INFORMATION TO USERS

The most advanced technology has been used to photograph and reproduce this manuscript 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.
Distributed termination detection: Algorithms and performance evaluation

Yang, Tao-heng, Ph.D.
The Ohio State University, 1990
DISTRIBUTED TERMINATION DETECTION: ALGORITHMS AND PERFORMANCE EVALUATION DISSERTATION

Presented in Partial Fulfillment of the Requirements for the Degree Doctor of Philosophy in the Graduate School of The Ohio State University

By
Tao-heng Yang, B.S., M.S.

The Ohio State University

1990

Dissertation Committee

Adviser's Approval

Professor Ten-Hwang Lai
Professor Dik L. Lee
Professor Mukesh Singhal

Department of Computer and Information Science
To My Parents
ACKNOWLEDGEMENTS

I would like to express sincere appreciation to my adviser, Dr. Ten-Hwang Lai, for his guidance and help with my research. I also thank Dr. Dik Lee and Dr. Mukesh Singhal for serving as members of my dissertation committee and for their efforts in reviewing the manuscript.

My parents deserve the greatest merits. They have always been very supportive during my graduate study at The Ohio State University. I especially thank my brother, Tao Hsing Yang, who has always encouraged me to move forward.

I am greatly indebted to my supervisors, Dennis Slaggy, Frank Northrup and Gene Crawford. They were very supportive and thoughtful throughout the years I work for them at the Instruction and Research Computer Center.

Last but not least, I want to thank my friends, Chun-Wu Leng, Wen-Shyen Chen and Yow-Wei Yao, who have shared my ups and downs in these years.
VITA

Apr. 12, 1959    Born - Taipei, Taiwan, Republic of China

Sep. 1977 - June 1981  B.S., Department of Computer Science and Information Engineering, National Taiwan University, Taipei, Taiwan, Republic of China

Sep. 1983 - Mar. 1985  M.S., Department of Computer and Information Science, The Ohio State University, Columbus, Ohio

Mar. 1985 - Mar. 1990  Ph.D., Department of Computer and Information Science, The Ohio State University, Columbus, Ohio

PUBLICATIONS


FIELDS OF STUDY

Major Field: Computer and Information Science

Studies in Parallel and Distributed Algorithms:
Dr. Ten-Hwang Lai

Studies in Computer Architecture and Distributed Systems:
Dr. Ming T. Liu

Studies in Statistics:
Dr. Haikady N. Nagaraja
TABLE OF CONTENTS

DEDICATION ................................................................. ii

ACKNOWLEDGEMENTS ................................................ iii

VITA .............................................................................. iv

LIST OF TABLES ............................................................ viii

LIST OF FIGURES .......................................................... ix

CHAPTER PAGE

I INTRODUCTION ........................................................... 1
  1.1 Stable Properties and Distributed Termination ............... 2
  1.2 Survey of Distributed Termination Detectors ............... 4
  1.3 Motivations and Contributions of Research ............... 7
  1.4 Organization of Dissertation ............................... 10

II ON DISTRIBUTED SNAPSHOTs .......................... 13
  2.1 Introduction .................................................... 13
  2.2 Model of A Distributed System ............................... 14
  2.3 Snapshots ......................................................... 16
  2.4 An Efficient Snapshot Algorithm ............................ 19
  2.5 Strongly Stable Properties .................................. 23
  2.6 Remarks on Deadlock and Termination Detection ......... 26

III TERMINATION DETECTION IN A BROADCAST NET- 30
  WORK ................................................................. 30
  3.1 Introduction .................................................... 30
  3.2 Definitions and Assumptions .................................. 33
  3.3 An Approach to Reducing the Problem’s Complexity .... 36
3.4 Foundations of the Algorithms ............................................. 38
3.5 Algorithm Without Piggybacking ...................................... 42
3.6 Algorithm Using Piggybacking .......................................... 43
  3.6.1 Description of the Algorithm .................................. 44
  3.6.2 Correctness and Complexity .................................... 47
3.7 Lower Bounds ............................................................................. 52
3.8 Conclusion ....................................................................................... 57

IV PERFORMANCE EVALUATION OF DISTRIBUTED TERMINATION DETECTION ALGORITHMS .......... 59
  4.1 Introduction ................................................................................. 59
  4.2 Preliminary .................................................................................... 62
    4.2.1 Distributed Termination Detection ...................... 62
    4.2.2 Snapshot-based Algorithm ....................................... 63
    4.2.3 Chandrasekaran-Kannan-Venkatesan Algorithm 67
  4.3 Theoretical Analysis ......................................................... 71
    4.3.1 Random Distributed Systems .................................. 72
    4.3.2 Lifetime of a System ...................................................... 76
    4.3.3 Number of Control Messages .................................. 82
    4.3.4 Number of Underlying Messages ............................ 88
    4.3.5 Example ........................................................................... 88
    4.3.6 Comments ....................................................................... 89
  4.4 Simulation ....................................................................................... 91
    4.4.1 Simulated System ....................................................... 91
    4.4.2 Results .............................................................................. 93
    4.4.3 Comments ........................................................................ 98
  4.5 Experiment .................................................................................... 100
    4.5.1 DIB ..................................................................................... 101
    4.5.2 Experiment Model ....................................................... 102
    4.5.3 Comments ........................................................................ 105
  4.6 Comparison and Analysis ....................................................... 106
  4.7 Conclusion ....................................................................................... 107
  4.8 Proof of Theorem 10 ................................................................. 108

V CONCLUSION ......................................................................................... 114

BIBLIOGRAPHY ....................................................................................... 117
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Result from probabilistic model with $d = 0.2$, $n = 8$</td>
<td>90</td>
</tr>
<tr>
<td>2</td>
<td>Varying the (exponential) mean of inter-message time.</td>
<td>95</td>
</tr>
<tr>
<td>3</td>
<td>Varying the (exponential) mean of initial load.</td>
<td>95</td>
</tr>
<tr>
<td>4</td>
<td>Varying the (exponential) mean of processing overhead.</td>
<td>96</td>
</tr>
<tr>
<td>5</td>
<td>Varying the (exponential) mean of message delay.</td>
<td>96</td>
</tr>
<tr>
<td>6</td>
<td>Varying the (normal) mean of initial load.</td>
<td>97</td>
</tr>
<tr>
<td>7</td>
<td>Varying the (normal) mean of processing overhead.</td>
<td>97</td>
</tr>
<tr>
<td>8</td>
<td>Varying the (normal) mean of message delay.</td>
<td>98</td>
</tr>
<tr>
<td>9</td>
<td>Result from experiment with different problem size.</td>
<td>105</td>
</tr>
</tbody>
</table>
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Computation $f$</td>
<td>19</td>
</tr>
<tr>
<td>2</td>
<td>Computation $f'$</td>
<td>20</td>
</tr>
<tr>
<td>3</td>
<td>A non-feasible global snapshot.</td>
<td>27</td>
</tr>
<tr>
<td>4</td>
<td>Transition graph of $flag$</td>
<td>47</td>
</tr>
<tr>
<td>5</td>
<td>Termination detector for a refined system</td>
<td>48</td>
</tr>
<tr>
<td>6</td>
<td>Actions and states of a process on time axis</td>
<td>73</td>
</tr>
</tbody>
</table>
CHAPTER I
INTRODUCTION

Technology advances and user needs have been two major motivations for distributed systems during the past years. Architecture and implementation of distributed systems have become more and more important and are receiving considerable attention.

The objectives of distributed systems usually include increased performance, extensibility, availability, and resource sharing. This expectation has made distributed systems a hot topic in research.

What really characterizes a "distributed" system logically and physically to make it different from a traditional system? According to G. LeLann [8], a distributed system should have the following characteristics:

1. it includes an arbitrary number of systems and user processes;

2. the architecture is modular, consisting of a possibly varying number of processing elements;
3. communication is achieved via message passing on a shared communication structure, excluding shared memory;

4. some system-wide control is performed, so as to provide for dynamic inter-process cooperation and runtime management;

5. interprocess message transit delays are variable and some non-zero time always exists between the production of an event by a process and the materialization of this production at the destination process premise (different from the observation of the event by the destination process).

1.1 Stable Properties and Distributed Termination

Processes in a distributed system communicate exclusively by sending and receiving messages. A process is only aware of its own state and the messages it sends and receives. It has little information about other processes in the system. A problem arises: how does a process evaluate a function which requires the global state of the system as input while it only has limited information on other processes?

To determine a global system state, a process $p$ must get the help from all other processes which record their own local states and send the recorded local state to $p$. All processes cannot record their local states at precisely the same instant unless they have access to a common clock. So the challenge is to devise an algorithm by which processes record their own states and the states of communication channels so that the set of process and channel states recorded form a global system state.
Without a carefully designed algorithm, the collected information may not be used as an appropriate global system state.

An important class of problems that can be solved with the global-state-detection algorithm is "stability". Let \( y \) be a predicate function defined on the global state of a distributed system \( P \). We say predicate \( y \) is a stable property of system \( P \) if \( y(S) \) implies \( y(S') \) for all global states \( S' \) of \( P \) reachable from global state \( S \) of \( P \). Examples of stable properties are "the system is deadlocked", "all tokens in a token ring have disappeared", and "computation has terminated". In this dissertation, we are particularly interested in the last one – "computation has terminated", which is also known as "distributed termination".

The original problem of distributed termination arises in a concurrent program when it is desirable to know if termination has become a global property of the program [16]. Due to the limited information about other processes at each process, special algorithms are proposed to detect the situation.

One of the usefulness of termination detection can be viewed from the angle of resource management. In a distributed system with limited resources, sometimes it is necessary to relocate resources as soon as the assignment has accomplished by the requester. Without a consensus, a process may release its requested resource too early – the computation is not finished and will ask for the use of the resource later, or too late – it still holds the resource after the computation is done without realizing it.
The following model is usually used to model distributed systems for discussion and will be used throughout the rest of this dissertation: a distributed system consists of a finite number of processes which communicate with one another exclusively by sending and receiving messages.

A distributed termination detector is a distributed system that, when superimposed on another distributed system, called the underlying system, can detect if the underlying system is terminated.

1.2 Survey of Distributed Termination Detectors

In 1980, Francez [16] defined the problem of distributed termination and proposed a solution for it. The algorithm can achieve a joint decision of a group of communication processes to terminate, where each of them is aware only of its own local state. One drawback of this algorithm, which is recovered by Francez himself later [18], is that it “freezes” the underlying computation of processes.

Also in 1980, Dijkstra and Scholten [10] discussed the problem of termination detection for diffusion computation, which is a specific class of computation starting with a node called “the environment” and spreading out to other “internal nodes” of the system.

In 1982, Misra and Chandy [29] showed how Dijkstra-Scholten’s scheme in diffusing computation can be adapted to detect termination of diffusing computation in a system of communicating sequential processes. Synchronous communication
is assumed in the algorithm.

In 1982, Francez and Rodeh [18] proposed a distributed termination detection algorithm which introduced no “freezing” to underlying computation. The basic idea of this algorithm is using control waves to run up and down on an assumedly existing spanning tree among processes. Synchronous communication is also assumed in the algorithm.

Cohen and Lehmann [7] worked on the problem for diffusing computation in a dynamic system and assumed synchronous communication. The main principle of their algorithm is the maintenance of a responsibility tree while the system is executing.

Dijkstra, Feijen and Gasteren [12] presented an algorithm for termination detection on a ring network in 1983. The algorithm is spelled out in rules and the concept is simple and neat.

In 1983, Misra [30] proposed a simple algorithm for termination detection in an arbitrary network using a single marker. This algorithm assumed asynchronous first-in-first-out channels. It requires the network to be preprocessed to determine its maximal strongly connected components and a cycle for each component containing all edges.

In 1984, Topor [40] presented a simple derivation of a general solution to the problem of termination detection by combining the non-freezing algorithm by Francez and Rodeh [18] and the one by Dijkstra et al. [12]
In 1985, Lozinskii [27] presented an algorithm which has certain advantages over the known methods: shorter time delays, better distribution of control messages over the communication network, and suitability for concurrent job processing.

Kumar [22] presented a class of termination detectors in 1985. They do not require communications channels to be first-in-first-out, but are based on a cycle (or cycles) that includes all processes. Markers are sent through the cycle and when it returns to the initiator, it can be decided if the system is terminated.

A termination detector for dynamically distributed systems with non-first-in-first-out communication is proposed by Lai [23] in 1986. The paper also discussed how to cut down an unbounded number of control messages to within a certain bound.

In 1986, Chandy and Misra [6] showed that the lower bound of control message complexity for a termination detector is $\Omega(M)$ where $M$ is the number of messages required to carry out underlying computation.

In 1987, Chandrasekaran, Kannan and Venkatesan [3] proposed an algorithm which meets this lower bound for worst case message complexity. However, it also requires as many messages in the best case.

Also in 1987, Rozoy [33] proposed an algorithm which meets the lower bound but is suitable for diffusing computation only.

In 1987, Mattern [28] presented a few algorithms based on the idea of message counting under a non-first-in-first-out channel model. Although these algorithms
are not optimal in message complexity, they are very easy to understand and implement.

1.3 Motivations and Contributions of Research

Chandy and Lamport [5] proposed a concept of snapshot to detect the stability of a distributed system. It provides a clear view on the relationship among local process states, global system states, and points in a distributed computation. However, their snapshot algorithm is restricted to first-in-first-out channels and needs $O(|C|)$ control messages, where $C$ is the set of channels in the distributed system. We propose an algorithm which is no more confined to first-in-first-out channels and is more efficient as far as message complexity is concerned.

Snapshot algorithms can be used to detect termination detection which is one of the stable properties. However, because termination is a more specialized global property, a tailored termination detection algorithm can be simpler and more efficient than an augmented snapshot algorithm.

Distributed systems can be based on point-to-point communication or broadcasting communication media among processes. Almost all termination detection algorithms we are aware of (Section 1.2) are designed especially for systems with point-to-point communications. Although they can be easily modified to work for systems with broadcast communications, algorithms so obtained tend to be inefficient as they do not capitalize the broadcast power offered by a broadcast
network. We proposed two termination detectors for static systems (cf. dynamic system [23, 24]). Each algorithm is shown to be optimal, measured by the number of overhead messages introduced by the algorithm. More specifically, we show that any termination detection algorithm in the worst case requires $I$ overhead messages, where $I$ is the total number of times for which the process changes state from active to idle; and in the worst case, our algorithms require no more than $I$ overhead messages. An implication from this result is that the termination detection in a broadcast system is not easier than it in a point-to-point system as some people think, as far as message complexity is concerned.

As mentioned, almost all termination detectors proposed in the literature are based on point-to-point communication systems. Performance of such distributed termination detectors are usually measured by the worst-case message complexity (i.e., the total number of messages sent and received, in the worst case, by the processes of a termination detector), although the average message complexity seems to be a more important gauge. To the best of our knowledge, no termination detector has been analyzed for its average message complexity.

Chandy and Misra [6] showed in 1986 that any distributed termination detector requires, in the worst case, $\Omega(M)$ control messages in order to detect the termination of a non-diffusing distributed system, where $M$ is the number of underlying messages used by the underlying system. By that time, almost all existing termination detectors for non-diffusing distributed systems require in the worst case
control messages, where $M$ is as above and $N$ is the number of processes in the underlying system. The gap between the lower and upper bounds on the worst-case message complexity of distributed termination detection was recently closed by S. Chandrasekaran, C. Kannan and S. Venkatesan, who presented in [3] a termination detector that requires only $O(M)$ control messages.

Thus, as far as worst-case complexity is concerned, the CKV algorithm is a clear winner and can hardly be improved. On the other hand, the CKV algorithm has a drawback: it always requires $\Omega(M)$ control messages, even in the best case. By comparison, a snapshot-based algorithm such as the basic algorithm by Lai [18] and the Four-Counter algorithm by Mattern [28] may require an unbounded number of control messages in the worst case, but it uses only $O(N)$ control messages in the best case.

In real-world applications, it is often the average performance of an algorithm, rather than its worst-case performance, that is of concern. In such cases, a question arises: Which termination detector is more efficient on the average – the one requiring $O(N)$ to arbitrarily many control messages or the one always requiring $O(M)$ messages?

In this dissertation, we evaluate the average-case performance of three representative distributed termination detection algorithms: the CKV algorithm, a simple snapshot-based algorithm, and a more complicated snapshot-based algorithm. Their worst-case message complexities are $O(M)$, unbounded, and $O(MN)$, respec-
tively. The results to be reported later on favor, in order, the unbounded snapshot-based algorithm, the "improved" snapshot-based algorithm, and the CKV algorithm.

Our results are obtained by analysis, simulation, and actual implementation of termination detectors. First, we calculated the expected number of control messages required by the unbounded algorithm, as well as that required by the $O(M)$ algorithm, when it runs on a "random" distributed system. The random system is not intended to model a real-world distributed system. It is introduced to represent a general enough distributed system so that we can analyze the average performance of a termination detection algorithm. Next, we conducted simulations, with underlying systems belonging to even more general classes of distributed systems. Finally, we ran a distributed branch-and-bound algorithm for the 0/1 knapsack problem on a network of Sun workstations.

This three-step analysis, from the very theoretical analysis to the hybrid simulation followed by real-life experiment, gives us a complete understanding of the behavior of three different termination detectors.

1.4 Organization of Dissertation

This dissertation is concerned with the design of efficient snapshot algorithms, the design of termination detection algorithms in broadcasting networks, and performance evaluation of three termination detection algorithms.
In Chapter II, we develop an efficient snapshot algorithm that needs no control messages and does not require channels to be first-in-first-out. We also show that several stable properties (e.g., termination, deadlock) can be detected with un-coordinated distributed snapshots. For such properties, our algorithm can be further simplified.

Chapter III studies the problem of deciding whether a process of a distributed system is terminated in a broadcast network environment. We show that, by discriminating the underlying messages between two classes, the complexity of termination detection will most likely be reduced. We then propose two termination detectors, each shown to be message optimal. For a rich class of distributed systems, our algorithms require no overhead messages at all – all control messages can be piggybacked on underlying messages. The algorithms can be used to decide if the entire system is terminated. When so used, the algorithms are also message optimal and more efficient than existing algorithms, which are designed especially for use in a point-to-point network.

Chapter IV evaluates the performance of three termination detectors: a simple snapshot-based algorithm, an “improved” snapshot-based algorithm, and the one proposed by Chandrasekaran et al. Their worst case control message complexities are unbounded, $O(MN)$ and $O(M)$, respectively, where $M$ is the number of underlying messages issued in the system and $N$ is number of processes. We evaluate their performance in three ways. We start with proposing a probability model to
carry out the theoretical calculation, followed by running simulation programs to simulate a distributed system, and finally we solve a 0/1 knapsack problem with a distributed branch-and-bound algorithm to see what really happens in the real world.

