i(t')$  and  $r_i(t) \leq r_i(t')$ ,
- $t \leq t'$  implies that  $S(t) \leq S(t')$  and  $R(t) \leq R(t')$ ,
- $R^* \leq R(t_2)$ , and
- $S^* \geq S(t_3)$ .

It then follows that:

$$\begin{aligned}
 R^* = S^* &\Rightarrow R(t_2) \geq S(t_3) \\
 &\Rightarrow R(t_2) \geq S(t_2) \\
 &\Rightarrow R(t_2) = S(t_2)
 \end{aligned}$$

Then if  $R^*$  is equal to  $S^*$ , there are no outstanding messages at real time instant  $t_2$ , and the system must be terminated.

So-called *skeptic* algorithms are a variant of the four counter method and use flags to detect activity to the right of the first wave, which can corrupt the values of counters obtained from this wave. This activity can be detected by the use of flags, initialized by the first wave, and set by processes when they receive or send messages. Then the second wave only need check to see if any flags have been set, in which case a possible corruption is assumed [40, 31]. (Several synchronous algorithms, which do not require counters, are based on this principle [19, 11, 63].)

Two cycles are required in order to detect the termination using either of these approaches. Further, there is no upper bound on the number of control messages. Processes must guess when the algorithm should be restarted after an unsuccessful trial.

<i>Event</i>	<i>Action</i>
send application msg to $P_i$	$CNT \leftarrow CNT + 1$ send $msg(CLOCK, \dots)$ to $P_i$
receive application message $msg(TSTAMP, \dots)$	$CNT \leftarrow CNT + 1$ $TMAX \leftarrow \max(TSTAMP, TMAX)$
receive control message $msg(TIME, ACCU, INVALID, INIT)$	if $INIT = j$ then if $(ACCU = 0) \wedge (\neg INVALID)$ terminate else try again? else send $msg(TIME, ACCU + COUNT,$ $INVALID \vee (TMAX \geq TIME),$ $INIT)$ to $P_{(j \bmod n)+1}$
starting control round	$CLOCK \leftarrow CLOCK + 1$ send $msg(CLOCK, COUNT, false, j)$ to $P_{(j \bmod n)+1}$

Figure 3.6: Mattern's Logical Clock Protocol for Termination Detection

Some protocols take an approach similar to that used by Rana. These protocols are based on the use of some kind of logical clock. Eventually, some process will attain the highest logical clock value seen during the computation, and a token containing this timestamp will circulate and detect the termination. Mattern gave one such protocol for a ring of processes, as in the system of the previous section. Each process is assumed to have a local message counter  $CNT$ , which indicates the total number of messages sent minus the total number of messages received, a discrete  $CLOCK$ , which is initialized to zero, and a variable  $TMAX$ , which holds the latest send time of all messages received by  $P_j$ . The protocol is given in figure 3.6.

This protocol can detect termination in a single token cycle. However, control information must be appended to every application message. Huang and Lai presented protocols based on this same principle; Lai's protocol addressed dynamic systems [25, 32].

In the following section, we develop termination detection protocols for systems with reliable asynchronous message passing with arbitrary delivery order. As we will see, the use

of roughly synchronized clocks enables detection of termination in a single token cycle, in all cases, without requiring any control data within application messages.

### 3.1.3.2 Asynchronous Communication

Asynchronous, rather than synchronous, communication is a common complication to the simple system of the previous protocols. In systems with asynchronous communication, processes return from a message send immediately; that is, they do not wait to find out whether or not a sent message has been received. In this case, even if all processes are simultaneously passive, there may be a message in the channels which will reactivate one of the system processes. Thus, under asynchronous communication, we must detect an instant at which all processes are passive and all the channels are empty.

**Unbounded Message Delay** Probabilistic clock synchronization algorithms do not require a bound on message transmission time [10]. We can then assume a system with unbounded message transmission time, but in which clocks are roughly synchronized. Here we present an algorithm for this type of system.

We use message counters to detect whether or not the channels are empty. First, we modify the synchronous local property established by each process  $P_i$ , prior to initiating or propagating a token, to be the property " $P_i$  is passive and has a message deficit of  $lmsgs_i$ ", where the *message deficit* is the number of messages sent minus the number received by  $P_i$ . Each process tracks its local message deficit using a counter variable. As the token circulates, it collects each process' message deficit. As in the previous protocol, complete circulation of the token then indicates that all processes were passive at the same time instant. Further, in this case, the message deficit total in the token represents the system

*state<sub>j</sub>* indicates  $P_j$ 's state, initially *active*  
*TS<sub>j</sub>*  $P_j$ 's timestamp, initially 0  
*alarm<sub>j</sub>(T)*  $P_j$ 's alarm which signals  $P_j$  at  $C_j(t) = T$ ; *alarm<sub>j</sub>(0)* cancels the alarm  
*Γ<sub>j</sub>* set of all tokens received by  $P_j$  during the current *transition* state, initially empty; the operator *tmax* applied to  $\Gamma_j$  returns the timestamp and process index from the token with the largest timestamp value  
*clock<sub>j</sub>* current value of  $P_j$ 's clock  
*lmsgs<sub>j</sub>*  $P_j$ 's message deficit, initially 0  
*Token(TS<sub>i</sub>, i, gmsgs)* token initiated by  $P_i$  with timestamp  $TS_i$ ; the total of all message deficits for processes which have propagated the token is contained in *gmsgs*

<i>state<sub>j</sub></i>	Event	Action
<i>active</i>	satisfy local predicate	$TS_j \leftarrow clock_j$ $alarm_j(TS_j + \epsilon)$ $\Gamma_j \leftarrow \emptyset$ $state_j \leftarrow transition$
	receive <i>Token(TS<sub>i</sub>, i, gmsgs)</i>	discard <i>Token(TS<sub>i</sub>, i, gmsgs)</i>
	send application msg	$lmsgs_j^{++}$
	receive application msg	$lmsgs_j^{--}$
<i>transition</i>	receive application message	$alarm_j(0)$ $lmsgs_j^{--}$ $state_j \leftarrow active$
	receive <i>Token(TS<sub>i</sub>, i, gmsgs)</i>	$\Gamma_j \leftarrow \Gamma_j \cup \{Token(TS_i, i, gmsgs)\}$
	<i>alarm<sub>j</sub></i> expires	$TS_i, i, gmsgs \leftarrow tmax(\Gamma_j)$ if $(TS_i \geq TS_j)$ then propagate <i>Token(TS<sub>i</sub>, i, gmsgs + lmsgs<sub>j</sub>)</i> else initiate <i>Token(TS<sub>j</sub>, j, lmsgs<sub>j</sub>)</i> $state_j \leftarrow idle$
<i>idle</i>	receive application message	$lmsgs_j^{--}$ $state_j \leftarrow active$
	receive <i>Token(TS<sub>i</sub>, i, gmsgs)</i>	if $(TS_j \leq TS_i) \wedge (i = j) \wedge (gmsgs = 0)$ then declare termination if $(TS_j \leq TS_i) \wedge (i = j) \wedge (gmsgs \neq 0)$ then discard <i>Token(TS<sub>i</sub>, i, gmsgs)</i> if $(TS_j \leq TS_i) \wedge (i \neq j)$ then propagate <i>Token(TS<sub>i</sub>, i, gmsgs + lmsgs<sub>j</sub>)</i> if $(TS_j > TS_i)$ then discard <i>Token(TS<sub>i</sub>, i, gmsgs)</i>

Figure 3.7: Protocol (*TD-Asynch*): TD Under Asynchronous Communication

wide message deficit at this same time instant. If this cumulative deficit is zero, then the system is terminated.

**Algorithm** The modified protocol and associated process state are given in figure 3.7. In addition to the process state required for the previous protocol, each process keeps an additional variable containing its local message deficit. Once again, each process  $P_i$  initiates a token each time that it becomes passive. The token contains a timestamp, the process index of the initiating process  $i$ , and  $P_i$ 's local message deficit. A process receiving the token can either discard it or propagate it. The criteria for token initiation and propagation are the same as that for the previous protocol. Each process establishes an SLP indicating its passivity prior to initiating or propagating a token. Tokens received by passive processes, other than the token's initiator, and which have timestamps greater than or equal to the receiving process' timestamp are propagated; tokens with lower timestamps are discarded. Here, though, a process propagating the token will first add its own local message deficit to the cumulative deficit contained in the token. If a process receives its own token containing a cumulative message deficit of zero, then termination is declared.

**Correctness** In theorem 3, we showed that if a token circulated completely, in the manner of the previous protocol, then the local predicates were all true at the same real time instant. In the previous protocol, each local predicate  $A_i$  was the predicate " $P_i$  is passive." Here we have only modified the SLP asserted by each process, prior to initiating or propagating a token, to handle asynchronous communication. In protocol *TD-SLP*, given in figure 3.4, each process asserted the SLP  $\mathcal{S}(TS_i, i, "P_i \text{ is passive}")$ . In this algorithm, each process asserts the SLP  $\mathcal{S}(TS_i, i, A_i)$  where  $A_i$  is the property " $P_i$  is passive and has a message

deficit of  $lmsgs_i$ ." Since processes cannot send messages from the *transition* or *idle* states and since they *transition* to the *active* state upon receiving a message, each local message deficit remains constant during any *transition* and *idle* intervals. By theorem 3 then, complete circulation of the token, with a cumulative message deficit of zero, indicates that at the instant the token initiator's clock read the timestamp value all processes were passive and there were no outstanding messages.

**Theorem 8 (Safety)** *If any process  $P_j$  declares termination, then the application is terminated.*

PROOF: If  $P_j$  declares termination then every  $P_i$ ,  $i$  in  $SYS$ , has propagated  $P_j$ 's token  $Token(TS_j, j, gmsgs)$  and the received token contained a cumulative message deficit of zero.

By the protocol, every process establishes the SLP  $S(TS_i, i, A_i)$  where  $A_i$  is the property " $P_i$  is passive and has a message deficit of  $lmsgs_i$ ",  $i$  in  $SYS$ , prior to initiating or propagating the token. Further, since passive processes cannot send messages and processes which receive a message transition back to the *active* state, the property asserted by each SLP remains true whenever a process is in the *transition* or *idle* states. By theorem 3, all processes were passive at the earliest instant  $tr_j$  at which  $C_j(tr_j) = TS_j$ , and the global message deficit at that time was zero. Then, under asynchronous communication, the computation must be terminated. ■

The liveness argument is also similar to that made for the previous algorithm. The system is terminated at the earliest instant  $tr_j$  at which  $C_j(tr_j) = TS_{max} + \epsilon$ , where  $TS_{max}$  is the highest valued timestamp generated during the computation and  $P_j$  is a process which generated timestamp  $TS_{max}$ . The proof of this fact, given for the earlier protocol, is not affected by the fact that communication is asynchronous. By clock axiom C1, all

process clocks must read some value greater than  $TS_{max}$  at real time instant  $C_j(tr_j)$ . By the protocol, any process active after this time instant would generate a timestamp greater than  $TS_{max}$ , a contradiction. Then, since the token and process message deficits are not considered when a process decides whether or not to propagate the token, some token with timestamp  $TS_{max}$  will circulate completely. In this case, however, we must also show that the message deficit contained in the token will be zero.

**Theorem 9 (Liveness)** *If the application is terminated, then some  $P_j$  will declare termination.*

PROOF: Consider the highest valued timestamp  $TS_{max}$  generated during the computation. Let  $P_j$  be a process which generated timestamp  $TS_{max}$ . If more than one process generates timestamp  $TS_{max}$ , let  $P_j$  be the first such process to enter the *idle* state. By an argument identical to that made for theorem 7, a token with the highest valued timestamp  $TS_{max}$  will circulate completely. However, in order for termination to be declared, the message deficit contained in that token must be zero.

Each process adds its local message deficit to the token as it circulates. By an argument identical to that made for theorem 8, the message deficits contained in the token must represent the system-wide total at  $tr_j$ , where  $tr_j$  is the earliest instant at which  $C_j(tr_j) = TS_{max}$ . Then, if the system is terminated at  $tr_j$ , the sum of the message deficits contained in the token must be zero.

Thus, some token with timestamp  $TS_{max}$  will circulate completely, the token will contain a message deficit of zero, and the termination will be declared. ■

$state_j$	indicates $P_j$ 's state, initially <i>active</i>
$TS_j$	$P_j$ 's timestamp, initially 0
$alarm_j(T)$	$P_j$ 's alarm which signals $P_j$ at $C_j(t) = T$ ; $alarm_j(0)$ cancels the alarm
$\Gamma_j$	set of all tokens received by $P_j$ during the current <i>transition</i> state, initially empty; the operator $\text{tmax}$ applied to $\Gamma_j$ returns the timestamp and process index from the token with the largest timestamp value
$clock_j$	current value of $P_j$ 's clock
$lastsend_j$	value of $C_j$ at instant of most recent message send
$Token(TS_i, i)$	token initiated by $P_i$ with timestamp $TS_i$
$\delta$	maximum message transmission delay between any two system processes, as measured by any process clock

$state_j$	Event	Action
<i>active</i>	satisfy local predicate	$TS_j \leftarrow clock_j + \max(0, (\delta - clock_j + lastsend_j))$ $alarm_j(TS_j + \epsilon)$ $\Gamma_j \leftarrow \emptyset$ $state_j \leftarrow \text{transition}$
	receive $Token(TS_i, i)$	discard $Token(TS_i, i)$
	send application msg	$lastsend_j \leftarrow clock_j$
<i>transition</i>	receive application message	$alarm_j(0)$ $state_j \leftarrow \text{active}$
	receive $Token(TS_i, i)$	$\Gamma_j \leftarrow \Gamma_j \cup \{Token(TS_i, i)\}$
	$alarm_j$ expires	$TS_i, i \leftarrow \text{tmax}(\Gamma_j)$ if $(TS_i \geq TS_j)$ propagate $Token(TS_i, i)$ else initiate $Token(TS_j, j)$ $state_j \leftarrow \text{idle}$
<i>idle</i>	receive application message	$state_j \leftarrow \text{active}$
	receive $Token(TS_i, i)$	if $(TS_j \leq TS_i) \wedge (i = j)$ then declare termination if $(TS_j \leq TS_i) \wedge (i \neq j)$ then propagate $Token(TS_i, i)$ if $(TS_j > TS_i)$ then discard $Token(TS_i, i)$

Figure 3.8: Protocol (*TD-Bnd*): TD With Bounded Message Transmission Time



**Bounded Message Delay** Many clock synchronization algorithms assume that message transmission time is always within some known bound  $\delta$ . In this case, we can eliminate the message deficit counters of the previous protocol and use the bound on message transmission time to determine that the channels are empty.

The protocol modification becomes clear if we restate the termination predicate as follows. The system is terminated if there is an instant at which every process  $P_i$ ,  $i$  in  $SYS$ , is passive and all of  $P_i$ 's outstanding messages have been received. Since the message delay bound is known, each process knows that all its outstanding messages have been received after an interval of  $\delta$  has elapsed since the instant its last message was sent. Thus, each process  $P_i$  should establish the SLP " $P_i$  is passive and all its messages have been received." The token circulation will then detect an instant at which this property is true for all  $P_i$  and termination can be concluded.

**Algorithm** The protocol and associated process state are given in figure 3.8. Again, each process initiates a token each time that it becomes passive and propagates tokens based on their timestamp value. However, in this protocol, each process must establish the SLP " $P_i$  is passive and all its outstanding messages have been received." It makes this assertion by keeping track of the time at which it last sent a message and using this value to modify the alarm setting. On each transition from the *active* state,  $P_i$  sets its alarm for at least  $\epsilon$ . However, it adds an additional interval to this  $\epsilon$  to ensure that, when the alarm expires, it has been at least  $\epsilon + \delta$  since its last message was sent. By definition,  $\delta$  is the maximum message transmission time *as measured by any process clock*. Thus, when  $P_i$ 's alarm expires it can assert  $S(TS_i, i, A_i)$ , where  $TS_i$  is  $P_i$ 's timestamp and  $A_i$  is the property " $P_i$  is passive and all its outstanding have been received."

**Correctness** The liveness argument for this protocol is identical to that given on page 53 for protocol *TD-SLP*. The system is terminated at real time instant  $C_j^{-1}(TS_{max} + \epsilon)$ , where  $TS_{max}$  is the highest valued timestamp generated during the computation and  $P_j$  is a process which generated timestamp  $TS_{max}$ . Then by the protocol, some token with timestamp  $TS_{max}$  will circulate completely. Here we give only the safety argument.

Once again, we only need prove that each process which propagates the token asserts an SLP which remains true throughout the *transition* and *idle* intervals and which, when asserted by all processes, ensures termination. Once this is established, the safety argument is the same as that given for the previous protocols. Complete circulation of the token indicates that the property held for all processes at the same instant in real time. Here we only prove that each process  $P_i$  can assert the property " $P_i$  is passive and all of  $P_i$ 's messages have been received" and that this property remains true throughout the *transition* and *idle* intervals.

**Lemma 5** *If  $P_i$  propagates  $Token(TS_j, j)$  then  $S(TS_i, i, A_i)$ , where  $A_i$  is the property " $P_i$  is passive and all its outstanding messages have been received." Further, once the SLP is established, it remains true until the instant the token is propagated.*

**PROOF:** Let  $TS_i$  be  $P_i$ 's timestamp when it initiates or propagates  $Token(TS_j, j)$ . When  $P_i$  satisfies its local predicate, it sets its timer for the maximum value of  $\epsilon$  and  $\delta - clock_j + lastsend_j$ .  $C_j$ 's timer then cannot expire until it has been at least  $\epsilon + \delta$  since its last message send. In order for the timer to expire,  $P_i$  must remain continuously passive from the instant it set the timer and, by the protocol,  $P_i$ 's timer must expire before it will initiate or propagate a token. Then, since  $\delta$  is the maximum message transmission time as read by any process clock,  $P_i$  can assert the SLP  $S(TS_i, i, A_i)$  where  $A_i$  is the property " $P_i$  is passive

and all its outstanding messages have been received." Further, since passive process cannot send messages and  $P_i$  must be passive in order to remain in the *idle* state and propagate tokens, the property will remain true until the instant at which  $P_i$  propagates any token using timestamp  $TS_i$ . ■

**Discussion** Both asynchronous communication algorithms require  $O(MN)$  messages to detect the termination. The detection delay is  $O(N)$ . Each requires an additional delay prior to initiating the final token to account for the rough clock synchrony and, in the case of bounded message delay, the maximum message transmission interval. Further, both algorithms are fully distributed and symmetric.

A number of algorithms have been presented for static systems with reliable asynchronous communication including [1, 50, 32, 43, 5, 47, 12, 4]. However, many of these are not symmetric [50, 32, 43, 12, 4], or make additional constraints on message delivery order [5, 47].

Few of these algorithms are symmetric. Arora et. al introduced a symmetric algorithm for distributed termination detection in [1]. However, their algorithm required that each process  $P_i$  sends a message to every neighbor  $P_j$  each time that  $P_i$  transitions from active to passive.

Thus, their algorithm can require significantly more message passing. The detection delay of their protocol is also  $O(N)$ . Thus, by structuring the algorithm on the use of a global time base, we are able to develop a fully distributed algorithm that is simple and efficient. Further, if all system clocks are known to be accurate, then we know not only that the system is terminated, but we also know that it was terminated at the token timestamp value in real time.

*state<sub>j</sub>* indicates  $P_j$ 's state, initially *active*  
*TS<sub>j</sub>*  $P_j$ 's timestamp, initially 0  
*alarm<sub>j</sub>(T)*  $P_j$ 's alarm which signals  $P_j$  at  $C_j(t) = T$ ; *alarm<sub>j</sub>(0)* cancels the alarm  
*Γ<sub>j</sub>* set of all tokens received by  $P_j$  during the current *transition* state, initially empty  
*clock<sub>j</sub>* current value of  $P_j$ 's clock  
*Token(TS<sub>i</sub>, i)* token initiated by  $P_i$  with timestamp  $TS_i$   
*Reply(TS<sub>i</sub>)* reply sent to  $P_i$  in response to *Token(TS<sub>i</sub>, i)*  
*nreplies<sub>j</sub>* number of replies received in response to  $P_j$ 's most recently timestamped token

<i>state<sub>j</sub></i>	Event	Action
<i>active</i>	satisfy local predicate	$TS_j \leftarrow clock_j$ $alarm_j(TS_j + \epsilon)$ $\Gamma_j \leftarrow \emptyset$ $state_j \leftarrow transition$
	receive <i>Token(TS<sub>i</sub>, i)</i>	discard <i>Token(TS<sub>i</sub>, i)</i>
	receive <i>Reply(TS<sub>i</sub>)</i>	discard <i>Reply(TS<sub>i</sub>)</i>
<i>transition</i>	receive application message	$alarm_j(0)$ $state_j \leftarrow active$
	receive <i>Token(TS<sub>i</sub>, i)</i>	$\Gamma_j \leftarrow \Gamma_j \cup \{Token(TS_i, i)\}$
	<i>alarm<sub>j</sub></i> expires	broadcast <i>Token(TS<sub>j</sub>, j)</i> for all $Token(TS_i, i) \in \Gamma_j$ if $(TS_i \geq TS_j)$ then send <i>Reply(TS<sub>i</sub>)</i> to $P_i$ $nreplies_j \leftarrow 1$ $state_j \leftarrow idle$
	receive <i>Reply(TS<sub>i</sub>)</i>	discard <i>Reply(TS<sub>i</sub>)</i>
<i>idle</i>	receive application message	$state_j \leftarrow active$
	receive <i>Token(TS<sub>i</sub>, i)</i>	if $(TS_i < TS_j)$ then discard <i>Token(TS<sub>i</sub>, i)</i> else send <i>Reply(TS<sub>i</sub>)</i> to $P_i$
	receive <i>Reply(TS<sub>i</sub>)</i>	if $(TS_i = TS_j)$ then $nreplies_j^{++}$ if $(nreplies_j = N)$ then declare termination

Figure 3.9: Protocol (*TD-Bcast*): TD with Broadcasts

### 3.1.3.3 Token Broadcasts

We can reduce the detection delay of our earlier protocols significantly by broadcasting the token. We assume that broadcasts can be made by creating and transmitting a single message. Further, by using token broadcasts, only a single message is required to notify all processes of the termination, once it is detected.

If we also assume well-ordered message passing, that is, the first message sent by any process is the first message received by any process, then we can eliminate the need for message counters. We consider a message sent when it is “on the wire,” and we consider a message received when it is ready for processing by the application process. (This is described in more detail below.) Both of these assumptions (single message broadcasts and well-ordered message delivery) would apply to a set of processors connected with an Ethernet and using an IP network layer.

**Algorithm** We refine our system model for this protocol. First, each process is assumed to have a message buffer in which incoming messages are stored until they can be processed. Both application messages and control messages are stored in this same buffer. Application messages take highest priority and will always be acted upon first. Control messages are assumed to be stored and processed in FIFO order. A control message can only be processed when there are no application messages in the buffer. A message is considered sent when it is “on the wire;” all application messages generated during an active interval must be on the wire before a process becomes passive. A message is considered received when it has been stored in the receiving process’ message buffer. We assume well-ordered message delivery, that is, the first message sent to a process  $P_i$ , by any process, is the first message

received by  $P_i$ . (As we pointed out in our general system model, we further assume reliable message passing.)

Again, the algorithm is symmetric. Each process broadcasts a token each time that it becomes passive. The token contains the initiator's timestamp and process index. Each process receiving the token sends a *reply* to the token initiator if its own timestamp is less than or equal to the token timestamp. If a process receives a reply from all other processes, then the computation is terminated. The protocol and associated process state are given in figure 3.9.

In this case, token propagation establishes the SLP " $P_i$  is passive and has received all messages destined for  $P_i$ ." If each process asserts this property for the same instant in time, there can be no outstanding messages and the computation is terminated. The protocol requires no specific action to ensure that all outstanding messages have been received at the timestamp value. This is a result of assuming well-ordered lossless message passing. Under this assumption, the token will flush the channels of any messages sent at or before the token timestamp value.

**Correctness** The liveness argument for this protocol is virtually identical to that made for protocol *TD-Asynch* (figure 3.7). Some process  $P_j$  will generate a token with the highest valued timestamp  $TS_{max}$  generated during the computation and the system is terminated at the earliest real time instant  $tr_j$  for which  $C_j(tr_j) = TS_{max} + \epsilon$ . The channels will then be empty, all processes will then reply to  $Token(TS_{max}, j)$ , and  $P_j$  will detect the termination. We do not repeat the liveness argument here.

Also, the safety argument is similar to that of previous protocols. Once we prove that token propagation (in this case, sending a reply to the token initiator) establishes the

appropriate local property with the token timestamp value, propagation of the token by all processes ensures that the local property was true for all processes at the same time instant.

Here, we give only the proof of the SLP asserted by token propagation.

**Lemma 6** *If  $P_i$  propagates  $Token(TS_j, j)$  then  $S(TS_j, j, A_i)$ , where  $A_i$  is the property " $P_i$  is passive and all messages destined for  $P_i$  have been received." Further, this property will remain true until the earliest real time instant  $tr_j$  for which  $C_j(tr_j) = TS_j + \epsilon$ .*

**PROOF:** Let  $P_i$ 's timestamp be  $TS_i$  when it propagates  $Token(TS_j, j)$  and let  $tt_i$  be the real time instant at which it propagated the token. By an argument identical to that made in earlier protocols, if  $P_i$  propagates the token then it can assert  $S(TS_j, j, A_i)$ , where  $A_i$  is the property " $P_i$  is passive." Further,  $P_i$  must remain passive until the instant at which it propagates the token.

