

A General Model for Detecting Distributed Termination in Dynamic Systems

Xinli Wang

Department of Computer Science, Michigan Technological University, Houghton, MI 49931 USA
Email: xinlwang@mtu.edu

Abstract

A symmetric algorithm is proposed for detecting distributed termination in a dynamic system with asynchronous communication networks. Correctness of the algorithm is proven. The network model exploited in the algorithm is more general and more suitable for the computations on internet networks. It is more efficient than those in previous works in terms of control messages used in the algorithm.

Key words: distributed termination detection, general model

1 Introduction

The distributed termination problem is to detect whether a computation within a distributed system has terminated[11, 30]. It is a fundamental problem in distributed computations. Since the works of Dijkstra and Scholten[10] and Francez[11], many elegant algorithms have been proposed to solve the problem in distributed systems. These include the solutions for static systems[12, 13, 30, 31, 24, 15, 14, 17, 21] and dynamic systems[10, 28, 5, 18, 8, 6]. Some algorithms take advantage of a global time base in the system[30, 26, 7]. While most of the solutions are for non-faulty systems, some of them are fault-tolerant[1, 33, 20, 32, 7]. Works in[22] discuss how algorithms can be implemented in a mobile system more efficiently in terms of power usage. Although the so-called “probe waves” are used in most of the algorithms to detect the global state, techniques of credit distribution/recovery[25, 16, 32] and markers[27] are also employed. Message-optimal algorithms are proposed under certain assumptions on the network topology and load decomposition in the computations[3, 19, 21, 2]. A good review and taxonomic discussion can be found in[23].

For dynamic systems, the network models can be broadly classified as (1) a tree[10, 28, 8]; (2) a ring[6]; and (3) a general network[18]. In the tree model, one process is typically designated to detect the global state. Based on the diffusing tree model, [8] presents a symmetric algorithm; it requires that the initial processes are connected to each other with bi-directional links. The ring model is simple and symmetric; however, a logic ring must be maintained during the entire computation. Though the solution in[18] is not restricted to any specific

network model, it uses distributed snapshots[29] to collect the global information, which is not efficient for the termination detection problem[24].

In this paper, we take the advantages of both the tree and ring network models and propose a symmetric solution for the termination detection problem in a dynamic system that has asynchronous communications, while its network model is more general. The algorithm is more efficient in terms of the number of messages used in the detection protocol. Furthermore, it is (or close to) message-optimal in most cases. We will first describe the network model and then the algorithm is presented.

2 System Model

A distributed system consists of a finite number of processes which cooperate and coordinate through communications with one another by message passing. A distributed computation being performed in such a system is known as a *basic computation* and inter-process messages used in this computation are recognized as *basic messages*. The basic computation consists of a number of tasks. Each of the processes in the system is assigned with some of the tasks. We call the tasks assigned to a process the local tasks or work in this process. On the basic computation, a *control computation* is superimposed for termination detection. Messages used in the control computation are known as *control messages*. We assume that processes in the system are identified with integer numbers. The notation p_i represents a process which has an ID of i .

The basic computation starts with a nonempty set of processes. These processes are known as *initial processes*. Let P_0 be the set of the initial processes and $M = |P_0|$. During the computation, each initial process can create one or more processes which may further create other processes. An external process can participate in the computation at any time by sending a *joining message* to one of the active processes in the system. For the first step, we do not allow a process to be destroyed until the basic computation terminates and there is no faulty process in the system.

The network model in this system is a combination of a logic ring and a number of computation trees (Figure 1). The initial processes are connected with a logic ring. For initial processes p_i and p_j , p_j is the neighbor of p_i if and only if $j = (i + 1) \% M$. We assume that either p_i knows the value of M or it has the knowledge of its neighbor on the ring if p_i is an initial process. This can be done at compile time. Computation trees are established when new processes are created, external processes are accepted in the system, or basic messages are sent out to other processes.