Chapter V concludes this dissertation.
2.1 Introduction

Chandy and Lamport [5] proposed an elegant technique, called distributed snapshots, for detecting stability in a distributed system:

1. Every process takes a snapshot by recording its own state as well as the states of all channels incident upon it.

2. The local snapshots are collected and assembled to form a global picture (or global snapshot) of the system, from which it can be decided whether the system has reached a stable state.

To ensure that the scheme works correctly, Chandy and Lamport proposed that processes be somewhat coordinated in taking snapshots so that the resulting global snapshot is “meaningful”. They described a distributed algorithm for taking a meaningful global snapshot. The algorithm relies on channels being first-in-first-out (i.e. messages being delivered in the order sent), and it requires $O(|C|)$ control
messages, where \( C \) is the set of channels in the system.

This chapter presents a message-efficient algorithm for processes to take local snapshots. The algorithm does not require channels to be first-in-first-out, and it may require no control messages at all.

As mentioned, Chandy and Lamport suggested "meaningful" global snapshots for detecting stability in a system. A natural question then arises: Can an "un-coordinated" global snapshot, in which processes take local snapshots without any coordination among them, be useful for stability detection? We answer it in the affirmative by showing that several stable properties (e.g., termination, deadlock) can be detected with un-coordinated snapshots.

This chapter is concerned only with the problem of taking local snapshots. The issue of forming a global snapshot is discussed in [36]. Interesting derivatives of the Chandy-Lamport algorithm can be found in [11, 31].

### 2.2 Model of A Distributed System

We adopt the model of [5] with two modifications: (i) interprocess communications are not necessarily first-in-first-out, and (ii) a computation of a system is defined as a function of time. (Throughout this chapter, whenever time is used, it denotes global time, and this is for purpose of presentation only – the distributed system itself does not have global time.) Below we give a brief description of the model; the reader is referred to [5] for more information.
A distributed system is a strongly-connected directed graph, in which each node represents a process and each arc represents a unidirectional communication channel. (A directed graph is strongly connected if there is a directed path from p to q, for all nodes p and q.) The system has no shared variables; in particular, it has no global clocks. Processes in the system communicate with one another exclusively by messages. Communication is asynchronous; and messages sent along a channel are always delivered correctly, with arbitrary but finite delay, but not necessarily in the order they were sent (so, channels are not necessarily first-in-first-out).

During the operation of a distributed system, each channel is associated with a set of messages, called its state, that may grow and shrink. When a message is sent (received) along a channel, the message is added to (removed from) the set associated with the channel. A process consists of a set of states, an initial state in the set, and a set of events. An event of a process p is a 5-tuple < p, s, s', μ, c >, meaning that process p changes its state from s to s' and sends (receives) message μ along outgoing (incoming) channel c which is incident upon p; μ and c are null symbols if no message is involved in the event. A global state of a distributed system is a set of process and channel states – one state per process/channel. The initial global state is one in which each process state is an initial state and each channel is empty. An event < p, s, s', μ, c > can occur in global state S if and only if (i) the state of p in S is s, and (ii) if c is an incoming channel, then μ is
contained in the state of c in global state S. If event e can occur in global state S, then next(S, e) denotes the global state immediately after the occurrence of e in global state S.

Let E be the union of all events sets in the system. Let f be a function mapping the time-line (0, ∞) to E ∪ {Λ} such that f(t) = Λ for all but a finite number of t, where Λ is a special symbol not in E denoting “no event”. Let t_i, 1 ≤ i ≤ n, be all t such that f(t) ≠ Λ. Assume 0 = t_0 < t_1 < · · · < t_n < t_{n+1} = ∞. We say that f is a computation of the system if and only if event f(t_i) can occur in global state S_{i-1}, where S_0 is the initial global state and S_i = next(S_{i-1}, f(t_i)), 1 ≤ i ≤ n. We occasionally write a computation as {S_i : 0 ≤ i ≤ n}.

Global state S' is reachable from global state S (denoted as S → S') iff there is a computation {S_i : 0 ≤ i ≤ n} such that S = S_j and S' = S_k for some j, k, 0 ≤ j ≤ k ≤ n.

Let y be a predicate function on the set of all global states of a distributed system. The predicate is said to be a stable property of the system iff y(S) implies y(S') whenever S_0 → S → S'.

2.3 Snapshots

Consider a fixed computation f of a distributed system with process set P and channel set C. Let t_0, . . . , t_{n+1} and S_0, . . . , S_n be as before. All definitions below are made relative to computation f.
For $t \in (t_i, t_{i+1})$, $0 \leq i \leq n$, define

$$GS(t) = S_i \text{ (i.e., } S_i \text{ is the global state of the system at time } t),$$

$$state(p, t) = \text{the state of } p \text{ in global state } S_i,$$

$$sent(c, t) = \text{the set of all messages sent along channel } c \text{ by time } t,$$

$$received(c) = \text{the set of all messages received along channel } c \text{ by time } t.$$

A local snapshot of process $p$ taken at time $t$ consists of:

- $state(p, t)$,
- $sent(c, t)$, for every outgoing channel $c$ incident upon $p$,
- $received(c, t)$, for every incoming channel $c$ incident upon $p$.

A distributed snapshot of a system taken at times $\{t_p : p \in P\}$ is a set of local snapshots with $p$'s snapshot taken at time $t_p$; that is, the following set:

$$\{state(p, t_p) : p \in P\} \cup \{sent(c, t_p), received(c, t_q) : c = (p, q) \in C\}$$

A global snapshot of a system taken at times $\{t_p : p \in P\}$ is the following set:

$$\{state(p, t_p) : p \in P\} \cup \{sent(c, t_p) - received(c, t_q) : c = (p, q) \in C\}$$

Note that a global snapshot is an "assembled" distributed snapshot, and is always a global state.
A global snapshot $GSN$ (respectively, distributed snapshot $DSN$) is occasionally written as $GSN(t_p : p \in P)$ (respectively, $DSN(t_p : p \in P)$) to emphasize the timing of local snapshots.

A global snapshot $GSN(t_p : p \in P)$ taken between times $\tau_1$ and $\tau_2$ (i.e., $\tau_1 < t_p < \tau_2 \ \forall p \in P$) is meaningful iff $GSN$ is reachable from global state $GS(\tau_1)$ and global state $GS(\tau_2)$ is reachable from $GSN$. It is feasible iff $\text{received}(c, t_p) \subseteq \text{sent}(c, t_p)$ for every channel $c = (p, q) \in C$.

The following theorem was established in [5]. We give a simpler proof here.

**Theorem 1** [5] *All feasible global snapshots are meaningful.*

**Proof.** Let $GSN(t_p : p \in P)$ be a feasible global snapshot taken between times $\tau_1$ and $\tau_2$, during computation $f$ (Figure 1). Let $\delta = \tau_1 - \tau_2$. Construct computation $f'$ from $f$: $f'$ is the same as $f$ except that every post-snapshotting event in $f$ is now postponed for $\delta$ units of time. That is,

$$
 f'(x) = \begin{cases} 
 f(t) & \text{if } f(t) \text{ is an event at } p \in P \text{ and } t \leq t - t_p, \\
 f(t - \delta) & \text{if } f(t - \delta) \text{ is an event at } p \in P \text{ and } t_p < t - \delta, \\
 \Lambda & \text{otherwise.}
\end{cases}
$$

Figure 2 displays an example of $f'$. Since $GSN$ is feasible, $f'$ is easily seen to be a computation of the system. One may readily check that

$$
 GS(\tau_1, f) = GS(\tau_1, f') \\
 GSN = GS(\tau_2, f') \\
 GS(\tau_2, f) = GS(\tau_2 + \delta, f')
$$
where $GS(\tau, g)$ denotes the global state of the system at time $\tau$ during computation $g$. So, global snapshot $GSN$ is meaningful. \hfill \Box

2.4 An Efficient Snapshot Algorithm

The snapshot algorithm of [5] not only takes a distributed snapshot, but further distributively refines it to a global snapshot. It works basically as below.

1. Every process takes a local snapshot and immediately sends a marker along every outgoing channel; this must be done before or upon receiving the first
Figure 2: Computation $f'$. 

--- global snapshot or global state
\rightarrow message passing

$GS(\tau_1, f') \quad GS(\tau_2, f') \quad GS(\tau_2 + \delta, f')$
2. Upon receiving a marker along channel $c = (p, q) \in C$, process $q$ computes

$$M(c) = \text{sent}(p, t_p) - \text{received}(c, t_q).$$

3. The set $\{\text{state}(p, t_p) : p \in P\} \cup \{M(c) : c \in C\}$ is a meaningful global snapshot.

Note that a marker sent along channel $c = (p, q)$ serves two purposes: (i) to ensure that $t_p$ and $t_q$ are such that $\text{received}(c, t_q) \subseteq \text{sent}(c, t_p)$, and (ii) for process $p$ to implicitly inform process $q$ the value of $\text{sent}(c, t_p)$. (If channel $c$ is FIFO and $q$ receives $p$'s marker at time $t'_q$, then $\text{received}(c, t'_q) = \text{sent}(c, t_q)$.)

We now describe a snapshot algorithm that uses no markers and that does not require channels to be FIFO. The idea is simple: let the distributed snapshot be transformed into a global snapshot at one node (i.e., non-distributively), thereby releasing a marker from the duty of (ii). Then achieve (i) without markers as follows. Consider any channel $c = (p, q)$. If after time $t_p$, process $p$ will never send messages to process $q$, then clearly $\text{received}(c, t_q) \subseteq \text{sent}(c, t_p)$, and no marker is necessary for channel $c$. Otherwise, let the marker "piggyback" on every outgoing post-snapshotting message. These ideas lead to the snapshot algorithm below.

**Snapshot Algorithm.** The processes operate according to the following rules.

R0. Every process is initially white and turns red while taking a local snapshot.
R1. Every message sent by a white (red) process is colored white (red).

R2. Every white process takes a snapshot at its convenience – but no later than a red message is possibly received. (Thus, the arrival of a red message at a white process will invoke the process to take a snapshot before receiving the message.)

After the local snapshots are collected to a node, they are combined to form a global snapshot, which is readily seen, by Theorem 1, to be meaningful.

Our algorithm needs no control messages at all in taking local snapshots and does not require channels to be FIFO. It requires more space, however, in sending local snapshots to other nodes; for they now each contain the complete message history of a node instead of, as in the algorithm of [5], just messages in transit. Fortunately, this drawback can be overcome in important applications such as termination and deadlock detection (see Section 2.6), where the number of messages in send(c, tp) or received(c, tp) rather than the set itself is of concern.

Note that our algorithm heavily relies on every process’s being able to spontaneously take a local snapshot even if no red messages ever arrive – this would not be a problem if the system is weakly fair in the sense that every node eventually executes the snapshot algorithm. (In contrast, [5] requires only one process to have this “spontaneity” nature.) If only one process can spontaneously initiate the algorithm, a signal may be passed along a spanning tree to ask every process
to take a snapshot [18, 40]; in that case, our algorithm requires $O(|P|)$ control messages.

In applications, a system usually has to repeatedly take global snapshots until the stable property in concern occurs and is detected. It is a straightforward exercise to modify the above algorithm so that it can take a series of global snapshots. Besides, one may reduce the size of a local snapshot by resetting sent() and received() to $\emptyset$ after a local snapshot is taken, assuming that the latest global snapshot is still available at the node responsible for forming global snapshots.

### 2.5 Strongly Stable Properties

We now study the question raised in Section 2.1: Is there any stable property that can be detected with un-coordinated distributed snapshots? And more interestingly: Is there any stable property that can be detected by a non-meaningful global snapshot?

A stable property $y$ of a distributed system $D = (P, C)$ is **strongly stable** iff $y(GS(\tau_1))$ implies $y(GSN)$ and $y(GSN)$ implies $y(GS(\tau_2))$ for all global snapshots $GSN$ taken between time $\tau_1$ and $\tau_2$ during all computations of $D$.

A strongly stable property is obviously a stable property. The converse is in general not true. For instance, the property "Dijkstra's self-stabilization system has reached a stable state" [9, 13] is stable, but not strongly stable.

If a stable property in concern is known to be strongly stable, then it can
be detected with un-coordinated snapshots; as a result, our snapshot algorithm becomes tremendously simple:

*Every process takes a snapshot at its convenience.*

Note that a strongly stable property $y$ may or may not be detected by a non-meaningful global state. The latter can happen if, for example, $y(GSN) = \text{false}$ for all non-meaningful $GSNs$.

In what follows we show that the property "local deadlock" as defined in [4] is a strongly stable property that can be detected even with a non-meaningful global snapshot.

Consider a distributed system $D = (P, C)$ in which every process is either *active* or *idle*, and every process is associated with a set of processes called its *dependent set*. The active/idle status of a process together with its dependent set constitutes the *state* of the process. An idle process becomes active upon receiving a message from any process in its dependent set; otherwise, it stays idle without changing its dependent set. An active process is free to send or receive messages, and may become idle at any moment.

The system is said to be *locally deadlocked* in global state $S$ if there is a nonempty set $Q \subseteq P$ such that the following are true in $S$:

1. All processes in $Q$ are idle.
2. The dependent set of every process in $Q$ is a subset of $Q$. 
3. Every channel between processes in $Q$ is empty.

In that case, $Q$ is said to be deadlocked.

**Lemma 1** During a computation, if $Q \subseteq P$ is deadlocked in global snapshot $GSN(t_p : p \in P)$, then no process $q \in Q$ may resume active after time $t_q$.

**Proof.** Assuming that this is not true, let $R = \{r \in Q : r$ ever resumes active after time $t_r\}$, and let $q$ be the process in $R$ that resumes active earliest. The message that makes $q$ active was sent by some process $p \in Q$ before time $t_p$, or else the "earliest" nature of $q$ would be contradicted. But then the channel from $p$ to $q$ is nonempty in $GSN$, contradicting the "deadlock" nature of $Q$. $\square$

**Theorem 2** That a system is locally deadlocked is a strongly stable property and may be detected even by a non-meaningful global snapshot.

**Proof.** Let $GSN(t_p : p \in P)$ be any global snapshot taken between $\tau_1$ and $\tau_2$. Assume the system to be locally deadlocked at time $\tau_1$. By definition, there is a set of processes $Q$ that is deadlocked at time $\tau_1$. It is clear that no processes in $Q$, and no channels between processes of $Q$, may change their states after time $\tau_1$. Hence, the system is locally deadlocked in $GSN$.

Conversely, assume the system to be locally deadlocked in $GSN$. Then there exists a deadlocked subset $Q$ (relative to $GSN$). For contradiction, assume the system is not locally deadlocked at time $\tau_2$. In particular, $Q$ is not deadlocked at
By definition, at least one of the following holds at time $\tau_2$: (i) some process in $Q$ is still active; (ii) the dependent set of some process in $Q$ is not contained in $Q$; and (iii) some messages are still in transit between processes of $Q$. In all cases, there must be a process in $Q$ that resumes active after taking its local snapshot, in contradiction to Lemma 1. So the system must be locally deadlocked at time $\tau_2$. By definition, the property in concern is strongly stable.

We now show that a local deadlock may be detected by a non-meaningful global snapshot. Figure 3 displays a system of four processes and a non-feasible global snapshot $GSN$. It is not hard to actually define the system so that $GSN$ is also non-meaningful. Let $\{p_3, p_4\}$ be deadlocked at time $t$. This fact evidently can be detected by $GSN$. So, the theorem is proved.

Besides local-deadlock, "the entire system is deadlocked" and "the system is terminated [18]" are also strongly stable properties.

2.6 Remarks on Deadlock and Termination Detection

This section justifies our claim that several applications (e.g., deadlock and termination detection) need only message counters (instead of complete message histories) for each channel.

**Theorem 3** Let $GSN(t_p : p \in P)$ be any global snapshot. A nonempty set $Q \subseteq P$ is deadlocked in $GSN$ iff the following are true in $GSN$: (i) every process in $Q$ is idle, (ii) the dependent set of every process in $Q$ is a subset of $Q$, and (iii)
Figure 3: A non-feasible global snapshot.
\[|\text{sent}(c, t_p)| = |\text{received}(c, t_q)| \quad \text{for every channel } c = (p, q) \in Q \times Q.\]

**Proof.** Consider any distributed snapshot \(GSN(t_p : p \in P)\) and nonempty set \(Q \subseteq P\).

("\(\Rightarrow\)") Assume \(Q\) deadlocked in \(GSN\). By definition, (i) and (ii) hold and, for every channel \(c = (p, q) \in Q \times Q\), \(\text{sent}(c, t_p) \subseteq \text{received}(c, t_q)\). If \(\text{sent}(c, t_p) = \text{received}(c, t_q)\) for every \(c = (p, q) \in Q \times Q\), we are done. So assume \(\text{sent}(c, t_p) \neq \text{received}(c, t_q)\) for some \(c = (p, q) \in Q \times Q\). That is, at least one message in \(\text{received}(c, t_q)\) is sent by \(p\) after time \(t_p\), contradicting Lemma 1.

("\(\Leftarrow\)") Now assume that (i), (ii) and (iii) hold in \(GSN\). For contradiction, assume \(Q\) is not deadlocked in \(GSN\). That is, assume \(\text{sent}(c, t_p) - \text{received}(c, t_q) \neq \emptyset\) for some channel \(c = (p, q)\). Since \(|\text{sent}(c, t_p)| = |\text{received}(c, t_q)|\), it follows that \(\text{received}(c, t_q) - \text{sent}(c, t_p) \neq \emptyset\) and hence \(p\) sends at least one message after \(t_p\).

So, the set \(R = \{ r \in Q : \text{r ever resumes active after time } t_r \}\) is not empty. Let \(q'\) be the process in \(R\) that resumes active earliest, and \(p'\) be the sender of the message, say \(\mu\), that makes \(q'\) active. If message \(\mu\) is sent after time \(t_{p'}\), then \(p'\) resumes active earlier than \(q'\). If \(\mu\) is sent before time \(t_{p'}\), then since \(|\text{sent}(c', t_{p'})| = |\text{received}(c', t_{q'})|\), where \(c' = (p', q')\), \(q'\) must receive before \(t_{q'}\) a message that is sent by \(p'\) after time \(t_{p'}\). In both cases the "earliest" nature of \(q'\) is contradicted. \(\Box\)

Thus, in local-deadlock detection, instead of recording the "set" of messages sent (received) along a channel, it suffices to count the "number" of messages sent
(received).