Now, consider the instant  $tp_j$  at which  $P_j$  initiates the token. By the protocol,  $tp_j \geq tr_j$ . Given FIFO delivery, all application messages sent prior to  $tp_j$  must already have been received by  $P_i$ . If any message had been in the channels at that time,  $P_i$  would have processed the message, and been reactivated. If  $P_i$  remains passive, then there were no application messages in the channels at  $tp_j$ . Thus, if  $P_i$  propagates the token, it has received all messages destined for  $P_i$ . ■

**Performance** This algorithm is  $O(MN)$  in the number of messages required to detect the termination. However, the detection delay is lower than those in which the token is propagated in a unidirectional ring of the system processes. Let  $\delta_s$  be the maximum time required to send a control message and  $\delta_r$  be the time required to receive a message. Assume that the time the message is on the wire and to the time to execute protocol instructions

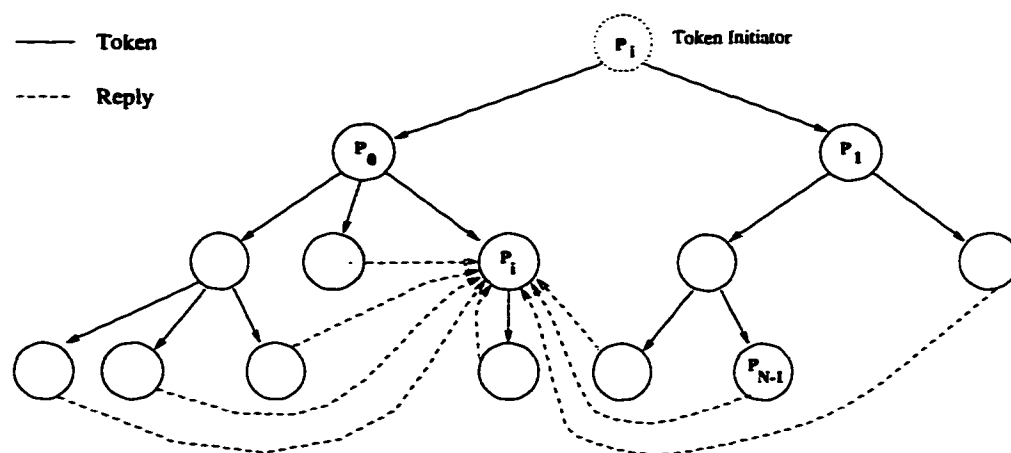


Figure 3.10: Tree Token Routing

once the message is received are negligible compared to the message delivery times. Then the earlier protocols require  $N(\delta_s + \delta_r) + 3\epsilon(1 + \rho_M)$  to detect the termination. They also require an additional delay of  $(N - 1)(\delta_s + \delta_r)$  to notify all processes of the termination. The broadcast algorithm only requires  $\delta_s + \delta_r + 3\epsilon(1 + \rho_M)$ , in the worst case, to detect the termination. Further, it only requires a delay  $\delta_s + \delta_r$  to notify all processes of the termination.

Thus, by a simple routing modification, we reduce the detection delay significantly, without an increase in the worst case message complexity.

#### 3.1.3.4 Tree-Based Routing

The previous algorithm reduced the detection delay from that of earlier protocols. However, it required well-ordered message passing and single message broadcasts. In the following algorithm, we achieve a similar result without making either of these restrictions on the system. This is accomplished by using a tree-based token routing scheme.



$state_j$	indicates $P_j$ 's state, initially <i>active</i>
$alarm_j(T)$	$P_j$ 's alarm which signals $P_j$ at $C_j(t) = T$ ; $alarm_j(0)$ cancels the alarm
$\Gamma_j$	set of all tokens received by $P_j$ during the current <i>transition</i> state, initially empty; the operator $t_{\max}$ applied to $\Gamma_j$ returns the timestamp, process index, and message deficit from the token with the largest timestamp value
$clock_j, TS_j$	current value of $P_j$ 's clock and $P_j$ 's timestamp (initially 0), respectively
$C_j, lchild$	set of all $P_j$ 's children and $P_j$ 's left child, respectively
$lmsgs_j$	$P_j$ 's message deficit, initially 0
$Token(TS_i, i, gmsgs)$	token initiated by $P_i$ with timestamp $TS_i$ ; the total of all message deficits for processes which have propagated the token is contained in $gmsgs$
$Reply(TS_i, gmsgs)$	reply containing message deficit $gmsgs$ sent in response to a token initiated by $P_i$ with timestamp $TS_i$
$nleaves$	number of leaf nodes in the system
$\mathcal{I}$	set of processes which first receive an initiated token
$msgtotal_j$	sum of message deficits from replies received by $P_j$
$nreplies_j$	number of replies received, for a given token, by $P_j$

$state_j$	Event	Action
<i>active</i>	satisfy local predicate	$TP_j \leftarrow clock_j + \epsilon$ $alarm_j(TP_j + \epsilon)$ ; $\Gamma_j \leftarrow \emptyset$ ; $nreplies_j \leftarrow 0$ ; $msgtotal_j \leftarrow 0$ if $(C_j = \emptyset)$ then $nreplies_j \leftarrow 1$ ; $msgtotal_j \leftarrow lmsgs_j$ $state_j \leftarrow transition$
	receive $Token(TS_i, i, gmsgs)$	discard $Token(TS_i, i, gmsgs)$
	receive $Reply(TS_i, gmsgs)$	discard $Reply(TS_i, gmsgs)$
	send (receive) application msg	$lmsgs_j^{++}$ ( $lmsgs_j^{--}$ )
<i>transition</i>	receive $Reply(TS_i, gmsgs)$	discard $Reply(TS_i, gmsgs)$
	receive $Token(TS_i, i, gmsgs)$	$\Gamma_j \leftarrow \Gamma_j \cup \{Token(TS_i, i, gmsgs)\}$
	application msg received	$alarm_j(0)$ ; $lmsgs_j^{--}$ ; $state_j \leftarrow active$
	$alarm_j$ expires	$TS_i, i, gmsgs \leftarrow t_{\max}(\Gamma_j)$ if $(TS_i \geq TS_j)$ then if $(C_j = \emptyset)$ then send $Reply(TS_i, gmsgs + lmsgs_j)$ to $P_i$ else send $Token(TS_i, i, gmsgs + lmsgs_j)$ to $lchild$ send $Token(TS_i, i, 0)$ to all $P_i \in C_j, P_i \neq lchild$ if $(TS_j > TS_i)$ then send $Token(TS_j, j, 0)$ to all $P_i \in \mathcal{I}$ $state_j \leftarrow idle$
<i>idle</i>	application msg received	$lmsgs_j^{--}$ ; $state_j \leftarrow active$
	receive $Token(TS_i, i, gmsgs)$	if $(TS_i \geq TS_j)$ then if $(C_j = \emptyset)$ then send $Reply(TS_i, gmsgs + lmsgs_j)$ to $P_i$ else send $Token(TS_i, i, gmsgs + lmsgs_j)$ to $lchild$ send $Token(TS_i, i, 0)$ to all $P_i \in C_j, P_i \neq lchild$ else discard $Token(TS_i, i, gmsgs)$
	receive $Reply(TS_i, gmsgs)$	if $(TS_i = TS_j)$ then $nreplies_j^{++}$ ; $msgtotal_j \leftarrow msgtotal_j + gmsgs$ if $(nreplies_j = nleaves) \wedge (msgtotal_j = 0)$ then declare termination else discard $Reply(TS_i, gmsgs)$

Figure 3.11: Protocol (TD-Tree): TD using Tree Routing

**Algorithm** Here again, each process initiates a token each time that it transitions from active to passive. The initiated token contains a timestamp, message deficit, and the initiator's process index.

The token routing is shown in figure 3.10. The system processes are assembled into a tree with an imaginary root. A process assumes the root position when it initiates a token. Thus, each process initiates a token by sending copies of the token to the same processes, e.g.,  $P_0$  and  $P_1$  of figure 3.10.

When an internal node  $P_i$  propagates a token, the token is replicated and a copy is sent to each child. The message deficit in the left child's copy of the token is the sum of the deficit in the received token and  $P_i$ 's own local deficit. The message deficit in the tokens sent to all other children is zero. The process id and timestamp are unchanged. When a leaf node receives the token, it propagates a *reply* to the token initiator. The reply contains the token timestamp and a message deficit, which is the sum of the leaf's local deficit and the deficit in the received token.

The criterion for token propagation is the same as in our earlier algorithms. Tokens that are received by a passive process and that have timestamps greater than or equal to the receiver's timestamp are propagated. All others are discarded. Termination is declared if the initiator receives replies from all leaf nodes, the replies contain the initiator's current timestamp, and the message deficits in the replies sum to zero. The protocol and associated process state are given in figure 3.11.

**Correctness** This protocol is similar to protocol *TD-Asynch* (figure 3.7) except for the modifications made to the token routing. The criteria for token propagation and declaration of termination are the same. Thus, by an argument similar to that made for theorem 8, if

$P_j$  initiates or propagates a token then it has established the SLP  $S(TS_j, j, A_i)$ , where  $A_i$  is the property “ $P_i$  is passive and has a message deficit of  $lmsgsi$ ”. Further, this property remains true while  $P_j$  is in the *transition* and *idle* states. Then, in order to apply theorem 3 we only need show that each process must propagate the token in order for termination to be declared, which is apparent from the protocol.

The liveness argument, too, is essentially unchanged from that given for the earlier protocol. We do not repeat it here.

**Performance** This algorithm is  $O(MN)$  messages in the number of messages required to detect the termination. The detection delay is  $O(H)$ , where  $H$  is the tree height. Nicol also gave a tree-based termination detection algorithm for systems with asynchronous communication [50]. He assumes that the system is formed into a complete binary tree. His algorithm is also  $O(\log N)$  in the detection delay. However, his algorithm’s best case performance can be considerably better, whereas our algorithm’s detection delay is not variable.

Lai gave a tree-based algorithm for dynamic systems with reliable asynchronous communication [32]. His algorithm is also  $O(H)$  in the detection delay and  $O(MN)$  in the number of messages required to detect the termination.

However, both Nicol’s and Lai’s algorithms require significantly more state information,  $O(N)$  space complexity, in each process and on each message.

### 3.1.3.5 Reduced Active Process Interruption

The following algorithm is a modification to protocol *TD-Asynch* given on page 61. Here the token routing is modified to reduce the number of interruptions to active processes. By the following protocol, no active process receives more than one control message during

$flag_j$	indicates whether or not $P_j$ can initiate a token when it transitions to the idle state, initially false for all processes except $P_0$ , which initially has $flag_j$ equal to true
$state_j$	indicates $P_j$ 's state, initially active
$TS_j$	$P_j$ 's timestamp, initially 0
$alarm_j(T)$	$P_j$ 's alarm which signals $P_j$ at $C_j(t) = T$ ; $alarm_j(0)$ cancels the alarm
$\Gamma_j$	triple containing the timestamp, process index, and message deficit from a token received in the transition state; initially (0, 0, 0)
$clock_j$	current value of $P_j$ 's clock
$lmsgs_j$	number of application messages sent minus then number received by $P_j$
$gmsgs$	cumulative total of send and receive counts contained in a token received or propagated by $P_j$
$Token(TS_i, i, gmsgs)$	token initiated by $P_i$ with timestamp $TS_i$ , the total application message sends and receives for all processes which have propagated the token are contained in $gmsgs$

$state_j$	Event	Action
active	satisfy local predicate	$TS_j \leftarrow clock_j$ $alarm_j(TS_j + \epsilon)$ $\Gamma_j \leftarrow (0, 0, 0)$ $state_j \leftarrow transition$
	receive $Token(TS_i, i, gmsgs)$	discard $Token(TS_i, i, gmsgs)$ $flag_j \leftarrow true$
	send application msg	$lmsgs_j^{++}$
	receive application msg	$lmsgs_j^{--}$
transition	receive application message	$lmsgs_j^{--}$ $state_j \leftarrow active$
	receive $Token(TS_i, i, gmsgs)$	$flag_j \leftarrow true$ if $(TS_i < TS_j)$ then discard $Token(TS_i, i, gmsgs)$ else $\Gamma_j \leftarrow (TS_i, i, gmsgs)$
	$alarm_j$ expires	$(TS_i, i, gmsgs) \leftarrow \Gamma_j$ if $(\Gamma_j \neq \emptyset)$ then propagate $Token(TS_i, i, gmsgs + lmsgs_j)$ $flag_j \leftarrow false$ if $(flag_j = true)$ then initiate $Token(TS_j, j, lmsgs_j)$ $flag_j \leftarrow false$ $state_j \leftarrow idle$
idle	receive application message	$alarm_j(0)$ $lmsgs_j^{--}$ $state_j \leftarrow active$
	receive $Token(TS_i, i, gmsgs)$	if $(TS_j \leq TS_i) \wedge (i = j) \wedge (gmsgs = 0)$ then declare termination if $(TS_j \leq TS_i) \wedge (i = j) \wedge (gmsgs \neq 0)$ then discard token if $(TS_j \leq TS_i) \wedge (i \neq j)$ then propagate $Token(TS_i, i, gmsgs + lmsgs_j)$ if $(TS_j > TS_i)$ then discard $Token(TS_i, i, gmsgs)$ initiate $Token(TS_j, j, lmsgs_j)$

Figure 3.12: Protocol (TD-Quiet): TD with Reduced Control Communication

any single active period. This is accomplished by ensuring that at any given instant in the computation either a single token is circulating, or there is no token circulating and a single process, currently in the *active* or *transition* states, will initiate a token when its alarm next expires.

**Algorithm** The algorithm and associated process state are shown in figure 3.12. Initially, only  $P_0$  can, and will, initiate a token. Once this token is in circulation, only a process which discards a received token can, and will, initiate a new token. Thus, only one token is ever in circulation. Otherwise, the protocol is the same as protocol *TD-Asynch* (figure 3.7). Each process  $P_i$  establishes the SLP " $P_i$  is passive and has a local message deficit of  $lmsgs_i$ ," prior to initiating or propagating a token. A token is only propagated by passive processes whose local timestamp is equal to, or less than, that on the received token. Each process adds its own local deficit to the total contained in the token prior to propagating it. Termination is declared when a process receives its own token containing a message deficit of zero.

**Correctness** In order for  $P_j$  to receive its own token, it must be propagated by each process and the criterion for token propagation is the same as that for protocol *TD-Asynch* (figure 3.7). Thus, the safety argument for this protocol is the same as that given for the earlier protocol. Here we give only the more complex liveness argument.

The following lemmas are useful in establishing the correctness of our protocol.

**Lemma 7** Suppose  $P_j$  asserts  $S(TS_j, j, A_j)$  at real time instant  $tr_j$  and that  $P_i$  receives this assertion at real time instant  $tt_i$ . If  $C_j(tr_j) > TS_j + \epsilon$  then  $C_i(tt_i) > TS_j$ .

**PROOF:** Clearly, given non-zero message transmission time,  $tt_i \geq tr_j$ . By clock axiom C1,  $C_i(tr_j) > TS_j$ . Then, by clock axiom C2,  $C_i(tt_i) > TS_j$ . ■

**Lemma 8** *Other than the first token initiated by  $P_0$ , for each token  $\text{Token}(TS_j, j, lmsgsj)$   $P_j$  initiates, it has previously discarded a token  $\text{Token}(TS_i, i, gmsgi)$  where  $TS_i < TS_j$ .*

**PROOF:** In order to initiate a token, a process  $P_j$  must either receive a token while in the *idle* state or it must have the variable  $flag_j$  set to **true**. By the protocol, initially only  $P_0$  has  $flag_j$  set to **true** and there are no tokens in circulation. The variable  $flag_j$  can only be set from **false** to **true** upon receipt of a token. Thus, only  $P_0$  can initiate the first token.  $P_0$  sets  $flag_0$  to **false** immediately after initiating the first token.

Tokens can only be initiated from the *idle* or *transition* states. By the protocol, clearly a process in the *idle* state can only initiate a token after it has discarded one with a lower timestamp.

$P_j$  can only initiate a token from the *transition* state if the variable  $flag_j$  is **true**. Initially,  $flag_j$  is **false** for all processes except  $P_0$ , which sets  $flag_0$  to **false** after initiating the first token.

By the protocol,  $P_j$  can only set  $flag_j$  to **true** while in the *active* or *transition* states. Now suppose  $P_j$  sets  $flag_j$  to **true** while in the *active* state. Then, by the protocol, it has discarded a token. By lemma 7 and the protocol,  $P_j$ 's next timestamp will be greater than that on the discarded token. Thus, if  $P_j$  initiates a token from the *transition* state after setting  $flag_j$  to **true** in the *active* state, then it has discarded a token with a lower timestamp.

Now suppose  $P_j$  sets  $flag_j$  to **true** in the *transition* state. Then it has received a token  $\text{Token}(TS_i, i, gmsgi)$ . If  $TS_i < TS_j$ , then  $P_j$  will discard the token and any subsequently initiated token will have a higher timestamp. Suppose that  $TS_i \geq TS_j$ . Then either  $P_j$  will propagate the token and reset  $flag_j$  to **false**, in which case no token is initiated, or  $P_j$

will transition back to the *active* state. If  $P_j$  transitions to the *active* state, then  $\Gamma_j$  will be reset prior to initiation of any token, effectively discarding the received token. By lemma 7 and the protocol, any subsequently initiated token will have a higher timestamp. Thus, if  $P_j$  initiates a token from the *transition* state after setting  $flag_j$  to true in the *transition* state, then it has discarded a token and any subsequently initiated token will have a higher timestamp. ■

**Lemma 9**  $P_j$  receives a single token during any interval in the *transition* state.

PROOF: By lemma 8 there can be only one token in circulation at any time. Thus, if  $P_j$  receives a token in the *transition* state, it cannot receive another token until it either propagates the token or initiates a token of its own. By the protocol,  $P_j$  only propagates or initiates tokens from the *transition* state when its alarm expires, immediately prior to entering the *idle* state. Thus,  $P_j$  receives a single token during any interval in the *transition* state. ■

**Lemma 10** If  $P_j$  discards a token  $Token(TS_i, i, gmsgs)$ , then  $TS_j > TS_i$  and  $P_j$  will initiate a token  $Token(TS_j, j, lmsgs_j)$

PROOF: A process can discard a token from any of the three states, *active*, *transition*, or *idle*. Suppose  $P_j$  discards a token in the *active* state. We must show that  $P_j$  will eventually initiate a token and that the timestamp on this token will be greater than that on the received token.

If  $P_j$  discards a token  $Token(TS_i, i, gmsgs)$  in the *active* state, it sets the variable  $flag_j$  to true.  $P_j$  will then initiate a token when its alarm next expires (which it must eventually, if the computation terminates) unless it sets  $flag_j$  to false prior to initiating

the token. In order for  $P_j$  to set  $flag_j$  to false,  $\Gamma_j$  must be non-empty. However, by lemma 8, only one token can be in circulation at any time. Thus,  $P_j$  cannot have received a token since it received  $Token(TS_i, i, gmsgs)$  and  $\Gamma_j$  must be empty. Further, by lemma 7, when the alarm expires,  $P_j$ 's timestamp will be greater than  $TS_i$ . Thus, if  $P_j$  discards a token  $Token(TS_i, i, gmsgs)$  from the *active* state, then it will eventually initiate a token  $Token(TS_j, j, lmsgs_j)$  with  $TS_j > TS_i$ .

Suppose that  $P_i$  discards a token while in the *idle* state. In this state, it only discards tokens which have timestamps less than its own and does so immediately prior to initiating a token with its own timestamp.

Suppose, finally, that  $P_j$  discards a token  $Token(TS_i, i, gmsgs)$  in the *transition* state. Then, by the protocol, it will only discard the token if  $TS_i < TS_j$ . At this point it will set  $flag_j$  to true. Then, by the protocol,  $P_j$  will initiate a token  $Token(TS_j, j, lmsgs_j)$  when its alarm next expires unless  $flag_j$  is set back to false before  $Token(TS_j, j, lmsgs_j)$  is initiated.  $P_j$  will only set  $flag_j$  back to false if  $\Gamma_j$  is non-empty when the alarm expires. In order for  $\Gamma_j$  to be non-empty, either it must already be non-empty, prior to the reception of  $Token(TS_i, i, gmsgs)$ ,  $P_j$  must receive another token between the time it received  $Token(TS_i, i, gmsgs)$  and the time its alarm expires. By the protocol,  $P_j$  empties  $\Gamma_j$  prior to entering the *transition* state. By lemma 9  $P_j$  can only receive a single token while in the *transition* state. By lemma 8, there can only be one token in circulation, and  $P_j$  cannot receive another token until it initiates one. Thus,  $P_j$  will not set  $flag_j$  back to false and will initiate a token, with a higher timestamp, when its alarm next expires.

Thus, if any process discards a token, it initiates a token of its own with a greater timestamp. ■



**Lemma 11**  $P_j$  discards a token  $\text{Token}(TS_i, i, gmsgs)$  if and only if it subsequently initiates a token  $\text{Token}(TS_j, j, gmsgs)$  where  $TS_j > TS_i$ .

PROOF: This is a direct result of lemmas 8 and 10.

**Theorem 10 (Liveness)** *If the application is terminated, then some  $P_j$  will declare termination.*

PROOF: Consider the process  $P_j$  which generates the highest valued timestamp  $TS_{max}$  generated during the computation. If more than one process generates this timestamp, consider any such process. By the same argument given for lemma 4, the system is terminated at the earliest instant  $tr_j$  for which  $C_j(tr_j) = TS_{max} + \epsilon$ .

By lemma 11, unless some process has already declared termination, some process with timestamp  $TS_{max}$  will eventually initiate a token. Clearly, this token will be initiated at an instant later than or equal to  $tr_j$ . Since the system is terminated at this instant, all processes receiving this token will be in the *transition* or *idle* states. By the protocol, all processes in the *transition* and *idle* states propagate tokens with equal or greater timestamps. Since no process has a timestamp greater than  $TS_{max}$ , all processes will propagate this token.

By the same argument given in theorem 9, the local message counts in the token represent the local totals at  $tr_j$ . Then, since the system is terminated, the message deficits must sum to zero and the termination will be declared. ■

**Performance** This algorithm trades an increase in detection delay for a decrease in the number of interruptions to active processes. In the worst case, it will require two token circulations to detect the termination rather than a single circulation. The worst case message passing is  $O(MN)$ . Thus, its performance is similar to that of the first algorithm

presented in this chapter, Dijkstra's termination detection algorithm for systems with synchronous communication. However, by basing the algorithm on the use of time, we achieve a simple symmetric algorithm capable of handling the more complex case of asynchronous communication.

### 3.1.4 Conclusions

The termination detection protocols presented in this section perform well in comparison to protocols which make no assumption about the existence of a global time base. Like protocols which require two passes in order to detect the termination [40], no information need be appended to application messages. Like protocols based on the use of logical clocks [40, 25] they are (always) able to detect termination in a single pass. Further, each process knows when it should initiate a control wave, so the number of control messages is bounded.

The same general approach is readily adapted to varying topologies and performance requirements. These protocols further demonstrated that the use of SLPs and GLPs simplifies protocol development and verification within a rough global time base.

## 3.2 Distributed Deadlock

The problem of distributed deadlock detection has been studied extensively [7, 46, 14, 17, 59, 20, 58]. It is a problem of concurrency control, coordinating the actions of processes that operate in parallel, access shared resources, and therefore potentially interfere with each other. Although some concurrency control protocols are deadlock free, most are vulnerable to deadlock [30].

Detecting and resolving distributed deadlock is a complex problem. The complexity

of the distributed deadlock detection problem is evident solely by the number of protocols proved correct [6, 46, 24, 51, 26, 59], which were then later proved incorrect [30, 20, 27, 15, 20, 17]. This complexity is partially attributable to the lack of a global time base [52]. Further, the problem is similar to that of detecting distributed termination. Therefore, before we turn our attention to general stable predicate evaluation, we study how a rough global time base can be applied to the problem of distributed deadlock detection.

There are several models of distributed deadlock including the One-resource, AND, OR, AND-OR,  $\binom{n}{k}$ , and Unrestricted models[30]. We first give a distributed database model followed, in section 3.2.1, by a description of the One-resource model of distributed deadlock. Then, in section 3.2.3, we present a brief overview of previous solutions to the general problem (all models) of distributed deadlock detection. We then give a time-based solution in section 3.2.4 for the One-resource model. Finally, we conclude in section 3.2.5 with a discussion of the application of our technique to the more general, and complex, deadlock models.

### 3.2.1 System Model

We consider a database system comprised of the following components:

- a static set  $D$  of  $n$  non-terminating *data manager* processes,  $\{D_1, D_2, \dots, D_n\}$ ,
- a set  $R$  of data resources,
- a set  $M$  of  $m$  non-terminating *transaction manager* processes  $\{M_1, M_2, \dots, M_m\}$ , and
- a set  $P$  of transaction processes.

Each data manager  $D_i$  is bound to a single node of the network and controls access to resource  $R_i$ , which is assumed to reside at that node. Similarly, transaction manager  $M_i$  executes at a single node of the network and controls a single transaction process  $P_i$ . All processes, and the network, are assumed to be fully connected and reliable.

In order to access a resource, a transaction must first receive permission from the data manager responsible for controlling the resource. Transactions do not request resources directly from data managers. A transaction's requests are handled by its transaction manager, which then communicates directly with the appropriate data manager.

We assume two-phase locking is used for concurrency control. In two-phase locking, a transaction which has released a lock may not obtain any more locks. A transaction manager sends a *Request* message to lock a data resource. If the resource is available, then the data manager will reply with a *Grant* message. Otherwise, the data manager sends a *Hold* message to indicate that another transaction currently has the resource. When a transaction has all the necessary locks, then it may read and write the data resource. After the transaction has committed or aborted its changes, it releases the resource by sending a *Release* message to the appropriate data manager.