When process p_i creates p_j , we say that p_i is dependent on p_j and p_j is related to p_i . The same rule applies when an external process p_k joins the system or when a process sends a basic message to another process. If p_k sends a joining request to p_i for participating in the computation and p_i accepts p_k , then p_i is dependent on p_k and p_k is related to p_i . When process p_i sends a basic message to p_j , p_j is related to p_i upon receiving this message and p_i is dependent on p_j . Each process will maintain a *dependent set*, DS , to hold the processes on which it is dependent and a *related set*, RS , to hold the processes to which it is related. A

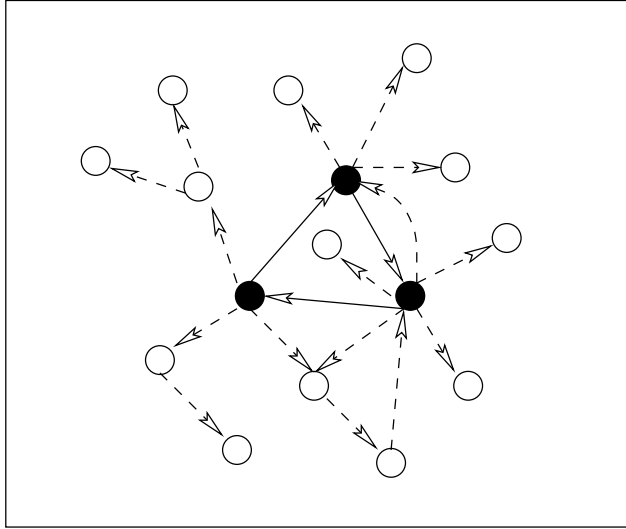


Figure 1: An example of the network topology. The three filled circles are initial processes and they are connected with solid lines on the logic ring. Open circles are created or joined processes. Computation trees are shown in dashed lines.

subscript to them will indicate which process they belong to. DS_i and RS_i is then, respectively, the dependent set and related set of p_i .

A computation tree is defined recursively. If $p_j \in DS_i$, then p_j is in the computation tree rooted with p_i . If $p_k \in DS_i$ and p_i is the process from which p_k receives a basic message for the first time, then p_k as well as processes on the computation tree rooted with p_k are on the computation tree rooted with p_i . In addition, the height of the tree rooted with p_i is one level higher than that of the tree rooted with p_k . By this definition, a process may belong to multiple computation trees if it receives basic messages from multiple processes.

Two processes p_i and p_j are connected if and only if $p_j \in DS_i$, or $p_i \in RS_j$, or p_i and p_j are initial processes and they are neighbors on the ring. Beyond this, a process does not have to have any other global knowledge about the system. There is a bi-directional link between any pair of connected processes. Communications through the link are asynchronous and reliable. Messages are delivered within an arbitrary but finite delay.

A process can be in one of the three states: *active*, *passive* and *terminable*. A process is active if it is currently engaged in the basic computation, working on some tasks of the basic computation. A process in the active state can send and receive basic messages, create processes, and accept external processes which are joining the system. When it completes its local work for the basic computation, it enters the passive state. While in the passive state, a process may become active again upon receiving a basic message from another active process or it may receive other control messages and stay in the passive state. When all of the processes in a computation tree rooted with a process become passive, this process becomes terminable. That means this process is ready to terminate. A terminable process can become active again upon receiving a basic message. A process in the passive and terminal states can

neither create new processes nor accept external processes that are joining the system.

A basic computation terminates when all processes involved are in the terminable state and no basic message is in transition. This is known as termination condition.

3 The Algorithm

In this section, we first define the messages and variables used to support the implementation of the algorithm. Then, the algorithm is presented.

3.1 Messages and Variables Used in the Algorithm

We define the following messages in implementing the algorithm:

$Bmsg(i, j, seq_j)$: A basic message sent from process p_i to p_j . The variable seq_j is the sequence number of the basic messages sent from process p_i to p_j . It starts with zero.

$Pmsg(i, k, s)$: A passive message sent from process p_i to p_k . This message informs process p_k that process p_i has done its local work and now is in the passive state. The variable s is the sequence number in the basic message which process p_k sent to p_i for the last time.

$Tmsg(i, k, s)$: A terminable message sent from process p_i to p_k . It informs p_k that p_i is in the terminable state. The variable s has the same meaning as in $Pmsg(i, k, s)$.

$TDmsg(i, j)$: A termination detection message. It is sent from process p_i to p_j to detect the termination condition.

The following variables are maintained in the process p_i :

DS_i : Dependent set. It consists of pairs of (j, seq_j) , where j is the ID of the process to which p_i has sent a basic message and seq_j is the sequence number stamped in this message. No matter how many times p_i has sent a basic message to p_j , at most one pair of (j, seq_j) exists in DS_i , in which seq_j is the sequence number in the basic message that p_i sent to p_j for the last time. In other words, seq_j is an accumulated sequence number.