For termination detection, it is even possible to use only one message counter without distinguishing between channels [23].
CHAPTER III
TERMINATION DETECTION IN A BROADCAST NETWORK

3.1 Introduction

One of the most fundamental problems in distributed programming is that of ensuring that a system is able to know whether it has terminated. A distributed system with this capacity is said to be terminable; otherwise, it is nonterminable.

Francez et al. [17] observed that constructing a nonterminable system may be much easier than a terminable one, and thus suggested relieving a programmer from the problem of termination detection by devising a termination detection algorithm (or termination detector) that, when superimposed on a possibly nonterminable distributed system, automatically makes the system terminable.

A number of termination detectors have been proposed for various models of distributed systems, including:

- [7], [10], [29], [38] that work for distributed systems with special properties such as diffusion;
that work for systems employing synchronous communications;

- [5], [30] that work for systems with asynchronous first-in-first-out communications;

- [22], [23], [28] that work for systems with non-first-in-first-out communications. [23] considers static systems as well as dynamic systems.

All of these algorithms were designed especially for systems with point-to-point communications. Although they can be easily modified so as to work for systems with broadcast communications, algorithms so obtained tend to be inefficient as they do not utilize the "broadcast" power offered by a broadcast network. Indeed, these algorithms (except for those that work only for special systems), when applied to a distributed system employing broadcast communication, require $O(MN)$ or even $O(MN^2)$ overhead messages, where $N$ and $M$ are, respectively, the number of processes and the number of messages received in the underlying system.

Since a great number of existing distributed systems (e.g., those running on a local area network) are based on broadcasting communication media such as contention bus, token ring, and token-passing bus [37], it is of practical as well as theoretical interest to devise efficient termination detectors exclusively for such systems.

In this chapter, we develop two such termination detectors for broadcast networks. Each algorithm is shown to be optimal, measured by the number of over-
head messages introduced by the algorithm. More specifically, we show that any termination detection algorithm in worst cases requires $l$ overhead messages, where $l$ is the total number of times for which the processes change state from active to idle; and in the worst cases, our algorithms require no more than $l$ overhead messages.

We also propose a simple strategy that may help reducing the message complexity of distributed termination. The problem of distributed termination detection normally requires each individual process of a distributed system to detect the termination of the system. In most (if not all) applications, however, it is not necessary for a process to know if the system has terminated; instead, it suffices for each process to know whether the process itself is terminated. We show that, by distinguishing the underlying messages between two classes – $\alpha$-messages and $\beta$-messages, the weaker version of termination detection is computationally easier than the original version. The fewer $\alpha$-messages are used, the easier is termination detection (the weaker version). We shall point out what messages may be regarded as $\beta$-messages, and what may not.

The rest of this chapter is organized as follows. Section 3.2 defines the problem of termination detection. Section 3.3 proposes an approach to reducing the complexity of termination detection. Section 3.2 presents basic concepts of the algorithms. Sections 3.5 and 3.6 describe two termination detectors, which are shown to be message optimal in Section 3.7.
3.2 Definitions and Assumptions

A distributed system consists of a collection of processes which communicate by messages. We assume that the processes are at a higher layer (e.g., the application layer) in the ISO network architecture, and the underlying communication medium is a broadcast channel such as a contention tree/bus or a token ring [37]. The communication medium at the physical layer is subject to contention and errors, but messages are delivered between processes (at the application layer) without errors, without duplicates, and in order. The delay time experienced by a message is arbitrary but finite.

A message may have multiple destinations. It is assumed in the physical layer that each time a message is transmitted over the channel it is received correctly by either all or none of its destination nodes; that is, if it is received correctly by one then it is received correctly by all. (This assumption is realistic in a token ring or a contention bus [37].)

Consider message $m_p$ from process $p$ to process $r$ and message $m_q$ from process $q$ to process $r$, where $p \neq q$. We assume that if $m_p$ arrives, without errors, at the physical layer of $r$ earlier than $m_q$, then $m_p$ is delivered to $r$ before $m_q$. (However, messages from the same source process are delivered to the destination process in the order they were sent.)

The following lemma follows directly from the above assumptions.
Lemma 2 Let \( p, q, r \) be processes. Suppose that \( p \) sends messages \( m_1 \) and \( m_2 \) in that order with \( m_1 \) addressed to \( q \) and \( m_2 \) to both \( q \) and \( r \), and suppose \( r \), after receiving \( m_2 \), sends message \( m_3 \) to \( q \). Then \( q \) receives \( m_1 \) before it receives \( m_3 \).

(Note that if \( m_2 \) is addressed to \( r \) only then the lemma is no longer true.)

Proof. Messages \( m_1 \) and \( m_2 \) are delivered to \( q \) in that order. On the other hand, at the physical layer, \( m_2 \) is correctly received by nodes \( q \) and \( r \) at about the same time, and that is before the time \( m_3 \) is transmitted. So, \( m_2 \) and \( m_3 \) are delivered to process \( q \) in that order. □

Each message is associated with an atomic action, which is executed without interruption immediately after the message is received.

During the operation of a distributed system, each process is either active or idle, with all processes initially being active. An active process is free at any moment to send a message, to receive (and execute) an arriving message, or to change state to idle. All these actions are atomic and cannot be interrupted. Thus, when a process is executing a message, unless required by the message, the process cannot, for example, send or receive another message. An idle process is free at any moment to receive an arriving message, but it cannot send a message. An idle process becomes active on receiving a message; and after the actions of all received messages have been executed, the process is free to become idle.

A distributed system is said to be terminated iff all processes are idle and no messages are still in transit in the system.
Throughout this chapter, we assume that every distributed system eventually becomes terminated. Under this assumption, a process of a distributed system is said to be terminated iff it is at present idle and will stay idle for the rest of the system's lifetime. That is, a process terminates when it becomes idle for the last time. Note that a process may terminates before the system does. Also note that a system is terminated iff every individual process in the system is terminated.

The problem of system termination detection is to devise an algorithm that, once the system terminates, is able to detect the occurrence of termination within a finite amount of time. The problem of process termination detection is to devise an algorithm that, once an individual process terminates, can detect the process's termination within a finite amount of time.

Thus, given a distributed system $P$, we are interested in devising another distributed system $\mathcal{P} = \{p : p \in P\}$ so that $p$ can determine if process $p$ (or system $P$) is terminated.

Conceptually, $p$ and $\bar{p}$ are two distinct processes running concurrently at the same node and communicating with each other through shared variables. In practice, it is usually more efficient to implement $p$ and $\bar{p}$ together as a single process, with messages of $p$ and messages of $\bar{p}$ sharing the same incoming (outgoing) queue.

Messages exchanged between processes of $P$ are called underlying messages. Messages exchanged between processes of $\mathcal{P}$ are called control messages. In our algorithms, a control message may piggyback on an underlying message (i.e., be
attached to an underlying message and broadcast as a part of the latter). A control message that does not piggyback on an underlying message is said to be an *overhead message*.

We assume that the overhead of piggybacking a message is negligible if the piggybacked message is short. In this case, the performance of a termination detector can be measured by the total number of overhead messages used by the detector.

### 3.3 An Approach to Reducing the Problem's Complexity

As mentioned in the introduction, many applications require only process (rather than system) termination detection. For instance, if the purpose of termination detection is for release of resources held by a process, then process termination detection is sufficient.

The problem of process termination detection is in general not easier than the problem of system termination detection, for a terminated process has no knowledge of its own termination until the time it knows the system is terminated. However, as to be seen in what follows, if users of a termination detector can provide such information as whether the execution of a received message requires sending of another message, process termination detection may require fewer control messages than system termination detection.

We first define a *refined* distributed system. (For discrimination, a distributed
In a refined distributed system, each process is either active, passive, or idle, and the processes communicate via two types of messages: α-messages and β-messages.

An active process is free to send a message, to receive an arriving message, or to become idle; but it cannot change state to passive. A passive or idle process is free to receive an arriving message, but neither can send a message. On receiving an α-message, a passive or idle process becomes active, and after the actions of all received messages have been executed the process is free to become idle. On receiving a β-message, an idle process becomes passive. A passive process is free to become idle after all received messages have been executed. Take a distributed branch-and-bound algorithm as an example. A message that carries an assignment to another process is an α-message since the receiver will turn active after receiving it and may send messages to other processes. A broadcast message carries a newly calculated value of cost function to all other processes is a β-message since it will not activate the receiver and is serving for informative purpose.

Like primitive systems, a refined distributed system is terminated iff every process is idle and no messages are in transit; and a process is terminated iff it is idle and will stay idle thereafter.

Let $P$ be a primitive distributed system. Let $B$ be the set of all messages $msg$ sent in $P$ such that no message needs to be sent in execution of $msg$. Suppose that a subset $B' \subseteq B$ is given together with $P$. Then $P$ can be transformed (using
a simple translator) into a refined system $P'$ as below:

1. Call messages in $B'$ $\beta$-messages and those not in $B'$ $\alpha$-messages.

2. For each process $p \in P$, introduce a new variable $state'(p)$. Initially, set $state'(p)$ to active.

3. Whenever $p$ becomes idle, set $state'(p)$ to idle.

4. Whenever $p$ receives an $\alpha$-message, set $state'(p)$ to active.

5. Whenever $p$ becomes active on receipt of a $\beta$-message, set $state'(p)$ to passive.

Let $P'$ be obtained as above, with $state'(p)$ regarded as the state of $p$ for every process $p$. Then $P'$ is a refined distributed system, which has the same function as $P$. Moreover, $P'$ is terminated iff $P$ is terminated, and a process is terminated in $P'$ iff it is terminated in $P$.

Later, we shall develop termination detectors that each require no more than $l$ overhead messages, where $l$ is the total number of times the processes change state from active to idle. By refining a primitive system as above, the $l$-value of the system will most likely be reduced and, thus, so is the number of overhead messages required for termination detection.

### 3.4 Foundations of the Algorithms

In the rest of this chapter, we consider only process termination detection in a refined system, with the understanding that a primitive system is simply a re-
fined system with no $\beta$ messages used, and that process termination detection is equivalent to system termination detection when no $\beta$ messages are used.

Let $P$ be a (refined) distributed system that starts running at time 0. (We assume that the network has no global clock. However, for purpose of presentation, we assume there is a global clock external to the network, and all "times" refer to this external clock.) For any two processes $p, q$ and time $t \geq 0$, define

$$sent(p, q, t) = \text{number of } \alpha\text{-messages sent from } p \text{ to } q \text{ before time } t;$$

$$received(p, q, t) = \text{number of } \alpha\text{-messages received by } p \text{ from } q \text{ before time } t;$$

$$sent(p, t) = \text{number of } \alpha\text{-messages sent by } p \text{ before time } t, \text{ with an } x\text{-destination message counted } x \text{ times;}$$

$$received(p, t) = \text{number of } \alpha\text{-messages received by } p \text{ before time } t;$$

$$deficit(p, t) = sent(p, t) - received(p, t).$$

A distributed system is said to be nearly terminated iff all processes are either idle or passive and no $\alpha$-messages are in transit in the system.

The theorem below shows that in order to detect if a process is terminated, it is first necessary to detect if the system is nearly terminated.

**Theorem 4** A terminated process may recognize itself as terminated only if the system is nearly terminated. Conversely, if the system is nearly terminated and no
outstanding $\beta$-messages are addressed to a currently idle process, then the process knows itself to be terminated.

**Proof.** If the system is not nearly terminated, then some processes are still active and/or some $\alpha$-messages are still in transit. In both cases, $p$ does not know if there will be no underlying messages arriving, and hence $p$ does not know whether it is already terminated. This proves the first part.

Now suppose the system is nearly terminated. Clearly, no process can send underlying messages any longer. Thus, if $p$ is idle and there are currently no pending $\beta$-messages addressed to it, then $p$ knows itself to be terminated. □

The next two theorems show sufficient conditions for recognizing a nearly terminated system; they form the basis of our termination detectors.

**Theorem 5** Let $P$ be a distributed system. Let $t^*$ and $t_p, p \in P$, be time instants such that $t_p < t^*$ for all $p \in P$. $P$ is nearly terminated at time $t^*$ if the following hold.

1. No process $p \in P$ is active at time $t_p$.
2. $\sum_{p \in P} \text{sent}(p, t_p) = \sum_{p \in P} \text{received}(p, t_p)$.
3. There exist no processes $p, q \in P$ such that $p$ sends an $\alpha$-message after $t_p$ and $q$ receives that message before $t_q$.

**Proof.** Assuming that conditions (1) and (2) are satisfied but $P$ is not nearly terminated at time $t^*$, we show (3) false.
Since $P$ is not nearly terminated at time $t^*$, there must be a process $q$ that resumes active after time $t_q$ on receiving an $\alpha$-message $msg$. Without loss of generality, assume that there is no similar event (i.e. a process $r$ becoming active after $t_r$) occurring earlier. Let $p$ be the sending process of $msg$. Then $p$ sends $msg$ before $t_p$; otherwise the "earliest" nature of $q$ becoming active after $t_q$ would be contradicted. Now, message $msg$ contributes to $\sum_{r \in P} sent(r, t_r)$ more than it does to $\sum_{r \in P} received(r, t_r)$. So there must be an $\alpha$-message that contributes more to $\sum_{r \in P} received(r, t_r)$ than $\sum_{r \in P} sent(r, t_r)$, which means (3) is false. □

Theorem 6 Let $P$ be a distributed system. Let $t^*$ and $t_p, p \in P$, be time instants such that $t_p \leq t^*$ for all $p \in P$. $P$ is nearly terminated at time $t^*$ if the following are satisfied.

(1) No process $p \in P$ is active at time $t_p$.

(2) $sent(p, q, t_p) = received(q, p, t_q)$ for all $p, q \in P$.

Proof. Assume (1) and (2) are true. Assume the system is not nearly terminated at time $t^*$. Then, as in the proof of Theorem 5, there is an "earliest" process $q$ receiving after time $t_q$ an $\alpha$-message that was sent by some process $p$ before time $t_p$. The message is counted in $sent(p, q, t_p)$ but not in $received(q, p, t_q)$. Since $sent(p, q, t_p) = received(q, p, t_q)$, there must be an $\alpha$-message from $p$ to $q$ that is sent after $t_p$ and received before $t_q$. That implies that $p$ resumes active earlier than $q$, in contradiction to the "earliest" nature of $q$. So, the system must be nearly
terminated at time $t^*$.

The following corollary is obvious.

**Corollary 1** If the conditions of Theorem 5 (6) hold, then no process $p$ may send any underlying message after time $t_p$.

### 3.5 Algorithm Without Piggybacking

Let $P$ be a distributed system. From Theorems 4 and 6, we immediately obtain the following algorithm for process termination detection.

**Algorithm.** Every process $p \in P$ keeps a list \{\(S_p(q,r), R_p(q,r) : q, r \in P\)\} and does the following:

- **R0.** Initially, \(S_p(q,r) := 0, R_p(q,r) := 0\) for all $q, r \in P$.

- **R1.** Whenever $p$ changes state from active to idle, say at time $t_p$, let $p$ broadcast (to all other processes) a control message carrying the set \(\{sent(p,q,t_p), received(p,q,t_p) : q \in P\}\).

- **R2.** Whenever $p$ receives from $q$ the control message containing \(\{sent(q,r,t_q), received(q,r,t_q) : q \in P\}\), let $S_p(q,r) := sent(q,r,t_q)$ and $R_p(q,r) := received(q,r,t_q)$ for all $r \in P$.

- **R3.** Whenever $p$ becomes idle, let $S(p,q) := sent(p,q,t_p)$ and $R(p,q) := received(p,q,t_p)$ for all $q \in P$. 
R4. Every time R2 or R3 is executed, check if \( p \) is idle and \( S_p(q, r) = R_p(q, r) \) for all \( q, r \in P, q \neq r \). If yes, then \( p \) claims itself to be terminated.

To show the above algorithm correct, let \( t^*_p \) for every \( p \in P \) denote the time process \( p \) claims itself to be terminated. We show \( p \) to be terminated at \( t^*_p \).

Note that process \( p \) has received at least one token from every other process by time \( t^*_p \). For \( q \neq p \), let \( msg_q \) be \( q \)'s latest control message received by \( p \) before time \( t^*_p \), and let \( t_q \) be the time \( msg_q \) was sent. Let \( t_p = t^*_p \). Clearly, the two conditions of Theorem 6 are satisfied relative to these \( t_p \)'s; therefore the system is nearly terminated at time \( t^*_p \). By Corollary 1, no process \( q \) may send any underlying message after time \( t_q \). So, \( p \) receives \( q \)'s underlying messages all before it receives \( msg_q \). By Theorem 4, \( p \) is terminated at \( t^*_p \).

On the other hand, one may readily check that once the system nearly terminates, every process will claim itself to be terminated within finite time. This proves the algorithm correct.

Clearly, the total number of control messages used in the algorithm is no more than \( l \), the number of times processes change state from active to idle.

### 3.6 Algorithm Using Piggybacking

We say that a control message \( c \) piggybacks on an underlying message \( u \) if \( c \) and \( u \) are merged to a single message with destinations being \( c \)'s and \( u \)'s all together. We assume that \( c \) and \( u \) (including their own destinations) can later be extracted from
the merged message by every receiving process. When $c$ is small (say, it contains no data or only one or two bytes of data), the cost of sending $c + u$ is normally far less than the cost of sending $c$ and $u$ separately. Thus, by piggybacking control messages on underlying messages, a termination detector may save lots of overhead messages.

The technique of piggybacking is not appealing for the algorithm in the preceding section, for each control message (except the one sent by $p_0$) carries $\Omega(n)$ pieces of data, where $n$ is the number of processes in the system. This section presents an algorithm that uses only short control messages, which makes piggybacking feasible.

3.6.1 Description of the Algorithm

This algorithm is based on Theorem 5. It is symmetric in the sense that every process executes the same algorithm. Thus, it suffices to describe the algorithm for an individual process, say, $p$. (Using notation of Section 3.2, the algorithm to be described is just $p$.) We assume the underlying system $P$ has $n$ processes which are numbered 1 to $n$; thus, $p$ is an integer satisfying $1 \leq p \leq n$. This assumption enables us to reduce the average-case complexity of the algorithm.

In the algorithm, processes communicate with two kinds of control messages: signals and tokens. A signal does not carry any data, while each token carries an integer value.
Each process sends a sequence of tokens and signals such that between any two consecutive tokens there is exactly one signal (i.e., tokens alternate with signals). A signal/token is addressed to all processes in the system (except the sender itself); thus, a group name will be sufficient for addressing. A signal and a token (in that order) are occasionally combined into a single message; when such a message is received, the signal is processed prior to the token.