Processes are assumed to communicate via reliable, asynchronous message passing with arbitrary delivery order. We define four types of messages through which transaction managers and data managers coordinate resource requests.

- *Request*( $i, q$ ) is a message sent from a transaction manager  $M_i$  to data manager  $DM_q$  requesting resource  $q$ .
- *Release*( $i, q$ ) is a message sent from a transaction manager  $M_i$  to data manager  $DM_q$  when  $P_i$  releases resource  $q$ .

- **Grant**( $q$ ) is a message sent from a data manager  $DM_q$  to a transaction manager  $M_i$  granting  $M_i$  access to resource  $q$ .
- **Hold**( $q$ ) is a message sent from a data manager  $DM_q$  to a transaction manager  $M_i$  notifying  $M_i$  that the requested resource  $q$  is not available.

We further define four interprocess communication primitives by which communication between a transaction and its transaction manager take place. These communications are assumed to fully reliable and synchronous.

- **Required**( $\mathcal{R}_i$ ) is a communication between transaction  $P_i$  and its transaction manager  $M_i$  of the set of resources  $\mathcal{R}_i$  required by  $P_i$ . Upon issuing **Required**( $\mathcal{R}_i$ ),  $P_i$  is blocked until it is granted the needed resource(s); which resources are required in order for the transaction to continue is a property of the deadlock model.
- **Obtained**( $q$ ) is a communication from transaction manager  $M_i$  to transaction  $P_i$  that the resource  $q$  has been granted to  $P_i$ .
- **Free**( $q$ ) is a communication from transaction  $P_i$  to its manager  $M_i$  that resource  $q$  can be released. Once  $P_i$  issues **Free**( $q$ ) it cannot request any more resources.
- **Abort** is a communication from transaction manager  $M_i$  to transaction  $P_i$  aborting the transaction.

Transactions are either *blocked* or *executing*. Each transaction presents a single resource request to its transaction manager. This request may be for a single resource or it may have a more abstract meaning, such as a request for multiple resources. A transaction process is blocked from the time it presents the request to its transaction manager until the request is

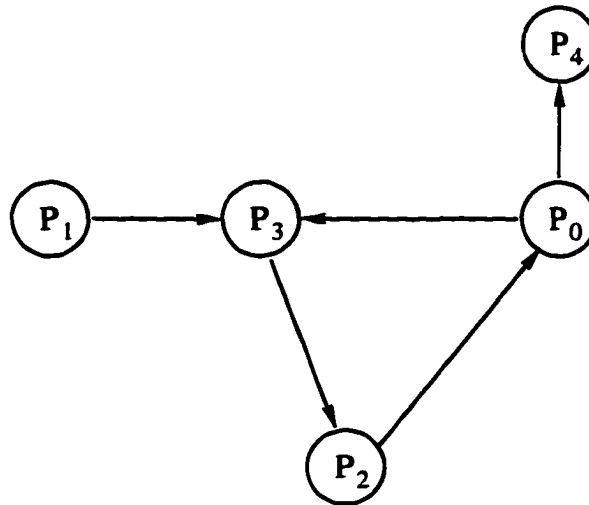


Figure 3.13: Transaction Wait For Graph (WFG)

granted. When it has received the resources it needs to proceed, then it is executing. (The deadlock model under consideration dictates which resources, of the set requested by the  $\text{Required}(\mathcal{R}_i)$  primitive, must be granted in order for the transaction to proceed.)

A transaction wait for graph (WFG) is a directed graph used to model resource requests. The vertices of the graph represent transactions. Directed edges represent the blocking relation between transactions. If transaction  $P_i$  is blocked waiting for a resource currently held by  $P_j$ , then there is a directed edge from  $P_i$  to  $P_j$ . A WFG is shown in figure 3.13. Whether or not the graph contains a deadlock depends upon the system deadlock model.

### 3.2.2 Problem

The algorithm we present is for the simplest model, the single resource model. Here transactions can have at most one outstanding resource request. Under this model, finding a deadlock corresponds to finding a cycle in the WFG.

The solution is very similar to the one given in figure 3.4 for termination detection. Since we are looking for a cycle in the WFG, the token does not traverse all system processes, as in

termination detection, but traverses the WFG via the appropriate transaction manager and data manager processes.  $P_i \rightarrow P_j$  denotes the dependence of transaction  $P_i$  on a resource currently held by transaction  $P_j$  at an instant in real time. The system is deadlocked if there is a set of transaction processes  $S = P_i, P_j, \dots, P_k$  such that at some real time instant

$$P_i \rightarrow P_j \rightarrow \dots \rightarrow P_k \rightarrow P_i, i \neq j \neq k, S \subseteq P, \text{ and } |S| > 1.$$

A transaction manager is *active* if it is not waiting for a request to be granted, more specifically, if the transaction manager has not received a *Wait* message. A transaction manager is *idle* if it has received a *Wait* message, but not yet received a *Grant* message. Peterson presented an algorithm, similar to Rana's termination algorithm, for detecting deadlock under this same model [52]. Her protocol is based on the assumptions that channels are FIFO and system clocks are perfectly synchronized.

### 3.2.3 Previous Work

Like termination, detection of distributed deadlock has received a great deal of attention. In traditional multiprocessing systems with centralized control and shared memory, deadlock detection protocols maintain TWF graphs. Early distributed deadlock detection protocols were based on this paradigm [46, 26, 3, 15]. However, accurately maintaining the global system view contained in the TWF turns out to be costly and difficult. This complexity leads to protocol errors.

As an example, consider the protocol proposed by Mensace and Muntz [46]. Their system model is slightly different from ours in that transaction managers are assumed to

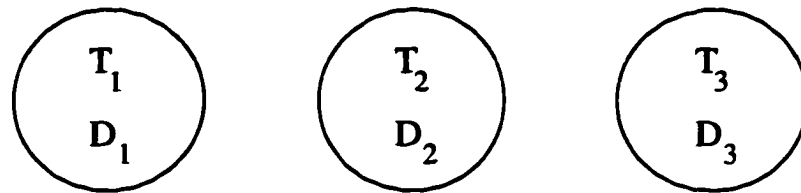


Figure 3.14: Mensace and Muntz Protocol Example - Initial Resource Allocation

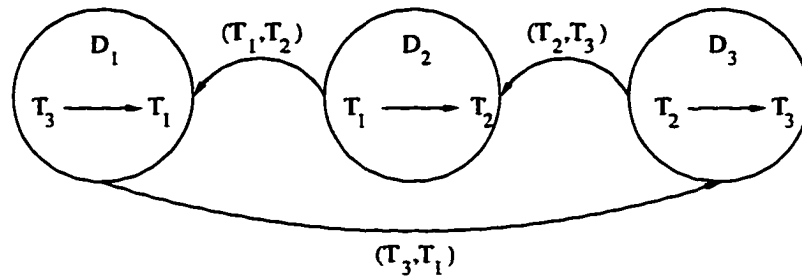


Figure 3.15: Mensace and Muntz Protocol Example - Requests, Local Blocking Pairs

reside at a data manager site. Transactions only send request messages when resources are needed at a site different from the one at which it resides. Each data manager maintains a condensed TWF graph. When a transaction manager  $TM_i$  requests a resource which cannot be granted, the data manager  $DM_q$  adds an arc  $(T_1, T_2)$  to its TWF, where  $T_2$  is the transaction which currently holds the resource requested by  $T_1$ . If  $T_1$  resides at a site different from  $D_q$ , the *blocking pair*  $(T_1, T_2)$  is sent to the site where  $T_1$  resides.

Upon receipt of a blocking pair, the data manager adds the arc to its local TWF and examines the graph. If a cycle exists, then a deadlock has been detected. If the second transaction,  $T_2$  in our example, is blocked by a transaction  $T_3$ , then a new blocking pair is generated, and sent to the site at which  $T_3$  resides.

Gligor and Shattuck showed that this protocol is incorrect [20]. They give the following counterexample. Suppose that initially  $T_1$  holds the resource managed by  $D_1$ ,  $T_2$  holds the resource managed by  $D_2$ , and that  $T_3$  holds the resource managed by  $D_3$ . Further assume that each of these transaction, data manager pairs reside at the same sight. This



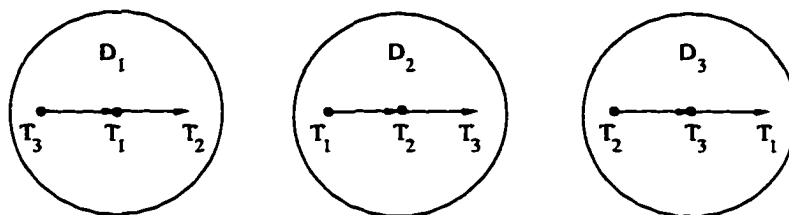


Figure 3.16: Mensace and Muntz Protocol Example - Final TWF Graphs

is depicted in figure 3.14. Now suppose that  $T_3$  requests the resource managed by  $D_1$ ,  $T_1$  requests the resource managed by  $D_2$ , and that  $T_2$  requests the resource managed by  $D_3$ . Further suppose that these requests are received simultaneously. Then each data manager will add an appropriate edge to its local TWF graph. Each data manager will then send the blocking pair to the appropriate remote site. This is depicted in figure 3.15. Suppose that each of these blocking pairs are received simultaneously. Then each data manager will add the appropriate edge to its local TWF graph and the resulting graphs will look like those in figure 3.16. The protocol requires that a new blocking pair be generated if the second transaction in the received pair is blocked. Thus, the resulting TWF graphs will not generate any new blocking pairs, and the deadlock will go undetected. If the blocking pair  $(T_1, T_2)$  had been received by  $D_1$  before the request from  $T_3$ , then the blocking pair  $(T_3, T_2)$  would have been generated and sent to  $D_2$  and  $D_3$ , and the deadlock would have been detected.

This exemplifies the complexity in constructing a current global view of a changing system using message passing. Gligor and Shattuck also showed that the protocol given by Isloor and Marsland [26] is incorrect. Further, Badal acknowledges that his protocol detects false deadlocks, but claims that this is not a significant problem [3].

Token, or probe, protocols abandon the attempt to maintain a global view of the system

state. By these protocols, a token traverses the edges of the TWF graph; return of the token to its initiator indicates deadlock. A simple protocol using this approach was proposed by Chandy, Misra, and Haas. Processes send two kinds of messages  $query(i, j, k)$  and  $reply(i, j, k)$ , which indicate they are part of a detection wave initiated by process  $P_i$  and are being sent from  $P_j$  to  $P_k$ . A blocked process  $P_i$  initiates detection of deadlock by sending queries to the data managers of all resources  $P_i$  is waiting to be granted ( $P_k$ 's *dependent set*).

An executing process  $P_k$  ignores all queries and replies. If  $P_k$  is blocked when it receives a query, it propagates the query to all processes in its dependent set. If this is the first query  $P_k$  has received since it was last executing, then  $P_k$  notes the number of queries it has sent in local variable  $num(i)$  and sets variable  $wait(i)$  to true, indicating that  $P_k$  has been blocked since it received the *engaging* query.  $P_k$  replies immediate to queries received while  $wait(i)$  is true.

$P_k$  replies to its engaging query only upon receiving replies to all its queries. When the process which initiated this diffusing computation receives replies to all its queries, then a deadlock has been detected.

By this protocol, processes may initiate several probes per blocked request. Further, every process in the cycle may detect the deadlock. This makes resolution difficult. Chandy, Misra, and Haas did not address deadlock resolution.

Sinha and Natarajan presented a protocol which addressed these deficiencies [59]. Their protocol ensures that at most one process will detect the deadlock. They further specify that only one token per blocked request will be generated. They increase the protocol's efficiency by transmitting state information, which can be saved and used in later deadlock

detection activity, within the token.

Choudhary et al. showed that their protocol leads to detection of false deadlocks [17]. The problems are caused by out-dated information. Choudhary suggests a modification to the protocol presented by Sinha and Natarajan, but makes no attempt to prove his protocol correct.

We believe that the approach used in termination detection will simplify development of protocols for distributed deadlock detection. The use of a timestamped token should eliminate problems with out-dated state information and allow simple and efficient deadlock resolution.

### 3.2.4 Algorithm

The algorithm for transaction manager  $M_j$  is given in figure 3.17. The algorithm for  $DM_q$  is given in figure 3.18. Once again, each is specified as a set of rules for the way that  $M_j$  and  $DM_q$  may respond to events that occur when they are in a given state.

Transaction managers can be in one of four states, *active*, *transition*, *idle*, or *done*. When in the *active* state, the transaction manager is either waiting to receive a resource request from its transaction process or it has posted a request to the appropriate data manager and is waiting for a reply. Upon receiving a *Wait* message, transaction manager  $M_j$  enters the *transition* state. In this state,  $M_j$  is waiting for the SLP interval to elapse prior to initiating a token. If  $M_j$  waits for an interval of  $\epsilon$ , as read by the local clock of  $M_j$ , then it will send a token to the data manager  $DM_q$  of the resource it is waiting for and enter the *idle* state. Prior to entering the *idle* state,  $M_j$  will propagate all tokens with equal or greater timestamps received during the transition period.  $M_j$  has then established the SLP

$\Gamma_j$	set of all tokens retained during the idle period, initially empty
$\Upsilon_j$	set of resources currently held by $P_j$ , initially empty
$TP_j$	$P_j$ 's clock value at the instant it receives a hold message
$TS_j$	$P_j$ 's timestamp, initially 0
$rq$	index of resource for which $P_j$ has an outstanding request
$state_j$	indicates $P_j$ 's state, initially active
$Token(TS_m, m, q, i, \mu)$	Token initiated by $P_m$ with timestamp $TS_m$ ; on an incoming token $q$ is the resource needed by $P_j$ 's predecessor in the control cycle routing and $i$ is the index of $P_j$ 's predecessor; on an outgoing token, $q$ is the resource $P_j$ needs and $i$ is $P_j$ 's process index; $\mu$ is the lowest index of any process which the token has visited
$alarm_j(T)$	$P_j$ 's alarm, signals $P_j$ at $C_j(t) = T$ ; $alarm_j(0)$ resets the alarm
$clock_j$	$P_j$ 's current clock reading

$state_j$	Event	Action
active	<b>Free</b> ( $q$ )	$\Upsilon_j \leftarrow \Upsilon_j - q$ ; send <i>Release</i> ( $j, q$ ) to $DM_q$ ; $state_j \leftarrow done$
	receive <i>Grant</i> ( $q$ ) from $DM_q$	$\Upsilon_j \leftarrow \Upsilon_j \cup q$ ; <b>Obtained</b> ( $q$ )
	<b>Required</b> ( $\{q\}$ )	send <i>Request</i> ( $j, q$ ) to $DM_q$ ; $rq \leftarrow q$
	receive <i>Wait</i> ( $q$ ) from $DM_q$	if $q \notin \Upsilon_j$ then $TP_j \leftarrow clock_j$ ; $alarm_j(TP_j + \epsilon)$ ; $\Gamma_j \leftarrow \emptyset$ $state_j \leftarrow transition$
	receive <i>Token</i> ( $TS_m, m, q, i, \mu$ )	discard <i>Token</i> ( $TS_m, m, q, i, \mu$ )
transition	receive <i>Grant</i> ( $q$ ) from $DM_q$	$alarm_j(0)$ ; $\Upsilon_j \leftarrow \Upsilon_j \cup q$ ; <b>Obtained</b> ( $q$ ); $state \leftarrow active$
	receive <i>Token</i> ( $TS_m, m, q, i, \mu$ )	if $(TS_m \geq TS_j) \wedge (k \in \Upsilon_j)$ then $\Gamma_j \leftarrow \Gamma_j \cup Token(TS_m, m, q, i, \mu)$ else discard <i>Token</i> ( $TS_m, m, q, i, \mu$ )
	$alarm_j$ expires	send <i>Token</i> ( $TS_j, m, rq, j, \min(j, \mu)$ ) to $DM_{rq}$ $\forall Token(TS_m, m, q, i, \mu) \in \Gamma$ do if $(TS_m = TS_j)$ then send <i>Token</i> ( $TS_m, m, rq, j, \min(j, \mu)$ ) to $DM_{rq}$ else send <i>Token</i> ( $TS_m, m, rq, j, \mu$ ) to $DM_{rq}$ $state_j \leftarrow idle$
	receive <i>Wait</i> ( $q$ ) from $DM_q$	discard <i>Wait</i> ( $q$ )
idle	receive <i>Grant</i> ( $q$ ) from $DM_q$	$\Upsilon_j \leftarrow \Upsilon_j \cup q$ ; <b>Obtained</b> ( $q$ ); $state_j \leftarrow active$
	receive <i>Token</i> ( $TS_m, m, q, i, \mu$ )	if $(TS_m < TS_j) \vee (q \notin \Upsilon_j)$ then discard <i>Token</i> ( $TS_m, m, q, i, \mu$ ) if $(TS_m = TS_j) \wedge (m = j)$ if $(\mu = j)$ then <b>Abort</b> else discard <i>Token</i> ( $TS_m, m, q, i, \mu$ ) if $(TS_m \geq TS_j) \wedge (q \in \Upsilon_j)$ then if $(TS_m = TS_j)$ then send <i>Token</i> ( $TS_m, m, rq, j, \min(j, \mu)$ ) to $DM_{rq}$ else send <i>Token</i> ( $TS_m, m, rq, j, \mu$ ) to every $DM_{rq}$
	receive <i>Wait</i> ( $q$ ) from $DM_q$	discard <i>Wait</i> ( $q$ )
done	<b>Free</b> ( $q$ )	send <i>Release</i> ( $j, q$ ) to $DM_q$
	receive <i>Token</i> ( $TS_m, m, q, i, \mu$ )	discard <i>Token</i> ( $TS_m, m, q, i, \mu$ )

Figure 3.17: Protocol (*DD-Oneres*): Transaction Manager State and Protocol

*holder* index of process which has been granted the resource  
*state<sub>q</sub>* Current state of  $DM_q$   
*ReqQ* Queue of unserved resource requests  
 The operator  $front(Q)$  applied to  $ReqQ$  returns the process number at the head of queue  $Q$ ;  $enqueue(Q, i)$  places  $i$  at the end of queue  $Q$ ;  $empty(Q)$  returns  $true$  when queue  $Q$  is empty,  $false$  otherwise.

$state_j$	Event	Action
available	receive $Token(TS_m, m, q, j, \mu)$	discard $Token(TS_m, m, q, \mu)$
	receive $Request(i, q)$	$holder \leftarrow i$ send $Grant(q)$ to $M_i$ $state_q \leftarrow held$
held	receive $Token(TS_m, m, q, j, \mu)$	if ( $j \neq holder$ ) then send $Token(TS_m, m, q, j, \mu)$ to $holder$
	receive $Request(i, q)$	$enqueue(ReqQ, i)$ send $Wait(q)$ to $M_i$
	receive $Release(i, q)$	if $empty(ReqQ)$ then $state_q \leftarrow available$ else $holder \leftarrow front(ReqQ)$ send $Grant(q)$ to $M_{holder}$ $state_q \leftarrow held$

Figure 3.18: Protocol (*DD-Oneres-Dm*): Data Manager Process State and Protocol

$S(TS_j, j, A_j)$ , where  $A_j$  is the property " $P_i$  holds all the resources in  $\Upsilon_j$  and is waiting for for resource  $r_q$ "

An incoming token contains the initiator's timestamp, the initiator's process index, the index of the resource required by  $P_j$ 's predecessor in the token's path, the index of  $P_j$ 's predecessor, and the minimum index of all processes which have seen the token. The token's minimum index field is used to ensure that a single process aborts the transaction. When a transaction manager process  $M_j$  receives a token, it is propagated if the timestamp on the token is greater than or equal to its own and if  $P_j$  holds the resource designated in the token.  $M_j$  puts its own index and the index of the resource it needs in the appropriate token fields prior to propagating it. Additionally, prior to propagating a token with a timestamp equal to its own,  $M_j$  checks the minimum index field. If the minimum index is greater than

its own index, it overwrites the field with its own index prior to propagating the token. Since transaction managers propagate all tokens with greater or equal timestamps, even in the *transition* state, all transaction managers with the same, highest valued, timestamp that control processes in a deadlock may receive their own token. However, only the single transaction process with the minimum index will be aborted.

When a transaction manager is in the *idle* state, it is waiting for a *Grant* message or receipt of its own token, with its index in the minimum index field. Once a resource has been released,  $M_j$  enters the *done* state. Once in the *done* state, the transaction managed by  $M_j$  can no longer request resources, and thus the transaction cannot be part of a deadlock cycle. Once a transaction has released all its resources, it is free to terminate. Its presence is not required for correct functioning of the protocol. A transaction aborts by releasing all its resources.

Data manager  $DM_q$  can be in one of two states, *available* or *held*. When in state *available*, the resource  $q$  which  $DM_q$  controls is available. Upon granting the resource to a transaction,  $DM_q$  enters state *held*. When  $DM_q$  receives a token in the available state, the token is discarded, since there can be no dependency based on the resource it manages. When  $DM_q$  receives a token in the *held* state, it forwards the token to the manager of the transaction which currently holds the resource. Using the token timestamp value, this process will determine if there was a dependency at the instant the token was initiated.

**Correctness** In order to establish the correctness of our protocol, we must show that if transaction process  $P_j$  is aborted, then there is a deadlock (safety) and that if a deadlock exists, then some transaction process  $P_j$  will be aborted (liveness).

The following lemmas are useful in establishing the correctness of our protocol. The

first lemma shows that the set of resources held by a transaction process remains constant while its transaction manager is in the *transition* and *idle* states.

**Lemma 12** *The set of resources  $\Upsilon_i$  held by transaction  $P_i$  remains constant while transaction manager  $M_i$  is in the transition and idle states.*

PROOF: This is a direct result of the protocol. The set of resources  $\Upsilon_i$  held by  $P_i$  is only modified upon receipt of a *Grant*( $q$ ) message by  $M_i$ . Whenever  $M_i$  receives a *Grant*( $q$ ) message while it is in the *transition* or *idle* states,  $M_i$  transitions back to the *active* state. Thus, the set of resources held remains constant while  $M_i$  is in the *transition* and *idle* states. ■

The next lemma shows that when a transaction manager  $M_j$  receives and propagates a token received from transaction manager  $M_i$ , the propagation establishes that a dependency existed between the transaction processes  $P_i$  and  $P_j$  at the instant the token initiator  $M_m$  released the token.

**Lemma 13** *Let  $Depends(P_i, P_j, t)$  denote that a dependency exists between transaction processes  $P_i$  and  $P_j$  at real time instant  $t$ , i.e.,  $P_i$  is waiting for a resource currently held by  $P_j$  at real time instant  $t$ . If  $M_j$  receives  $Token(TS_m, m, q, i, \mu)$  and  $M_j$  and subsequently propagates it, then  $Depends(P_i, P_j, tr_m)$  where  $tr_m$  is the earliest instant for which  $C_m(tr_m) = TS_m$ .*

PROOF: Let  $TS_i$  be  $M_i$ 's timestamp when it sends  $Token(TS_m, m, q, i, \mu)$  to  $M_j$  (via  $DM_q$ , the data manager of the resource requested by  $M_i$ ). If  $M_i$  sends the token to  $M_j$ , then  $M_i$  is in the *transition* or *idle* states. In order for  $M_i$  to enter the *transition* or *idle* states,  $alarm_i(TS_i + \epsilon)$  must have expired. If  $alarm_i(TS_i + \epsilon)$  has expired, then  $M_i$  has received a

$Wait(q)$  message for a resource  $q$  which is not in  $\Upsilon_i$  at the time the  $Wait(q)$  message was received.

By the protocol,  $M_i$  adds resource  $q$  to  $\Upsilon_i$  whenever it receives a  $Grant(q)$  message. Also by the protocol, whenever  $M_i$  receives a  $Free(q)$  communication from  $P_i$ ,  $M_i$  releases resource  $q$  and enters the *done* state, from which it cannot transition to any other state. Thus, when  $M_i$  sets  $alarm_i$ ,  $M_i$  has never received a  $Grant$  message for resource  $q$ . Since reception of a  $Grant(q)$  message causes  $M_i$  to modify  $\Upsilon_i$ , by lemma 12, when  $M_i$  sends  $Token(TS_m, m, q, i, \mu)$  from the *transition* or *idle* states, it is still waiting for resource  $q$ . Thus, if  $M_i$  sends  $Token(TS_m, m, q, i, \mu)$  it has established the SLP  $\mathcal{S}(TS_i, i, A_i)$ , where  $A_i$  is the property “ $M_i$  has not received a  $Grant(q)$  message for requested resource  $q$ ” and this property remains true until  $P_i$  sends the token to  $M_j$ .

$M_i$  only receives a  $Wait(q)$  message in response to a  $Request(i, q)$  message it has sent to  $DM_q$ .  $M_i$  only sends  $Request(i, q)$  when it receives  $Required(\{q\})$  from  $P_i$ . Further,  $P_i$  is blocked from the time it sends  $Required(\{q\})$ , until  $M_i$  receives a  $Grant(q)$  from  $DM_q$ . Thus, when  $M_i$  sends the token to  $M_j$ , it has also established the SLP  $\mathcal{S}(TS_i, i, A_i)$ , where  $A_i$  is the property “ $P_i$  is waiting for resource  $q$ .”

Let  $TS_j$  be  $M_j$ 's timestamp when it propagates  $Token(TS_m, m, q, i, \mu)$ .  $M_j$  only propagates the token from within the *idle* or *transition* states after  $alarm_j(TS_j + \epsilon)$  has expired. Then  $M_j$  only propagates the token if  $q \in \Upsilon_j$ . By lemma 12,  $\Upsilon_j$  has not changed since  $M_j$  set  $alarm_j$ . Thus, if  $M_j$  propagates the token, it has established the SLP  $\mathcal{S}(TS_j, j, A_j)$ , where  $A_j$  is the property “ $M_j$  has received a  $Grant(q)$  message.” Further, this property remains true until  $M_j$  propagates the token.

