Improving the Efficacy of a Termination Detection Algorithm

Sathya Peri   Neeraj Mittal

Department of Computer Science

The University of Texas at Dallas,

Richardson, TX 75083

sathya.p@student.utdallas.edu   neerajm@utdallas.edu

A preliminary version of the paper has appeared earlier in a conference proceeding [1].

Abstract

An important problem in distributed systems is to detect termination of a distributed computation. A distributed computation is said to have terminated when all processes have become passive and all channels have become empty. We focus on two attributes of a termination detection algorithm. First, whether the distributed computation starts from a single process or from multiple processes: *diffusing computation* versus *non-diffusing computation*. Second, whether the detection algorithm should be initiated along with the computation or can be initiated anytime after the computation has started: *simultaneous initiation* versus *delayed initiation*. We show that any termination detection algorithm for a diffusing computation can be transformed into a termination detection algorithm for a non-diffusing computation. We also demonstrate that any termination detection algorithm for simultaneous initiation can be transformed into a termination detection algorithm for delayed initiation. We prove the correctness of our transformations, and show that our transformations have minimal impact on the complexity of the given termination detection algorithm.

Index Terms

distributed system, termination detection, algorithm transformation, diffusing and non-diffusing computations, simultaneous and delayed initiations, worst-case and average-case message complexities

I. Introduction

One of the fundamental problems in distributed systems is to detect termination of an ongoing distributed computation. The distributed computation, whose termination is to be detected, is modeled as follows. A process can either be in *active* state or *passive* state. Only an active process can send an application message. An active process can become passive at anytime. A passive process becomes active on receiving an application message. A computation is said to have *terminated* when all processes have become passive and all channels have become empty.

The problem of termination detection was independently proposed by Dijkstra and Scholten [2] and Francez [3] more than two decades ago. Since then, many researchers have worked on this problem and, as a result, a large number of termination detection algorithms have been developed (*e.g.*, [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], [18], [19]). Although most of the research work on termination detection was conducted in 1980s and early

1990s, a few papers on termination detection still appear every year (*e.g.*, [15], [16], [17], [18], [19]).

We focus on two attributes of a termination detection algorithm. First, whether the distributed computation (whose termination is to be detected) starts from a single process or from multiple processes: *diffusing computation* versus *non-diffusing computation*. Second, whether the detection algorithm should be initiated along with the computation or can be initiated anytime after the computation has started: *simultaneous initiation* versus *delayed initiation*.

One of the earliest termination detection algorithms, which was proposed by Dijkstra and Scholten [2], assumes that the underlying computation is diffusing. Shavit and Francez [6] generalize Dijkstra and Scholten's algorithm to work for any non-diffusing computation. Other examples of termination detection algorithms developed for diffusing computation are Mattern's echo algorithm [7], Mattern's credit distribution and recovery algorithm [9] and Huang's weight throwing algorithm [10]. The last two algorithms are based on a similar idea and were proposed independently by Mattern and Huang. While modifications have been proposed to the credit distribution and recovery algorithm so that it can be used for a non-diffusing computation, they are ad hoc in nature and are, therefore, specific to the algorithm.

Matocha and Camp, in their paper on taxonomy of termination detection algorithms, write "[termination detection] algorithms that require diffusion place restrictions on the kinds of computations with which they may execute" [20]. In this paper, we show that the restriction is not rigid and can be easily relaxed. Specifically, we give a transformation that can be used to convert *any* termination detection algorithm for a diffusing computation to an algorithm for detecting termination of a non-diffusing computation. We later use this transformation to enable delayed initiation with any termination detection algorithm.

Note that, when an arbitrary subset of processes may be active initially, any termination detection algorithm must exchange at least $N-1$ control messages in the worst-case, where N is the number of processes in the system. Also, the worst-case detection latency of any termination detection algorithm for a non-diffusing computation, when the communication topology is arbitrary, is $\Omega(D)$, where D is the diameter of the topology. We show that, under certain realistic assumptions, if message-complexity and detection latency of the given termination detection algorithm are $O(C)$ and $O(L)$, respectively, then message complexity and detection latency of the resulting termination detection algorithm are given by $O(C+N)$ and $O(L+D)$, respectively. Observe that the additional overhead is not due to the inefficiency of our transformation but because of the nature of the problem itself.

Chandy and Misra [21] prove that any termination detection algorithm, in the worst case, must exchange at least M control messages, where M is the number of application messages exchanged. As a result, if the underlying computation is message-intensive, then a termination detection algorithm may exchange a large number of control messages. Chandrasekaran and Venkatesan [11] give a termination detection algorithm that can be initiated anytime *after* the computation has started. The advantage of their approach is that the number of control messages exchanged by their algorithm depends on the number of application messages exchanged by the computation *after* the termination detection algorithm began. As a result, their algorithm is especially suited to detect termination of a message-intensive computation for which it is preferable to initiate the algorithm when the computation is “close” to termination. Chandrasekaran and Venkatesan [11] show that, with delayed initiation, any termination detection algorithm must exchange at least E control messages, where E is the number of channels in the communication topology. Moreover, delayed initiation is not possible unless all channels are first-in-first-out (FIFO).