Process $p$ may send a token only when it is idle or passive, or when it is about to become idle (i.e., it becomes idle immediately after the token is sent). Each token carries the following value

$$d(p) = \text{deficit}(p, t) - \text{deficit}(p, t')$$

where $t$ and $t'$ are respectively the times at which the current token and the previous token are sent ($t' = 0$ if there is no previous token).

After process $p$ has received at least one token from every other process, adding together $p$'s current deficit and all the data received by $p$ yields

$$D(p) = \sum_{q \in P} \text{deficit}(q, t_q)$$

where $t_q$, for $q \neq p$, is the time the last received token from $q$ was sent, and $t_p$ is the last time $p$ became idle. Clearly, if $p$ is idle and $D(p) = 0$ then both (1) and (2) of Theorem 5 are satisfied. Thus, in order to know if the system is nearly terminated, we only need a mechanism for deciding if condition (3) of Theorem 5 holds.
For this purpose, whenever process \( p \) resumes active after it sends a token, let \( p \) send a signal to inform all other processes of its becoming active. To save communication cost, the signal is not sent immediately after \( p \) resumes active, but instead it is held until \( p \) has an underlying message or a token to send, then the signal takes a free ride by piggybacking on the underlying message or token.

To decide the timing of sending tokens, process \( p \) maintains a variable, \( N_{tokens}(p) \), which is initially set to zero and later incremented (decremented) by one whenever \( p \) receives a token (signal).

If process \( p \) has an underlying message to send immediately before it changes state from active to idle (such a message is called a pre-idle message), then it piggybacks a token on the underlying message. Otherwise, \( p \) sends a token only when it is idle/passive and \( N_{tokens}(p) \geq p - 1 \). The ideas behind this scheme are explained below.

A token sent by process \( q \) and received by \( p \) is said to be a latest token (from \( q \)) if \( p \) does not receive any signal from \( q \) following that token. At any moment \( N_{tokens}(p) \) indicates the number of latest tokens that \( p \) has received. Process \( p \) may possibly recognize the system as nearly terminated only if \( N_{tokens}(p) = n - 1 \). (If \( N_{tokens}(p) < n - 1 \) then it is not known whether all processes are idle/passive.) When \( N_{tokens}(p) \) is small, say \( N_{tokens}(p) < p - 1 \), process \( p \) presumes that the system is still operating and there is no hurry for \( p \) to send a token (to report its status). Indeed, we let it wait until \( N_{tokens}(p) \geq p - 1 \). While waiting, \( p \) may
resume active, in which case $p$ does not need to send any token until $p$ becomes idle again. In this way, lots of tokens might be saved.

The above concepts are spelled out in Fig. 5 as a set of "rules", where each rule can be implemented as a procedure which is called when the corresponding event occurs. In this sense, the algorithm is event-driven. Note in the algorithm that process $p$'s timing of sending signals and tokens is controlled by $flag$ as well as $Ntokens$, where $flag$ has the following meanings:

$$
flag = \begin{cases} 
0 & \text{initially} \\
1 & \text{has sent a token} \\
2 & \text{needs to send a signal} \\
3 & \text{needs to send a token} 
\end{cases}
$$

Figure 4 illustrates how $flag$ changes value.

**3.6.2 Correctness and Complexity**

We now establish the correctness of the algorithm by showing (a) that a process may announce itself to be terminated only if the process is really terminated, and
Each process $p$, $1 \leq p \leq n$, keeps four variables $d$, $D$, $flag$, $Ntokens$ (all initialized to 0) and operates according to the following rules.

**R1.** Whenever process $p$ changes state to active, do:
\[ \text{if } flag = 1 \text{ then } flag := 2. \]

**R2.** Whenever $p$ sends an underlying message with $x$ destinations, do:
\[ \text{if it is an } \alpha \text{-message then } d := d + x; \]
\[ \text{if } flag = 2 \text{ then} \]
\[ \text{piggyback a signal on the message, and set } flag := 3; \]
\[ \text{if the message is pre-idle then} \]
\[ \text{piggyback a token (carrying } d \text{) on the message, and set } \]
\[ D := D + d, \quad d := 0, \quad flag := 1. \]

**R3.** Whenever $p$ receives an (underlying or control) message, do:
\[ \text{if it is (or contains) an } \alpha \text{-message having } p \text{ as a destination then} \]
\[ d := d - 1; \]
\[ \text{if the message contains a signal then} \]
\[ Ntokens = Ntokens - 1; \]
\[ \text{if the message contains a token then} \]
\[ D := D + d \text{-value carried by the token} \]
\[ Ntokens = Ntokens + 1 \]
\[ \text{if } p \text{ is idle then execute } R5 \text{ below} \]
\[ \text{endif.} \]

**R4.** Whenever $p$ changes state to idle, execute $R5$ below.

**R5.** if $flag \neq 1$ and $Ntokens \geq p - 1$ then
\[ \text{if } flag = 2 \text{ then} \]
\[ \text{send a control message containing a signal and a token carrying } d \]
\[ \text{else /* } flag = 0 \text{ or } flag = 3 */ \]
\[ \text{send a control message containing a token carrying } d \]
\[ \text{endif} \]
\[ D := D + d; \quad d := 0; \quad flag := 1 \]
\[ \text{endif;} \]
\[ \text{if } D = 0 \text{ and } Ntokens = n - 1 \text{ then} \]
\[ \text{"Announce process } p \text{ terminated."} \]

---

**Figure 5:** Termination detector for a refined system.
(b) that once the system nearly terminates every process will claim itself to be terminated in finite time.

**Proposition 1. If a process claims itself to be terminated, it is really terminated.**

**Proof.** Let \( p \) be any arbitrary but fixed process, and assume \( p \) claims itself terminated at time \( t^* \). We show that, at time \( t^* \), the system is nearly terminated and there are no pending messages addressed to \( p \). The proposition then follows from Theorem 4.

It is not hard to check that each process really sends its tokens and signals alternately (just as we intend it to), and thus variable \( N_{\text{tokens}} \) has exactly the meaning as explained in Section 3.6.1.

At time \( t^* \), \( p \) is idle, \( D = 0 \), and \( N_{\text{tokens}} = n - 1 \) (according to R5 of the algorithm). Since \( N_{\text{tokens}} = n - 1 \), process \( p \) has received from every other process a "latest" token, following which no signal (from that same process) has been received. For every \( q \neq p \), let \( t_q \) be the time the "latest" token from \( q \) was sent; and let \( t_p \) be the last time process \( p \) became idle. Then, every process \( r \in P \) is idle at time \( t_r \). Since messages are delivered in a first-in-first-out manner, all of \( q \)'s tokens sent before time \( t_q \) have been received by \( p \) by time \( t^* \), and thus \( D = \sum_{r \in P} \text{deficit}(r, t_r) \) at time \( t^* \). So, conditions (1) and (2) of Theorem 5 are satisfied.

Now assume condition (3) of that same theorem does not hold. Let \( q, r \) be processes such that \( q \) sent \( r \) an \( \alpha \)-message \( msg \) after time \( t_q \), and \( r \) received that
message before time $t_r$. (Notice that $q \neq p$, for $p$ has been idle since time $t_p$.) Since $q$ was idle at time $t_q$, $q$ must have resumed active (and set its flag to 2) before it could send message $msg$ to $r$. According to R2 of the algorithm, either a signal (of $q$) had been sent after time $t_q$, or a signal would piggyback on message $msg$. This signal arrived at process $r$ before time $t_r$, i.e., before $r$ sent its “latest” token. Therefore, the signal arrived at process $p$ before $r$’s “latest” token did, and that was before time $t^*$. This contradicts the assumption that $q$ sent its “latest” token at time $t_q$. Thus, condition (3) also holds. By Theorem 5, the system is nearly terminated at time $t^*$.

By Corollary 1, no process $q \neq p$ may send any underlying message after time $t_q$ (i.e., after it sent its “latest” token). Since messages from $q$ to $p$ are delivered in the first-in-first-out manner, by the time $p$ received $q$’s “latest” token, all underlying messages from $q$ to $p$ have been received. Thus, no outstanding messages are addressed to $p$ at time $t^*$. By Theorem 4, process $p$ is terminated at $t^*$. □

**Proposition 2.** Once the system becomes nearly terminated, every process will recognise itself as terminated within a finite amount of time.

**Proof.** Assume that the system is nearly terminated. Within a finite amount of time, the system will be terminated. So, let’s assume that the system is already terminated.

It is not hard to show, by induction on $p$, that every process $p$ sent a token
(its last one) after or immediately before it changed state from active to idle for the last time. After all of these tokens are received, the conditions “$D = 0$ and $N_{tokens} = n - 1$” will hold at every process.

Consider any process $p$. When $N_{tokens}(p)$ reaches $n - 1$ (i.e., when $p$ receives the last token), if $p$ is idle, it will announce itself terminated by calling $R5$ from $R3$; or if $p$ is not idle at that time, then when $p$ eventually changes state to idle, it will call $R5$ from $R4$ and recognize itself as terminated. □

As for message complexity, one readily sees that the algorithm requires at most $l$ tokens and $l - n$ signals, where $n$ is the number of processes in the system and $l$ is the total number of times the processes change state from active to idle. Since signals always piggyback on either underlying messages or tokens, only tokens may cause overhead messages. Thus the algorithm needs at most $l$ overhead messages.

Because of the strategy that $p$ does not send a token until $N_{tokens}(p) \geq p - 1$, the number of overhead messages is normally expected to be less than $l$.

It is not unusual in distributed computing that a process needs to send a pre-idle message every time it is about to change state from active to idle (e.g., to inform other processes of its new computational results). For such a distributed system, our algorithm requires no overhead message at all.
3.7 Lower Bounds

Each of the foregoing algorithms requires no more than $l$ overhead messages, where $l$ is the number of times the processes change state from active to idle. We will show in this section that any process termination detector requires $l$ overhead messages in the worst case. Thus, our algorithms are optimal, measured by the number of overhead messages required in the worst case. We will also show that the problem of system termination detection is intrinsically harder than the problem of process termination detection in terms of overhead messages required.

Lemma 3. Let $P$ be a system of $n$ processes. Suppose at time $t^\ast$ no messages are in transit and all the $n$ processes change state from active to idle simultaneously. If there is a process, say $p$, that does not send any overhead message after time $t^\ast$, then no process other than $p$ can recognize itself as terminated.

Proof. Note that every process terminates at time $t^\ast$. Also note that after time $t^\ast$ no underlying messages are available for piggybacking and hence all control messages must be overhead messages.

If process $p$ does not send any control (overhead) message after time $t^\ast$, then process $q$ ($q \neq p$) has no knowledge of $p$'s status. In particular, $q$ does not know whether $p$ is idle and whether $p$ will no longer send underlying messages. Hence, $q$ cannot recognize itself as terminated. □
It follows from Lemma 3 that in order for every process to recognize its own termination, every process must send an overhead message after time $t^*$. The following theorem shows that every process-termination detector requires at least $l$ overhead messages in the worst case.

**Theorem 7** Let $n, l$ be integers such that $1 < n \leq l$. For any process-termination detector $A$, there exists a system of $n$ processes with the following properties:

1. The processes change state from active to idle for a total number of $l$ times.
2. In order for every process to recognize itself as terminated, the detector needs at least $l$ overhead messages.

**Proof.** Given $A, n, l$ as above, we construct a system of $n$ processes with the required properties.

Let $P$ consist of $n$ processes and start running at time $t_0$. During time interval $[t_0, t_1)$, where $t_0 < t_1$, let every process be active and send no underlying messages.

For $i = 1, 2, \ldots, l - n$, define, interval by interval, the computation of the system for time interval $[t_i, t_{i+1})$ as below:

1. Let $p_i \in P$ become idle at $t_i$, where $p_i$ is selected as below:

   (a) Imagine there is an adversary that makes a duplicate $P' + A'$ of the combined system $P + A$ (i.e., system $P$ with detector $A$ superimposed on it) such that $P' + A'$ is identical to $P + A$ over time interval $[t_0, t_i)$. 


(b) Let every process in $P'$ change state from active to idle at time $t_i$.

(c) Suppose $p'_j$ is the process in $P'$ that is first to send an overhead message after $t_i$ (by Lemma 3, there is always such a process, or $A'$ would not be a correct process-termination detector). Then let $p_i \in P$ be the process corresponding to $p'_j$.

2. After an overhead message has been sent in $P$ (in $A$, strictly speaking), let any arbitrary active process in $P$ send an $\alpha$-message to $p_i$. (After time $t_i$ at least one overhead message will be sent for following reasons. Processes $p_i$ and $p'_i$ (including their superimposed detectors) are identical up to time $t_i$; and if $p_i$ receives no overhead messages after time $t_i$, $p_i$ and $p'_i$ will remain identical until $p'_i$ sends an overhead message; at that time, $p_i$ will also send an overhead message.)

3. Let $t_{i+1}$ be any time instant after $p_i$ has received the $\alpha$-message and resumed active.

Then let all processes in $P$ become idle at time $t_{i-n+1}$.

Let $P$ be constructed as above. During each time period $[t_i, t_{i+1})$, $1 \leq i \leq n-1$, exactly one process becomes idle and at least one overhead message is sent. Furthermore, all the $n$ processes become idle at time $t_{i-n+1}$ and, by Lemma 3, at least $n$ overhead message will be sent after time $t_{i-n+1}$. Thus, the processes
change state from active to idle for a total number of \( l \) times, and \( A \) uses at least \( l \) overhead messages. The theorem is proved.

\[ \square \]

**Theorem 8** Let \( n, l, m \) be integers such that \( 1 < n \leq l \). For any system-termination detector \( A \), there exists a system of \( n \) processes with the following properties:

1. The processes change state from active to idle for a total number of \( l \) times.

2. The processes send a total number of \( m \) \( \beta \)-messages.

3. Algorithm \( A \) requires at least \( l + m \) overhead messages in order for every process to detect the system's termination.

**Proof.** Given \( A, n, l, m \) as above, let \( P \) be constructed as in the proof of Theorem 7 for the time from \( t_0 \) to (excluding) \( t_{i-n} \). Let \( p, q \) be any two distinct processes in \( P \).

For \( l - n \leq i \leq l - n + m - 1 \), define the computation of \( P \) for period \([t_i, t_{i+1})\) as follows. Let \( p \) send a \( \beta \)-message \( msg \) to \( q \) at time \( t_i \). We claim that at least one process will send a control message after time \( t_i \). Assuming the claim, let time \( t_{i+1} > t_i \) be such that message \( msg \) has been received by \( t_{i+1} \), at least one control message has been sent during \((t_i, t_{i+1})\), and no control messages are in transit at time \( t_{i+1} \).
After $P$ is defined for $[t_{n+m-1}, t_{n+m})$, let all processes in $P$ become idle at time $t_{n+m}$. It is not hard to check that system $P$ has the properties required.

It remains to prove the above claim that some process must send a control message after time $t_i$. Assume that no process sends a control message after time $t_i$, or we are done. Imagine an adversary who makes a copy $P' + A'$ of $P + A$ such that the two are identical up to time $t_i$. Let all processes be idle immediately after $p'$ sends $q'$ a $\beta$-message $msg'$ at time $t_i$, where $p'$ and $q'$ are duplicate of $p$ and $q$, respectively. Besides, let message $msg'$ arrive at $q'$ exactly when $msg$ arrives at $q$.

Now, since the message could be delayed for an arbitrary amount of time, no process is able to know whether $q'$ has received the message (and, hence, whether the system has terminated) if no process ever hears from $q'$ after time $t_i$. And since $A'$ is a termination detector by which every process eventually detects the system's termination, $q'$ must be heard some time after $t_i$ (i.e., $q'$ must send a control message, say $cmsg$, sometime after $t_i$). Up to the time message $cmsg$ is sent, $q'$ and $q$ are identical, and so are their associated termination detectors. Therefore, $q$ must send a control message when $q'$ sends $cmsg$. This proves the claim and, hence, the theorem.

\[\square\]

The problem of termination detection is usually studied under the assumption that no global clock is available in the network. While this assumption is reasonable in a point-to-point network, it might be too restricted in a broadcast network, especially a local-area network, where clock synchronization is not a big problem.
A natural question then arises: With a global clock available, will termination detection become easier? The answer is no. To see this, just observe that no assumption is required about the existence or non-existence of a global clock in the network. Thus, Theorems 7 and 8 still hold even if the network has a global clock.

### 3.8 Conclusion

This chapter studied the problem of termination detection in the environment of a broadcast network. We studied the possibility of reducing the complexity of termination detection, and noticed that, in many applications, the problem of termination detection may be formulated as deciding if a process (rather than the entire system) is terminated. We then showed that if users of a termination detector can provide such information as whether a message \( \text{msg} \) requires its receiving process to send another message in execution of \( \text{msg} \), then a distributed system can be automatically transformed to another (functionally equivalent) system so that the latter normally requires fewer overhead messages for process-termination detection than the former. Such transformation, however, does not help reducing the complexity of system-termination detection.

The technique of piggybacking proved to be useful in reducing the number of overhead messages. While these algorithms were developed for process-termination detection in a refined system, they can be used for system-termination detection
in a primitive system in which all messages are $\alpha$-messages. When so used, the algorithms are also message optimal.
CHAPTER IV

PERFORMANCE EVALUATION OF DISTRIBUTED TERMINATION DETECTION ALGORITHMS

4.1 Introduction

Since N. Francez published his pioneering work on distributed termination detection [16], a number of termination detectors with different characteristics have been proposed. There are algorithms that work for distributed systems with special properties such as diffusion [10], [29, 33]; algorithms that work for systems using synchronous communication [7, 12, 18, 27, 29, 40]; algorithms that work for systems with asynchronous first-in-first-out communication [5, 30]; algorithms that work for dynamic systems [7, 23]; and algorithms that work for systems with non-first-in-first-out communications [22, 23, 28].

Like most distributed algorithms, the performance of a distributed termination detector is most often measured by the worst-case message complexity (i.e., the total number of messages sent in the worst case by the processes of a termination
detector). Chandy and Misra [6] showed that any distributed termination detector requires, in the worst case, $\Omega(M)$ control messages in order to detect the termination of a non-diffusing distributed system, where $M$ is the number of messages used by the underlying system. Until recently, all existing termination detectors for non-diffusing distributed systems require in the worst case $\Omega(MN)$ control messages, where $M$ is as above and $N$ is the number of processes in the underlying system. The gap between the lower and the upper bounds above was recently closed by S. Chandrasekaran, C. Kannan, and S. Venkatesan, who presented in [3] a termination detector using only $O(M)$ control messages.