As  $P_j$  is blocked whenever  $M_j$  is in the *transition* or *idle* states,  $M_j$  has established the



equivalent SLP  $\mathcal{S}(TS_j, j, P_j \text{ holds resource } q)$ . Further, this property remains true until  $M_j$  propagates the token.

Then, since  $TS_i \leq TS_m$  and  $TS_j \leq TS_m$ , by lemma 3,  $P_i$  is waiting for resource  $q$  and  $P_j$  holds resource  $q$  at real time instant  $tr_m$ , where  $tr_m$  is the earliest instant for which  $C_m(tr_m) = TS_m$ . By definition, then,  $Depends(P_i, P_j, tr_m)$ . ■

Note that during the previous argument, we made little mention of data manager processes. This is because the dependence of  $P_i$  on the resource held by  $P_j$  can be determined only by the fact that  $M_i$  sent a token that  $M_j$  subsequently propagated. As we showed in the previous lemma, the set of resources held (or requested) by a process  $P_i$  in the *transition* or *idle* states is constant. Thus, if  $P_i$  sends a token indicating that it is waiting for resource  $q$  and  $P_j$  subsequently propagates this same token, indicating that it holds resource  $q$ , the data manager  $DM_q$  serves only to route the token from  $P_i$  to  $P_j$ , and the state of  $DM_q$  is irrelevant. The interval over which  $P_i$  was waiting for  $q$  and the interval over which  $P_j$  held  $q$  overlap. As only one transaction can hold a resource, if, at some real time instant  $t$ ,  $P_j$  holds resource  $q$ , then no messages releasing or granting resource  $q$  can be in transit at this same time instant.

The next lemma shows that no data manager will reflect a token back to the transaction manager it received the token from. Thus, in order for some transaction manager  $M_j$  to receive its own token, it must travel through a cycle of transaction managers.

**Lemma 14** *Suppose  $M_j$  sends token  $Token(TS_m, m, q, j, \mu)$  to  $DM_q$ . Then  $M_j$  will not receive  $Token(TS_m, m, q, j, \mu)$  back from  $DM_q$ .*

**PROOF:** This is a direct result of protocol *DD-Oneres-Dm*. By the protocol,  $M_j$  always adds its index to the appropriate field of the token prior to its initiation or propagation.

$DM_q$  then checks this field prior to propagating the token to the transaction manager which holds resource  $q$ . If  $M_j$  holds resource  $q$  then the token is discarded. ■

**Theorem 11 (Safety)** *If  $P_m$  aborts then a deadlock exists.*

**PROOF:** If transaction manager  $M_m$  aborts transaction  $P_m$ , then  $M_m$  has received its own token. If  $M_m$  receives its own token then, by the protocol, the token has traversed a set of edges  $S_m$ , where  $S_m = \{(M_m, M_i), (M_i, M_j), \dots, (M_k, M_m)\}$ . (Note that the token must pass through the appropriate data manager process in order to traverse an edge  $(M_m, M_i)$ .) By lemma 14, for each edge  $(M_i, M_j)$  it must be the case that  $i \neq j$ . Then  $|S| \geq 2$ . By lemma 13, if the token traverses edge  $(M_i, M_j)$  then  $P_i \rightarrow P_j$  at real time instant  $tr_m$ , where  $tr_m$  is the earliest real time instant for which  $C_m(tr_m) = TS_m$ . Thus, if transaction manager  $M_m$  receives its own token, there is cycle in the WFG at real time instant  $tr_m$  and the system is deadlocked. ■

**Theorem 12 (Liveness)** *If some set  $S$  of transactions is deadlocked then a transaction process  $P_m$  in  $S$  will be aborted by its transaction manager  $M_m$ .*

**PROOF:** Consider a set  $S$  of deadlocked processes. Each transaction manager  $M_i$  of a transaction  $P_i$  in  $S$  will receive a final  $Wait(q)$  message that will send  $M_i$  into the *transition* and *idle* states until the deadlock is broken. Let the set of transaction managers of processes in  $S$  be denoted  $S_m$ .

By the protocol, each  $M_i$  in  $S_m$  will generate a timestamp  $TS_j$ , and initiate a token, prior to entering the *transition* state. Now consider the highest valued timestamp generated by some  $M_i \in S_m$ . If more than one transaction manager initiates a token with this same

timestamp, then consider the transaction manager  $M_j$  with the lowest index.  $M_j$  will send  $Token(TS_j, j, q, j, j)$  to  $DM_q$ , where  $q$  is the resource  $M_j$  last requested.

When  $DM_q$  receives this token, it can be in either the *available* or *held* state. If  $DM_q$  is in the *available* state, then the resource is available and  $M_j \notin S_m$ . If  $DM_q$  is in the *held* state, then it will send the token to the transaction manager  $M_i$  to which  $DM_q$  last granted the resource.

When  $M_i$  receives the token, it can be in either the *active*, *transition*, *idle*, or *free* states. From our assumption,  $Token(TS_j, j, q, j, j)$  is the highest valued token initiated by some  $M_j$  in  $S_m$ . By lemma 7, a transaction manager receiving the token in the *active* state would subsequently generate a higher valued token. Thus, the token cannot be received by a transaction manager in the *active* state. If  $M_i$  is in the *free* state when it receives the token, then  $M_i$  is not in  $S_m$ . Thus,  $M_i$  must be in the *transition* or *idle* states when it receives then token. By our assumption,  $M_i$  will have a timestamp less than or equal to that on the token. Thus,  $M_i$  will propagate the token unless it is not waiting for resource  $q$ . If  $M_i$  is not waiting for resource  $q$ , then it has sent a release message to  $DM_q$  which has not yet been received. However, if  $M_i$  has released the resource, then  $M_i$  is not in  $S_m$ , a contradiction. Thus  $M_i$  will propagate the token to the data manager  $DM_q$  of the resource  $P_i$  is waiting for.

By an argument similar to the one given above, each data manager of a resource requested by some transaction in  $S$ , and the corresponding transaction managers, will propagate the token. Then,  $M_j$  will receive its own token and, since it has the lowest index of any transaction manager in  $S_m$ ,  $M_j$  will abort its transaction process  $P_j$ . ■

**Theorem 13** *At most one transaction process in any deadlock will abort its transaction.*

PROOF: Consider transaction manager  $M_m$ , in a set of deadlocked transactions, whose clock reads its timestamp value  $TS_m + \epsilon$  last (prior to some process breaking the deadlock.) Let  $tr_m$  be the earliest instant for which  $C_m(tr_m) = TS_m + \epsilon$ . By theorem 11 and the protocol, at this instant, no token could have circulated completely.

Suppose a process  $M_i$  were in the *active* state at real time instant  $tr_m$ . Clearly, by the protocol,  $M_i$ 's clock  $C_i$  would reach  $TS_i + \epsilon$  at an instant later than  $tr_m$ , contradicting our assumption. Then all transaction manager processes must be in the *transition* or *idle* states.

Consider any transaction manager  $M_k$  in the *idle* state. By clock axiom C1,  $C_k(tr_m) < TS_m + 2\epsilon$ . If  $M_k$  is in the *idle* state then its alarm has expired and, by the protocol, its timestamp must be less than  $TS_m$ . Then  $M_m$  will not propagate a token from  $M_k$  and no token from a transaction manager in the *idle* state at real time instant  $tr_m$  will detect the deadlock.

Now consider any transaction manager in the *transition* state. By the protocol, no transaction manager in the *transition* state could have initiated a token yet. Thus, any token generated by a transaction manager in the *transition* state must traverse all the transaction manager processes which control a process in  $\mathcal{S}$ . By the protocol, then, only the token with the highest valued timestamp  $TS_{max}$  of all these processes can circulate completely. By the protocol, if there is more than one token with timestamp  $TS_{max}$ , then only the token initiated by the transaction manager with the lowest index can circulate completely. Thus, at most one transaction manager will declare the deadlock and abort its transaction. ■

**Performance** A number of solutions have been proposed for the deadlock detection problem. In [14], Mitchell and Merritt proposed a solution for the single resource model, but their solution has no apparent extension for the more general AND model. In [7], Chandy, Misra, and Haas proposed a token-based solution. Their solution detects the deadlock in  $n\delta$  time, where  $n$  is the number of sites in the deadlock cycle and  $\delta$  is the inter-site communication delay. However, a transaction may generate several tokens per blocked request. Further, every process in a transaction may detect deadlock, which makes resolution difficult.

### 3.2.5 Discussion

This technique can be extended to the more complex distributed deadlock models. Within the AND model of distributed deadlock, processes may request multiple resources. A transaction is blocked until it is granted all the resources it has requested. For example, in the WFG shown in figure 3.13, transaction  $P_0$  is waiting for two resources held by transactions  $P_3$  and  $P_4$ . Since  $P_0$  must get all the resources it has requested, the system is deadlocked. Thus, as in the One-resource model, if a single cycle exists within the WFG then the system is deadlocked. Our protocol could be extended to detect deadlocks under this model by having processes send tokens to the data manager of *every* requested resource which is in use. The propagation scheme would be similar to the previous protocol with the exception that processes would send tokens to the data managers of *all* resources the process has been waiting for since its alarm expired. If a transaction manager receives one of the tokens it initiated, then there is a deadlock.

As in the AND model, transactions may make multiple resource requests under the OR model. Here, though, transactions need only one of the requested resources in order to

continue. Tokens would then be initiated, and propagated, to the data managers of all requested resources. A deadlock exists if some transaction manager receives all the tokens that it initiated.

Thus, as in the case of distributed termination detection, the approach is flexible. The token routing can be modified to ensure that complete circulation indicates the desired global predicate.

### 3.3 Global Snapshots

So far we have looked at special global predicates. We have considered only conjunctive predicates over the local process states. In this section, we discuss the problem of detecting more general predicates. This includes predicates which cannot (practically) be stated as a conjunction of predicates of the local process states, i.e. the predicate  $x_i < y_j$ , where  $x_i$  is a variable in the state of process  $P_i$  and  $y_j$  is a variable in the state of process  $P_j$ .

#### 3.3.1 Problem

Most predicate evaluation algorithms detect general stable predicates by repeatedly executing a *snapshot algorithm*. A snapshot algorithm coordinates the taking of each local process state so that the resulting global state is consistent.

If a predicate is stable then (1) if it is true in a consistent global state, it is still true after the algorithm has detected the predicate and (2) if the predicate is true at some instant, then a snapshot, either taken at that instant or some later instant, will detect the predicate's truth.

It is important to clarify this definition by describing predicates which are not stable,

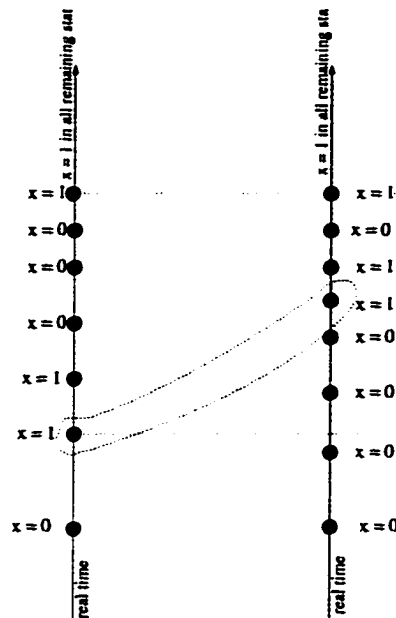


Figure 3.19: Execution with Time Driven Global State Predicate Changes

and therefore cannot be detected with a distributed snapshot algorithm. The fact that the predicate must remain true once it is true in *any* consistent state eliminates many predicates whose truth oscillations are not (always) communication driven. For example, consider the simple execution of a two process system shown in figure 3.19. Here we show the values of local variable  $X$  in the various process states as they occur in real time within each process. The value of the global predicate oscillates as a function of time and, once globally true at some real time instant, remains true indefinitely. Thus it is *instantaneously stable*. However, it is true in several consistent states prior to becoming globally true in an instantaneous global state. One might at first consider such predicates to be stable, but a global snapshot algorithm could not be applied in this case. The snapshot could falsely indicate the predicate's global truth at some real time instant..

### 3.3.2 Previous Work

Chandy and Lamport introduced the first distributed snapshot algorithm [5]. Their algorithm was designed for systems with FIFO communication and required  $O(N^2)$  message passing. Later efforts focused on removing the FIFO communication restriction and increasing the efficiency of that first algorithm [60, 62, 42].

Lai and Yang [62] introduced an algorithm which removes the FIFO restriction and reduced the message complexity of Chandy and Lamport's algorithm. According to their protocol, each process takes a snapshot at its convenience, but adheres to the following rules:

- Every process is initially white and turns red when it takes a snapshot.
- Every messages sent by a red(white) process is colored red(white).
- A white process must take a snapshot before it receives a red message. (Thus, the arrival of a red message will cause a white process to take a snapshot.)

If the channel states are required for predicate evaluation, then processes keep a complete message history and forward that along with the local state information. The channel states are then determined from the set difference.

Mattern proposed modifications to Lai and Yang's algorithm to eliminate the need for keeping and transmitting complete message histories[42]. He assumed that a single process initiates the snapshot and collects the local process states. A red process receiving a white message forwards the message to the collector process. Each process keeps a deficiency counter of the number of messages sent minus the number received. These counters are forwarded as part of the local process state to the collector after the local snapshot has



been taken. These counters are used by the collector process to determine the number of white application messages which should be received in order to complete the snapshot.

### 3.3.3 Discussion

In each of our earlier algorithms, we used traversal of a timestamped token to detect the truth of the global predicate. In snapshot algorithms, every process takes its state and replies to the snapshot initiator. Thus, every "token" circulates completely. In our algorithm, some "snapshots" are aborted when a process determines locally that the global predicate will not be true. Thus, our algorithm will either require less message passing or will detect the predicate more quickly. If every process initiates a snapshot each time it becomes passive, our algorithm will require less message passing. If a single process initiates successive snapshots, then our algorithm will, in the worst case, detect the predicate more quickly. Further, our algorithm detects a broader class of predicates in that it can detect processes which are instantaneously stable.

However, a snapshot algorithm is actually collecting a picture of some consistent global state. Thus, it could evaluate predicates such as " $x_i < y_j$ ", where  $x_i$  is the value of variable  $x$  in process  $P_i$  and  $y_j$  is the value of variable  $y$  in process  $P_j$ . Our algorithm could not detect such a predicate. Further, it is not clear that a time-based algorithm would provide any advantage in detecting such predicates. Clearly, any algorithm will have to collect the state of every process over whose state the predicate is made. Thus, a true snapshot could not be "aborted". The use of a global time base to ensure the collected states are consistent provides no clear advantage over either Mattern's or Lai and Yang's algorithm.

There is one exception, however. One of the benefits of a global time base is that

processor actions can be coordinated without message passing. If a time is chosen in advance and known to all processes, then the chosen clock value can be used to initiate taking of the local states by each process so that the accumulated global state will be consistent. These “scheduled snapshots” are also useful for evaluation of unstable predicates. We then defer the presentation of algorithms for scheduled snapshots to the next chapter, which covers evaluation of unstable predicates.

### 3.4 Conclusions

Each of the protocols presented in this chapter perform well in comparison to those which perform similar functions within a similar system architecture and which are based on the use of consistent global states rather than a global time base. Further, unlike their consistent state counterparts, they can detect a broader class of predicates, those which are instantaneously stable.

It is not immediately clear that taking of a global “snapshot” could be done more efficiently by using a global time base, with the exception of scheduled snapshots. However, no consistent snapshot algorithm could be designed to detect an instantaneously stable predicate whereas we could use the techniques of this chapter to design a spontaneously initiated snapshot algorithm for detecting such predicates. We discuss this in more detail in the concluding chapter.

The single technique of using a timestamped token is highly configurable for the underlying system architecture and performance goals. The use of SLPs and GLPs facilitates this approach in realistic systems in which the process clocks are only roughly synchronized. Their use simplifies both protocol development and verification.

## Chapter 4

# Unstable Predicates

In the previous chapter we examined the use of roughly synchronized clocks in the evaluation of stable predicates. We now turn our attention to the evaluation of unstable predicates. Unlike stable predicates, no restriction is placed on how the truth of an unstable predicate may vary. The predicate's truth may vary arbitrarily.

Algorithms for unstable predicate detection which assume the existence of a global time base can provide a significant advantage over algorithms which evaluate these predicates over consistent global states. When predicates are evaluated over a consistent global state, it is impossible to develop an algorithm which determines whether or not an arbitrary unstable predicate was true at some instant in real time during the computation. No such restriction exists in systems with a perfect global time base. For example, consider a system in which clocks are perfectly synchronized. In such a system, the global state of the computation at any point in time can, with significant overhead, be established using the system clocks, assuming that the clocks have sufficient resolution. It is then possible to determine whether or not *any* unstable predicate was true at some point during the computation. If the global

time base is not perfect, it is not possible to detect the truth of an arbitrary unstable predicate. However, it is possible to detect the occurrence of certain unstable predicates which are not detectable over consistent global states. In this chapter, we will demonstrate this and other benefits of assuming a rough global time base in unstable predicate detection.

First, we will describe the problem of unstable predicate detection in detail. Then, in section 4.2, we will review unstable predicate evaluation algorithms from the literature. In section 4.3, we present algorithms for scheduled evaluation of unstable (and stable) predicates. In section 4.4, we present a centralized algorithm for detection of unstable predicates whenever they occur during the computation. The algorithms presented evaluate only those predicates which remain globally true for a real time interval of  $2\epsilon(1 + \rho_M)$ , twice the maximum clock skew as read by any process clock. A number of physical systems have periods greater than the length of this interval for typical values of drift rate  $\rho_M$  and clock skew  $\epsilon$ . The ability to evaluate predicates over the system state would be valuable to these systems' distributed controllers and monitors. The presented algorithms arise naturally from the use of SLPs and GLPs. We conclude with a summary, in section 4.5.

## 4.1 Problem

Unlike a stable predicate which must remain true once it becomes true in some consistent global state, an unstable predicate's truth may vary arbitrarily. Detection of this type of predicate in a distributed system is very difficult. These difficulties are linked to the fundamental characteristics of distributed systems: lack of a common clock and common memory.

The lack of common memory requires that all synchronization be done by passing in-

$P_i$	$P_j$
$x=0;$	$x=0;$
$x=1;$	$x=1;$
$x=0;$	$x=0;$

(A)

$P_i$	Time Vectors	$P_j$	Time Vectors
$x=0;$	(1,0)	$x=0;$	(0,1)
$x=1;$	(3,3)	$x=1;$	(0,2)
$x=0;$	(5,5)	$x=0;$	(0,4)

(B)

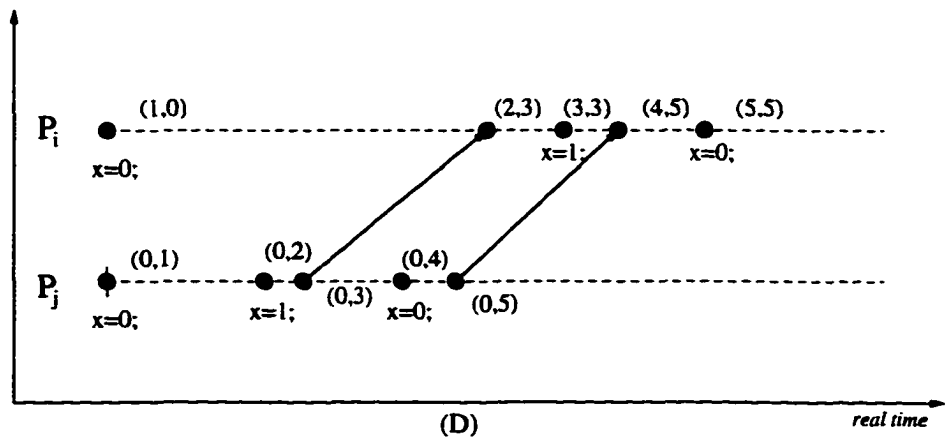
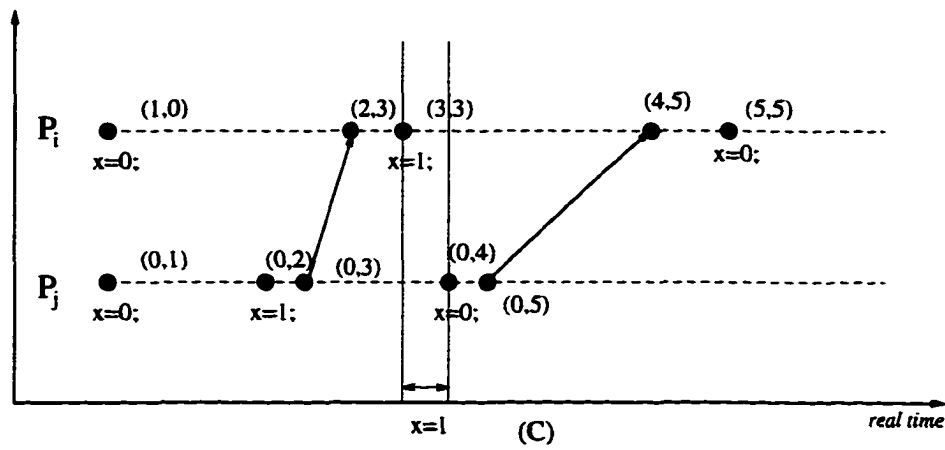


Figure 4.1: Varying Timelines for Two Process Execution History

formation between processes. There is no natural single global view of the system; it must be constructed via message passing. Thus, all local state changes which might affect the truth of the global predicate must be collected in order to detect the global predicate. For example, consider a system in which system clocks are perfectly synchronized, an unrealistic assumption. Let some global state predicate  $A$  be comprised of the conjunction of predicates over the local process state of each process; i.e.  $A = A_0 \wedge A_1 \wedge \dots \wedge A_{N-1}$ , where  $A_i$  is a local predicate over the state of process  $P_i$ . Now each process  $P_i$  can monitor the truth of the local predicate  $A_i$  and timestamp the endpoints of each interval over which the predicate is true. Thus, we have all the information required to detect occurrence of the predicate. However, these intervals must be accumulated and compared to determine whether or not the predicate was true at some time instant. Clearly, an algorithm to accomplish this will have significant computational complexity and, if the intervals are accumulated as they occur, significant message complexity. This approach would be well-suited to post-mortem analysis.

The lack of a common clock further complicates the task. If there is no common time base, rough or perfect, either the computation must be frozen, so that the state of each process can be determined, or the predicate must be evaluated over consistent global states. Currently, there are no other mechanisms for determining whether or not states in separate local processes (could have) occurred simultaneously.

If the computation is frozen, then the application may be adversely affected. If the computation cannot be frozen to allow synchronization, and the predicate can only be evaluated using consistent global states, then two difficulties arise. First, if the predicate is evaluated over consistent global states, then some subset of all possible consistent states must be con-

structured, either during or after the computation. This will be expensive computationally. If the process states are accumulated as they occur, the evaluation will have significant message complexity as well. The second problem is that development of an algorithm which detects whether an arbitrary unstable predicate was ever true during the computation is impossible. Consider the simple code fragments from a two process system given in part A of figure 4.1. Now suppose that a particular execution of these code fragments results in the vector times shown in part B of the figure and that we want to determine whether or not the variable  $x$  had the value one in every process at some point during the computation. Part C of the figure shows one possible time history. Here the execution of the events corresponding to the code fragments of part A of the figure are shown in an absolute real time frame. In this time history, it is clear that the predicate holds at an instant of real time. However, if there is no global time base and processes are not frozen to allow synchronization, there is no way to determine absolutely whether or not this reflects the actual execution; we know only that it could have. If we try to reconstruct the time history using only the information from part B of the figure, we can construct time histories in which the predicate holds, and ones in which it does not hold. A time history in which the predicate does not hold is shown in part D of the figure.

In physical systems, change in the truth of a predicate is often driven by external events, not by message exchange. Further, monitoring and control of a system may not require significant interprocess communication. For example, each monitor process may be responsible for a separate component of the system. In the absence of message passing, any event in one process is concurrent with all events in any other process. This is the most complex case for predicate evaluation over consistent states and will preclude determination of whether or

not certain unstable predicates occurred during the computation. However, the ability to detect whether or not an unstable predicate was true at some point during the computation, not whether it *could* have occurred, would clearly be useful to distributed systems which control and monitor these physical systems. Thus, development of an alternative to the algorithms which use consistent global states would be beneficial.

Although evaluation of unstable predicates using consistent global states will have a high computational complexity, it does provide a benefit which is not available when structuring predicate evaluation over instantaneous global states. That is, it is possible to determine whether the predicate *could* have occurred *during a given run*. It is important to emphasize both points: first, that we can determine if the predicate could have occurred and second, that this determination is limited to the execution being monitored. If the predicate is detected in a consistent global state, even though it may not have occurred in the current execution, it can occur in a subsequent execution in which the time intervals between local process events are different. However, the use of consistent states only enumerates the possibilities for the current execution path. Other execution paths, which did not occur in the current computation, may be possible. The predicate could then be true in a subsequent execution, which took a different execution path, and this kind of evaluation might not detect it. Thus, it is somewhere between a static analysis, which evaluates all potential execution paths, and a perfect global time based execution analysis, which can determine whether arbitrary unstable predicates actually held during a given execution.