RS_i : Related set. Similar to DS_i , RS_i consists of pairs of (j, s) , where j is the ID of the process from which p_i has received a basic message and s is the sequence number in this message. At most one pair of (j, s) exists in RS_i , in which s is the sequence number in the basic message that p_i receives from p_j for the last time.

FR_i : First related process. A process from which p_i receives a basic message for the first time. It is a pair of (j, s) , where j is the process ID and s the sequence number in a basic message p_j sent to p_i for the last time.

$State_i$: State of p_i . It has the domain of $\{active, passive, terminable\}$.

$seqNum_i[j]$: Sequence number used in a basic message sent to p_j , $j \in DS_i$. p_i maintains a sequence number for each of the processes in DS_i .

I_i : Indicator of whether p_i is an initial process. We assume that $I_i = 1$ if p_i is an initial process and $I_i = 0$ if not.

3.2 The Protocol of the Algorithm

The basic idea exploited in the algorithm is as follows. When a process has completed its local work for the basic computation, it sends a passive message to each of the processes in its related set. After that, the process stays in the passive state until it has received a passive or terminable message from every process to which it had sent a basic message. Then the process sends a terminable message to the process from where it receives a basic message for the first time. At this point, this process sends a termination detection message to its neighbor if it is an initial process. When an initial process receives a termination detection message which was sent out from itself, it declares a termination. A termination can be detected only by the initial processes.

The algorithm is presented as the protocol for process p_i in Table 1. Actions process p_i should take upon some event are enumerated in the protocol according to the states it is in when the event occurs. If actions are not specified in the protocol for a particular event, the event is ignored.

Table 1 The protocols for process p_i

$State_i$	event	actions
Active	send p_j basic message or create process p_j or receive a joining request from p_j	$seqNum_i[j] \leftarrow seqNum_i[j] + 1;$ $DS_i \leftarrow DS_i - \{(j, -)\};$ $DS_i \leftarrow DS_i \cup \{(j, seqNum_i[j])\};$ send $Bmsg(i, j, seqNum_i[j]);$
	finish local work	for $\forall(k, s), \{(k, s)\} \in RS_i$ send $Pmsg(i, k, s);$ if($DS_i = \emptyset$) Set_Terminable ; else $State_i \leftarrow Passive;$ DeadlockQ ;
	receive $Pmsg(j, i, s)$ or $Tmsg(j, i, s)$	$DS_i \leftarrow DS_i - \{(j, s)\};$
Active or Passive	receive $Bmsg(j, i, s)$	if($(FR_i = \emptyset) \vee ((j, -) \in FR_i)$) $FR_i \leftarrow \{(j, s)\};$ else $RS_i \leftarrow RS_i - \{(j, -)\};$ $RS_i \leftarrow RS_i \cup \{(j, s)\};$ if($State_i = Passive$) $State_i \leftarrow Active;$
Passive	receive $Pmsg(j, i, s)$ or $Tmsg(j, i, s)$	$DS_i \leftarrow DS_i - \{(j, s)\};$ if($DS_i = \emptyset$) Set_Terminable ; else DeadlockQ ;
Terminable	receive $Bmsg(j, i, s)$	$FR_i \leftarrow \{(j, s)\};$ $State_i \leftarrow Active;$
	receive $TDmsg(j, i)$	if($i = j$) declare a termination; else forward $TDmsg(j, i)$ to $p_{(i+1)\%M};$

A process being created or an external process joining the system is in the passive state before any basic message is received. When an active process p_i creates process p_j or receives a joining request from an external process p_j , p_i sends a basic message to p_j and puts p_j into DS_i , indicating that the completion of tasks in p_i for the basic computation is now dependent on the completion of tasks in p_j . This is the same action as when p_i needs to send a basic message to p_j . In addition to the sender and receiver IDs, a basic message is stamped with a sequence number. This sequence number is used to record how many basic messages process

p_i has sent to p_j . It is useful in determining whether process p_j has finished tasks that are assigned or resulted when basic messages from p_i are received.

An active process p_i sends out a *Pmsg* message to each process in its related set when it finishes its local tasks for the basic computation. At this point, process p_i can enter two different states according to the situations it is in:

(1) If its dependent set is empty, it enters the terminable state and sends a *Tmsg* message to the process where it receives a basic message for the first time because all of the processes on the computation tree rooted with p_i are in the passive or terminable state. In addition, if p_i is an initial process, it informs its neighbor on the ring of this situation by sending a termination detection message. These actions are done through function **Set_Terminable** which is given in Table 2.