In terms of worst-case complexity, the Chandrasekaran-Kannan-Venkatesan (CKV) algorithm is a clear winner and can hardly be improved. However, the CKV algorithm has a drawback: it always requires $\Omega(M)$ control messages. By comparison, snapshot-based algorithms [23, 28] use only $O(N)$ control messages in best cases, although their worst-case message complexities are unbounded.

In most applications, an algorithm's average-case performance is far more critical than its worst-case performance, and comparing different algorithms by their average-case performance has become an important step in selecting an appropriate algorithm.

In this chapter, we evaluate the average-case performance of three representative distributed termination detection algorithms: the CKV algorithm, a basic snapshot-based algorithm that has an unbounded worst-case message complexity,
and an "improved" version of the snapshot-based algorithm that reduces the worst-case complexity to $O(MN)$. We first analytically compare the basic snapshot-based algorithm with the CKV algorithm on a simple random underlying system. Then we compare all three algorithms by simulation, with underlying systems belonging to a more general class of distributed systems. Finally, we run a distributed branch-and-bound algorithm for the 0/1 knapsack problem on a network of Sun workstations, and actually observe the performance of the three termination detectors. Our results indicate that should $M$ not be very small, the optimal (CKV) algorithm tends to use more control messages on the average than the other two, and the basic snapshot-based algorithm tends to outperform its "improved" version on the average. In other words, our results favor the basic snapshot-based algorithm in terms of average-case performance.

The rest of this chapter is organized as following. Section 4.2 defines the problem of distributed termination detection and describes the three detectors to be evaluated. Section 4.3 describes a simple random distributed system and our mathematical analysis. Simulation and experimental results are reported in Section 4.4 and Section 4.5, respectively. Comparisons and analysis are made in Section 4.6. Finally, Section 4.7 concludes this chapter.
4.2 Preliminary

In this section, we will review some background. First we define distributed system and what the termination detection problem is in Section 4.2.1. Then we describe a snapshot-based algorithm in Section 4.2.2. The algorithm in [3] is described in Section 4.2.3 after some modification to make it deal with non-first-in-first-out channels.

4.2.1 Distributed Termination Detection

A distributed system is a collection of processes which communicate exclusively by sending and receiving messages. At any moment a process is either active or idle. An active process may send a message to any other process, may receive an incoming message, and may turn idle at any time. An idle process cannot send any message, but it can receive incoming messages. When an idle process receives a message, it becomes active immediately.

When a process sends a message, we assume that either the message is correctly delivered to the destination after an arbitrary but finite delay, or the message is returned to the sender in which case the underlying network cannot deliver the message. Messages are not necessarily delivered in the first-in-first-out order.

A distributed system $S$ is said to be terminated if and only if all processes in $S$ are idle and all messages that have been sent by these processes have either been received by the receivers or returned to the senders. Given a distributed
system $S$, the problem of termination detection is to devise another distributed system $T$ that, when running concurrently with $S$, can detect the termination of the latter within a finite amount of time. A distributed system such as $T$ is called a distributed termination detector, while $S$ whose termination is to be detected is called the underlying distributed system (or underlying system for short).

For an underlying system $S$ and termination detector $T$, we distinguish between the messages with which the processes of $S$ communicate and the messages with which the processes of $T$ communicate. The former are called underlying messages, and the latter control messages.

4.2.2 Snapshot-based Algorithm

Most existing termination detectors differ from each other primarily in three parameters: (1) When to take a snapshot; (2) What to snapshot; (3) How to collect the snapshots.

In this subsection, we will describe a generalized snapshot algorithm based on the following theorem in [23].

**Theorem 9** [23] Let $s(p, t)$ denote the number of messages sent at $p$ by time $t$; $r(p, t)$ be the number of messages received at $p$ by time $t$; $t_p$ be the time process $p, \in S$, takes its last snapshot. A distributed system is terminated at time $\alpha$,

---

*In the rest of this chapter, a lower case $p$ or $q$ with a subscript always refers to a process, e.g., $p_i, q_0$. A $p$ or $q$ without a subscript may refer to a process or a probability, depending on the text. Where there is a possibility of confusion, we will state it clearly in the text.*
> \max\{t_p : p \in S\}, if the following conditions are satisfied:

1. For all process \( p \in S \), \( p \) is idle at time \( t_p \).
2. \( \sum_{p \in S} s(p, t_p) = \sum_{p \in S} r(p, t_p) \).
3. There exist no processes \( p, q \) in \( S \) such that \( p \) sends \( q \) a message after \( t_p \) and \( q \) receives that message before \( t_q \).

In the following algorithm, a spanning tree is assumed as in most other snapshot-based algorithms. The basic idea is that the root process of the spanning tree uses control waves to trigger snapshots at each process and collect these snapshots. Each process provides information which should be enough for the root process to make decision based on Theorem 9. Counters are used to test condition (1) and (2) in the theorem and clocks are used to check if condition (3) is satisfied. For easy understanding, we call a control message in a downward wave a signal, a control message in an upward wave a token. A process advances to a new phase each time it takes a snapshot.

The following variables are used by each process in the algorithm:

- receiving counter: number of (underlying) messages received;
- sending counter: number of messages sent;
- phase clock: current phase the process is in;
- max clock: maximum timestamp on incoming messages the process has received;
• **signal\_clock**: the timestamp on the signal the process received most recently.

**R0.** Each process initializes `receiving\_counter`, `sending\_counter`, `phase\_clock` and `max\_clock` to 0; `signal\_clock` to -1.

**R1.** Each process increments `sending\_counter` when sending a message, and stamps on every outgoing message with the value of `phase\_clock`. When it receives a message, it increments `receiving\_counter` and sets `max\_clock` to the value of the timestamp on the message if it is larger than the original value.

**R2.** The root process increments `signal\_clock` and starts a wave by sending each child process a signal stamped with the new value of `signal\_clock`.

**R3.** When a (non-root) process receives a signal, it sets `phase\_clock` and `signal\_clock` to the value of the timestamp on the message and sends a copy of the signal to each of its children.

**R4.** When a (non-root) process becomes idle, if `phase\_clock` is equal to `signal\_clock` and all its children have sent it a token whose timestamp matches its `phase\_clock`, it increments `phase\_clock` and sends its parent a token carrying the maximum value of `max\_clock` on this subtree and the deficit of total messages sent and received on this subtree.

**R5.** When the root process becomes idle and it has received from each child a
token whose timestamp matches its phase\_clock, it increments phase\_clock, calculates the deficit of total messages sent and received in the system, and checks the maximum max\_clock exists in the system. If the deficit is 0 and the maximum max\_clock matches is (one) less than its phase\_clock, it declares termination. Otherwise it executes R2.

This generalized algorithm has some advantages over existing termination detectors. For example, the Four-Counter-Method algorithm \cite{28} needs at least two waves to detect termination and may need two more waves after the termination occurs, while this algorithm may do the detection in one round and at most one more round after the termination occurs. This algorithm deals with non-first-in-first-out communication channels while most others do not.

One drawback of this algorithm, as in other algorithms based on asynchronous communication channels, is that it may need unbounded number of control messages in the worst case. To see this, consider the situation when all processes in the system are idle but there are still some pending messages. Suppose these messages are delayed for a long time, during which control waves have been traveling up and down the spanning tree a few times. This drawback can be overcome by using flags and adding another type of control message into the system. The basic idea is that each process keeps a flag to indicate whether it has received any message after reporting a token to its parent. In the above R5, if the root process realizes the maximum max\_clock is less than its phase\_clock but the deficit is not 0, it
sends red signals to ask if any process has received a message since the last time
the process took a snapshot. A process recently receiving a message replies to such
an inquiry with a red token. After receiving such a reply, the root process starts
another phase. This function is now spelled out as augmented rules.

R6. Each process keeps a flag which is initialized to false. Whenever a process
receives a message, it sets the flag to true; whenever it sends a token, it
reset the flag to false.

R7. If a process has a true flag and the stamp on red_signal.clcok is equal to
its phase.clcok, or it has received a token whose stamp value is the same as
red_signal.clcok, then it sends a red token stamped with red_signal.clcok if
it has not reported to that red signal yet.

R8. The root process executes R2 if its signal.clcok is less than phase.clcok and
either its flag is true or it has received a red token with timestamp equal
to phase.clcok.

The correctness of the above algorithm can be found in [23].

4.2.3 Chandrasekaran-Kannan-Venkatesan Algorithm

In the (CKV) algorithm by Chandrasekaran, Kannan and Venkatesan [3], a sim-
ilar spanning tree is assumed and asynchronous communication is allowed. But
messages have to be delivered in the correct order as they are sent.
The basic idea of CKV algorithm is that each process keeps track of every single message sent in the system after the termination detection starts and reports to its neighbor who issues the message (and therefore triggers its computation). In a model of first-in-first-out channels, it is very easy to tell which messages are sent before and which are after the termination detection starts by sending control messages through all channels as a phase separator. When we want to adjust the algorithm for non-first-in-first-out channels, we have to make sure this basic phase separation function preserved by some mechanism one way or the other.

If we mark each message with a sequence number and use coloring technique to separate detection phases, plus that the transmission delay is finite, we can turn the original CKV algorithm into one that deals with non-first-in-first-out channels as in the Snapshot-based algorithm. The modified algorithm is now depicted in rules:

R0. Initially, all messages sent are colored white and all processes are in initial state NDT (Not Detecting Termination).

R1. The root process changes its state from NDT to DT (Detecting Termination) and sends a warning message to all its neighbor processes when it decides to detect termination.

R2. When a process receives a warning from its neighbor, it marks that channel.

If it is in state NDT, it sends a warning to all its neighbors, and changes
state to DT.

R3. When a process in state DT sends a message (generated by the underlying computation) to its neighbor \( q \), it numbers the message, colors it red and pushes an entry \( TO(q) \) on its local stack. Different sequence numbers apply to different addressees and are updated when a message is sent.

R4. When a process receives a red message (which is generated by the underlying computation) from a neighbor \( p \) on a marked channel and it is already in DT, it pushes \( FROM(p) \) on its stack. If it is in NDT (and the channel is not marked), the message is treated as a warning as in R2 before it is pushed on stack.

R5. When a process turns idle and no messages are missing according to sequence numbers, it pops the entry on the top of the stack if it is a \( FROM(p) \), and sends a remove_entry message to process \( p \). If the top entry is a \( TO(q) \), it waits until \( q \) reports.

R6. When a process receives a remove_entry message from its neighbor \( q \), it scans the stack and deletes the first entry of the form \( TO(q) \). If it is idle, it clears the stack as in R5.

R7. A node sends the terminate message to its parent when it is idle, each of its channels are marked, its local stack is empty, no messages are supposedly
coming, and it has received the *terminate* message from each of its children.

**R8.** If a process receives a *terminate* message while it is in NDT, it treats the message like a *warning* as in **R2**.

**R9.** When the condition in **R7** is satisfied at the root, it concludes that the underlying computation has terminated.

If channels are FIFO, the correctness of CKV algorithm has been proved in [3]. We now argue this modified algorithm is also correct. When channels become non-FIFO, the disordering may result in (1) *warning* arrives after red message or *terminate*, (2) *terminate* arrives before last red message, and (3) red messages arrive out of order. Since a red message or a *terminate* is also treated as a *warning* if the latter has not arrived, this takes care of (1). When a process sends out *terminate*, the stack must be empty. So if a red message arrives after *terminate* (from the same channel), it is either a delayed message or one that is triggered by some other process and that process will wait for the current computation to finish before sending *terminate* to its parent. This takes care of (2). The sequence number associated with each message is an indication of the disordering of messages and a process will wait for missing message before clearing up the stack. This takes care the effect of the last possibility of disordering.

Let \( e \) denote the number of one-way channels in the system. (A two-way channel is counted as two channels.) Since all \( M \) underlying messages are acknowledged, \( e \)
warnings and $N - 1$ terminates are sent, the message complexity is $O(e + M + N)$, or simply $O(M)$ if $e, N < M$.

4.3 Theoretical Analysis

The purpose of this chapter is to quantitatively compare the average message-complexities of the two termination detectors. For that purpose, we shall first make a theoretical analysis by describing a random distributed system and compute the expected number of control messages required for each of the two termination detectors to detect the termination of such a random system. Other approaches will be given in later sections.

A distributed system without any concrete parameters is hard to analyze. We therefore associate each possible action with some probabilities to construct a random distributed system.

Based on this random distributed system, we calculate the number of control messages required by the snapshot-based algorithm, and the number of underlying messages, which is also the number of control messages required in the CKV algorithm excluding warning messages since one underlying message results in exactly one control message, in the system.

The number of control messages issued in the system can be calculated by multiplying the number of control waves by the number of control messages in one wave. To get the number of control waves, we analyze the expected lifetime of the
In Section 4.3.1, we describe random distributed systems based on probabilities. In Section 4.3.2, we show how to calculate the expected lifetime of such a system. In Section 4.3.3, we show how to derive the expected time of a control wave, and therefore the expected number of control waves and control messages. The number of underlying messages in a random system (or the number of control messages in a CKV algorithm) is calculated in Section 4.3.4. We use an example to show the result of calculation on a random system with specific parameters in Section 4.3.5.

### 4.3.1 Random Distributed Systems

A random distributed system $R$ with parameters $(n, p, q, d)$ – where $n$ is a natural number, $p$, $q$, and $d$ are real numbers between 0 and 1 such that $p + q < 1$ – consists of $n$ processes and operates on the discrete time axis. Each process is a sequence of actions or events. There are six types of events:

1. An active process turns idle.
2. An active process sends an (underlying) message.
3. An active stays active without sending an (underlying) message.
4. An active process receives all incoming (underlying) messages.

---

*The discretization of time has proved to be useful for analysis of distributed systems [2]; it will enable us to model a distributed system as a discrete-time Markov chain.*
5. An idle process receives all incoming (underlying) messages and turns active.

6. Null event (an idle process stays idle).

Each action is assumed to be atomic. The first three types of actions are self-initiated actions, as opposed to the passive actions of the other three. For ease of performance analysis, we assume that self-initiated events may occur only at times $0, 1, 2, \ldots$, and passive events may occur only at times $0^+, 1^+, 2^+, \ldots$, where $i^+$ means the instant a little bit after time $i$, e.g., time $i + 0.1$ (see Fig. 6).

At any tick of the discrete time axis, exactly one self-initiated event occurs at each active process:

- With probability $p$, the process turns idle;
- With probability $q$, the process sends a message;
• With probability $1 - p - q$, the process stays active without sending any message.

Whatever action of the above is taken by a process, it is assumed to be finished immediately. A self-initiated action may be immediately followed by a passive action. Thus, for example, an active process may turn idle at time $t$ and resumes active at time $t^+$ by receiving a message.

Recall that in a general distributed system, a message sent by a process is either correctly delivered to the destination after an arbitrary delay, or is returned to the sender if the underlying network cannot deliver the message. To capture this feature, we assume in our random distributed system that when an active process sends a message, it randomly chooses a process as the receiver (thus, every process in the system, including the sender itself, has equal probability to receive the message), and assume that the underlying network is reliable and will correctly deliver the message to the destination. Note that we have modeled the possible event in a real system that “a message is returned to its sender by an unreliable network” by that “a message is addressed to the sender itself and is correctly delivered by a reliable network.”

There is a possible delay for each message. If a message is sent at time $t$ and does not arrive its destination immediately by $t^+$, we say this message is in transit. At any time $t$, a message in transit will be delayed with probability $d$, and be delivered to its destination at time $t^+$ with probability $1 - d$. Thus, a message sent
at time $t$ will, with probability $d^k(1 - d)$, be delivered to its destination at time $(t + k)^+$, for $k \geq 0$. Consequently, an active process may turns idle at time $t$ and immediately resume active by receiving a message at $t^+$. 

Note that in our random distributed system the delay time of a message is independent of its sender and receiver. Thus, we may assume without loss of generality that when a process sends a message, the receiver's address is written on the message; without reading the address, the communication system waits for a random amount of time (according to the above delay-time distribution); after the delay time, the address is read and the message is delivered to the receiver. The main point here is that the receiver's address is not read or used until the moment the message is to be handed to the receiver. Because of that we may think that the sender of a message does not specify the receiver but it relies on the communication network to randomly pick a receiver at the moment the communication network is about to deliver the message. This point of view will prove to be useful in our analysis of termination detectors.

A process is alternately active and idle throughout the lifetime of the system. A time interval $(t_1, t_2)$ is said to be an active session of a process if the process changes from idle to active at time $t_1^+$, stays active all the time, and finally turns idle at $t_2$. (If the process resumes active at time $t_2^+$, then another active session starts.) If $(t_1, t_2)$ is an active session of a process, then for any moment $t$ in the interval, $t_2 - t$ is said to be the process's remaining active time as of time $t$. 
If at time $t$, exactly $l$ active processes turn idle, $s$ active processes each sends a message, and $a$ active processes stay active without sending any message, then we say that the global event $<l,s,a>$ occurs at time $t$.

The global state of a distributed system is the number of active processes together with the number of still-in-transit (underlying) messages.

### 4.3.2 Lifetime of a System

Let $R$ be a random distributed system with parameters $(n, p, q, d)$. Let $r = 1 - p - q$. We compute in this section the expected lifetime of $R$, treating $R$ as a Markov chain.

Let $X_t$ denote the system's global state at time $t^-$, $t = 0, 1, \ldots$, where $t^-$ denote the instant just before time $t$, say $t - 0.1$. (See Fig. 6.) $X_t = (i, j)$ if and only if at time $t^-$ the system has exactly $i$ active processes and exactly $j$ still-in-transit messages. The sequence of random variables $X_0, X_1, \ldots$ evidently forms a homogeneous Markov chain$^c$.

Assume that the system starts with $n$ active processes, i.e., $X_0 = (n, 0)$. By definition, the system will terminate once it reaches the absorption state $(0, 0)$.

Let $P_{i,j,i',j'}$ be the probability of going to state $(i', j')$, given that the system is currently at state $(i, j)$. That is,

$$P_{i,j,i',j'} = P\{X_{t+1} = (i', j')|X_t = (i, j)\} \quad 0 \leq i \leq n, 0 \leq j < \infty.$$  

$^c$A Markov chain is homogeneous if the transition probabilities are independent of $t$. 

In the following (Lemmas 4 – 8), we show how to compute the matrix of transition probabilities, from which we shall be able to calculate the system’s expected lifetime (Eq. 4.2).

If there are \( i \) active processes and \( j \) in-transit messages in the system, the following lemma gives the probability with which the global event \(< l, s, a >\) will occur. Recall that in the event \(< l, s, a >\), exactly \( l \) active processes turns idle, \( s \) processes send messages, and \( a \) processes stay active without sending messages.