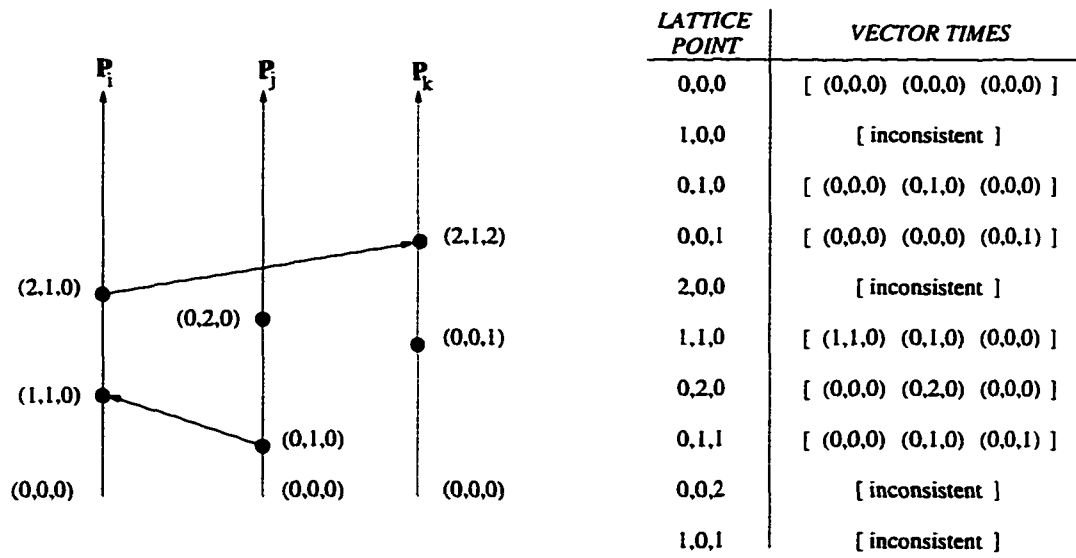
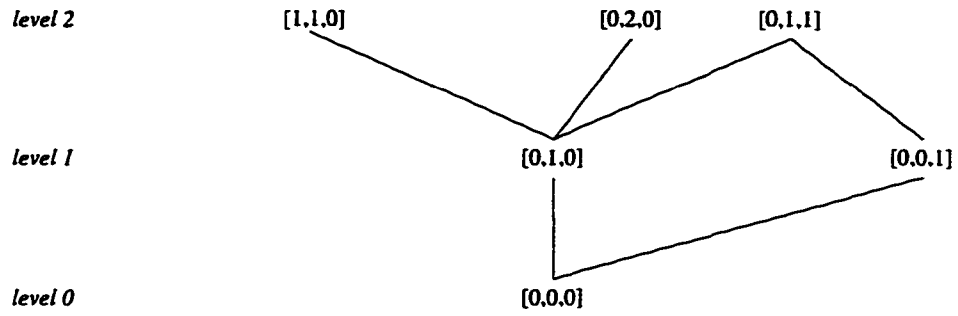


Figure 4.2: Consistent Global State Lattice

## 4.2 Previous Work

Most of the distributed systems literature on unstable predicates is devoted to algorithms which evaluate these predicates over consistent global states. Cooper and Marzullo introduced three modalities for evaluation of a predicate  $A$  within a given execution: *possibly*( $A$ ), *definitely*( $A$ ), and *currently*( $A$ ) [9]. They also gave algorithms for classification. Briefly, *possibly*( $A$ ) and *definitely*( $A$ ) are detected by first constructing all executions which are consistent with the observed execution. *Possibly*( $A$ ) then holds if  $A$  is true in some global state within any execution consistent with the observed execution. *Definitely*( $A$ ) holds if  $A$  held in some global state within every execution consistent with the observed execution. *Currently*( $A$ ) is detected by external control and monitoring of the program's execution by the detection algorithm. *Currently*( $A$ ) then holds if  $A$  held at any time instant within the *controlled* execution.

The set of all executions consistent with the observed execution is obtained through construction of a global state lattice. Each point in the lattice represents a single consistent global state, a state which could have occurred during the execution. The lattice is built by connecting a given consistent global state, the state of origin, with every other global state that is consistent and that is obtained from the state of origin by execution of a single event within a single process. Thus one, higher level, state is reachable from another, lower level, state by execution a single event system-wide. Each level  $n$  in the lattice then contains all consistent global states reachable by execution of a sequence of  $n$  events within the system.

Figure 4.2 shows a simple execution, the corresponding time vectors, and partial construction of a lattice for the execution. Lattice point  $[p, q, r]$  represents execution of  $p, q,$  and  $r$  events in processes  $P_i, P_j,$  and  $P_k$  respectively. Any path through the lattice represents

a possible execution. *Possibly(A)* holds if the predicate holds at any point in the lattice. *Definitely(A)* then holds if all paths through the lattice pass through a point in which *A* is true. For example, *Definitely(A)* holds if *A* is true within all states at a given level, because all execution paths must pass through some state within a given level. Note that detection of *definitely(A)* only implies that predicate *A* occurred. It does not imply that if predicate *A* occurs during a given execution then *definitely(A)* will hold. If a predicate were true at lattice point  $[0, 1, 1]$  then it occurred at an instant of real time according to the time history shown in figure 4.2. However, *definitely(A)* would not be asserted. *Possibly(A)* would be asserted, indicating that *A* may or may not have occurred. Here, we have assumed knowledge of the time history. Without a global time base, we would not have this knowledge.

Their method for detecting *currently(A)* requires blocking the monitored program. Process execution which might affect the truth of *A* is serialized by the monitor process. Each process must request a state change, which could affect the truth of *A*, from the monitor process. A process with an impending state change request is blocked until the request is granted. These requests are enqueued by the monitor process, which then allows their execution sequentially. When  $P_i$ 's state request change is granted,  $P_i$  changes its state and sends the modified state to the monitor process. The monitor process then checks the truth of the predicate, prior to allowing the next requested state change. If *A* is ever detected, then *currently(A)* holds.

Garg and Waldecker proposed an algorithm for run-time detection of *weak conjunctive* predicates [65]. These are conjunctive predicates over the local process states for which there exists a global state, consistent with the execution, in which the predicate holds. It is

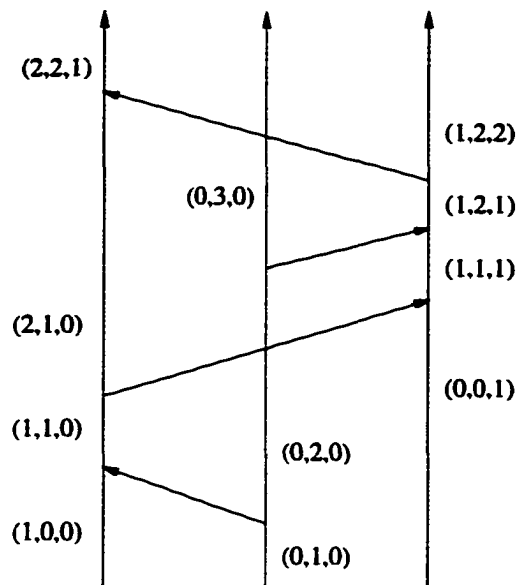


Figure 4.3: Examples of lmvectors

similar to Cooper and Marzullo's *possibly(A)*, except the predicate  $A$  must be a conjunctive predicate over the local process states. They propose a centralized scheme. Detection is based upon the use of *lmvectors*. An *lmvvector* is similar to a virtual time vector [43] except the vectors are updated only upon the sending and receiving of a message. They are not updated when the local state changes. For process  $P_j$ ,  $lmvvector[i]$ ,  $i \neq j$ , is the message id of the most recent message from  $P_i$  which has a causal relationship to  $P_j$ ;  $lmvvector[j]$  is the next message id that  $P_j$  will use. Figure 4.3 gives an example.

Each process  $P_i$  sends its *lmvvector* to the central checker process whenever the local predicate is true and  $P_i$  has sent a message since the last time it sent the *lmvvector* to the checker process. The channels between the checker process and each local process are assumed to be FIFO. The central checker process maintains a queue for each process' *lmvectors*. It then looks through the *lmvvector* queues to find a set of consistent values.

Lcmvectors  $u$  and  $v$  are from concurrent states if

$$\neg(u < v), \text{ where } u < v \text{ if and only if } (\forall i : u[i] \leq v[i]) \wedge (\exists j : u[j] < v[j]).$$

When a set of concurrent states is found, the predicate has been detected. They also describe how the algorithm can be decentralized by breaking down the predicate detection to hierarchical groups of processes.

Later they proposed a scheme for detection of *strong conjunctive predicates* [66]. A strong conjunctive predicate is true if and only if the system will always reach a global state in which a conjunctive predicate over the local process states is true. It is similar to Cooper and Marzullo's *definitely(A)*, except the evaluation is restricted to conjunctive predicates over the local process states. They again use lcmvectors. Here lcmvector intervals are sent each time the predicate transitions from true to false and a message has been received since the last time the lcmvector interval was sent. For any process  $P_i$ , the interval contains the lcmvector value  $lcmvector[i].lo$  when the local predicate transitioned from false to true and the lcmvector value  $lcmvector[i].hi$  when the local predicate transitioned from true to false. Once again, the channels between the checker process and the other system processes are assumed to be FIFO. The checker process maintains queues for the incoming data, in this case lcmvector intervals, from each process. It then searches these queues for overlapping lcmvector intervals, those in which  $lcmvector[i].lo \rightarrow lcmvector[j].hi$ , for every  $P_i, P_j$  in the system. If states  $s$  and  $t$  have vectors  $u$  and  $v$ , respectively, then  $s \rightarrow t$  if and only if  $u < v$ , as defined earlier.

In this paper, they also discuss detection of strong linked predicates. Informally, a

strong linked predicate is a conjunctive predicate over the local process states in which each component has a specified causal order in relation to the other components. It is of the form  $LP = A_0(s_{q0}) \wedge A_1(s_{q1}) \wedge \dots \wedge A_m(s_{qm})$ , where  $A_n(s_{qn})$  is a predicate over the local process state of  $P_q$ ,  $s_{qn}$  is a state observed within  $P_q$  during the execution and in which  $A_n$  holds, and there exists  $s_{q0}, s_{q1}, \dots, s_{qm}$  such that  $s_{q0} \rightarrow s_{q1} \rightarrow \dots \rightarrow s_{qm}$  in all executions consistent with the observed execution.

In order to detect the linked predicate, it is broken down into a sequence of local predicates which must be true in order for the linked predicate to be true. Each local predicate has an associated index, indicating its order of occurrence within the linked predicate. Each process keeps, in ascending order, a list of those components of the linked predicate, along with the associated index, which must be true locally in order for the linked predicate to be true. A local counter variable  $C_i$ , with initial value 1, reflects  $P_i$ 's most recent knowledge of which predicate in the list is being evaluated, as reflected by the predicate's index. Every process adds its value for  $C_i$  to every message it sends. When a process  $P_i$  receives a message, it sets its value for  $C_i$  to the maximum of its current local value and the received value. If the value of  $C_i$  matches the index value at the top of  $P_i$ 's local predicate list, and the local predicate becomes true, it increases its value for  $C_i$ , and deletes the head of its local list. If any process' value for  $C_i$  becomes  $m + 1$ , where  $m$  is the number of local predicates in the linked predicate, then the linked predicate is detected.

In the following sections, we present protocols for detecting unstable predicates which are based on the use of SLPs. The first approach is scheduled evaluation, similar to scheduling the taking of a global snapshot. This facilitates detection of predicates on attainment of some system state at an instant of real time. In section 4.4 we present a protocol for

runtime detection of certain unstable predicates. The protocol structure is similar to that of the protocols presented by Garg and Waldecker [65, 66] in that a central checker process monitors FIFO queues of local predicate values. Unlike the approaches outlined above, these protocols detect, with certainty, whether or not certain unstable predicates occurred during the computation.

### 4.3 Scheduled Evaluation

One of the advantages of a global time base is the ability to synchronize process actions without message passing. In this section, we present two algorithms which exploit this advantage for global state predicate evaluation. These algorithms schedule predicate evaluation at an agreed upon clock value and are based upon the use of SLPs.

Scheduling the evaluation provides another advantage. We can readily evaluate predicates which are based on attainment of a certain global state at a specified clock value. For example, we can evaluate predicates such as “Were all valves closed at two o’clock?”. In order to meaningfully evaluate such a predicate, the system clocks must be accurate, as well as precise. If we are evaluating attainment of a global system state at a specified real time instant, then we assume that some process clock reads absolute real time when it reaches the specified real time instant. For example, if we are evaluating a predicate  $A(t)$ , where  $A$  is a predicate whose attainment at real time instant  $t$  is being detected, then  $C_i(t) = t$  for some  $C_i$  in the system.

The evaluation is restricted to predicates whose truth remains constant for at least a real time interval of  $2\epsilon(1 + \rho_M)$ . We will show that this is the minimum interval over which a predicate must be globally true in order to ensure that all processes can assert SLPs with

*flicker* indicates whether  $A_i$ 's value has changed during the  $\epsilon$  interval, initially false  
*a* indicates value of  $A_i$  at the start of the SLP interval  
*state<sub>i</sub>*:  $p_i$ 's state, initially *computation*

<i>state<sub>j</sub></i>	Event	Action
<i>computation</i>	$C_i(t) = T$	$flicker \leftarrow \text{false}$ $a \leftarrow \text{false}$ if ( $A_i$ ) then $a \leftarrow \text{true}$ $state_i \leftarrow \text{transition}$
<i>transition</i>	$(a = \text{true}) \wedge \neg A_i$	$flicker \leftarrow \text{true}$
	$(a = \text{false}) \wedge A_i$	$flicker \leftarrow \text{true}$
	$C_i(t) \geq T + \epsilon$	if $(\neg flicker) \wedge (a = \text{true})$ assert $A_i$ if $(\neg flicker) \wedge (a = \text{false})$ assert $\neg A_i$ if <i>flicker</i> assert <i>unknown</i> $state_i \leftarrow \text{computation}$

Figure 4.4: Protocol (*Sched-Conj*): Scheduled Conjunctive Predicate Evaluation

equal timestamps.

We present two algorithms. The first considers conjunctive predicates over the process states. The second incorporates more general stable predicates, including predicates over the channel states.

### 4.3.1 Conjunctive Process State Predicates

If process clocks are perfectly synchronized, the state of all system processes at some real time instant can be obtained by having each process take its local state at some agreed-upon clock value. SLPs facilitate this approach in systems with rough clock synchronization. If each process can assert  $\mathcal{S}(T_i, i, A_i)$  with identical SLP timestamps then, by theorem 2, every local predicate  $A_i$ ,  $i$  in  $SYS$ , must have been true at the same real time instant. If the value of any local predicate changes during the  $\epsilon$  interval over which it is being monitored, then nothing definitive is asserted about the simultaneous truth of the local predicates.



In this section we present an algorithm for evaluation of a global state predicate  $A$  of the form  $A = (A_0 \wedge A_1 \wedge \dots \wedge A_{N-1})$ , where  $A_i$  is a predicate over the state of process  $P_i$ . By the algorithm, the local states are not collected prior to evaluation of the global state predicate. This excludes evaluation of a number of predicates over the channel states. It also excludes predicates in which process states must be compared in order to determine the truth of the global predicate. For example, the predicate " $x_i > y_j$ ", where  $x_i$  is a variable in the state of process  $P_i$  and  $y_j$  is a variable in the state of process  $P_j$ , could not be evaluated with this algorithm. We look at evaluation of more general predicates in the next section.

**Algorithm** The algorithm, which we presented in an earlier work [29], is shown in figure 4.4. Each process starts in state *computation*. When the process clock reaches  $T$ , the value of the local predicate is recorded and the *transition* state is entered. In the *transition* state, processes monitor the truth of the local predicate for an interval of  $\epsilon$ . If the predicate's truth changes during this interval, then the variable *flicker* is set to true. When the local process clock reaches  $T + \epsilon$ , processes either assert the local predicate's truth value or, if *flicker* is true, they assert *unknown*. If one or more processes assert *unknown*, nothing certain can be said about the value of the global predicate. No specific method is specified for accumulating the local process assertions. This may be done in a number of obvious ways depending on the network topology.

**Discussion** We now present several results which are useful in evaluating the performance of this protocol. The first two results bound the real time interval that can elapse between the instants that two process clocks read the same value  $T$ . In order to bound this interval, we must assume a tight bound on the clock skew  $\epsilon$ . Thus, we assume that  $|C_i(t) - C_j(t)| \leq \epsilon$ ,

rather than requiring that  $|C_i(t) - C_j(t)| < \epsilon$ .

The following lemma shows that a clock interval of  $\gamma$ , on any single process clock, corresponds to a real time interval of at most  $\gamma(1 + \rho_M)$ .

**Lemma 15** *Suppose that  $0 \leq C_i(tf) - C_i(ts) < \gamma$ , then  $tf - ts < \gamma(1 + \rho_M)$ .*

**PROOF:** By clock axiom C3,

$$(1 - \rho_M)(tf - ts) \leq \gamma \leq (1 + \rho_M)(tf - ts).$$

Then,

$$\frac{1 - \rho_M}{\gamma} \leq \frac{1}{tf - ts} \leq \frac{1 + \rho_M}{\gamma},$$

and by clock axiom C2,

$$\frac{\gamma}{1 - \rho_M} * \frac{1 + \rho_M}{1 + \rho_M} \geq tf - ts \geq \frac{\gamma}{1 + \rho_M} * \frac{1 - \rho_M}{1 - \rho_M}.$$

Finally, neglecting  $\rho_M^2$  terms, we get

$$\gamma(1 + \rho_M) \geq tf - ts \geq \gamma(1 - \rho_M).$$

■

The next theorem bounds the interval that can elapse between the real time instants when any two process clocks read the same value  $T$ .

**Theorem 14** *Let  $ts$  be the earliest real time instant at which any process clock reads  $T$ .*

*Let  $tf$  be the latest instant at which any process clock reads  $T$ . Then  $(tf - ts) < \epsilon(1 + \rho_M)$ .*

PROOF: Let  $C_i(ts) = T$ , and let  $C_j(tf) = T$ . By clock axiom C1,  $C_i(tf) < T + \epsilon$ . By clock axiom C2,  $0 \leq C_i(tf) - C_i(ts) < \epsilon$ . By lemma 15 then,  $tf - ts < \epsilon(1 + \rho_M)$ . ■

Suppose then that we want use the previous algorithm to detect a conjunctive global predicate at real time instant  $t$  by scheduling the evaluation a clock time  $T$  (so that  $T = t$ ). By theorem 2, from chapter 2, we know that at some instant when the intervals overlap, a process clock reads  $T$ . By theorem 14, we then know that if we detect the predicate, it occurred within a real time interval of at last  $\epsilon(1 + \rho_M)$  of real time instant  $t$ .

The following two lemmas bound the interval over which a global predicate must remain true to ensure that all process will be able to assert SLPs with equal timestamps. In order to bound this interval, we again require a tight bound on the clock skew  $\epsilon$ , and thus assume that  $|C_i(t) - C_j(t)| \leq \epsilon$ , rather than requiring that  $|C_i(t) - C_j(t)| < \epsilon$ .

The first theorem shows that if a conjunctive predicate over the local process states remains true for a real time interval of duration  $2\epsilon(1 + \rho_M)$  then all processes will be able to assert SLPs with equal timestamps.

**Theorem 15** *Suppose the conjunctive global predicate  $(A_0 \wedge A_1 \wedge \dots \wedge A_{N-1})$  is true for a real time interval of duration  $\mathcal{I}$ . Then all processes can assert  $S(TS, i, A_i)$  for some  $TS$  if  $\mathcal{I} \geq 2\epsilon(1 + \rho_M)$ .*

PROOF: Let  $ts$  be the first instant that  $(A_0 \wedge A_1 \wedge \dots \wedge A_{N-1})$  is true. Let  $TS$  be the maximum clock value at this time, and let  $C_i(ts) = TS$ . Then, by clock axiom C1,  $TS \geq C_j(ts) > TS - \epsilon$  for every  $j$  in  $SYS$ . Let  $tl$  be the latest instant that some process clock reads  $TS$ . By lemma 14,  $tl - ts < \epsilon(1 + \rho_M)$ . Thus, all process clocks will read  $TS$  within a real time interval of  $\epsilon(1 + \rho_M)$  of the first instant at which the predicate becomes

true.

Let  $tf$  be the latest instant that some process clock reads  $TS + \epsilon$ . By lemma 14,  $tf - tl < \epsilon(1 + \rho_M)$ . Thus, if the predicate is true for a real time interval of duration  $2\epsilon(1 + \rho_M)$  then all process clocks will read the values  $TS$  and  $TS + \epsilon$  during this interval and, by definition, can assert  $S(TS, i, A_i)$ . ■

The next theorem shows that if the conjunctive predicate  $A_i \wedge A_j$  is not true for at least  $2\epsilon(1 + \rho_M)$  then valid clock functions  $C_i$  and  $C_j$  exist for which  $P_i$  and  $P_j$  cannot assert SLPs with equal timestamps. In other words, if the conjunctive predicate is not true for an interval of duration  $2\epsilon(1 + \rho_M)$ , then we cannot be sure that both processes will be able to assert SLPs with equal timestamps.

**Theorem 16** *Suppose that the conjunctive predicate  $A = (A_i \wedge A_j)$  is true for a real time interval of duration  $\mathcal{I}$ , and that  $A$  is false outside this interval. Further suppose that  $C_i$  and  $C_j$  obey the clock axioms with the exception that  $|C_i(t) - C_j(t)| \leq \epsilon$  rather than  $|C_i(t) - C_j(t)| < \epsilon$ . Then if  $\mathcal{I} < 2\epsilon(1 + \rho_M)$ , there exists  $C_i$  and  $C_j$  for which there does not exist a clock value  $TS$  such that  $S(TS, i, A_i) \wedge S(TS, j, A_j)$  holds.*

**PROOF:** Let  $ts$  be the first instant that  $A$  is true. Let  $TS$  be the maximum value of  $C_i(ts)$  and  $C_j(ts)$ , and let  $C_i(ts) = TS$ . Let  $tm = ts + \epsilon(1 + \rho_M)$ . Then, we can use an argument similar to that given for lemma 15 to show that  $C_i(tm) = TS + \epsilon$  does not violate the clock axioms.

Then, by our assumption, we can let  $C_j(tm) = T$ . Further, by clock axiom C3,  $C_j$  can read  $TS + \epsilon$  as late as  $tm + \epsilon(1 + \rho_M)$ .

In order for both processes to assert SLPs with equal timestamps,  $A$  must then be true

over the interval  $[ts, tm + \epsilon(1 + \rho_M)]$ . However,

$$tm + \epsilon(1 + \rho_M) - ts = 2\epsilon(1 + \rho_M).$$

Then, since  $\mathcal{I} < 2\epsilon(1 + \rho_M)$ , both processes cannot assert SLPs with the same timestamp. ■

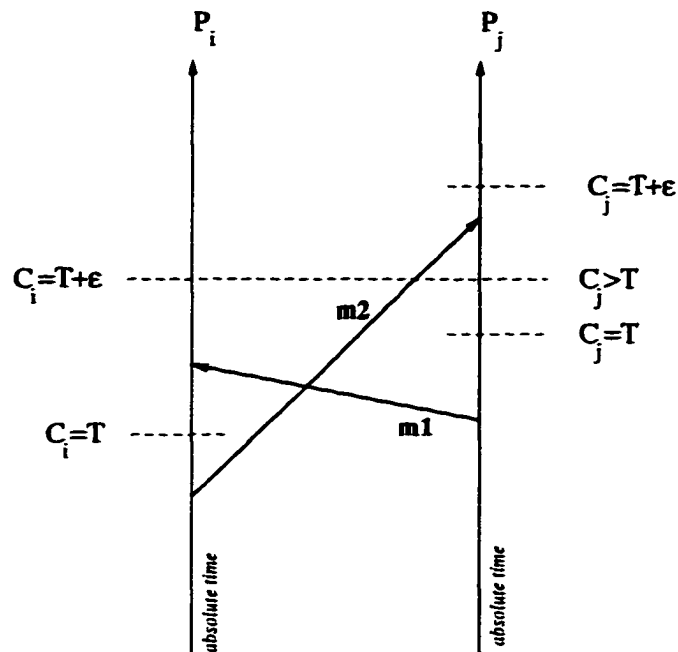
Thus, the previous protocol, and the protocols given throughout the remainder of this chapter, are limited to global predicates which remain true for at least  $2\epsilon(1 + \rho_M)$ .

Clearly, this protocol is efficient in terms of message complexity. Only a single message from each process is required to obtain the truth of the global predicate. The protocol might at first seem optimal in terms of message complexity, however, if message delay is bounded, we can develop a protocol that is more efficient. If message delay is bounded, we can modify the protocol so that processes only send a message if the local predicate is false at some point during the SLP interval. In this case, processes can wait the maximum message transmission delay  $\delta$  after the SLP interval. If no messages indicating that some local predicate  $A_i$  was false on the SLP interval, then we know that all the local predicates were true. In this case, the predicate is detected with *no* message passing.

By using SLPs, we have eliminated concerns with granularity and roughness in clock synchronization in construction of the algorithm. The approach is clear and, from theorem 2, its correctness is apparent.

### 4.3.2 General Predicates

We have already shown, by theorem 2 in chapter 2, that if two or more processes assert SLPs with equal timestamps, then the intervals will overlap at some real time instant.

Figure 4.5: Real Time  $\epsilon$  Interval Overlap

We also know that some process clock will read  $T$  at an instant when the intervals overlap. However, we don't know the order in which these intervals occurred in real time. This makes it impossible to attain an instantaneous channel state without restricting communication. For example, consider the execution given in figure 4.5. Both  $P_i$  and  $P_j$  could have equal clock values at the instant they send their respective messages. Similarly, they could have equal clock values when they receive their respective messages. However, message  $m1$  is not in the channels during any instant at which the intervals overlap; whereas, message  $m2$  is in the channels during every instant at which the intervals overlap. Thus, given a message containing the sender's clock value at the instant the message was sent, as well as the receiver's clock value at the instant the message was received, we have no way to determine whether or not the message was in the channels at the instant the intervals overlapped. In order to get an instantaneous channel state, we prohibit message activity during the SLP interval; that is, no messages are sent during the interval, and any received messages are

buffered for action after the SLP interval.