(2) If its dependent set is nonempty, then at least one active process exists on the computation tree rooted with p_i . The computation cannot terminate. Process p_i enters the passive state and waits there for the completion of tasks in other processes. At this point, however, process p_i might be involved in a deadlock. Since a process sends *Tmsg* message to the process from which it receives a basic message for the first time only when its dependent set is empty, p_i and p_j may be in a circular waiting situation if p_i is first related to p_j and p_j is first related to p_i . In this case, to break the deadlock, we force p_i to send a *Pmsg* message to p_j if $i > j$. These actions are encapsulated in function **DeadlockQ** which is given in Table 2. The function does nothing if there is more than one process in the dependent set of the process which invokes it.

Table 2 Functions of **Set_Terminable** and **DeadlockQ**

<pre> procedure Set_Terminable $State_i \leftarrow Terminable;$ send $Tmsg(i, k_0, s_0) : \{(k_0, s_0)\} = FR_i;$ if($I_i = 1$) send $TDmsg(i, (i + 1)\%M);$ </pre>
<pre> procedure DeadlockQ if($DS_i = 1$) $k \leftarrow j : \{(j, -)\} = FR_i;$ $l \leftarrow d : \{(d, -)\} = DS_i;$ if($k = l$) if($i > k$) send $Pmsg(i, j, s) : \{(j, s)\} = FR_i;$ </pre>

When an active process p_i receives a *Pmsg* message or a *Tmsg* message from process p_j , it removes this process from its dependent set if this process has completed the task resulted from the basic message that process p_i sent to it for the last time. Otherwise, this message is discarded. This is done by using the sequence number in the *Pmsg* and *Tmsg* messages. The sequence number used in the *Pmsg* and *Tmsg* messages serves as an accumulative acknowledgment to the *Bmsg* messages a process has received from another. When process p_i in the active state receives a basic message from process p_j , it records p_j as the process to which p_i is first related if this is the first basic message to process p_i . Otherwise, process p_i puts this message in its related set.

A process in the passive or terminable state becomes active again when it receives a basic message, reflecting that the basic computation has not terminated yet and it is assigned some new tasks by the system. When a passive process p_i receives a $Pmsg$ message or a $Tmsg$ from process p_j , it first removes p_j from its dependent set if p_j has finished the task resulted from the basic message that p_i sent to it for the last time. Then p_i enters the terminable state, if its dependent set is empty, sends a $Tmsg$ message to the process to which p_i is related for the first time and a termination detection message to its neighbor if p_i is an initial process. If its dependent set is nonempty, it might be involved in a deadlock with the same reason as described above when p_i in the active states completes its local tasks. Therefore, it invokes function **DeadlockQ** to avoid a deadlock.

When a termination detection message arrives at a terminable process, if this message is sent out from itself, it declares a termination; otherwise, it forwards this message to its neighbor on the ring. A process in the passive or terminable state does not respond to a joining message from an external process because no task can be assigned to the external process.

4 Correctness Arguments

To prove the correctness of the algorithm, we are going to show that when a basic computation terminates, the algorithm will declare the termination; on the other hand, if the algorithm declares a termination, the basic computation has terminated.

Lemma 1 *A process is in the terminable state if and only if*

- (1) *Every process on the computation tree rooted with this process is in the passive or terminable state;*
- (2) *No basic message is in transition which is sent out from a terminable process on the computation tree; and*
- (3) *The passive processes on the computation tree must belong to another computation at the same time in the system.*

Proof Let p_i be the process and h be the height of the computation tree rooted with p_i . We will prove Lemma 1 with mathematical induction on h .

Basis: Show that Lemma 1 is true for $h = 1$.

In this case, DS_i is empty and it is a leaf process on the computation tree. According to the algorithm, p_i becomes terminable if and only if it has completed local tasks and no basic message is sent out from it.

Induction hypothesis: For each $n > 1$, assume that Lemma 1 is true for $h = n$.

Induction step: We shall show that Lemma 1 is true for $h = n + 1$.

Suppose $p_j \in DS_i$. Process p_i becomes terminable if and only if it has completed local tasks and $DS_i = \emptyset$. If p_j is in the passive state and sends a $Pmsg$ message to p_i , then p_j must belong to another computation tree at the same time. Otherwise, it should send a $Tmsg$ message and be in the terminable state. If p_j is in the terminable state and sends a $Tmsg$ message to p_i , then no basic message is sent out from processes on the computation tree rooted with p_j by the induction hypothesis. Furthermore, no basic message is sent out from p_i because its dependent set is empty.