**Lemma 4** If \( X_t = (i, j) \), then the probability that the global event \(< l, s, a >\) occurs at time \( t \) is given by

\[
P\{< l, s, a >\} = \binom{i}{l} p^l q^{i-l} r^a
\]

where \( i = l + s + a \).

**Proof.** Each of the \( i \) active processes will take exactly one of the three self-initiated actions with probability \( p, q \) and \( r \) respectively. If we regard the event at a process as a trial with three possible outcomes, \( P\{< l, s, a >\} \) is the probability that in \( i \) trials, each outcome occurs \( l, s \) and \( a \) times respectively. This is a generalized Bernoulli trials (or multinomial distribution) [14] and the probability is

\[
P\{< l, s, a >\} = \frac{i!}{l!s!a!} p^l q^s r^a
\]

which is an equivalent of Eq. 4.1. \( \Box \)

Let \( I_t \) and \( J_t \) be random variables such that \( X_t = (I_t, J_t) \).
Lemma 5 If $X_t = (i, j)$ and global event $< l, s, a >$ occurs at time $t$, the probability that there will be $j'$ messages in transit at time $(t + 1)^-$ is

$$P\{J_{t+1} = j'\} = \begin{cases} \binom{j+s}{j'}(1-d)^{j+s-j'} & \text{if } s \leq j' \leq s+j \\ 0 & \text{otherwise} \end{cases}$$

Proof. $j$ messages are in transit at time $t^-$ and $s$ newly issued messages will join them at time $t$. Each of these $j + s$ messages will be delayed with probability $d$ or delivered with probability $1 - d$. This is another generalized Bernoulli trial with $j + s$ trials and outcome probability $d$ and $1 - d$. So the probability is as in the above equation. □

Lemma 6 [26] The number of combinations of distributing $r$ distinct objects into $n$ distinct cells with no cell left empty is $n!S(r, n)$, where $S(r, n)$ is the Stirling number of the second kind and is defined as

$$S(r, n) = \frac{1}{n!} \sum_{i=0}^{n} (-1)^i \binom{n}{i} (n-i)^r$$

Lemma 7 If $X_t = (i, j)$ and global event $< l, s, a >$ occurs at time $t$, then the probability that $i'$ processes are active given that $j'$ messages are in transit at time $(t + 1)^-$ is

$$P\{I_{t+1} = i'|J_{t+1} = j'\} = \begin{cases} A & \text{if } a + s \leq i' \leq 2s + j \\ 0 & \text{otherwise} \end{cases}$$

where

$$A = \sum_{k=i'-s-a}^{j+s-j'} \binom{j+s-j'}{k} \binom{n-s-a}{i'-s-a} S(k, i'-s-a)(i'-s-a)! \frac{(s+a)^{j+s-j'-k}}{n^{j+s-j'}}$$
and $S$ is the Stirling number of the second kind given in Lemma 6.

**Proof.** At time $t$, there are $j + s$ messages need to be decided by probability for their delay. $j'$ of them will still be in transit after $t^+$ and $j + s - j'$ of them will arrive at some processes at $t^+$. Since $s + a$ processes will remain active after $t$ and $i'$ processes will be active in $X_{t+1}$, there must be $i' - s - a$ processes, out of $n - s - a$ idle processes, wakened by some messages at $t^+$. Each of these $i' - s - a$ processes must receive at least one of the $j + s - j'$ arriving messages. This is the same as distributing $k$, $i' - s - a \leq k \leq j + s - j'$, messages into $i' - s - a$ processes with no process left empty. Remaining $j + s - j' - k$ messages will be addresses to some of the $s + a$ active processes. Each message has equal chance to be addressed to any process in the state with probability $1/n$. □

The probability of one-step transition can now be derived as in the following lemma.

**Lemma 8** The one-step transition probability is

$$p_{ij,i'j'} = \sum_{s=0}^{i-s} \sum_{l=0}^{i-s} P\{<l,s,a>\} P\{J_{t+1} = j'\} P\{I_{t+1} = i'| J_{t+1} = j'\}$$

**Proof.**

$$P_{ij,i'j'} = P\{X_{t+1} = (i',j')|X_t = (i,j)\}$$

$$= \sum_{<l,s,a>} P\{X_{t+1} = (i',j')|X_t = (i,j),<l,s,a>\} P\{<l,s,a>\}$$
\[
\begin{align*}
&= \sum_{<l,s,a>} P\{<l,s,a>\} P\{I_{t+1} = i', J_{t+1} = j'\} \\
&= \sum_{s=0}^{i} \sum_{l=0}^{i-s} P\{<l,s,a>\} P\{J_{t+1} = j'\} P\{I_{t+1} = i'|J_{t+1} = j'\}
\end{align*}
\]

With the transition probabilities \( P_{i,j,i',j'} \), we are able to compute the expected lifetime of the system, denoted \( \eta \), which is simply the expected number of steps required for the process to move from the initial state \((n,0)\) to the absorbing state \((0,0)\).

Let \( W_{i,j,i',j'} \) denote the expected number of times state \((i',j')\) is to be visited, given that the system starts with state \((i,j)\). Then

\[
\eta = \sum_{j=0}^{\infty} \sum_{i=0}^{n} W_{i0,ij} \tag{4.2}
\]

However, it is impossible to evaluate the summation with infinite terms in real computation, we can only approximate it by finite terms. But if the system will not terminate forever, we should not approximate an infinite value with a finite one. Therefore we are interested in finding under what conditions the system will terminate with probability one and this will allow us to use approximation.

**Theorem 10** In a random distributed system \( R \) as proposed in Section 4.3.2, if \( p \), the probability of turning idle by an active process, is greater than or equal to \( q \), the probability of sending message and staying active, the system will terminate
with probability one. That is,

\[ P\{R \text{ will terminate}\} = 1 \text{ if } p \geq q \]

The proof is given in the last section of this chapter. In the following, we use \( b \) as our bound in computation.

The \( W_{n0,ij} \) variables satisfy the following equations [39]:

\[
W_{i,j,i',j'} = \delta_{ij,i'j'} + \sum_{j''=0}^{b} \sum_{i''=0}^{n} P_{i,i''j''} W_{i''j'',i'j'} \quad 0 \leq i, i' \leq n, 0 \leq j, j' \leq b
\]  

(4.3)

where

\[
\delta_{ij,i'j'} = \begin{cases} 
1 & \text{if } ij = i'j' \\
0 & \text{otherwise}
\end{cases}
\]

Note that for each \((i',j')\), one set of linear equations have to be solved. Summing up \( W_{n0,i',j'} \), \( 0 \leq i' \leq n, 0 \leq j' \leq b \) yields \( \eta \).

Let \( \gamma \) be the expected total active time of a process in the lifetime of the system.

**Lemma 9** The expected total active time of a process in the lifetime of the system is

\[
\gamma = \frac{\sum_{j=0}^{b} \sum_{i=0}^{n} i W_{n0,ij}}{n}
\]

(4.4)

**Proof.** Each time state \((i, j)\) is visited, there are \( i \) processes active. Weigh each \( W_{n0,ij} \) by the number of active processes then divide by total number of processes yields average number of active time for each process. \( \square \)

\( \eta \) can be solved in one set of Eq. 4.3 if \( \delta_{ij,i'j'} = 1 \), for all \( i, j, i' \) and \( j' \). But we still need \( W_{n0,i'j'} \) in Eq. 4.4.
4.3.3 Number of Control Messages

In this section, we analyze the Snapshot-based algorithm with our probabilistic model. We want to find out how many control messages are sent in an algorithm like this.

To count the number, $C$, of control messages used, we first need to know how many waves are issued in the algorithm since the number of waves determines the number of control messages. We start with some definitions.

$\alpha$: The probability that a process is active when a control message arrives at it.

$A_i$: A random variable denoting the duration of an active session of process $i$.

$D_i$: A random variable denoting the transmission time of a message in a virtual communication channel between process $i$ and one of its children process.

$S_i$: A random variable denoting the remaining active time of process $i$ when a control message arrives at it.

From definition, it is clear that $A_i$ and $D_i$ have geometric distribution with parameter $p$ and $1 - d$, respectively. $P\{A_i = k\} = p(1 - p)^{k-1}$ and $P\{D_i = k\} = (1 - d)d^{k-1}$. Here we consider the delay of a control message to be geometrically distributed and is different from underlying messages whose delay is geometric distributed only if it can get to the underlying network. We assume control messages have higher priority and can always get to the underlying network.
We approximate $\alpha$ by the following equation:

$$\alpha \approx \frac{\gamma}{\eta}$$

where $\gamma$ is the expected total active time of a process in the lifetime of the system as defined in Lemma 9 and $\eta$ is the expected lifetime of the system as in Eq. 4.2.

That is, we approximate $\alpha$ by the probability that a process is active at a randomly picked instant during the lifetime of the system. This is also based on the fact that a control message may arrive at a process any time before the system terminates.

Lemma 10 $S_i = YA_i$, where $Y$ is a Bernoulli function

$$Y = \begin{cases} 
0 & \text{with probability } 1 - \alpha \\
1 & \text{with probability } \alpha 
\end{cases}$$

Proof. If a process is idle when a control message arrives at it, the remaining active time is 0. If the process is active, the remaining active time is the same as $A_i$, a geometric distribution with parameter $p$, since the remaining active time is independent of the past activeness in that active session. When a control message arrives at a process, the probability that it is active is $\alpha$. □

The time, $T_0$, of a wave in the Snapshot-based algorithm described in Section 4.2.2 is from the moment the root process, $p_0$, sends out the wave till it gets the last token reported from its children, plus the remaining active time of $p_0$ after
receiving that last token. Of course if $p_0$ has already been idle then this remaining active time is 0. $p_0$ will then decide if the system has been terminated.

In the probabilistic model we described in Section 4.3.1, we assume that a process can send a message to any process in the system. The probabilistic delay is applied to the message when it is in the virtual communication channel between processes. This implies that the system is a complete communication system and therefore its spanning tree is virtually a star-shape spanning tree whose depth is 2.

Since it is possible to complicate our model and make $d$ denote the delay probability of messages in each physical link, we assume the spanning tree to be a general one in the following analysis. For the case of a star-shape spanning tree, one can ignore the part for processes which are neither the root nor a leaf process.

Let $T_i$ denote the time from the moment process $p_i$ receives a signal from its parent to the moment it sends a token back to its parent. (Recall that a process will send out a token only after receiving tokens from all its children.) In other words, $T_i$ is the wave time for subtree rooted at $p_i$ and is determined recursively by the wave time of its children.

Formally,

$$T_i = \begin{cases} 
\max_{j \in C} \{D_j + T_j + D'_j\} + S_0 & \text{if } i = 0 \\
S_i & \text{if } p_i \text{ is a leaf process} \\
S'_i + \max_{j \in C} \{D_j + T_j + D'_j\} + S_i & \text{otherwise}
\end{cases}$$

here $C$ is the set of children of $p_i$; $S_i$ is the remaining active time for current active

*We will discuss it in the conclusion of this chapter.
session when $p_i$ receives all tokens; $S'_i$ is the remaining active time for current active session when $p_i$ receives signal from its parent. $D_j$ is the transmission delay for downward signal to $p_j$ and $D'_j$ for upward token from $p_j$. $S_i$ and $S'_i$ have same distribution, so do $D_j$ and $D'_j$.

To calculate the expected time of a wave, $T_0$, we use the following equivalence:

$$E[T_0] = \sum_{k=0}^{\infty} k f_0(k) = \sum_{k=0}^{\infty} (1 - F_0(k))$$

$F_0$ and subsequent $F_i$'s are derived as following.

$$F_0(k) = P\{T_0 \leq k\}$$

$$= P\{\max_{j \in \mathbb{C}}\{D_j + T_j + D'_j\} + S_0 \leq k\}$$

$$= \sum_{n=0}^{k-2} P\{\max_{j \in \mathbb{C}}\{D_j + T_j + D'_j\} \leq k - n | S_0 = n\} P\{S_0 = n\}$$

$$= \sum_{n=0}^{k-2} P\{\max_{j \in \mathbb{C}}\{D_j + T_j + D'_j\} \leq k - n\} P\{S_0 = n\}$$

$$= \sum_{n=0}^{k-2} P\{S_0 = n\} \prod_{j \in \mathbb{C}} P\{D_j + T_j + D'_j \leq k - n\}$$

$$= \sum_{n=0}^{k-2} P\{S_0 = n\} \prod_{j \in \mathbb{C}} \sum_{m=1}^{k-n} \sum_{m'=1}^{k-n-m} P\{T_j \leq k - n - m - m'|D_j = m, D'_j = m'\} P\{D_j = m\} P\{D'_j = m'\}$$

$$= \sum_{n=0}^{k-2} P\{S_0 = n\} \prod_{j \in \mathbb{C}} \sum_{m=1}^{k-n} \sum_{m'=1}^{k-n-m} P\{T_j \leq k - n - m - m'|D_j = m\} P\{D_j = m\} P\{D'_j = m'\}$$
If \( p_i \) is a leaf process, then

\[
P\{T_i \leq k\} = P\{S_i \leq k\}
\]

\[
= \sum_{n=0}^{k} P\{S_i = n\}
\]

\[
= \beta + \alpha(1 - q^k)
\]

where \( l \) is the level of the process on the spanning tree and \( \alpha \) is as defined in Eq 4.5.

If the process \( p_i \) is neither the root nor a leaf process, then

\[
P\{T_i \leq k\} = P\{S'_i + \max_{j \in C}\{D_j + T_j + D'_j\} + S_i \leq k\}
\]

\[
= \sum_{n=0}^{k-2} \sum_{n'=0}^{k-n-2} P\{\max_{j \in C}\{D_j + T_j + D'_j\} \leq k - n - n'\} P\{S_i = n\} P\{S'_i = n'\}
\]

\[
= \sum_{n=0}^{k-2} \sum_{n'=0}^{k-n-2} \prod_{j \in C} P\{D_j + T_j + D'_j \leq k - n - n'\} P\{S_i = n\} P\{S'_i = n'\}
\]

\[
= \sum_{n=0}^{k-2} \sum_{n'=0}^{k-n-2} P\{S_i = n\} P\{S'_i = n'\} \prod_{j \in C} \sum_{m=1}^{k-n-n' - k-n-n'-m} \sum_{m'=1}^{k-n-n'-m-m'} P\{T_j \leq k - n - n' - m - m'\} P\{D_j = m\} P(D'_j = m')
\]

Since \( A_i \) is geometric and from Lemma 10,

\[
P\{S_i = n\} = \begin{cases} \beta & \text{if } n = 0 \\ \alpha pq^{n-1} & \text{otherwise} \end{cases}
\]

Since \( D_j \) is geometric,

\[
P\{D_j = m\} = (1 - d)d^{m-1} \quad m \geq 1
\]

As in other probabilistic model [35], some simplifying assumptions of independence are made to make the preceding calculation possible:
1. \(S_i\) is independent of \(D_j + T_j + D_j'\): when \(p_i\) sends out a signal, the remaining active time is independent of the time from that moment to the moment when it receives a token from that child.

2. \(D_j\) is independent of \(T_j\): the delay time of a message in the channel between \(p_j\) and its parent \(p_i\) is independent of wave time of the subtree rooted at \(p_j\).

3. \(D_i\) is independent of \(D_i'\): the delay time of a downward signal is independent of an upward token in the same wave cycle.

4. \(D_i + T_i + D_i'\) is independent of \(D_j + T_j + D_j'\) if \(p_i\) and \(p_j\) are sibling: the wave time of the subtree rooted at \(p_i\) is independent of wave time of the subtree rooted at \(p_j\).

With \(\eta\) and \(E[T_0]\), we can finally calculate \(X\), the expected number of control messages, as

\[
X = (\eta/E[T_0] + O(1)) \times 2 \times (n - 1)
\]

The extra \(O(1)\) is added since the algorithm may need one more rounds of detection wave even after the termination of the system occurs. The snapshot-based algorithm requires at most one more round of waves after termination. Whether this extra round is needed depends on which part of the spanning tree the last basic communication occurs. Each round of wave requires \(2 \times (n - 1)\) control message.
4.3.4 Number of Underlying Messages

The estimate of the number of underlying messages issued in the system is stated in the following two lemmas.

**Lemma 11** Let the number of messages issued in the system be $M$, then

$$M = \eta aqn$$

**Proof.** On the average, a process is active for $\eta \alpha$ units of time. When a process is active, the probability that it will issue a message is $q$. There are totally $n$ processes in the system. So the number of messages issued is the product of $\eta$, $\alpha$, $q$ and $n$. □

**Lemma 12** Let the number of messages issued after $p_0$ starts the first detection wave be $M'$, then

$$M' = (\eta - 1/p)\alpha qn$$

**Proof.** Since $A_t$ is geometric with parameter $p$, the expected time of the first active session of $p_0$ is $1/p$. The remaining system time after the first wave starts is $\eta - 1/p$. □

4.3.5 Example

We use a spanning trees of size 8, as an example to calculate the number of control messages required under some combinations of $p$, $q$ and $r$. This should provide us a better understanding of the methodology proposed in previous subsections.
The result is given in Table 1. The value of \( b \) (Eq. 4.3) used in calculation is 16. We assume \( O(1) \) to be \( 1/2 \) in Eq. 4.6 since 0 or 1 extra round of control wave may be needed. Note that the number of control messages needed by CKV algorithm is \( M' \) if the detector is started as soon as the computation starts. If the detector starts after the root process turns idle for the first time, then it needs to add 56 control messages – one for each channel, to trigger the detection at each process. Recall that the system will terminate with probability 1 only if \( p \geq q \). For \( p < q \), the termination is not guaranteed and therefore the numbers shown in the table are pretty rough approximations in some sense. But the implication on magnitude is by no means less.