Here we are attempting to establish an instantaneous global state, not a consistent state. Global snapshots, which establish a consistent state, can also be scheduled. Neiger and Toueg outline such a protocol [49]. Their algorithm requires only that sent messages be timestamped with the sender's clock value at the time of the send and that received messages be buffered until the receiver's clock reaches the message timestamp value. Each process then records its state at an agreed upon clock value  $T$ . (Note that if the minimum message passing delay were known to be greater than  $\epsilon$ , then no message could be received before the receiver's clock read the timestamp value, and timestamps would not be required.) By establishing an instantaneous state, we can absolutely detect occurrence of certain unstable predicates, as well as detect predicates which are instantaneously stable.

As state changes can occur which do not necessarily negate the global predicate's truth, each process records all state changes. The collector process then ascertains whether or not the global predicate was true continuously over the SLP interval. We record the channel states, via message deficits, at the beginning of the SLP interval. Received messages are buffered for action after the SLP interval has elapsed.

**Algorithm** The algorithm is shown in figure 4.6. Here again, each process starts in state *computation*. Processes track the local message deficit while in this state. When the clock reaches the agreed upon value  $T$ , the local message deficit and process state are recorded. Then the *transition* state is entered, during which the the local state is monitored for an interval of  $\epsilon$ . Any change which might affect the truth of the global predicate is recorded. If a message is received, then it is buffered for processing after the SLP interval. When the  $\epsilon$  interval has elapsed, each process sends its local state, including all changes, and the

$state_j$   $P_j$ 's state, initially *computation*  
 $lmsgs_j$   $P_j$ 's message deficit, initially 0  
 $msg_j$  message sent by  $P_j$   
 $seq_j$  number of times  $P_j$ 's process state has changed during the  $\epsilon$  interval; initially 0  
 $\mathcal{V}_j(seq_j)$  value of  $P_j$ 's process state;  $seq_j$  corresponds to the number of times the local process state has changed during the  $\epsilon$  interval  
 $\Gamma_j$  messages received during the SLP interval

$state_j$	Event	Action
<i>computation</i>	$C_j(t) = T$	$\mathcal{V}_j(seq_j) \leftarrow P_j$ 's process state $state_j \leftarrow transition$
	send $msg_j$	$lmsgs_j^{++}$
	receive $msg_i$	$lmsgs_j^{--}$
<i>transition</i>	process state change	$seq_j^{++}$ $\mathcal{V}_j(seq_j) \leftarrow P_j$ 's process state
	receive $msg_i$	$\Gamma_j \leftarrow \Gamma_j \cup msg_i$
	$C_j(t) \geq T + \epsilon$	send $(\mathcal{V}_j, lmsgs_j)$ to collector

Figure 4.6: Protocol (*Sched-Gen*): Scheduled Predicate Evaluation

message deficit to the collector process. The message deficits here are solely recorded as part of the process' state. (We assume that a single process collects the local process states and channel states. However, the algorithm could easily be modified to be fully distributed and symmetric.) The collector process then looks at the collection of states  $state_j(k_j)$ ,  $j \in SYS$ , where  $k_j$  is the number of relevant changes in the state of  $P_j$  during the SLP interval. If there is some collection of states, one from each process, in which the predicate is not true, then nothing certain can be asserted about the global predicate's truth. However, if the predicate is true for all states from each process, then the global state predicate was true at some real time instant.

### 4.3.3 Discussion

Like the previous protocol, this one clearly requires less message passing than traditional snapshot algorithms. Further, it can evaluate instantaneously stable predicates. However,



$state_i$   $P_i$ 's state, initially *unsatisfied*  
 $timer_i(\gamma)$   $P_i$ 's timer, signals  $P_i$  at  $C_i(t) = \gamma$ ;  $timer_i(0)$  resets the timer

state	event	action
<i>unsatisfied</i>	$A_i$	$TS_i \leftarrow C_i(t)$ $timer_i(TS_i + \epsilon)$ $state_i \leftarrow transition$
<i>transition</i>	$\neg A_i$	$timer_i(0)$ $state_i \leftarrow unsatisfied$
	$timer_i$ expires	send $m(TS_i, 0, i)$ to $P_0$ $state_i \leftarrow satisfied$
<i>satisfied</i>	$\neg A_i$	$TF_i \leftarrow C_i(t) - \epsilon$ send $m(TS_i, TF_i, i)$ to $P_0$ $state_i \leftarrow unsatisfied$

Figure 4.7: Protocol (*Unstable-App*): Global Predicate Detection - Application Process

if there are a large number of relevant state changes during the  $\epsilon$  interval, then the amount of state information stored and transmitted, as well as the processing required to determine whether the predicate was true in every state, will be large. Further, message activity must be restricted during the SLP interval.

This algorithm could be used to detect general stable predicates by repeated application at scheduled intervals. The times at which the SLPs are evaluated could either be agreed upon in advance or could be distributed by the collector process if message delivery time is bounded and that bound is known.

#### 4.4 Centralized Evaluation

Although scheduled evaluation is useful for detecting certain time-based predicates, it is difficult to extend this approach so that we can detect unstable predicates which can become true at any point during a run. For example, in detecting stable predicates, it is sufficient to schedule evaluations periodically, either dynamically or statically. If a stable predicate becomes true, it remains true indefinitely. Thus, the scheduled intervals cannot skip over

the interval in which the predicate is true. This is not the case for unstable predicates.

In order to detect unstable predicates whose truth changes dynamically, a scheme like Garg and Waldecker's protocol for detecting weak unstable predicates is more practical. We give such an algorithm below. It is based on the use of SLPs. A version based on the use of GLPs is presented in appendix A, to contrast the use of GLPs versus SLPs.

The algorithm is centralized. One process,  $P_0$ , called the monitor process, is not part of the computation. Its purpose is to collect information from the application processes and evaluate the global state predicate. In addition to being delivered reliably, we assume that messages sent on the channel between any process  $P_i$  and  $P_0$  are delivered in FIFO order. Values received by  $P_0$  are assumed to be buffered in FIFO order until they are used in the detection algorithm. We further assume that message delivery time is bounded within some known interval  $\delta$ , as read by any  $C_i$ ,  $i$  in  $SYS$ .

Since we are considering only conjunctive predicates which are *globally* true for a real time interval of at least  $2\epsilon(1 + \rho_M)$ , each local predicate must also be true for the same duration. Suppose each process  $P_i$  then monitors the truth of its local predicate and asserts  $S(TS_i, i, A_i)$  whenever  $\mathcal{L}(T, i, A_i)$ ,  $TS_i \leq T \leq TS_i + \epsilon$  holds. Then, by theorem 15, the processes will assert a set of SLPs with equal timestamps. By lemma 2, if all processes assert SLPs with equal timestamps then there is an instant in time at which the predicate is globally true. Thus, to determine when the predicate is true, we only need find a set of equal SLP timestamps.

**Algorithm** Algorithms for the application and monitor processes are given in figures 4.7 and 4.8, respectively. Each application process sends a message to the monitor process whenever the value of the local predicate changes. These messages contain the minimum

<i>CTime</i>	value which is current candidate for $T_{detect}$ , initially 0
$\tau_i$	interval endpoints ( $TS_i, TF_i$ ) most recently read from $P_i$ 's buffer
BufferEmpty( <i>i</i> )	returns a boolean indicating whether or not the buffer associated with $p_i$ is empty
AgeTimestamp( <i>T</i> )	blocks until the local clock value is $T + \delta + \epsilon$
STime( $\tau_i$ )	$\tau_i$ 's interval start time, $TS_i$
ETime( $\tau_i$ )	$\tau_i$ 's interval end time, $TF_i$
GetValidInterval( <i>i</i> , <i>T</i> )	reads interval values from $P_i$ 's buffer and returns a potentially valid interval; returns the first interval in the queue for which $TF_i \geq T$ or $(TF_i = 0) \wedge (TS_i > \max(TF_i : TF_i < T, 0))$ ; the function blocks until such an interval is available; if the start time for this interval appears in both a closed and open interval then the closed interval values are used
Overlap( $\tau_i, T$ )	returns true if any value on the interval $\tau_i$ is equal to $T$ , false otherwise; if $\tau_i$ denotes an open interval, then only the value for STime( $\tau_i$ ) is used.

```

1   for  $i \leftarrow 1$  to  $N - 1$  do
2        $\tau_i \leftarrow$ GetValidInterval( $i, 0$ )
3   end for

4    $CTime \leftarrow \min\{S\text{Time}(\tau_i) : 1 \leq i \leq N - 1\}$ 
5    $noverlaps \leftarrow 0$ 
6    $found \leftarrow \text{false}$ 
7   while not( $found$ ) do
8       for  $i \leftarrow 1$  to  $N - 1$ 
9           if Overlap( $\tau_i, CTime$ )  $\vee ((E\text{Time}(\tau_i) = 0) \wedge (S\text{Time}(\tau_i) \leq CTime) \wedge (\text{BufferEmpty}(i)))$  then
10               $noverlaps++$ 
11          else
12              if  $((E\text{Time}(\tau_i) = 0) \wedge (\neg \text{BufferEmpty}(i) \wedge (S\text{Time}(\tau_i) \leq CTime))) \vee$ 
13                  $((E\text{Time}(\tau_i) \neq 0) \wedge (E\text{Time}(\tau_i) < CTime))$  then
14                  GetValidInterval( $i, CTime$ )
15                  if Overlap( $\tau_i, CTime$ )  $\vee ((E\text{Time}(\tau_i) = 0) \wedge (S\text{Time}(\tau_i) \leq CTime))$  then
16                      $noverlaps++$ 
17              endif
18          end for

19      if ( $noverlaps = N - 1$ ) then
20          ASSERT  $A$ 
21           $found \leftarrow \text{true}$ 
22      else
23           $noverlaps \leftarrow 0$ 
24           $CTime \leftarrow \min\{S\text{Time}(\tau_i) : S\text{Time}(\tau_i) > CTime, 1 \leq i \leq N - 1\}$ 
25          AgeTimestamp( $Ctime$ )
26      endif
27  endwhile

```

Figure 4.8: Protocol (*Unstable-Mon*): Global Predicate Detection - Monitor Process

and maximum SLP timestamps of every SLP interval over which the local predicate  $A_i$  holds. Let  $TS_i$  be the minimum clock value at which  $A_i$  can be asserted by  $P_i$ . Let  $TF_i$  be the maximum clock value from the same interval. Then  $\mathcal{L}(T_i, i, A_i)$  for  $TS_i \leq T \leq TF_i$  holds. When the value of the local predicate changes from false to true, and then remains true for  $\epsilon$ , as read by  $C_i$ , the application process will send a message to the monitor process containing the interval values  $(TS_i, 0)$ . The monitor then knows that  $A_i$  changed from false to true at  $P_i$  and that  $P_i$ 's initial SLP timestamp was  $TS_i$ . An interval ending timestamp of zero indicates that  $TS_i$  is the only SLP timestamp that can be asserted for the current interval. When  $A_i$  changes from true to false,  $P_i$  sends the interval values  $(TS_i, TF_i)$  to  $P_0$ , where  $TF_i$  is equal to the value of  $C_i(t)$ , the local clock value when the predicate's value changed, minus  $\epsilon$ ; i.e.,  $TF_i = C_i(t) - \epsilon$ . If  $TF_i$  is zero, then we refer to the interval as "open". Otherwise, the interval is said to be "closed".

The algorithm for  $P_0$  is given in figure 4.8. By the algorithm for the application processes, each  $P_i$  generates an interval  $\tau_i$  of SLP timestamps  $(TS_i, TF_i)$  for which  $\mathcal{L}(T, i, A)$ ,  $TS_i \leq T \leq TF_i + \epsilon$ , holds. By theorem 15, if the global predicate is true for a real time interval of at least  $2\epsilon(1 + \rho_M)$  then all processes will assert SLPs with equal timestamp values. Then the monitor  $P_0$  will receive a set of intervals  $\tau_{sat}$ , one from each process, in which the interval  $\tau_i$  from each process  $P_i$  contains the same SLP timestamp value. Let  $\Sigma_{sat}$  contain the starting timestamps  $TS_i$  from the earliest such set of intervals  $\tau_{sat}$ , as indicated by the interval start times. Let  $T_{detect}$  be the maximum of  $\Sigma_{sat}$ . Each timestamp in  $\Sigma_{sat}$  is part of some interval of SLP timestamps. Clearly,  $T_{detect}$  must then be the starting time  $TS_i$  of the interval  $\tau_i$  from which it was taken. Otherwise, there would be some smaller valued set of intervals  $\tau_{sat}$ , contradicting our definition.

The monitor process then collects a set of SLP timestamp intervals, one from each  $P_i$ . From this set, it selects one starting timestamp  $TS_i$  as a candidate for  $T_{detect}$ . This value is kept in variable  $CTime$ .

The monitor compares each of the intervals  $\tau_i$ , in turn, to  $CTime$ . If some interval contains a timestamp equal to  $CTime$ , a counter is incremented. If the interval is closed and is too early to contain a timestamp equal to  $CTime$ , the interval is discarded. The monitor then waits until it receives a potentially valid interval to replace the one which was discarded. A potentially valid interval is the first interval in the queue for which

- $TF_i \geq CTime$ , or
- $TF_i = 0$  and  $TS_i > \max(TF_i : TF_i < CTime)$ .

Thus, an interval is potentially valid if it contains a timestamp equal to  $CTime$  or some later valued  $T_{detect}$  candidate. This new interval value is then compared to  $CTime$ . If the interval contains a timestamp equal to  $CTime$ , the counter is incremented.

If the current value of  $CTime$  is  $T_{detect}$ , then the counter will have been incremented for each process and the predicate will be detected. If the counter was not incremented for each process, then  $CTime$  was not  $T_{detect}$ , a new candidate will be chosen from the set of global timestamp intervals, and the process will continue.

**Correctness** Our obligation in showing that the protocol is correct is twofold. First, we must establish that if the monitor process declares that the global predicate was satisfied then there was an instant in time  $t$  at which  $\mathcal{L}(C_i(t), i, A_i)$  for every  $i$  in  $SYS$ (safety). Then we must show that if the global state predicate was true over a real time interval of at least  $2\epsilon(1 + \rho_M)$  then the monitor process will declare it (liveness).

**Theorem 17 (Safety)** *If  $P_0$  asserts  $A$ , then there exists some real time instant  $t$  for which  $\mathcal{L}(C_i(t), i, A_i)$  for every  $i$  in  $SYS$ .*

**PROOF:** Clearly, by the algorithm,  $P_0$  will assert  $A$  if and only if *noverlaps* is incremented within line 10 or line 15 on every iteration of the for loop (figure 4.8).

Suppose that *noverlaps* is incremented in line 10. Then either  $Overlap(\tau_i, CTime)$  is true or the predicate  $((ETime(\tau_i) = 0) \wedge (STime(\tau_i) \leq CTime) \wedge BufferEmpty(i))$  is true.

By definition,  $Overlap(\tau_i, CTime)$  is only true if  $\tau_i$  contains a timestamp equal to  $CTime$ . If  $ETime(\tau_i)$  is equal to zero then  $\tau_i$  is an open interval. The value for  $CTime$  remains constant within the for loop. Further, each new value for  $CTime$  is aged prior to its use. Thus any timestamp which would close this open interval at a time which is less than  $CTime$  would have been received prior to this iteration (and would be at the head of the appropriate queue, since channels are FIFO). Thus, if  $BufferEmpty(i)$  is true and  $STime(\tau_i)$  is less than or equal to  $CTime$ , then  $\mathcal{L}(CTime, i, A_i)$  holds.

Suppose then that *noverlaps* is incremented within line 15. Then  $Overlap(\tau_i, CTime)$  is true or the predicate  $((ETime(\tau_i) = 0) \wedge (STime(\tau_i) \leq CTime))$  is true.

Again, if  $Overlap(\tau_i, CTime)$  is true then  $\mathcal{L}(CTime, i, A_i)$  holds, by definition. If  $ETime(\tau_i)$  is equal to zero then  $\tau_i$  is an open interval. By the protocol,  $\tau_i$  was returned by a call to *GetValidInterval*. Since  $STime(\tau_i)$ , the initial SLP timestamp of the  $\tau_i$  interval, is less than or equal to  $CTime$  and since  $CTime$  was aged prior to its use, if a closed interval, with this same start time  $STime(\tau_i)$  and which ended prior to  $CTime$  existed, it would already have been received and  $\tau_i$  would have been discarded by the call to *GetValidInterval*. Thus  $\mathcal{L}(CTime, i, A_i)$  holds.

Thus, *noverlaps* is incremented for each process, and  $P_0$  asserts  $A$ , only if there exists a set of global timestamps, one from each process, with equal values. Finally, by lemma 2, there exists  $t$  such that  $\mathcal{L}(C_i(t), i, A_i)$  for all  $i$  in  $SYS$ . ■

The following results are useful in establishing liveness. The first one shows that the value for *CTime* will continue to increase until *CTime* is greater than or equal to  $T_{detect}$ .

**Lemma 16** *Let  $CTime(j)$  be the value of *CTime* at the start of iteration  $j$  of the while loop in the algorithm of figure 4.8. If  $CTime(j) < T_{detect}$  then at the end of execution of the while loop there exists an interval  $\tau_i$  of SLP timestamps such that  $STime(\tau_i) > CTime(j)$ .*

**PROOF:** If  $CTime(j)$  is less than  $T_{detect}$  then some process never produced an interval of SLP timestamps which contains *CTime*. Let this process be  $P_q$ .

Execution of the while loop does not begin until there is an interval  $\tau_i$  at the head of the queue for each process  $P_i$ , for all  $i$  in  $SYS$ . An interval  $\tau_i$  can only be removed from the queue by a call to *GetValidInterval*, which then blocks until another interval is available. Thus, during any iteration of the while loop, outside the call the *GetValidInterval*, there is an interval  $\tau_i$  at the head of the queue for process  $P_i$ .

Let  $\tau_q$  be the interval at the head of the queue for  $P_q$  at the start of iteration  $j$  of the while loop. Then  $\tau_q$  must be earlier or later than an interval which would overlap  $CTime(j)$ . Suppose it is earlier; then it must have ended prior to  $CTime(j)$ . Since each *CTime* value is aged prior to its use, the closed interval values for  $\tau_q$  must have already been received. Thus, *GetValidInterval* will be called. By definition, it will return an interval  $\tau_{over}$  with  $STime(\tau_{over}) > CTime$ , since no valid interval from  $P_q$  with an earlier start time exists. Thus,  $STime(\tau_{over})$  will be available for selection as the next value of *CTime*.

If  $\tau_q$  is later than any interval which would overlap  $CTime(j)$ , then it can be selected as the next value of  $CTime$ .

In either case, an interval start time which is greater than  $CTime(j)$  will be available for selection as the next value of  $CTime$ . ■

The next result shows that  $T_{detect}$  will never lie between two successive values of  $CTime$ .

**Lemma 17** *If closed interval  $\tau_i$  is discarded then  $\tau_i \notin \Sigma_{sat}$ .*

PROOF: The proof is by induction. Let  $S_j$  be the start times of the intervals  $\tau_i$  at the head of the queues at the beginning of iteration  $j$  of the while loop, i.e.  $S_j = \{STime(\tau_i) : i \text{ in } SYS, i \neq 0\}$ . By the protocol for the application processes given in figure 4.7,  $S_0$  contains the timestamp of the first SLP interval generated by each process.  $CTime(0)$  is the minimum of this set. The second value in each queue will be the closed interval values that correspond to the open interval values in  $S_0$ . An interval can only be discarded through a call to *GetValidInterval*. By definition, any call to this function during the first iteration will return the second value in the queue, since all end times in the second interval are greater than or equal to  $CTime(0)$ . Thus, no closed interval will be discarded.

Now suppose that no closed interval value in  $\Sigma_{sat}$  has been discarded during the first  $m$  iterations. Consider iteration  $m + 1$ . Let  $CTime(m + 1) = STime(\tau_q)$ . All earlier closed intervals generated by  $P_q$  have been discarded. By our assumption, none of these intervals contained  $T_{detect}$ . Then  $CTime(m + 1) \leq T_{detect}$  and  $\tau_q$  is the earliest interval from  $P_q$  which could contain  $T_{detect}$ .

By definition, *GetValidInterval* will only discard closed intervals which have ending interval SLP timestamps which are less than  $CTime(m + 1)$ . Such an interval does not



overlap  $\tau_q$ . Then, since  $\tau_q$  is the earliest interval from  $P_q$  which could contain  $T_{detect}$ , none of the discarded intervals are in  $\Sigma_{sat}$ .

**Theorem 18 (Liveness)** *If there exists real time instants  $t_1$  and  $t_2$  such that  $C_i(t_2) - C_i(t_1) \geq 2\epsilon(1 + \rho_M)$  and  $\mathcal{L}(C_i(t), i, A_i), t_1 \leq t \leq t_2$ , for all  $i$  in  $SYS$ , then  $P_0$  will assert  $A$ .*

**PROOF:** If there exists real time instants  $t_1$  and  $t_2$  such that  $C_i(t_2) - C_i(t_1) \geq 2\epsilon(1 + \rho_M)$  and  $\mathcal{L}(C_i(t), i, A_i), t_1 \leq t \leq t_2$ , for all  $i$  in  $SYS$ , then by lemma 15, there exists a set of equal SLP timestamps, one from each process.

By lemma 16 and the protocol,  $CTime$  takes on monotonically increasing values at each iteration of the while loop. By lemma 17, no closed interval which contains  $T_{detect}$  is ever discarded. Then eventually  $CTime$  will equal  $T_{detect}$ .

Now suppose that for iteration  $j$ ,  $CTime$  is equal to  $T_{detect}$ . Consider the interval  $\tau_i$  of SLP timestamps at the head of the queue for any process  $P_i$ ,  $i$  in  $SYS$ , at the start of iteration  $j$ . By lemma 17 no valid interval has been discarded. Since  $CTime(j)$  is equal to  $T_{detect}$ ,  $STime(\tau_i)$  is then less than or equal to  $CTime(j)$ . This interval  $\tau_i$  can either be open or closed.

Suppose that it is open. If the buffer is empty, then *noverlaps* will be incremented. If the buffer is not empty, then *GetValidInterval* will be called. By definition, *GetValidInterval* will return the interval for  $\tau_i$  that contains  $T_{detect}$  and *noverlaps* will be incremented.

Suppose then that the initial value of  $\tau_i$  is closed. Then, since no valid interval has been discarded and every process has generated an interval containing  $CTime$ , either

- $STime(\tau_i) < CTime$  and  $ETime(\tau_i) < CTime$ , or
- $STime(\tau_i) \leq CTime$  and  $ETime(\tau_i) \geq CTime$ .