On the other hand, if $\forall p_j, \{(j, -)\} \in DS_i$, are either in the passive or terminable state, each of them will send a *Pmsg* or a *Tmsg* message to p_i . Process p_i will be in the terminable state when it completes local tasks for the basic computation. Lemma 1 is then proven.

Theorem 2 *The algorithm declares a termination if and only if the basic computation has terminated.*

Proof From the algorithm we know that a termination is declared only by the initial processes. A termination detection message is sent out by an initial process p_i when it enters the terminable state and this message is forwarded around only by an initial process in the terminable state. When this detection message is circulated around on the logic ring and gets back to the process p_i , then every process on the ring is in the terminable state. In addition, all of the processes in the computation trees rooted with those processes must be in the terminable state and no basic message is in transition according to Lemma 1. At this point, process p_i declares a termination and the basic computation has terminated.

On the other hand, when the basic computation has terminated, all of the initial processes will be in the terminable state according to Lemma 1. At least one of the initial processes will initiate a *TDmsg* message and this message will be circulated around and get back to this process. Then this process will declare the termination. We have then proven Theorem 2.

5 Conclusion and Discussion

We present a symmetric algorithm for distributed termination detection in a dynamic system with asynchronous communication. Correctness proof is given. The number of control messages used in the computation trees is less than or equal to that of basic messages, which is message-optimal[4, 3, 19]. Control messages used in the logic ring circulate the ring one round at most, which is more efficient than those in previous works[30, 9, 26]. Multiple processes on the ring may declare the same termination. As pointed out in[30], instead of being a problem, this situation will accelerate the termination wave.

Compared with previous works[10, 28, 5, 18, 8, 6], the network model is more general and more efficient. It is a combination of a logic ring for the initial processes and a number of computation trees. Any process in the system can be involved in one or more computation trees. If the basic computation starts with only one process, the network is a diffusion tree[10, 28, 5]. On the other hand, if neither a basic message is sent out, nor an external process is accepted, nor a process is created in the entire computation, the network is a logic ring[9, 30, 14, 26]. This network model is more appropriate for a distributed computation in an internet network. There might be multiple networks participating in a distributed computation, which starts with one or more processes in each of the networks. Then each of the initial processes can create more processes within its local network.

In diffusing termination detection[10], a passive process initiates a “probe-wave” for detecting the termination. When this wave arrives at the process where it was initiated, it comes back to this process again. As a result, this wave travels the tree twice, forward and backward, in the system. In our algorithm, the control message travels only in one direction. It starts at the leaf processes and travels upward to the root in a computation tree. The sequence

number in a control message is useful in serving as an accumulative acknowledgment to basic messages.

The algorithm requires that no faulty process exist in the system. It allows creating processes and accepting external processes during the computation, but does not deal with the destruction of processes in the system. This will be discussed in another paper.

Acknowledgments

I would like to thank Dr. Jean Mayo for her comments and suggestions.

References

- [1] Afek, Y. and Saks, M. Detecting global termination conditions in the face of uncertainty. In *Proceedings of Sixth Annual ACM Symposium on Principles of Distributed Computing*, pages 109–124, Vancouver, British Columbia, Canada, 1987. ACM Press.
- [2] Al-Fayoumi, N. I. and Hanson, E. N. Testing for termination of asynchronous parallel computations. In *Proceedings of the 36th Annual Southeast Regional Conference*, pages 201–206. ACM Press, 1998.
- [3] Chandrasekaran, S. and Venkatesan, S. A message-optimal algorithm for distributed termination detection. *Journal of Parallel and Distributed Computing*, 8:245–252, 1990.
- [4] Chandy, K. M. and Misra, J. How processes learn. *Distributed Computing*, 1:40–52, 1986.
- [5] Cohen, S. and Lehmann, D. Dynamic systems and their distributed termination. In *Proceedings of the first ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 29–33, New York, NY, USA, 1982. ACM Press.
- [6] Darling Jr, D. and Mayo, J. Stable predicate detection in dynamic systems. *submitted for publication*, 2003.
- [7] Dash, P. K. and Hansdah, R. C. A fault-tolerant distributed algorithm for termination detection using roughly synchronized clocks. In *Proceedings of 1997 International Conference on Parallel and Distributed Systems (ICPADS'97)*, pages 736–743, Seoul, Korea, 1997. IEEE Press.
- [8] Dhamdhere, D. M., Iyer, S. R., and Reddy, E. K. K. Distributed termination detection for dynamic systems. *Parallel Computing*, 22(14):2025–2045, 1997.
- [9] Dijkstra, E. W., Feijen, W. H. J., and van Gasteren, A. J. M. Derivation of a termination detection algorithm for distributed computation. *Information Processing Letters*, 16(1983):217–219, 1983.