4.3.6 Comments

In Table 1, we did not list probabilistic systems with certain parameters combination when less than 8 underlying messages may be issued since we expect each process to issue at least one message during its lifetime. Systems which are extreme cases, such as \( |p - q| \geq 0.6 \), are not listed either. We can see that when processes have smaller probability of sending messages, \( q \leq 0.4 \), or have larger probability of turning idle, \( p \geq 0.5 \), the number of control wave is more likely to be in the very low order. This means the number of control messages is close to \( O(N) \), where \( N \) is the number of processes, as the optimal case for a termination detector. When the number of waves increases, the number of underlying message increases in a much
Table 1: Result from probabilistic model with $d = 0.2$, $n = 8$

<table>
<thead>
<tr>
<th>$p$</th>
<th>$q$</th>
<th>$r$</th>
<th>$\eta$</th>
<th>$E(T_0)$</th>
<th>$X$</th>
<th>$M$</th>
<th>$M'$</th>
</tr>
</thead>
<tbody>
<tr>
<td>.1</td>
<td>.1</td>
<td>.8</td>
<td>45.6</td>
<td>23.7</td>
<td>34.0</td>
<td>15.3</td>
<td>68.0</td>
</tr>
<tr>
<td>.1</td>
<td>.2</td>
<td>.7</td>
<td>113.1</td>
<td>26.7</td>
<td>66.3</td>
<td>94.3</td>
<td>142.0</td>
</tr>
<tr>
<td>.2</td>
<td>.2</td>
<td>.6</td>
<td>23.4</td>
<td>13.2</td>
<td>31.8</td>
<td>16.2</td>
<td>68.8</td>
</tr>
<tr>
<td>.25</td>
<td>.25</td>
<td>.5</td>
<td>19.0</td>
<td>11.1</td>
<td>30.9</td>
<td>16.7</td>
<td>69.2</td>
</tr>
<tr>
<td>.1</td>
<td>.3</td>
<td>.6</td>
<td>390.6</td>
<td>30.0</td>
<td>189.0</td>
<td>608.7</td>
<td>649.1</td>
</tr>
<tr>
<td>.2</td>
<td>.3</td>
<td>.5</td>
<td>37.2</td>
<td>13.9</td>
<td>44.5</td>
<td>42.8</td>
<td>93.0</td>
</tr>
<tr>
<td>.3</td>
<td>.3</td>
<td>.4</td>
<td>16.1</td>
<td>9.7</td>
<td>30.1</td>
<td>17.2</td>
<td>69.7</td>
</tr>
<tr>
<td>.4</td>
<td>.3</td>
<td>.3</td>
<td>9.9</td>
<td>8.0</td>
<td>24.4</td>
<td>10.6</td>
<td>63.9</td>
</tr>
<tr>
<td>.33</td>
<td>.33</td>
<td>.33</td>
<td>14.6</td>
<td>9.1</td>
<td>29.6</td>
<td>17.6</td>
<td>70.0</td>
</tr>
<tr>
<td>.1</td>
<td>.4</td>
<td>.5</td>
<td>1529.8</td>
<td>32.3</td>
<td>669.2</td>
<td>3656.2</td>
<td>3688.3</td>
</tr>
<tr>
<td>.2</td>
<td>.4</td>
<td>.4</td>
<td>67.9</td>
<td>14.7</td>
<td>71.4</td>
<td>118.3</td>
<td>165.6</td>
</tr>
<tr>
<td>.3</td>
<td>.4</td>
<td>.3</td>
<td>22.5</td>
<td>10.0</td>
<td>38.5</td>
<td>34.26</td>
<td>85.18</td>
</tr>
<tr>
<td>.4</td>
<td>.4</td>
<td>.2</td>
<td>12.5</td>
<td>8.0</td>
<td>28.8</td>
<td>18.4</td>
<td>70.7</td>
</tr>
<tr>
<td>.5</td>
<td>.4</td>
<td>.1</td>
<td>8.4</td>
<td>7.0</td>
<td>23.9</td>
<td>12.4</td>
<td>65.5</td>
</tr>
<tr>
<td>.1</td>
<td>.5</td>
<td>.4</td>
<td>5926.1</td>
<td>33.8</td>
<td>2465.5</td>
<td>19212.8</td>
<td>19236.4</td>
</tr>
<tr>
<td>.2</td>
<td>.5</td>
<td>.3</td>
<td>141.6</td>
<td>15.7</td>
<td>133.6</td>
<td>350.8</td>
<td>394.4</td>
</tr>
<tr>
<td>.3</td>
<td>.5</td>
<td>.2</td>
<td>34.1</td>
<td>10.3</td>
<td>53.2</td>
<td>70.5</td>
<td>119.6</td>
</tr>
<tr>
<td>.4</td>
<td>.5</td>
<td>.1</td>
<td>16.6</td>
<td>8.2</td>
<td>35.4</td>
<td>31.8</td>
<td>83.0</td>
</tr>
<tr>
<td>.5</td>
<td>.5</td>
<td>.0</td>
<td>10.5</td>
<td>7.0</td>
<td>27.8</td>
<td>19.7</td>
<td>71.9</td>
</tr>
<tr>
<td>.1</td>
<td>.6</td>
<td>.3</td>
<td>21864.8</td>
<td>34.7</td>
<td>8838.7</td>
<td>89484.0</td>
<td>89499.1</td>
</tr>
<tr>
<td>.2</td>
<td>.6</td>
<td>.2</td>
<td>328.0</td>
<td>16.5</td>
<td>285.6</td>
<td>1085.1</td>
<td>1124.5</td>
</tr>
<tr>
<td>.3</td>
<td>.6</td>
<td>.1</td>
<td>56.6</td>
<td>10.8</td>
<td>80.6</td>
<td>154.3</td>
<td>201.2</td>
</tr>
<tr>
<td>.2</td>
<td>.7</td>
<td>.1</td>
<td>816.4</td>
<td>17.1</td>
<td>674.2</td>
<td>3416.7</td>
<td>3451.8</td>
</tr>
</tbody>
</table>

$p$: probability of turning idle  
$q$: probability of sending a message and staying active  
$p$: probability of staying idle without sending message  
$\eta$: expected system lifetime  
$E(T_0)$: expected time of a wave  
$X$: expected number of control messages  
$M$: expected number of underlying messages  
(CKV algorithm starts from the very beginning)  
$M'$: adjusted expected number of underlying messages  
(CKV algorithm starts after first active session)
faster pace. This means the increasing speed of control messages can hardly catch that of underlying messages, not to mention the worst case complexity $O(MN)$. This is extremely obvious when $q > 0.5$.

4.4 Simulation

In the previous section, we compare two algorithms — a snapshot-based algorithm (including both the basic and the improved versions) and CKV algorithm, by a theoretical approach using a probabilistic model. This approach is purely from the theoretical point of view as no distributed system in the real world is actually operated based on probability. To extend this theoretical work to a more realistic model, we ran programs to simulate distributed systems. We will now compare three algorithms: the CKV algorithm and two snapshot-based algorithms — the basic one with unbounded control message complexity and the more complicated one with bounded complexity.

4.4.1 Simulated System

The system we simulate has following features:

- Each process is assigned some work load, according to some statistic distribution, to start with.

- When a process is active, it may send messages to other processes in the system. The inter-message period is based on some statistic distribution and
the addressee is randomly chosen.

- When a process receives an underlying message, a small amount is added to its work load as processing overhead caused by new information.

- Each message has transmission delay according to some distribution.

- When a process becomes idle, it sends request for work to a randomly picked process. Requests that are not granted will be discarded.

- A snapshot-based termination detector is added to the system.

Note that if the statistic distributions we use are exponential, the message delay and inter-message period are the continuous version of geometric distribution in the random distributed system.

The idea of work request and the strategy for load balance is adopted from [15]. We have the following policy in our simulated system: if the receiver has a load more than the mean initial load, it transfers part, depends on a uniform distribution between 0.25 and 0.5, of the load to the requester.

The simulated system is assumed to have 8 processes and each process can communicate with any other process in the system. So the spanning tree used by the termination detector has only two levels as in the example in previous section.

Program is written in SIMSCRIPT II and ran on IBM 3081-D running MVS/XA 2.1.7.
4.4.2 Results

There are four variables used in the simulation: the time between message sendings, the initial load of a process, the processing overhead of a message, and the message delay. We will assume exponential distribution for inter-message time, while exponential and normal distributions for the other three variables.

We use the following notations in our tables:

\( \lambda_m \) – (exponential) mean of time between message sendings,

\( \lambda_i \) – (exponential) mean of initial load,

\( \lambda_p \) – (exponential) mean of processing time for underlying messages,

\( \lambda_d \) – (exponential) mean of message delay,

\( \mu_i \) – (normal) mean of initial load (normal),

\( \mu_p \) – (normal) mean of processing time for underlying messages,

\( \mu_d \) – (normal) mean of message delay,

\( \sigma_p \) – (normal) standard deviation of processing time,

\( \sigma_i \) – (normal) standard deviation of initial load,

\( \sigma_d \) – (normal) standard deviation of message delay.
We varied the mean (and deviation) of these variables to see how they may affect the performance of the message complexity of distributed termination detection algorithm. In normal distribution, if the generated value is less than 0, it will be replaced by 0. We repeated simulation 30 times for each set of data.

Each table has three columns for control message complexity of three different termination detectors: unbounded snapshot-based algorithm, bounded snapshot-based algorithm, and the CKV algorithm. The number of control messages in the first two algorithms are measures from the simulation, while the number of control messages for the CKV algorithm is the number of underlying messages issued in the system since each underlying message causes exactly one control message. In the first two algorithms, a process will not pass a signal or token until it becomes idle. (So the detection does not start until the root process becomes idle for the first time.) We do not add the number of warning messages, which is 7 in our case, to the result of CKV algorithm since we assume the algorithm is initiated immediately as the computation starts (or 56 if we follow closely to original CKV algorithm). This slight edge of CKV algorithm does not change the result of simulation.

We assume the system is strongly connected so the 8-node spanning tree consists of one root and 7 leaves.
Table 2: Varying the (exponential) mean of inter-message time.

<table>
<thead>
<tr>
<th>$\lambda_m$</th>
<th>unbounded snapshot</th>
<th>bounded snapshot</th>
<th>CKV</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.5</td>
<td>64.9</td>
<td>63.2</td>
<td>1454.1</td>
</tr>
<tr>
<td>1.0</td>
<td>33.6</td>
<td>41.3</td>
<td>176.3</td>
</tr>
<tr>
<td>2.0</td>
<td>33.1</td>
<td>29.7</td>
<td>73.4</td>
</tr>
<tr>
<td>3.0</td>
<td>30.3</td>
<td>37.2</td>
<td>47.6</td>
</tr>
<tr>
<td>4.0</td>
<td>28.9</td>
<td>29.0</td>
<td>42.0</td>
</tr>
</tbody>
</table>

$\lambda_m$: inter-message time
$\lambda_i = 10.0$: mean initial load
$\lambda_p = 0.5$: mean processing overhead
$\lambda_d = 0.1$: mean message delay

Table 3: Varying the (exponential) mean of initial load.

<table>
<thead>
<tr>
<th>$\lambda_i$</th>
<th>unbounded snapshot</th>
<th>bounded snapshot</th>
<th>CKV</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.0</td>
<td>35.5</td>
<td>38.6</td>
<td>93.2</td>
</tr>
<tr>
<td>10.0</td>
<td>33.6</td>
<td>41.3</td>
<td>176.3</td>
</tr>
<tr>
<td>20.0</td>
<td>33.6</td>
<td>39.5</td>
<td>312.4</td>
</tr>
<tr>
<td>30.0</td>
<td>32.7</td>
<td>38.1</td>
<td>496.7</td>
</tr>
<tr>
<td>40.0</td>
<td>32.2</td>
<td>36.3</td>
<td>690.1</td>
</tr>
</tbody>
</table>

$\lambda_i$: mean initial load
$\lambda_m = 1.0$: mean inter-message time
$\lambda_p = 0.5$: mean processing overhead
$\lambda_d = 0.1$: mean message delay
Table 4: Varying the (exponential) mean of processing overhead.

<table>
<thead>
<tr>
<th>$\lambda_p$</th>
<th>unbounded snapshot</th>
<th>bounded snapshot</th>
<th>CKV</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.3</td>
<td>28.9</td>
<td>32.7</td>
<td>127.1</td>
</tr>
<tr>
<td>0.5</td>
<td>33.6</td>
<td>41.3</td>
<td>176.3</td>
</tr>
<tr>
<td>0.7</td>
<td>39.2</td>
<td>53.6</td>
<td>249.8</td>
</tr>
<tr>
<td>0.9</td>
<td>57.9</td>
<td>67.0</td>
<td>564.8</td>
</tr>
<tr>
<td>1.0</td>
<td>63.0</td>
<td>79.3</td>
<td>1751.1</td>
</tr>
</tbody>
</table>

$\lambda_p$: mean processing overhead
$\lambda_m = 1.0$: mean inter-message time
$\lambda_l = 10.0$: mean initial load
$\lambda_d = 0.1$: mean message delay

Table 5: Varying the (exponential) mean of message delay.

<table>
<thead>
<tr>
<th>$\lambda_d$</th>
<th>unbounded snapshot</th>
<th>bounded snapshot</th>
<th>CKV</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>33.6</td>
<td>41.3</td>
<td>176.3</td>
</tr>
<tr>
<td>0.2</td>
<td>37.3</td>
<td>47.6</td>
<td>179.8</td>
</tr>
<tr>
<td>0.3</td>
<td>38.3</td>
<td>43.3</td>
<td>189.2</td>
</tr>
<tr>
<td>0.4</td>
<td>35.5</td>
<td>46.4</td>
<td>171.8</td>
</tr>
<tr>
<td>0.5</td>
<td>38.3</td>
<td>42.7</td>
<td>185.3</td>
</tr>
<tr>
<td>1.0</td>
<td>43.4</td>
<td>52.7</td>
<td>188.1</td>
</tr>
</tbody>
</table>

$\lambda_d$: mean message delay
$\lambda_m = 1.0$: mean inter-message time
$\lambda_l = 10.0$: mean initial load
$\lambda_p = 0.5$: mean processing overhead
Table 6: Varying the (normal) mean of initial load.

<table>
<thead>
<tr>
<th>$\mu_I$</th>
<th>$\sigma_I$</th>
<th>unbounded snapshot</th>
<th>bounded snapshot</th>
<th>CKV</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.0</td>
<td>2.5</td>
<td>34.1</td>
<td>41.1</td>
<td>96.7</td>
</tr>
<tr>
<td>10.0</td>
<td>5.0</td>
<td>34.5</td>
<td>43.0</td>
<td>160.1</td>
</tr>
<tr>
<td>20.0</td>
<td>10.0</td>
<td>34.5</td>
<td>34.0</td>
<td>344.0</td>
</tr>
<tr>
<td>30.0</td>
<td>10.0</td>
<td>36.4</td>
<td>42.5</td>
<td>474.1</td>
</tr>
<tr>
<td>40.0</td>
<td>10.0</td>
<td>34.5</td>
<td>37.6</td>
<td>619.3</td>
</tr>
</tbody>
</table>

$\mu_I/\sigma_I$: mean/deviation initial load

$\lambda_m = 1.0$: mean inter-message time

$\mu_p = 0.5(\sigma_p = 0.25)$: mean processing overhead

$\mu_d = 0.1(\sigma_d = 0.05)$: mean message delay

Table 7: Varying the (normal) mean of processing overhead.

<table>
<thead>
<tr>
<th>$\mu_p$</th>
<th>$\sigma_p$</th>
<th>unbounded snapshot</th>
<th>bounded snapshot</th>
<th>CKV</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.3</td>
<td>0.15</td>
<td>30.8</td>
<td>35.7</td>
<td>126.1</td>
</tr>
<tr>
<td>0.5</td>
<td>0.25</td>
<td>34.5</td>
<td>43.0</td>
<td>160.1</td>
</tr>
<tr>
<td>0.7</td>
<td>0.25</td>
<td>40.1</td>
<td>47.7</td>
<td>265.1</td>
</tr>
<tr>
<td>0.9</td>
<td>0.25</td>
<td>51.8</td>
<td>69.8</td>
<td>492.3</td>
</tr>
<tr>
<td>1.0</td>
<td>0.25</td>
<td>70.5</td>
<td>93.1</td>
<td>1481.7</td>
</tr>
</tbody>
</table>

$\mu_p/\sigma_p$: mean/deviation processing overhead

$\lambda_m = 1.0$: mean inter-message time

$\mu_i = 10.0(\sigma_i = 5.0)$: mean initial load

$\mu_d = 0.1(\sigma_d = 0.05)$: mean message delay
Table 8: Varying the (normal) mean of message delay.

<table>
<thead>
<tr>
<th>$\mu_d$</th>
<th>$\sigma_d$</th>
<th>unbounded snapshot</th>
<th>bounded snapshot</th>
<th>CKV</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>0.05</td>
<td>34.5</td>
<td>43.0</td>
<td>160.1</td>
</tr>
<tr>
<td>0.2</td>
<td>0.10</td>
<td>36.4</td>
<td>35.4</td>
<td>170.4</td>
</tr>
<tr>
<td>0.3</td>
<td>0.10</td>
<td>39.2</td>
<td>32.0</td>
<td>167.5</td>
</tr>
<tr>
<td>0.4</td>
<td>0.10</td>
<td>35.0</td>
<td>42.2</td>
<td>170.0</td>
</tr>
<tr>
<td>0.5</td>
<td>0.10</td>
<td>36.4</td>
<td>40.1</td>
<td>168.9</td>
</tr>
</tbody>
</table>

$\mu_d/\sigma_d$: mean/deviation message delay  
$\lambda_m = 1.0$: mean inter-message time  
$\mu_l = 10.0(\sigma_l = 5.0)$: mean initial load  
$\mu_p = 0.5(\sigma_p = 0.25)$: mean processing overhead

4.4.3 Comments

In Table 2, 3, 4 and 5, we assume all distributions are exponential. This is a commonly accepted approach in simulation. However, since we cannot verify this assumption of exponential distribution, we adopt normal distribution on message delay, initial load and processing overhead in Table 6, 7 and 8.

In Table 2, when we extend the inter-sending time between messages, the number of underlying message decreases dramatically as expected, while the number of control messages decreases in a slower pace and is very steady.

In Table 3, the system lasts longer when we increase initial load and it results in more communication. But the complexity of control messages is again showing steadiness and has very little to do with the increasing complexity of underlying messages.
In Table 4, when the processing time of an underlying message is shortened, the message complexity decreases while control message complexity decreases too but in a slower pace. If processing time of underlying messages is larger than inter-sending time, the system will explode since more messages arrive than the process can handle. So $\lambda_p = 1.0$ is the marginal value for a limited system in our simulation.

In Table 5, the complexity of both underlying and control messages seem to be affected very little by message delay time.

Table 6, 7 and 8 do not change the figures much.

As we can see in these tables, both bounded and unbounded snapshot-based algorithm keep a very steady order of control messages. Only when the processing overhead increases do they increase by a little. They are barely affected by other parameters. On the other hand, CKV algorithm is sensitive to inter-message time, initial load and processing overhead parameters: the longer the system is, the more control messages it requires.

If we mix up those varied parameters, we can expect the same result: the complexity of control messages will always be steady and more likely in the low order of $N$. When it increases, the complexity of underlying message would increase in a much faster pace. Table 2 – 8 show this tendency and are consistent with what we get from theoretical analysis in Table 1.