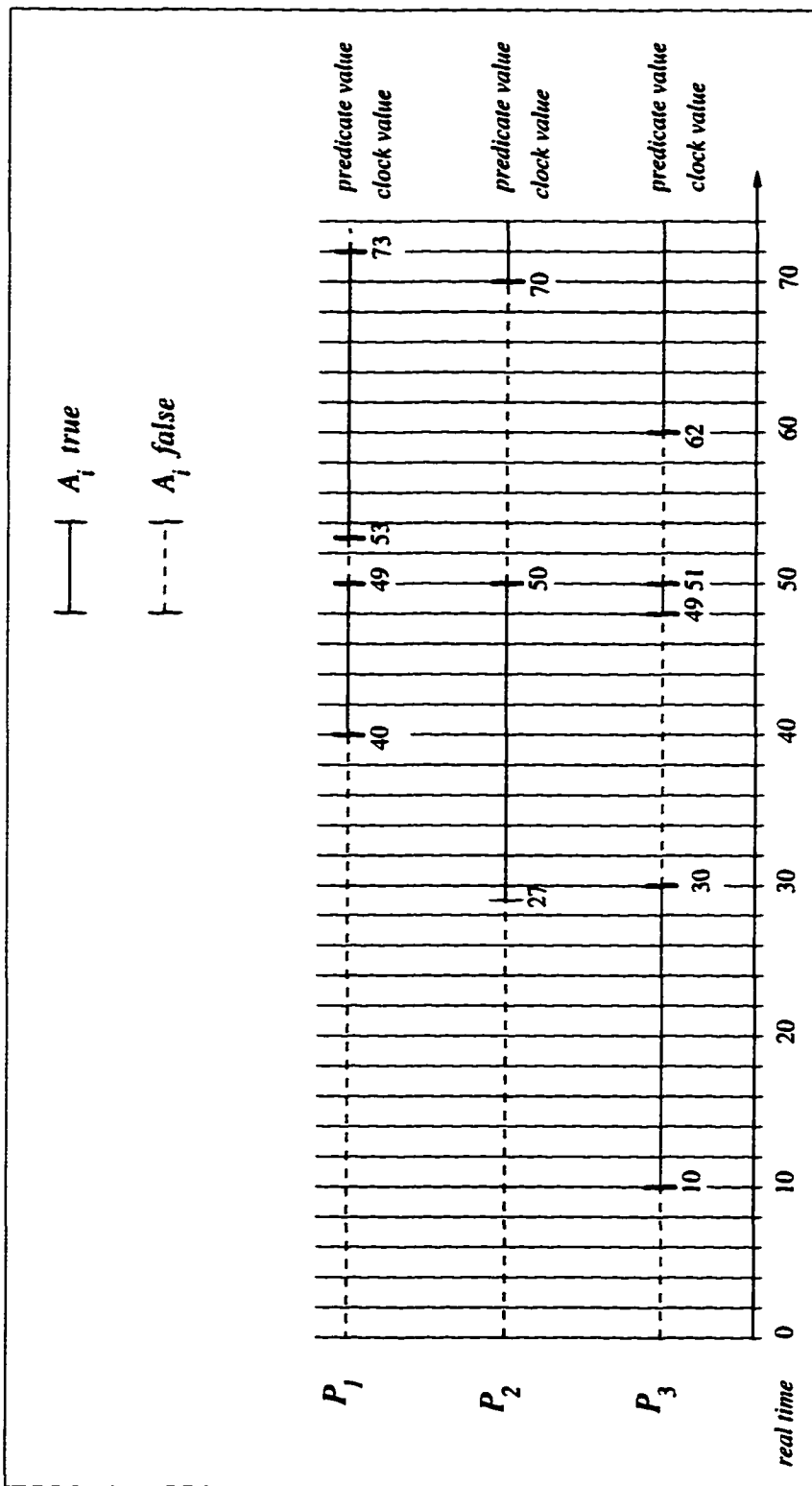


Figure 4.9: Exemplary Execution for Protocols *Unstable-Mon* and *Unstable-App*

Step	Event	t	ptr	$\tau_1$	$\tau_2$	$\tau_3$	$Buf_1$	$Buf_2$	$Buf_3$	i	CTime	noverlaps
1	t ← 0	0	1	(0,0)	(0,0)	(0,0)	$\emptyset$	$\emptyset$	$\emptyset$	0	0	0
2	t ← 13	13	3						(10,0)			
3	t ← 30	30	↑						(10,0), (10,27)			
4	t ← 33	33	↑					(27,0)	$\emptyset$			
5	t ← 43	43	↑				(40,0)	$\emptyset$				
6	ptr ← 1		1	(40,0)			$\emptyset$					
7	ptr ← 1		1		(27,0)							
8	ptr ← 4		4			(10,27)						
9	ptr ← 7		7								10	0
10	ptr ← 17									1		0
11	ptr ← 17									2		1
12	ptr ← 17									3		2
13	ptr ← 26											0
14	ptr ← 17									1	27	
15	ptr ← 17									2		1
16	ptr ← 17									3		2
17	ptr ← 26										40	0
18	ptr ← 17									1		1
19	ptr ← 17									2		2
20	t ← 53	53	13				(40,46)	(27,47)				
21	t ← 56	56					(40,46), (53,0)	$\emptyset$				
22	t ← 63	63					$\emptyset$		(62,0)			
23	ptr ← 17		17			(62,0)			$\emptyset$	3		2
24	t ← 65	65	25								62	0
25	ptr ← 17			(53,0)			$\emptyset$			1		1
26	t ← 72	72					(53,70)					
27	t ← 73	73					$\emptyset$	(70,0)				
28	ptr ← 17				(70,0)			$\emptyset$		2		1
29	ptr ← 17									3		2
30	ptr ← 26										70	0
31	ptr ← 17			(53,70)						1		1
32	ptr ← 17									2		2
33	ptr ← 17									3		3

Figure 4.10: Execution of Protocol Unstable-Mon

If  $S\text{Time}(\tau_i) < C\text{Time}$  and  $E\text{Time}(\tau_i) < C\text{Time}$ , then by the test in line 12, a call will be made to *GetValidInterval*. By definition, this call will return the earliest overlapping interval and *noverlaps* will be incremented. If  $S\text{Time}(\tau_i) < C\text{Time}$  and  $E\text{Time}(\tau_i) \geq C\text{Time}$ , then *noverlaps* will be incremented by the test in line 9.

The variable *noverlaps* will then be incremented for each  $P_i$  when  $C\text{Time}$  is equal to  $T_{\text{detect}}$ . Since  $C\text{Time}$  must eventually equal  $T_{\text{detect}}$  if the predicate remains true for a real time interval of  $2\epsilon(1 + \rho_M)$ , the predicate will be detected. ■

**Example** We will now trace execution of the protocol for the execution time history shown in figure 4.9. The figure shows execution for a three process system. The axis for process  $P_i$  shows the value of local predicate  $A_i$ , as well as certain values for local clock  $C_i$ , as a function of absolute real time. If a local clock value is not given for some value of real time, the clock value is assumed to be the same as the value of real time. We assume that the value of the maximum clock skew  $\epsilon$  for this system is three. For simplicity, we further assume that message delivery is instantaneous, that the monitor process clock skew from real time is zero, and that program execution time is negligible.

We monitor the execution using table 4.10. Each entry in the table gives program variable values for an arbitrary step of the protocol's execution. If no value is given for some variable, the value is assumed to be the same as that for the previous step. Each entry was chosen to correspond with some event of interest from the execution. The table entries, or the events, occur sequentially in real time, each event being at the same real time instant, or later, than the previous event. Entries correspond with two types of events: reaching a certain value of absolute real time, or reaching a certain statement within the program. Thus, between any two events, either some interval of real time has elapsed, or

the program counter has advanced. The variable *ptr* reflects the current point of execution within the program code. Its value indicates the next program statement to be executed. A *ptr* table entry of † indicates that the value of *ptr* has not changed since the previous event.

Steps one through nine in the table trace execution of the initialization code, lines one through six. The first step indicates initial values, prior to execution of line one. Step two corresponds to receipt of the open interval values for the first interval values received from process  $P_3$ . Note that *ptr* has value three. Here a call to *GetValidInterval* has been made, at line two, and execution is stalled waiting for a valid interval from process  $P_1$ . Steps three through five correspond with receipt of interval values from processes  $P_3$ ,  $P_2$ , and finally  $P_1$ . Note that the value of *ptr* has not changed. Steps six through eight correspond with execution of the initial for loop, filling in values for  $\tau_1$ ,  $\tau_2$ , and  $\tau_3$ . Step nine corresponds with completion of execution of lines four through seven, filling in initial values for *CTime* and *noverlaps*.

Steps 10 through 33 corresponds to the execution of the while loop, until the predicate is finally detected. Execution of steps 10 through 17 are straightforward. These steps correspond to two iterations of the while loop. During these iterations, no calls to *GetValidInterval*, at line 13, are required, and thus no intervals are discarded. At step 17, *CTime* has taken the value of  $S\text{Time}(\tau_1)$ , the value it will have for the next iteration of the while loop.

Between steps 17 and 18, one pass is made through the for loop which begins at line eight. Here, *noverlaps* is incremented because *CTime* is equal to  $S\text{Time}(\tau_1)$ . Step 19 corresponds to completion the next pass through the for loop. Here, *noverlaps* is incremented because

$S\text{Time}(\tau_2)$  is less than  $C\text{Time}$ ,  $E\text{Time}(\tau_2)$  is zero, and the buffer for  $P_2$  is empty. Since the value  $C\text{Time}$  was aged prior to its use, we know that if the open interval  $\tau_2$  had been closed prior to  $C\text{Time}$ , then the closed interval values would already have been received and the buffer for  $P_2$  would not be empty. The third pass through the for loop occurs between steps 19 and 23. Between steps 19 and 20, a call to *GetValidInterval*, at line 13, is made, as  $\tau_3$  is a closed interval which ended prior to  $C\text{Time}$ . While execution is suspended, pending receipt of a valid interval value for  $P_3$ , interval values are received from  $P_1$  and  $P_2$ , step 20, and then  $P_2$  again, step 21. Finally, a valid interval value is received for  $P_3$ , step 22. Note that the local clock values corresponding to the real time interval  $[48, 50]$ , in which  $A_3$  was true, were never sent because the predicate did not remain true for  $\epsilon$  as read by  $C_3$ . During this real time interval, the conjunctive predicate was globally true. This is consistent with our assumption that if the predicate is not true for  $2\epsilon$ , as read by any  $P_i$ , the protocol may not detect it.

Step 24 corresponds to the aging of the next value for  $C\text{Time}$  prior to its use. When line 24 is reached, the value of  $C_0$  is 63 and the value of  $C\text{Time}$  is 62. Thus,  $P_0$  must wait until its clock reaches 65 before it can proceed with the next iteration of the while loop.

Between steps 24 and 25, the first pass through the for loop of line eight is made. A call to *GetValidInterval* was made at line 13 and the value (53, 70) was returned from the buffer for  $P_1$ . The value for *noverlaps* was incremented at line 15. Between steps 25 and 26, a call to *GetValidInterval* was made in order to get a valid interval for  $P_2$ . While execution was suspended by this call, a closed interval value for  $P_1$  was received. Step 27 corresponds with receipt of a valid interval value for  $P_2$ . Note that *noverlaps* is not incremented since the start time for this interval is greater than  $C\text{Time}$ . Between steps 28 and 30, the final

two passes are made through the for loop, *noverlaps* is incremented twice, at line 10, and a new value for *CTime* is chosen.

Steps 31 through 33 correspond to the final pass through the while loop, during which the predicate is detected. Note that the predicate did not remain true for  $2\epsilon(1 + \rho)$ , yet it was still detected. Again, if the protocol detects the predicate, it was true at some real time instant, and if the predicate remains globally true for a real time interval of  $2\epsilon(1 + \rho)$ , then the protocol will detect it.

**Discussion** Unlike protocols which evaluate predicates over consistent global states, this protocol doesn't require that vector time be kept. Thus, no control information must be added to the application messages. Although we have required that message delivery time be bounded, the algorithm could be structured so that the evaluation only takes place based on closed interval values. This would obviate the need for the monitor process to age timestamps prior to their use.

The protocol requires that the predicate being evaluated remain globally true for at least a real time interval of  $2\epsilon(1 + \rho_M)$ . Recall that typical values of  $\rho_M$  are on the order of  $10^{-6}$  and that protocols which guarantee synchronization can provide maximum clock skews on the order of tens of milliseconds. Many physical systems have periods greater than this interval. Using this approach, we are then able to detect predicates which could not (easily) be detected by traditional methods.

## 4.5 Conclusions

We have presented several algorithms for detection of global predicates on the state of distributed systems with roughly synchronized clocks. Structuring these algorithms on the assumption of a rough global time base provided several advantages.

First, by scheduling the evaluation at an agreed upon clock value, we could develop an algorithm that is message optimal, in that the only message required were those with which the process states were accumulated. Further, we could evaluate predicates that were based on attainment of a specified system state at a given real time instant, plus or minus a real time interval of  $\epsilon(1 + \rho_M)$ . None of the algorithms required that vector time be kept. Thus, there is no increase in application message size as the number of system processes increases.

Evaluation in each of these algorithms is limited to predicates which remain globally true for at least a real time interval of  $2\epsilon(1 + \rho_M)$ . However, we are able to detect occurrence of predicates which are not detectable by traditional methods.



## Chapter 5

# Conclusions

We have divided this chapter into two sections. Our results are summarized in the first section. We conclude by outlining directions for further research in the final section.

### 5.1 Summary

All potentially instantaneous global states, for a given execution sequence, can be determined from a causal ordering of the events comprising the execution. Evaluating global predicates over these potentially instantaneous states, or consistent global states, allows detection of predicates which did not occur in the current execution, but may occur in subsequent executions. This type of analysis is particularly useful for program debugging and verification. Causal ordering is based on the global ordering of local events imposed by message passing and can then be efficiently constructed without assuming the existence of either shared memory or a global time base, the dominant characteristics of distributed systems.

The vast majority of global state predicate evaluation protocols structure the evaluation

over consistent global states. This is partially attributable to the assumption, for whatever reason, that no global time base is available within a distributed system. However, numerous fault-tolerant clock synchronization protocols have been presented in the literature [21, 38, 13, 37, 61, 57, 53, 35, 64]. Thus, the assumption that a global time base cannot be efficiently constructed seems overly restrictive in certain cases. Further, these protocols provide maximum clock skews at least on the order of message passing delays [55].

Protocols based on the use of consistent global states can sometimes be used to determine the state of the system at some real time instant. For example, the evaluation protocol of Cooper and Marzullo reconstructs all paths that an execution could have taken, based on an observed execution sequence [9]. If their protocol detects that some predicate definitely occurred during the computation, then, at some real time instant, the predicate was true. However, their protocol does not allow detection of whether or not an arbitrary predicate occurred during the execution. In other words, if their protocol detects that the predicate definitely occurred, then it occurred at some real time instant. However, the predicate could occur at some real time instant, and their protocol may not detect it. Garg and Waldecker developed a similar protocol, designed for runtime evaluation, rather than postmortem evaluation [66].

The fundamental problem with detecting predicates over consistent global states is that causal ordering only partially orders the events of an execution. In order to develop a necessary and sufficient test for occurrence of an arbitrary predicate, a total ordering of events within an absolute time frame is required. One must be able to reconstruct the instantaneous global states that actually comprised the execution.

The ability to construct an instantaneous global state is especially important to dis-

tributed monitoring and control systems. These distributed systems control and monitor physical systems, and thus must determine the state of the system being controlled within an absolute real time frame. For example, a monitoring system may want to determine the state of the system being monitored at a certain point in real time; i.e. "What was the tank water level at two o'clock?". Such systems also may need to determine if, or when, some predicate occurred. An example might be detecting a predicate such as "Are all valves closed?", so that some alarm function can be executed.

Evaluation of these kinds of predicates over consistent states is difficult. A predicate such as "Did all valves close?" cannot be evaluated using consistent global states, assuming the valves may open and close arbitrarily. Detection of such a predicate would require the ability to construct all global states that occurred during the computation. Without some global time base, it is also impossible to evaluate a predicate on the state of the systems at a specific value of real time. For example, for a predicate such as "Were all tanks full at two o'clock?", all tank monitoring processes must have roughly the same idea of when it is two o'clock.

Detection of these kinds of predicates arises naturally within systems in which a rough global time base can be assumed. In section 4.3 of chapter 4, we gave a protocol for scheduled predicate evaluation. This protocol allows detection of predicates at a certain instant in real time, assuming that some process clock is accurate. Thus, a predicate such as "Were all tanks full at two o'clock?" is readily detected. In section 4.4 of chapter 4, we developed a protocol to detect conjunctive predicates over the local process states. This facilitates detection of a specific predicate whenever it occurs during the computation. With this protocol, it is possible to detect a predicate such as "Did all valves close?".

The global time base is rough, thus our detection techniques are limited to predicates which remain true for a sufficiently long interval, an interval of duration  $2\epsilon(1+\rho_M)$ . However, the magnitude of this interval is on the order of tens of milliseconds, well below the period of many physical systems.

In addition to facilitating more efficient detection of certain unstable predicates, the practical assumption of a rough global time base allows development of simple and efficient protocols for stable predicate evaluation. As we showed in section 3.1.3 of chapter 3, assuming a global time base we can develop a protocol which always detects termination with a single pass through the system processes, without requiring the addition of control information to basic application messages. Protocols based on the use of consistent global states typically either detect the termination with two passes through the system processes, without requiring control information on application messages, or they detect the termination in a single pass, and require control information on application messages [19, 11, 63, 31, 40].

Protocol development within a rough global time base can be complex because we tend to think of processor actions as occurring in a single time frame. It can be confusing to deal with several processors acting independently within unique and different time bases. The SLP and GLP formalizations ease development of protocols for predicate evaluation within a rough global time base. By globalizing the timestamp on a local property according to the protocol we presented in section 2.5.2 of chapter 2, a processor can make a time-stamped statement about its local state at a time which is valid by any process clock. Thus, the timestamps on such assertions can all be referenced to a single process' clock, and we are once again dealing with the more easily understood single time frame.

By using SLPs, we can more efficiently detect the simultaneous truth of local properties.

This leads to the intuitive task of detecting equal timestamps on assertions about the truth of local predicates. For example, we readily developed an unstable predicate detection protocol by having each process assert the truth of some local property via SLPs. The monitor process then looks for equal timestamps on the local assertions. A similar approach could be taken in a system with a perfect global time base. We can then focus on ways to more efficiently determine when processes make assertions with equal timestamps, without the distraction of dealing with roughness within the global time base.

Thus, by basing predicate evaluation on the assumption of a rough global time base, we can detect unstable predicates which are not readily detected over consistent global states and we can develop simple and efficient stable predicate evaluation protocols. The use of SLPs and GLPs facilitated protocol development within this rough time base.

## 5.2 Directions for Further Research

The efficiency and simplicity of the stable predicate evaluation protocols we have developed prompt further investigation into more complex cases of both distributed termination detection and distributed deadlock detection. Specifically, our results indicate that we might be able to develop simple and efficient protocols for termination detection within dynamic systems. Dynamic systems are systems in which the set of system processes is not fixed; processes are allowed to join and leave the system. The techniques used in our earlier protocols should apply equally well to dynamic termination detection and may lead to simpler, more efficient solutions.

Another area within stable predicate evaluation that we plan to investigate in more depth is distributed deadlock detection. We have developed a protocol for detecting dead-

lock under the most simple model of distributed deadlock. This protocol is efficient and facilitates deadlock resolution, in that a unique process detects the deadlock and aborts its transaction. This protocol should be extensible to more complex deadlock models. By our earlier protocol, processes are in one of two states, one in which they are active and one in which they are waiting for a resource. While in this second state, the set of resources held, and requested, by a process remain constant. Circulation of a token then indicates that all processes are simultaneously in this second state.

That the approach is extensible to more complex models is exemplified by its application to the OR model of deadlock. Under the OR model of deadlock, processes post a request for some set of resources, and can proceed upon receiving any single resource from the set. It would seem, then, that our protocol could be extended only by modifying the token routing. Within a protocol for deadlock detection under the OR model, processes would post their request for some set of resources, and enter the "second" state upon receipt of a *Hold* message. Upon receipt of any *Grant* message, processes re-enter the "first" state. Tokens would be sent to the data managers of all requested resources and complete circulation of any single token would indicate deadlock. Here again, in this second state, the set of all resources held and request by a process are constant. Complete token circulation then indicates that all processes are simultaneously in this second state. The token routing ensures the correct dependency chain.

We have not encountered a fully distributed and symmetric protocol for unstable predicate evaluation within the literature. Our scheduled evaluation protocol is distributed and symmetric, but cannot detect predicates whenever they become true. We plan to investigate whether or not the assumption of a global time base might facilitate development of

such a protocol.

A platform which is particularly attractive for the application of our techniques is local networks of workstations. These distributed systems are receiving a lot of attention as economical alternatives to traditional parallel processors. The fact that these systems are local make them good candidates for tight clock synchronization, in that the uncertainty in message delivery times introduced by multiple hops between source and destination is eliminated. The ways in which the assumption of a global time base might make the tasks performed by these systems more efficient bears further investigation.

Finally, the argument that the assumption of a rough global time base is a practical one is not completely convincing without an implementation to bear out the arguments. We then plan to implement both clock synchronization and some of our protocols to demonstrate the protocols' practicality and efficiency. Here again, we will focus on local networks of workstations.

Thus, we plan deeper research in areas we touched upon within the dissertation, specifically, dynamic termination detection, general distributed deadlock detection, and distributed unstable predicate detection. Further, we plan to investigate other areas of application of our techniques, starting with the tasks commonly performed within local networks of workstations for parallel processing. Finally, we plan to implement our protocols to demonstrate that the approach is practical.

## Appendix A

# Unstable Predicate Evaluation

## Using GLPs

We present a centralized protocol for the detection of conjunctive predicates, over the local process states, as they occur. Unlike the protocol presented in section 4.4 of chapter 4, the evaluation is based upon the use of GLPs. This protocol has been published previously [29].

Other than basing the evaluation on the use of GLPs, the protocol is identical to the earlier protocol. The evaluation is unscheduled; the protocol detects the predicate when it first becomes true. The algorithm is not symmetric. One process,  $P_0$ , called the monitor process, is not part of the computation. Its purpose is to collect information from the other application processes and evaluate the global state predicate. We assume that message passing is reliable, and that messages sent on the channel between  $P_0$  and any other  $P_i$  are delivered in FIFO order. Values received by  $P_0$  are assumed to be buffered in FIFO order until they are used. We further assume that message delivery time is bounded within some known interval  $\delta$ , as read by any  $C_i$ ,  $i \in SYS$ .



$state_i$ :  $P_i$ 's state, initially *unsatisfied*  
 $alarm_i(T)$ :  $P_i$ 's alarm, signals  $P_i$  at  $C_i(t) = T$ ;  $alarm_i(0)$  cancels the alarm  
 $clock_i$ :  $P_i$ 's current clock reading

state	event	action
unsatisfied	$A_i$	$TS_i \leftarrow clock_i + \epsilon$ $alarm_i(TS_i + \epsilon)$ $state_i \leftarrow transition$
transition	$\neg A_i$	$alarm_i(0)$ $state_i \leftarrow unsatisfied$
	$alarm_i$ expires	send $m(TS_i, 0, i)$ to $P_0$ $state_i \leftarrow satisfied$
satisfied	$\neg A_i$	$TF_i \leftarrow clock_i - \epsilon$ send $m(TS_i, TF_i, i)$ to $P_0$ $state_i \leftarrow unsatisfied$

Figure A.1: Protocol (*UnstableGLP-App*): Modification to Protocol *Unstable-App*

Since we are considering only predicates which are *globally* true for an interval of at least  $2\epsilon$ , as read by any  $C_i$ , each local predicate must also be true for at least  $2\epsilon$ . Suppose each  $P_i$  then monitors the truth of its local predicate and asserts the local predicate  $A_i$  whenever  $\mathcal{L}(T_i, i, A_i)$ ,  $TS_i - \epsilon \leq T_i \leq TS_i + \epsilon$ . Then, by the clock axiom, there must be a set of global timestamps within  $\epsilon$  of each other. As we will show later, if the timestamps are within  $\epsilon$ , then there is an instant in time at which the predicate is globally true. Thus, to determine when the predicate is true, we only need find a set of global timestamps within  $\epsilon$  of each other.

## Protocol

Algorithms for the application and monitor processes are given in figures A.1 and A.2. Each application process sends a message to the monitor process whenever the value of the local predicate changes. These messages contain the starting and ending global timestamps of the interval over which the local predicate  $A_i$  holds. Let  $TS_i$  be the minimum global timestamp for which  $A_i$  can be asserted by  $P_i$ . Let  $TF_i$  be the maximum timestamp from

**CTime** value which is current candidate for  $T_{detect}$   
 $\tau_i$  interval endpoints  $(TS_i, TF_i)$  most recently read from  $P_i$ 's buffer  
**BufferEmpty(i)** returns a boolean indicating whether or not the buffer associated with  $P_i$  is empty  
**AgeTimestamp(T)** blocks until the local clock value is  $T + \delta + 2\epsilon$   
**STime( $\tau_i$ )**  $\tau_i$ 's interval start time,  $TS_i$   
**ETime( $\tau_i$ )**  $\tau_i$ 's interval end time,  $TF_i$   
**GetValidInterval(i,T)** reads interval values from  $P_i$ 's buffer; returns a potentially valid interval; an interval is potentially valid if  $TF_i \geq T - \epsilon$  or  $TF_i = 0 \wedge TS_i > \max(TF_i : TF_i < T - \epsilon, 0)$ ; the function blocks until such an interval is available; if the start time for this earliest interval appears in both a closed and open interval then the closed interval values are used  
**Overlap( $\tau_i, T$ )** returns true if any value on the interval  $\tau_i$  is within  $\epsilon$  of  $T$ , false otherwise; if  $\tau_i$  denotes an open interval, then only the value for  $STime(\tau_i)$  is used.

```

for i ← 1 to N - 1 do
   $\tau_i \leftarrow$  GetValidInterval(i, 0)
end for

CTime ← min{STime( $\tau_i$ ) : 1 ≤ i ≤ N - 1}
noverlaps ← 0
found ← false
while not(found) do
  for i ← 1 to N - 1
    if Overlap( $\tau_i$ , CTime) ∨ ((ETime( $\tau_i$ ) = 0) ∧ (STime( $\tau_i$ ) ≤ CTime +  $\epsilon$ ) ∧ (BufferEmpty(i))) then
      noverlaps++
    else
      if ((ETime( $\tau_i$ ) = 0) ∧ (¬BufferEmpty(i))) ∨ (ETime( $\tau_i$ ) < CTime -  $\epsilon$ ) then
        GetValidInterval(i, CTime)
        if Overlap( $\tau_i$ , CTime) ∨ ((ETime( $\tau_i$ ) = 0) ∧ (STime( $\tau_i$ ) ≤ CTime +  $\epsilon$ )) then
          noverlaps++
        endif
      endif
    end for
  end for

  if (noverlaps = N - 1) then
    ASSERT A
    found ← true
  else
    noverlaps ← 0
    CTime ← min{STime( $\tau_i$ ) : STime( $\tau_i$ ) > CTime, 1 ≤ i ≤ N - 1}
    AgeTimestamp(CTime)
  endif
endwhile
  
```

Figure A.2: Protocol (*UnstableGLP-Mon*): Modification to Protocol *Unstable-Mon*

the same interval. When the value of the local predicate changes from false to true, and then remains true for  $2\epsilon$ , the application process will send a message to the monitor process containing  $(TS_i, 0)$ . Thus, the monitor knows that  $A_i$  has become true for  $P_i$  and that its initial global timestamp was  $TS_i$ . When  $A_i$  changes from true to false at  $C_i(t) = TF_i + \epsilon$ ,  $P_i$  sends  $(TS_i, TF_i)$  to  $P_0$ . If  $TF_i = 0$ , then we refer to the interval as “open”. Otherwise, the interval is said to be “closed”.