- [10] Dijkstra, W. and Scholten, C. Termination detection for diffusing computations. *Information Processing Letters*, 11(1):1–4, 1980.
- [11] Francez, N. Distributed termination. *ACM Transactions on Programming Languages and Systems*, 2(1):42–55, 1980.
- [12] Francez, N., Rodeh, M., and Sintzoff, M. Distributed termination with interval assertions. In *Proceedings of International Colloquium on Formalization of Programming Concepts, Lecture Notes in Computer Science 107*, pages 280–291, Peniscola, Spain, 1981. Springer-Verlag.
- [13] Francez, N. and Rodeh, M. Achieving distributed termination without freezing. *IEEE Transactions on Software Engineering*, SE-8(3):287–292, 1982.
- [14] Haldar, S. and Subramanian, D. K. Ring based termination detection algorithm for distributed computations. *Information Processing Letters*, 29(1988):149–153, 1988.
- [15] Huang, S.-T. A fully distributed termination detection scheme. *Information Processing Letters*, 29(1988):13–18, 1988.
- [16] Huang, S.-T. Detecting termination of distributed computations by external agents. In *Proceedings of IEEE 9th International Conference on Distributed Computing Systems*, pages 79–84, Newport Beach, CA, USA, 1989. IEEE Press.
- [17] Kumar, D. Development of a class of distributed termination detection algorithms. *IEEE Transactions On Knowledge and Data Engineering*, 4(2):145–155, 1992.
- [18] Lai, T.-H. Termination detection for dynamically distributed systems with non-first-in-first-out communication. *Journal of Parallel and Distributed Computing*, 3:577–599, 1986.
- [19] Lai, T.-H., Tseng, Y.-C., and Dong, X. A more efficient message-optimal algorithm for distributed termination detection. In *Proceedings of Sixth International Parallel Processing Symposium*, pages 646–649, Beverly Hills, CA, USA, 1992. IEEE Press.
- [20] Lai, T.-H. and Wu, L.-F. An $(N - 1)$ -resilient algorithm for distributed termination detection. In *Proceedings of the Fourth IEEE Symposium on Parallel and Distributed Processing*, pages 274–281, Arlington, TX, USA, 1992. IEEE Press.
- [21] Leung, H.-F. and Ting, H.-F. An optimal algorithm for global termination detection in shared-memory asynchronous multiprocessor systems. *IEEE Transactions on Parallel and Distributed Systems*, 8(5):538–543, 1997.
- [22] Matocha, J. Distributed termination detection in a mobile wireless network. In *Proceedings of the 36th Annual Southeast Regional Conference*, pages 207–213. ACM Press, 1998.

- [23] Matocha, J. and Camp, T. A taxonomy of distributed termination detection algorithms. *Journal of Systems and Software*, 43(3):207–221, 1998.
- [24] Mattern, F. Algorithms for distributed termination detection. *Distributed Computing*, 2:161–175, 1987.
- [25] Mattern, F. Global quiescence detection based on credit distribution and recovery. *Information Processing Letters*, 30(1989):195–200, 1989.
- [26] Mayo, J. and Kearns, P. Distributed termination detection with roughly synchronized clocks. *Information Processing Letters*, 52(1994):105–108, 1994.
- [27] Misra, J. Detecting termination of distributed computations using markers. In *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing*, pages 290–294, Montreal, Quebec, Canada, 1983. ACM Press.
- [28] Misra, J. and Chandy, K. M. Termination detection of diffusing computations in communicating sequential processes. *ACM Transactions on Programming Languages and Systems*, 4(1):37–43, 1982.
- [29] nd Leslie Lamport, K. M. C. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, 1985.
- [30] Rana, S. A distributed solution of the distributed termination problem. *Information Processing Letters*, 17(1983):43–46, 1983.
- [31] Topor, R. W. Termination detection for distributed computations. *Information Processing Letters*, 18(1984):33–36, 1984.
- [32] Tseng, Y.-C. Detecting termination by weight-throwing in a faulty distributed system. *Journal of Parallel and Distributed Computing*, 25:7–15, 1995.
- [33] Venkatesan, S. Reliable protocols for distributed termination detection. *IEEE Transactions on Reliability*, 38(1):103–110, 1989.