Both the unbounded and bounded snapshot-algorithms do better than CKV
algorithm in all aspects we evaluated on, while the unbounded algorithm does slightly better than the bounded one. The reason for the first situation is obvious since we expect the worst case complexity for a snapshot-based algorithm rarely occurs. As to the second situation, it is because the bad case which bounded algorithm covers well does not occur in simulation: an underlying message takes a long delay during transmission thereafter result in more rounds of control waves. We used system-provided random number in generating message delay. So an underlying message is unlikely to travel longer than all control messages in a wave. It then sounds that bounded algorithm should do much worse than unbounded since it issues extra control message. This is compensated by issuing few rounds of white signals due to the red signals. Therefore bounded algorithm is still doing better than unbounded algorithm occasionally. We expect the unbounded snapshot-based algorithm outperforms bounded algorithm when the spanning tree of the system has many levels.

4.5 Experiment

As simulation is still a mixture of theoretical and practical approach for analysis, we are interested in what may happen in the world of pure reality. We conducted experiment.

A 0/1 knapsack problem is solved by a distributed branch-and-bound algorithm running on several Sun workstations based on the concept used in DIB [15]. In
this real-world experiment, we examine the number of control messages issued by the three termination detectors as in last section: the unbounded snapshot-based algorithm, the bounded snapshot-based algorithm, and the CKV algorithm. In Section 4.5.1, we briefly review the basic algorithm underlying the operation of DIB – a Distributed Implementation of Backtracking. In Section 4.5.2, the experiment model is described. The experiment result and a few observations are given in Section 4.5.3.

4.5.1 DIB

DIB – a Distributed Implementation of Backtracking, is a general-purpose package which supports the class of problems that use tree-traversal algorithms. The basic idea is to arbitrarily divide initial work among processes. When process \( p_i \) finishes the work it was given, it sends a request for work to another process \( p_j \). If \( p_j \) is still working, it divides its work and sends part of it to \( p_i \). Otherwise it forwards the request to another process or discards it.

The underlying distributed algorithm has following characteristics:

- Each process maintains two table, WorkGotten and WorkGiven. WorkGotten records problems for which this process is responsible for and WorkGiven records problems the process has distributed. When a process finishes a problem, it reports to its parent which then removes the entry from WorkGiven. Each process can determine which work it is responsible for is still outstanding-
ing and to redo it in case of failure.

- When a process has finished a problem and reported its result, it picks an unassigned problem from WorkGiven. If there is no entry in WorkGiven, the process sends a request to a helper. There are three algorithms for choosing helpers and each works best in certain environments:

  - Assigned initially. Process \( p_i \) has a fixed helper \((i \mod n) + 1\). If the helper cannot grant the request, it forwards the request to its successor.

  - Selected randomly. A process sends requests for work to \( k \) other processes selected at random when it becomes idle.

  - Assigned initially in cyclic order. A process sends a request for work to the next helper in cyclic order, as in the first method, but not to forward requests.

- As to fault tolerance, when a process finishes its work and its WorkGiven table is not empty, it undoes some outstanding work in WorkGiven while waiting for another machine to grant its request.

4.5.2 Experiment Model

We modified the Least Cost branch-and-bound technique for 0/1 knapsack problem given in [19, page 388-397] for a distributed environment described in [15]. Initially, all processes are connected through "sockets" [1] to form a fully connected
distributed system, and waiting for computation to start. The process which is assigned the parameters of the problem sends initialization messages to all other processes. This starting process then divides the problem into two subproblems which will be broken down into subproblems recursively. A subproblem will survive only if it satisfies certain conditions of a cost function. If the process which breaks the problem finds there is no subproblem in its queue, it will keep them for itself. Otherwise it will assign them to two designated helpers. A process stores arriving subproblems in a table (as WorkGotten table in [15]) and works on them. It also checks mails from time to time for updated information and new subproblems coming in.

Most strategy used in the experiment followed from DIB described in last section. But there are still some major differences. Firstly, our system does not handle fault tolerance which is a feature of DIB. That is, a process will not redo the subproblem it assigned to its helpers in any circumstance. Secondly, each process does not report to its parent when the computation it inherited from its parent process is done. This is not necessary as we do not consider fault tolerance and this can also simplify each process which does need to keep a WorkGotten table any more. Each process broadcasts when it comes up with a new value for the cost function in branch-and-bound algorithm or whenever it thought it got a possible solution for the knapsack problem. Thirdly, a process does not send request to other processes for work when it becomes idle. Load balancing is not an issue in
our experiment. However, we did try to balance load by designating appropriate helpers to each process. In the experiment, process $i$ have process $i + 1 \pmod{N}$ and process $i + 3 \pmod{N}$ as its helpers. We believe assigning $i + 3$ instead of $i + 2$ as a helper can help load balance a little bit, especially when we look at a solution path from root to leaf. If we use $i + 2$, the path is more likely to have a cycle of length 4 (i.e., 4 processes are involved in this solution path) due to the characteristic of 0/1 knapsack problem (once the pack is almost full, all remaining nodes on the path form a skewed subtree). If we choose $i + 3$, then all 8 processes will appear in the cycle.

Programs are written in C and run on Sun 3/50 workstations running 4.2 BSD Unix system and connected through Ethernet network with TCP/IP protocols. Processes share data files stored on Network File System but do not share variables. The interprocess communication is made through sockets and is fully connected among processes. Number of processes in the experiment is 8 which is also the problem size of the analysis made in previous sections. Each process resides on a workstation in our experiment.

We solve the problem with (array) size 50, 100, 150 and 200. Each profit and weight is randomly generated between 1 and 1000. Knapsack capacity is randomly generated between 5000 and 20000 for problem size 50 so we can expect half of the objects will be included in the knapsack. The numbers are adjusted accordingly when problem sizes are different. We ran 30 sets of such random data for each
Table 9: Result from experiment with different problem size.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>32.67</td>
<td>34.13</td>
<td>51.27</td>
</tr>
<tr>
<td>100</td>
<td>36.40</td>
<td>43.50</td>
<td>63.97</td>
</tr>
<tr>
<td>150</td>
<td>45.73</td>
<td>54.93</td>
<td>116.53</td>
</tr>
<tr>
<td>200</td>
<td>48.07</td>
<td>52.67</td>
<td>155.52</td>
</tr>
</tbody>
</table>

A: unbounded snapshot algorithm  
B: bounded snapshot algorithm  
C: CKV algorithm

problem size.

4.5.3 Comments

In Table 9, one can see that the average control message complexity of CKV algorithm is clearly worse than both the basic and the improved snapshot-based algorithms in all circumstances. The margin is getting bigger when problem size grows larger and the underlying message are issued more. When we look at raw data, it is again true that only when the number of underlying messages is small does CKV algorithm have a chance to be better than the others.

Recall that the CKV algorithm needs to send warning messages through each channel to make it function correctly, unless we assume the detection starts as soon as the system does. In our example, if the CKV algorithm does not starts immediately, then 7 should be added to column C in Table 9.
4.6 Comparison and Analysis

In all three approaches of evaluation, we assume the termination detector starts as soon as the system starts. Actually, this is not necessary in real system detection. As in the knapsack application of problem size 50, we know the tree will grow to level 50 sooner or later, we can set a triggering event such as reaching level 40 and do not start the detection until the event occurs. Alternatively, we can delay the issuing of a wave, say by one second, when it is to be issued. This delay will allow more messages to arrive and result in fewer control messages.

However, these strategies seem to be more appropriate for snapshot-based algorithms if we don’t want to issue extra control messages which are needed for CKV algorithm to clear channels. But the snapshot algorithms do need to keep counting underlying messages at the very beginning, as does CKV algorithm require each process to number its messages if it is applied to systems with non-first-in-first-out channels.

All evaluation showed that both snapshot-based algorithms are usually, if not always, better than CKV algorithm. This is more obvious when we look at the average figures. CKV algorithm is better than snapshot-based algorithms only when the number of underlying messages issued is small. This is exactly what we concluded in the probabilistic analysis.

While comparing unbounded and bounded snapshot-based algorithm, neither
one dominates the other. But the unbounded algorithm performed better than or as good as the bounded algorithm in about 3/4 occasions of simulation and experiment, although the margin is small.

4.7 Conclusion

We evaluated three distributed termination detection algorithms: a basic snapshot-based algorithm with unbounded control message complexity, an improved snapshot-based algorithm with bounded message complexity $O(MN)$, and the algorithm by Chandrasekaran et al. with optimal complexity $O(M)$. We did the evaluation in three ways, from a very theoretical one to a very practical one. We started from a random distributed system which is based on probabilities. Then we ran simulation which is a mixture of theoretical and practical approaches. Finally we solved a knapsack problem in a distributed environment as a real-world experiment.

The analysis in a random distributed system showed that the number of control messages in a snapshot-based algorithm is either close to a low order of the number of processes in the system, or grows much slower comparing to the number of underlying messages, which happens to be the number of control messages in CKV algorithm, when system lasts longer. The result from simulation backed up the theoretical analysis. In real-world experiment, although the chosen problem is just a particular one, it is still consistent with what we got in the other two analysis approaches.
Snapshot-based algorithms performed better than CKV algorithm in most cases. While the latter did better in situations which underlying messages are extremely few. There is no clear cut on which is better in what circumstances. But it is obvious that snapshot-based algorithms are simple and flexible, even though they are theoretically worse in the unlikely worst cases.

The basic snapshot-based algorithm did better than the more complicated snapshot-based algorithm in most occasions. But the difference is not so clear as in comparing snapshot-based algorithm and CKV algorithm.

The random distributed system can be improved to be a more reliable one if we complicate each global state to be an \((n + c)\)-tuple, where \(n\) is the number of processes in the system and \(c\) is the number of channels. Then we can rule out the possibility of undelivered messages caused by the unreliability of underlying network. But we believe this will not affect the result by much.

4.8 Proof of Theorem 10

Before proving Theorem 10, we introduce some notations and quote a theorem from [21].

In a two-type branching process, let \(U_n\) and \(V_n\) be the number of individuals of type \(I\) and \(II\), respectively, in the \(n\)th generation. If the process begins with a single individual, then either

\[ U_0 = 1 \quad \text{and} \quad V_0 = 0 \]
or

\[ U_0 = 0 \text{ and } V_0 = 1. \]

We introduce the pair of two-dimensional probability generating functions

\[
\phi_n^{(1)}(s, t) = \sum_{k,l=0}^{\infty} P\{U_n = k, V_n = l|U_0 = 1, V_0 = 0\} s^k t^l,
\]

\[
\phi_n^{(2)}(s, t) = \sum_{k,l=0}^{\infty} P\{U_n = k, V_n = l|U_0 = 0, V_0 = 1\} s^k t^l.
\]

Let

\[
m_{11} = E\{U_1|U_0 = 1, V_0 = 0\}
\]

\[
m_{12} = E\{V_1|U_0 = 1, V_0 = 0\}
\]

\[
m_{21} = E\{U_1|U_0 = 0, V_0 = 1\}
\]

\[
m_{22} = E\{V_1|U_0 = 0, V_0 = 1\}
\]

and introduce the matrix of expectations

\[
M = \begin{pmatrix}
m_{11} & m_{12} \\
m_{21} & m_{22}
\end{pmatrix}
\]

(4.7)

Thus \( m_{11} \) and \( m_{12} \) are the expected numbers of offspring of type I or type II, respectively, produced by a single parent of type I.

For the two-dimensional branching process, the extinction probabilities

\[
\pi^{(1)} = P\{U_n = V_n = 0 \text{ for some } n|U_0 = 1, V_0 = 0\},
\]

\[
\pi^{(2)} = P\{U_n = V_n = 0 \text{ for some } n|U_0 = 0, V_0 = 1\}.
\]
We use the following vector notations for convenience

\[ \Phi(u) = (\varphi^{(1)}(s, t), \varphi^{(2)}(s, t)) \]

\[ \pi = (\pi^{(1)}, \pi^{(2)}). \]

**Theorem 11** [21] Assume that the components of \( \Phi(u) \) are not linear functions of \( s \) and \( t \) and that \( M \gg 0 \) (every element of \( M \) is positive). Then \( \pi = 1 \) if the largest eigenvalue \( \rho \) of \( M \) does not exceed one and \( \pi \ll 1 \) if \( \rho > 1 \). (The notation \( u \ll v \) signifies that \( v - u \) has positive (nonnegative) components.)

**Proof of Theorem 10.** We will use a two-type branching process \( R' \) as our asymptotic system whose termination will assure the termination of the original random distributed system \( R \). However, the reverse is not true.

We create another random distributed system \( R' \) which have infinite processes. In \( R' \), a message always wakes up an idle process and no two messages will be addressed to the same process. We want to show that \( R' \), under certain conditions, will terminate with probability one. Since the number of messages in \( R \) is always the same as in \( R' \) while the number of active process is always no more than \( R' \), it is obvious that \( R \) will terminate with probability one under the same condition as \( R' \).

System \( R' \) can be viewed as a branching process: processes are type I population and messages are type II population. Each process gives birth to a process and a message with probability \( q + r \) and \( r \), respectively. Each message gives birth
to a message and a process with probability $d$ and $1 - d$, respectively. The total number of active processes and number of messages in transit at any time is the population of each generation in the branching process. We want to know under what condition will the population extinct.

Whether the initial population is one or $N$ will not affect the condition which guarantees the branching process to terminate. We assume the initial population in $R'$ is one.

Based on the definition of random distributed system, we define the following probabilities

$$P\{U_1 = 0, V_1 = 0|U_0 = 1, V_0 = 0\} = p$$
$$P\{U_1 = 1, V_1 = 0|U_0 = 1, V_0 = 0\} = r$$
$$P\{U_1 = 2, V_1 = 0|U_0 = 1, V_0 = 0\} = q(1 - d)$$
$$P\{U_1 = 1, V_1 = 1|U_0 = 1, V_0 = 0\} = qd$$
$$P\{U_1 = 0, V_1 = 1|U_0 = 0, V_0 = 1\} = d$$
$$P\{U_1 = 0, V_1 = 0|U_0 = 0, V_0 = 1\} = (1 - d)p$$
$$P\{U_1 = 1, V_1 = 1|U_0 = 0, V_0 = 1\} = (1 - d)q$$
$$P\{U_1 = 1, V_1 = 0|U_0 = 0, V_0 = 1\} = (1 - d)r$$

Note that we are observing the branching process every two steps as our generation population.
The generating functions are

\[ \varphi_1^{(1)}(s, t) = p + qdst + q(1 - d)s^2 + rs \]

\[ \varphi_1^{(2)}(s, t) = (1 - d)p + (1 - d)rs + dt + (1 - d)qst \]

and the components of the matrix of expectation are

\[ m_{11} = r + qd + 2q(1 - d) \]

\[ m_{12} = qd \]

\[ m_{21} = (1 - d)(q + r) \]

\[ m_{22} = d + (1 - d)q \]

Since the components of \( \Phi(u) \) are not linear functions of \( s \) and \( t \) and \( M \gg 0 \), we can apply Theorem 11 to our branching process \( R' \).

The eigenvalue of Eq. 4.7 is obtained by solving \( det(M - \lambda I) = 0 \), or, equivalently,

\[ det \begin{bmatrix} r + qd + 2q(1 - d) - \lambda & qd \\ (1 - d)(q + r) & d + (1 - d)q - \lambda \end{bmatrix} = 0 \]

which is a quadratic equation of \( \lambda \). Using the formula for the root of quadratic equation, i.e.

\[ \lambda = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \]

The largest eigenvalue should not exceed one. Take the "+" sign in above formula, let \( \lambda \leq 1 \) and solve for the equation. After some calculation, we come up with

\[(p - q)(1 - d)(1 + qd - q) \geq 0.\]
Since $d$, the probability of message delay, and $q$, the probability of sending messages, are less than 1, $p$ must be greater than or equal to $q$ to satisfy the equation. □
CHAPTER V

CONCLUSION

This dissertation covered three topics in distributed algorithms. Firstly, we proposed a distributed snapshot algorithm which is more message-efficient than the previous one proposed by Chandy and Lamport. Secondly, we designed termination detection algorithms for broadcasting networks. Thirdly, we studied the performance of some distributed termination detection algorithms for point-to-point communication systems.

Due to the lack of shared memory and global clock in a distributed system, it is always difficult for a process to judge the system status from limited information. Distributed snapshot provides a clear concept on the relationship among local process states, global systems states, and points in a distributed computation. We extended a snapshot algorithm for first-in-first-out channels to one which is for more general models. Our algorithm also requires fewer control messages than the original one. We also pointed out that some stable properties can be detected even with an un-coordinated snapshot algorithm. This can further simplify the
algorithm we proposed.

Termination detection is one of the stable properties people are interested in. Most proposed termination detectors are designed for point-to-point communication systems. Although they can be modified for broadcasting communication systems, it would be more efficient to have a specially designed algorithm which takes full advantage of broadcasting power. We proposed two algorithms for broadcasting networks. We also showed they were optimal algorithms as far as message complexity is concerned.

Termination detectors designed for point-to-point communication system are usually labeled by their worst case message complexity. No one has tried to compare those algorithms in an average case. We are interested in doing the comparison of three representative algorithms: a simple snapshot-based algorithm, an improved snapshot-based algorithm, and the one proposed by Chandrasekaran et al. The snapshot-based algorithms are simple but look bad in the worst case, while the CKV algorithm are more complicated but message-optimal in the worst case. We believe the behavior of a termination detector in an average case should be a more important factor than the worst case performance when doing the comparison.

We started the evaluation from proposing a probabilistic model. But probabilistic model alone is not convincing enough since it is a pure theoretical analysis. No such system exists in the real world. We then ran a simulation to bridge the
gap between pure theory and the real world. Simulation is a mixture of theory and experiment. It simulates a real system but still assumes some activities to occur according to some statistic distribution.

The last step in our comparison was running an experiment on a distributed environment. We solved a 0/1 knapsack problem with a distributed branch-and-bound algorithm on several Sun workstations and observed the result of the experiment.

The theoretical approach and simulation strongly indicated that snapshot-based detectors are better than the CKV detector in general. In the branch-and-bound experiment, it showed that the particular problem we chose will have some impact on the performance of different termination detector. But in any case, the snapshot-based detector can always outdo the CKV detector.

The probability model seems to be an interesting tool in quantifying activities in a system. With it, it should be possible to analyze the behavior of other algorithms in a system which is not necessarily distributed.

In the algorithm proposal and performance evaluation, we assume that all messages will be delivered to their destinations without duplication and without any error. This is an ideal assumption and is unlikely in the real world. The importance of fault tolerance will become more obvious when systems get more and more complicated as technology evolves. We believe design of fault tolerant termination detection algorithms will be an important issue in the future.
Bibliography