The algorithm for  $P_0$  is given in figure A.2. By the algorithm for the application processes, each  $P_i$  generates global timestamp intervals  $(TS_i, TF_i)$  for which  $\mathcal{L}(T, i, A)$ ,  $TS_i - \epsilon \leq T \leq TF_i + \epsilon$ . By the clock axiom, if the global predicate is true for at least  $2\epsilon$ , as read by any process clock, then a set,  $\tau_{sat}$ , of intervals will be generated, one from each  $P_i$ , which contain timestamps that differ by no more than  $\epsilon$ . Let  $\Sigma_{sat}$  contain the starting timestamps  $TS_i$  from earliest set of intervals  $\tau_{sat}$ , as indicated by the starting timestamps. Let  $T_{detect}$  be the maximum of  $\Sigma_{sat}$ . Each timestamp in  $\Sigma_{sat}$  is part of some global timestamp interval. Clearly,  $T_{detect}$  must then be the starting time of the interval from which it was taken. Otherwise, there would be some smaller valued set of timestamps which differ by at most  $\epsilon$ , contradicting our definition of  $\Sigma_{sat}$ .

The monitor process then collects a set of global timestamp intervals, one from each  $P_i$ . From this set, it selects one starting timestamp as a candidate for  $T_{detect}$ . This value is kept in variable  $CTime$ .

The monitor compares each of the intervals, in turn, to  $CTime$ . If some process interval contains a timestamp within  $\epsilon$  of  $CTime$ , a counter is incremented. If the interval is closed and is too early to contain a timestamp within  $\epsilon$  of  $CTime$ , the interval is discarded. The monitor then waits until it receives a potentially valid interval to replace the one which was

discarded. A potentially valid interval is one for which

- $TF \geq CTime - \epsilon$ , or
- $TF = 0$  and  $TS > \max(TF : TF < CTime - \epsilon)$ .

Thus, an interval is potentially valid if it contains a timestamp within  $\epsilon$  of  $CTime$  or some later valued  $T_{detect}$  candidate. This new interval value is then compared to  $CTime$ . If the interval contains a timestamp within  $\epsilon$  of  $CTime$ , the counter is incremented.

If the current value of  $CTime$  is  $T_{detect}$ , then the counter will have been incremented for each process and the predicate will be detected. If the counter was not incremented for each process, then  $CTime$  was not  $T_{detect}$ ; a new candidate will be chosen from the set of global timestamp intervals, and the process will continue.

## Correctness

The following results are useful in establishing the correctness of our protocol. It shows that if a predicate remains globally true for at least  $2\epsilon$ , as read by any  $C_i$ ,  $i \in SYS$ , then all processes will generate global timestamps within  $\epsilon$  of each other.

**Lemma 18** *If  $\mathcal{L}(C_i(t), i, A_i)$ ,  $t_1 \leq t \leq t_2$ , and  $C_i(t_2) - C_i(t_1) \geq 2\epsilon$  for every  $P_i, i \in SYS$ , then there exists  $T_i$  and  $T_j$  such that  $\mathcal{L}(T_i, i, A_i)$ ,  $TI - \epsilon \leq T_i \leq TI + \epsilon$ ,  $\mathcal{L}(T_j, j, A_j)$ ,  $TJ - \epsilon \leq T_j \leq TJ + \epsilon$  and  $|TI - TJ| < \epsilon$ .*

**PROOF:** Let  $C_i(t_1) = TI - \epsilon$  and let  $C_j(t_1) = TJ - \epsilon$ . By our assumptions,  $A_i$  is true from  $[TI - \epsilon, TI + \epsilon]$  as read by  $C_i$  and  $A_j$  is true over the interval  $[TJ - \epsilon, TJ + \epsilon]$  as read by  $C_j$ . By the clock axiom,

$$TI - 2\epsilon < C_j(t_1) < TI,$$

and

$$TI - \epsilon < TJ < TI + \epsilon.$$

■

Thus if a predicate remains true for at least  $2\epsilon$ , there will be a set of global timestamps in which no two timestamps differ by more than  $\epsilon$ .

The next result shows that when process assert GLPs with timestamps that are within  $\epsilon$  of each other, then the asserted local properties were true at some real time instant.

**Lemma 19** *Suppose that  $\mathcal{L}(T_i, i, A_i)$ ,  $TI - \epsilon \leq T_i \leq TI + \epsilon$ , and  $\mathcal{L}(T_j, j, A_j)$ ,  $TJ - \epsilon \leq T_j \leq TJ + \epsilon$ . Further suppose that  $|TI - TJ| \leq \epsilon$ . Then there exists a real time instant  $t$  for which  $\mathcal{L}(C_i(t), i, A_i) \wedge \mathcal{L}(C_j(t), j, A_j)$ .*

PROOF: If  $|TI - TJ| \leq \epsilon$  then  $TI - \epsilon \leq TJ \leq TI + \epsilon$ . By our assumption then,  $A_i$  was true for  $P_i$  at  $C_i(t) = TJ$ . By the clock axiom,  $TJ - \epsilon < C_j(t) < TJ + \epsilon$ . Thus at real time instant  $t$ ,  $A_i$  held at  $P_i$  and  $A_j$  held at  $P_j$ . ■

The next two results are useful for establishing liveness. The first one shows that the value for  $CTime$  will continue to increase until  $CTime$  is greater than or equal to  $T_{detect}$ .

**Lemma 20** *The value of  $CTime$  increases monotonically for every iteration of the while loop (figure A.2) until  $CTime \geq T_{detect}$ .*

PROOF: Let  $CTime_j$  be the value of  $CTime$  at iteration  $j$  of the while loop. If  $CTime_j$  is less than  $T_{detect}$ , then it is not within  $\epsilon$  of a global timestamp from every other process and there must be some process  $P_q$  which produced no interval that contained a timestamp within  $\epsilon$  of  $CTime_j$ .

Clearly, since *GetValidInterval* blocks until a valid interval is available, each  $\tau_i$  always contains some set of interval values. For  $\tau_q$  this interval must then be earlier or later than an interval which would overlap  $CTime_j$ . Suppose it is earlier; then it must have ended prior to  $CTime_j - \epsilon$ . Since each  $CTime$  value is aged prior to its use, the closed interval values for  $\tau_q$  must have already been received. Thus, *GetValidInterval* will be called. By definition, it will return an interval  $\tau_{over}$  with  $STime(\tau_{over}) > CTime + \epsilon$ , since no valid interval from  $P_q$  with an earlier start time exists. Thus,  $STime(\tau_{over})$  will be available for selection as the next value of  $CTime$ .

If the current value for  $\tau_q$  is later than any interval which would overlap  $CTime_j$ , then it can be selected as the next value of  $CTime$ .

In either case, an interval start time which is greater than  $CTime_j$  will be available for selection as the next value of  $CTime$ . By the algorithm, some greater value will always be chosen and  $CTime_{j+1} > CTime_j$  for all  $CTime_j < T_{detect}$ . ■

The final result shows that  $T_{detect}$  will never lie between two successive values of  $CTime$ .

**Lemma 21**  $\neg(CTime_{j-1} < T_{detect} < CTime_j)$  for all  $j$ .

PROOF: The proof is by induction. Let  $T_j = \{STime(\tau_i) : 1 \leq i \leq N - 1\}$  be the interval start times at the beginning of iteration  $j$  of the while loop.  $T_0$  contains the starting time of the first global timestamp interval generated by each process.  $CTime_0$  is the minimum of this set. Let  $P_q$  be the process which generated  $CTime_0$ . The only way a value from  $T_0$  can be modified is through a call to *GetValidInterval*. The next interval in each process buffer, if there is one, will be the closed interval times for the current (open) interval. *GetValidInterval*, by definition, will not change the start time of any interval  $\tau_i$  for which  $STime(\tau_i) \geq CTime - \epsilon$ . Thus,  $T_1 = T_0$ , and only  $P_q$  can generate intervals

for which  $CTime_0 < STime(\tau) < CTime_1$ . Suppose  $STime(\tau) = T_{detect}$  for one of these intervals; then  $STime(\tau)$  must overlap the remaining process intervals. However, each of these intervals has a starting time which is greater than  $STime(\tau)$ . This contradicts our definition of  $T_{detect}$ . Thus  $\neg(CTime_0 < T_{detect} < CTime_1)$

Now suppose that  $\neg(CTime_{j-1} < T_{detect} < CTime_j)$ ,  $1 \leq j \leq n$ . Then either  $CTime_n = T_{detect}$  or  $CTime_n < T_{detect}$ . If  $CTime_n = T_{detect}$  then, as we will show later,  $P_0$  will assert  $A$ ; the algorithm will terminate, and the assumption will hold. Suppose then that  $CTime_n < T_{detect}$ . We have already shown that  $CTime_{j+1} > CTime_j$  for all  $j$  such that  $CTime_j < T_{detect}$ . Suppose then that  $CTime_n < T_{detect} < CTime_{n+1}$ . By definition, any interval  $\tau$  for which  $STime(\tau) = T_{detect}$  must overlap intervals for all other processes. This includes the interval which has start time  $CTime_{n+1}$ , which contradicts our definition of  $T_{detect}$ . Thus,  $\neg(CTime_n < T_{detect} < CTime_{n+1})$ . ■

Our obligation in showing that the protocol is correct is twofold. First, we must establish that if the monitor process declares that the global predicate was satisfied, then there was an instant in time  $t$  at which  $\mathcal{L}(C_i(t), i, A_i)$  for every  $i \in SYS$  (safety). Then we must show that if there was a  $2\epsilon$  interval over which the predicate was globally true, the monitor process will declare it (liveness).

**Theorem 19 (Safety)** *If  $P_0$  asserts  $A$ , then there exists some real time instant  $t$  for which  $\mathcal{L}(C_i(t), i, A_i)$  for every  $i \in SYS$ .*

**PROOF:** Clearly, by the algorithm,  $P_0$  will assert  $A$  if and only if *noverlaps* is incremented on every iteration of the for loop (figure A.2.) The variable *noverlaps* can only be incremented for  $P_i$  if

- $Overlap(\tau_i, CTime)$  is true, or if
- $ETime(\tau_i) = 0$  and  $STime(\tau_i) \leq CTime + \epsilon$ .

By definition,  $Overlap(\tau_i, CTime)$  is only true if  $\tau_i$  contains a timestamp within  $\epsilon$  of  $CTime$ . If  $ETime(\tau_i) = 0$ , then  $\tau_i$  is an open interval.  $CTime$  is aged prior to its use. Thus any timestamp which would close this open interval at a time which is less than  $CTime - \epsilon$  would have been received prior to this iteration. Thus, if  $BufferEmpty(i)$  is true, then  $\mathcal{L}(CTime, i, A)$ . If the buffer is not empty, then a call to  $GetValidInterval$  was made. This call, by definition, returns the earliest interval succeeding the latest invalid interval. Thus, if  $STime(\tau_i) \leq CTime + \epsilon$  then the returned interval has a timestamp within  $\epsilon$  of  $CTime$ .

The value of variable  $CTime$  only changes when  $noverlaps$  is set to zero. All tests to increment  $noverlaps$  must then use the same value for  $CTime$ .

Thus,  $noverlaps$  is incremented for each process, and  $P_0$  asserts  $A$ , only if there exists a set of global timestamps, one from each process, which differ by at most  $\epsilon$ . Finally, by lemma 19, there exists  $t$  such that  $\mathcal{L}(C_i(t), i, A)$ ,  $i \in SYS$ . ■

**Theorem 20 (Liveness)** *If there exists real time instants  $t_1$  and  $t_2$  such that  $C_i(t_2) - C_i(t_1) \geq 2\epsilon$  and  $\mathcal{L}(C_i(t), i, A_i)$ ,  $t_1 \leq t \leq t_2$ , for all  $i \in SYS$ , then  $P_0$  will assert  $A$ .*

**PROOF:** If there exists real time instants  $t_1$  and  $t_2$  such that  $C_i(t_2) - C_i(t_1) \geq 2\epsilon$  and  $\mathcal{L}(C_i(t), i, A_i)$ ,  $t_1 \leq t \leq t_2$ , for all  $i \in SYS$ , then, by lemma 18, there exists a set of global timestamps, one from each process, which differ by at most  $\epsilon$ . Clearly, there must then be some smallest valued set of such timestamps  $\Sigma_{sat}$ . Let  $T_{detect}$  be the maximum of  $\Sigma_{sat}$ . Each timestamp in  $\Sigma_{sat}$  is part of some global timestamp interval.  $T_{detect}$  must then be the start time of the interval from which it was taken. Otherwise, there would be some



smaller valued set of timestamps which differ by no more than  $\epsilon$ , which would contradict our definition of  $\Sigma_{sat}$ .

By lemma 20,  $CTime$  takes on monotonically increasing values at each iteration of the while loop. Thus,  $CTime$  will eventually equal  $T_{detect}$  unless  $CTime_{j-1} < T_{detect} < CTime_j$  for successive iterations  $j-1$  and  $j$ . However, in lemma 21 we showed that this can never happen. Thus, eventually  $CTime = T_{detect}$ .

Now suppose that for some iteration  $CTime = T_{detect}$ . We consider an interval discarded if it is not the current value for some  $\tau_i$  or is not in a process buffer. By the algorithm, on a given iteration  $j$ , an interval is discarded only if

- it is closed and its end time is less than  $CTime_j - \epsilon$ , or
- if it is open, and the corresponding closed interval is either used as the current value for  $\tau_i$  or has an end time less than  $CTime_j - \epsilon$ .

Since the buffers are FIFO and successive values of  $CTime$  are monotonically increasing, no interval which overlaps  $T_{detect}$  has been discarded during any preceding iteration. Thus, either the closed or open interval values of any interval containing a timestamp in  $\Sigma_{sat}$  will be the current value for  $\tau_i$ , or it will be in a process buffer.

Now consider a single iteration of the for loop. The initial value of any  $\tau_i$  can be either open or closed. Suppose that it is open. If the buffer is empty, then *noverlaps* will be incremented. If the buffer is not empty, then *GetValidInterval* will be called. Since no valid interval has been discarded, clearly *GetValidInterval* will return the overlapping interval values, and *noverlaps* will be incremented.

Suppose then that the initial value of  $\tau_i$  is closed. If it overlaps  $CTime_j$ , then *noverlaps* will be incremented. If it does not overlap  $CTime_j$ , then it must have ended earlier than  $CTime - \epsilon$  and *GetValidInterval* will be called. Again, since no valid interval has been discarded, *GetValidInterval* will return the overlapping interval values, and *noverlaps* will be incremented.

The variable *noverlaps* will then be incremented for each  $P_i$  when  $CTime$  is equal to  $T_{detect}$ . Since  $CTime$  must eventually equal  $T_{detect}$ , the predicate will be detected. ■

# Bibliography

- [1] R.K. Arora, S.P.Rana, and M.N.Gupta. Distributed termination detection algorithm for distributed computations. *Information Processing Letters*, pages 311–314, May 1986.
- [2] A. Avizienis. The N-version approach to fault-tolerant software. *IEEE Transactions on Software Engineering*, SE-11(12):1491–1501, 1985.
- [3] D. J. Badal. The distributed deadlock detection algorithm. *ACM Transactions on Computer Systems*, pages 320–337, November 1986.
- [4] S. Chandrasekaran and S. Venkatesan. A message optimal algorithm for distributed termination detection. *Journal of Parallel and Distributed Computing*, pages 244–252, August 1990.
- [5] K.M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, 1985.
- [6] K.M. Chandy and J. Misra. A distributed algorithm for detecting resource deadlocks in distributed systems. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, pages 157–164, 1982.
- [7] K.M. Chandy, J. Misra, and L. Haas. Distributed deadlock detection. *ACM Transactions on Computer Systems*, 1(2):144–156, 1983.
- [8] C.Hazari and H. Zedan. A distributed algorithm for distributed termination. *Information Processing Letters*, pages 293–297, Mar 1987.
- [9] R. Cooper and K. Marzullo. Consistent detection of global predicates. *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging, ACM SIGPLAN Notices*, 26(12), December 1991.
- [10] Flaviu Cristian. Probabilistic clock synchronization. *Distributed Computing*, 3:146–158, 1989.
- [11] E. Dijkstra, W. Feijen, and A. van Gasteren. Derivation of a termination detection algorithm for distributed computations. *Information Processing Letters*, 16:217–219, 1983.
- [12] E.W. Dijkstra and C.S. Scholten. Termination detection of diffusing computations. *Information Processing Letters*, 11(1):1–4, August 1980.

- [13] D. Dolev, N.A. Lynch, S.S. Pinter, E.W. Stark, and W.E. Weihl. Reaching approximate agreement in the presence of faults. *Journal of the ACM*, 33(3):499–516, 1986.
- [14] D.P. Mitchell and M.J. Merritt. A distributed algorithm for deadlock detection and resolution. In *Proceedings of the ACM Conference on Principles of Distributed Computing*, pages 282–284, August 1984.
- [15] A. Elmagarmid. An effective on-line deadlock detection technique for distributed database management. In *ACM Sigmod Records*, Sep 1986.
- [16] O. Eriksen. A termination detection protocol and its formal verification. *Journal of Parallel and Distributed Computing*, pages 82–91, September 1988.
- [17] A. N. Choudhary et al. A modified priority based probe algorithm for distributed deadlock detection and resolution. *IEEE Transactions on Software Engineering (SE)*, ; *ACM CR 89-8-0570*, 15(1), January 1989.
- [18] N. Francez. Distributed termination. *ACM Transactions on Programming Languages and Systems*, 2(1):42–55, January 1980.
- [19] N. Francez and M. Rodeh. Achieving distributed termination without freezing. *IEEE Transactions on Software Engineering*, SE-8(3):287–292, 1982.
- [20] V.D. Gligor and S.H. Shattuck. On deadlock detection in distributed systems. *IEEE Transactions on Software Engineering*, pages 434–440, September 1980.
- [21] Joseph Halpern, Barbara Simons, Ray Strong, and Danny Dolev. Fault-tolerant clock synchronization. In *Proceedings of Symposium on Principles of Distributed Computing*, pages 89–102. ACM SIGPLAN/SIGOPS, August 1984.
- [22] S. P. Harbison and G. L. Steele. *C, A Reference Manual*. Prentice Hall, Englewood Cliffs, NJ, third edition, 1991.
- [23] H. Kopetz. Sparse time versus dense time in distributed real time systems. In *Proceedings of Twelfth International Conference on Distributed Computing Systems*, pages 460–467, Jun 1992.
- [24] G.S. Ho and C. V. Ramamoorthy. Protocols for deadlock detection in distributed database systems. *IEEE Transactions on Software Engineering*, 8(6):554–557, November 1982.
- [25] S. Huang. Termination detection by using distributed snapshots. *Information Processing Letters*, pages 113–119, Aug 1989.
- [26] S. Isloor and T. Marsland. An effective on-line deadlock detection technique for distributed database management. In *Proc. Compsac 78*, pages 283–288, Nov 1978.
- [27] J.R. Jagannathan and R. Vasudevan. A distributed deadlock detection and resolution scheme; performance study. In *Proceedings of Third International Conference on Distributed Computing Systems*, pages 496–501, 1982.

- [28] J.L.W.Kessels. Two designs of a fault-tolerant clocking system. *IEEE Transactions on Computers*, C-33(10):912-919, October 1984.
- [29] J.Mayo and P.Kearns. Global predicates in rough real time. In *Proceedings of the Seventh IEEE Symposium on Parallel and Distributed Processing*, pages 17-24, 1995.
- [30] Edgar Knapp. Deadlock detection in distributed databases. *ACM Computing Surveys*, 19(4):303-328, December 1987.
- [31] D. Kumar. A class of termination detection algorithms for distributed computations. In N Maheshwari, editor, *5th Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 206 of *Lecture Notes in Computer Science*, pages 73-100. Springer, 1985.
- [32] T.H. Lai. Termination detection for dynamically distributed systems with non-first-in-first-out communication. *J. Parallel and Distributed Computing*, 3:577-599, 1986.
- [33] T.H. Lai. An  $(n-1)$ -resilient algorithm for distributed termination detection. *IEEE Transactions on Parallel and Distributed Systems*, 6(1):63-78, January 1995.
- [34] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558-565, 1978.
- [35] L. Lamport, R. Shostak, and M. Pease. The Byzantine Generals problem. *ACM Transactions on Programming Languages and Systems*, 4:382-401, 1982.
- [36] B. Liskov. Practical uses of synchronized clocks in distributed systems. In *Proceedings of the Tenth Annual ACP Symposium on Principles of Distributed Computing*, pages 1-9, August 1991.
- [37] L.Lamport and P.M.Melliar-Smith. Synchronizing clocks in the presence of failures. *Journal of the ACM*, 32(1):52-78, January 1985.
- [38] J. Lundelius and N. Lynch. A new fault-tolerant algorithm for clock synchronization. In *Proceedings of the Third ACM SIGACT/SIGOPS Symposium on Principles of Distributed Computing*, pages 75-88, August 1984.
- [39] F. Mattern. New algorithms for distributed termination detection in asynchronous message passing systems. Report 42/85, University of Kaiserslautern, 1985.
- [40] F. Mattern. Algorithms for distributed termination detection. *Distributed Computing*, 2:161-175, 1987.
- [41] F. Mattern. Global quiescence detection based on credit distribution and recovery. *Information Processing Letters*, pages 195-200, Feb 1989.
- [42] F. Mattern. Efficient algorithms for distributed snapshots and global virtual time approximation. *Journal of Parallel and Distributed Computing*, pages 423-434, August 1993.

- [43] Friedemann Mattern. Virtual time and global states of distributed systems. In M. Cosnard et. al., editor, *Parallel and Distributed Algorithms: Proceedings of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226. Elsevier Science Publishers B. V., 1989.
- [44] J. Mayo and P. Kearns. Distributed termination detection with imprecise clocks. *Information Processing Letters*, 54:105–108, 1994.
- [45] J. Mayo and P. Kearns. Efficient distributed termination detection with roughly synchronized clocks. In *Proceedings of the Seventh IASTED/ISMM International Conference Parallel and Distributed Computing Systems*, pages 304–307, October 1995.
- [46] D. Menasce and R. Muntz. Locking and deadlock detection in distributed data bases. *IEEE Transactions on Software Engineering (SE)*, Vol.SE-5, (3), May 1979.
- [47] J. Misra. Detecting termination of distributed computations using markers. In *ACM Symposium on Principles of Distributed Computing*, pages 290–294, 1983.
- [48] J. Misra and K. Chandy. Termination detection of diffusing computations in communicating sequential processes. *ACM Transactions on Programming Languages and Systems*, 4(1):37–43, January 1982.
- [49] G. Neiger and S. Toueg. Simulating synchronized clocks and common knowledge in distributed systems. *Journal of the ACM*, 40(3):334–367, 1993.
- [50] D. Nicol. Noncommittal barrier synchronization. *Parallel Computing*, 21:529–549, 1995.
- [51] R. Obermarck. Distributed deadlock detection algorithm. *ACM Transactions on Database Systems*, 7(2):187–208, June 1982.
- [52] S. Peterson. *Causal Synchrony in the Design of Distributed Protocols*. PhD thesis, The College of William and Mary, 1994.
- [53] P Ramanathan, D. D. Kandlur, and K. G. Shin. Hardware-assisted software clock synchronization for homogeneous distributed systems. *IEEE Transactions on Computers*, C-39(4):514–524, April 1990.
- [54] P. Ramanathan and K. Shin. Use of a common time base for checkpointing and rollback recover in a distributed system. *IEEE Transactions on Software Engineering (SE)*, 19(6), June 1993.
- [55] P. Ramanathan, K. G. Shin, and R. W. Butler. Fault-tolerant clock synchronization in distributed systems. *IEEE Computer*, 23(10):33–44, [10] 1990.
- [56] S. Rana. A distributed solution of the distributed termination problem. *Information Processing Letters*, 17:43–46, 1983.
- [57] K. G. Shin and P Ramanathan. Clock synchronization of a large multiprocessor system in the presence of faults. *IEEE Transactions on Computers*, C-36(1):2–12, January 1987.

- [58] M. Singhal. Deadlock detection in distributed systems. *IEEE Computer*, pages 37–48, November 1989.
- [59] M. K. Sinha and N. Natarajan. A priority based distributed deadlock detection algorithm. *IEEE Transactions on Software Engineering (SE)*, 11(1), January 1985.
- [60] M. Spezialetti and J.P. Kearns. Efficient distributed snapshots. In *Proceedings of the Sixth International Conference on Distributed Computing Systems*, pages 382–388, 1986.
- [61] T. K. Srikanth and Sam Toueg. Optimal clock synchronization. *Journal of the ACM*, 34(3):626–645, July 1987.
- [62] T.H.Lai and T.H.Yang. On distributed snapshots. *Information Processing Letters*, pages 153–158, May 1987.
- [63] R. Topor. Termination detection of distributed computations. *Information Processing Letters*, 18:33–36, 1984.
- [64] N. Vasanthavada and P.N. Marinos. Synchronization of fault-tolerant clocks in the presence of malicious failures. *IEEE Transactions on Computers*, 37(4):440–448, April 1988.
- [65] V.Garg and B. Waldecker. Detection of weak unstable predicates in distributed programs (to appear). *IEEE Transactions on Parallel and Distributed Systems*, 1994.
- [66] V.Garg and B. Waldecker. Detection of strong unstable predicates in distributed programs. *IEEE Transactions on Parallel and Distributed Systems*, 7(12):1323–1333, December 1996.
- [67] J. Welch and N. Lynch. A new fault-tolerant algorithm for clock synchronization. *Information and Computation*, 77(1):257–274, April 1988.
- [68] Z. Yang and T. Marsland. *Global States and Time in Distributed Systems*. IEEE Computer Society Press, 1994.

## VITA

Jean Ann Mayo was born in Newport News, Virginia on March 31, 1962. She received the Bachelor of Mechanical Engineering in 1985 from the Georgia Institute of Technology. She was employed by Newport News Shipbuilding, in Newport News, Virginia, from 1985 until 1992. During this time she completed the requirements for the Bachelor of Science, with a concentration in Physics, which she received from Longwood College in 1990. She entered the College of William and Mary in 1992 as a candidate for the M.S. degree with a concentration in Computer Science. She remained with the department until the present, receiving the Master of Science in Computer Science in 1992.