In “strict” delayed initiation, as described by Chandrasekaran and Venkatesan [11], no control information can be maintained about the state of the computation (for example, the number of messages that have been sent or received along a channel) until the termination detection algorithm has started. Lai *et al* [22] consider a variation of “strict” delayed initiation in which some control information is maintained about the state of the computation before the termination detection algorithm has started; however, no control messages can be exchanged until after the termination detection algorithm has actually begun. We refer to this approach as *quasi-delayed initiation*. Lai *et al* [22] modify Dijkstra and Scholten’s algorithm, which assumes diffusing computation and simultaneous initiation, to work with quasi-delayed initiation.

Mahapatra and Dutt [17] observe that the approach used by Chandrasekaran and Venkatesan [11] to achieve delayed initiation is applicable to many other termination detection algorithms as well, especially those based on acknowledgment (*e.g.*, [14]) and message-counting (*e.g.*, [7]). However, from their discussion, it is not clear whether *any* termination detection algorithm designed for simultaneous initiation can be modified to start later without increasing its message complexity in any way (except when required to solve the problem) and, if yes, how. In this paper, we provide two transformations—for delayed and quasi-delayed initiations—that can be used to initiate *any* termination detection algorithm for a non-diffusing computation *later* after the computation has already begun. Moreover, if the *worst-case* message-complexity of the given termination detection algorithm is $O(C)$, then the *worst-case* message complexity of the resulting termination detection algorithm with delayed initiation is $O(C + E)$ and with quasi-delayed initiation is $O(C)$. However, we expect the *average* message complexity to be much lower because fewer number of application messages need to be tracked. The detection latency of the resulting termination detection algorithm, with both types of initiations, is same as that of the given termination detection algorithm.

Besides efficiency, another advantage of delayed initiation is that it can be used to make a termination detection algorithm tolerant to detectable faults such as crash failure in a system equipped with perfect failure detector. For example, consider a distributed system in which processes can fail by crashing. On detecting failure of a process, the termination detection algorithm can be simply restarted and the earlier initiations of the detection algorithm be simply ignored [23].

All our transformations are general in the sense that they do not make any additional assumptions about the communication topology or the ordering of messages (except those made by the given termination detection algorithm or required to solve the problem).

The paper is organized as follows. In Section II, we describe the system model and notation used in this paper and also describe the termination detection problem. We describe and analyze our transformations in Section III and Section IV. We discuss the related work and outline possible directions for future research in Section V.

II. System Model and Problem Statement

A. Model and Notation

We assume an asynchronous distributed system consisting of N processes $P = \{p_1, p_2, \ldots, p_N\}$, which communicate with each other by sending messages over a set of bidirectional channels. We do not assume that the communication topology is fully connected. There is no common clock or shared memory. Processes are non-faulty and channels are reliable. Message delays are finite but may be unbounded.

Processes execute *events* and change their states. An event causes the state of a process to be updated. In addition, a send event causes a message to be sent and a receive event causes a message to be received. Events on a process are totally ordered. However, events on different processes are only partially ordered by the Lamport's happened-before relation [24], which is defined as the smallest transitive relation satisfying the following properties:

1) if events e and f occur on the same process, and e occurred before f in real time then e happened-before f, and
2) if events e and f correspond to the send and receive, respectively, of a message then e happened-before f.

A state of a system can be captured by the set of events that have been executed on each process so far. Not every set of events correspond to a valid state of the system. A set of events form a *consistent cut* if for every event contained in the cut, all events that happened-before it also belong to the cut. Formally,

$$C \text{ is a consistent cut} \quad \triangleq \quad \langle \forall e, f :: (e \rightarrow f) \wedge (f \in C) \;\Rightarrow\; e \in C \rangle$$

Intuitively, a set of events correspond to a valid state of the system if and only if the set forms a consistent cut.

B. The Termination Detection Problem

The termination detection problem involves detecting when an ongoing distributed computation has terminated. The distributed computation is modeled using the following four rules:

Rule 1: A process can be either in an *active* state or a *passive* state.

Rule 2: A process can send a message only when it is active.

Rule 3: An active process can become passive at anytime.

Rule 4: A passive process becomes active on receiving a message.

The computation is said to have *terminated* when all processes have become passive and all channels have become empty.

To avoid confusion, we refer to the messages exchanged by a computation as *application messages*, and the messages exchanged by a termination detection algorithm as *control messages*. Moreover, we refer to the actions executed by a termination detection algorithm as *control actions*. Clearly, it is desirable that the termination detection algorithm exchange as few control messages as possible, that is, it has low message complexity. Further, once the underlying computation terminates, the algorithm should detect it as soon as possible, that is, it has low detection latency. Detection latency is measured in terms of number of message hops. For determining detection latency, it is assumed that each message hop takes at most one time unit and message processing time is negligible.

A computation is said to be *diffusing* if only one process is active initially; otherwise it is *non-diffusing*. If the termination detection algorithm has to be initiated along with the computation, then we refer to it as *simultaneous initiation*. On the other hand, if the termination detection algorithm can be initiated anytime after the computation has started, then we refer to it as *delayed initiation*. In case it is possible to maintain some control information about the state of the computation before the termination detection algorithm has started, we refer to it as *quasi-delayed initiation*. However, even with quasi-delayed initiation, no control messages can be exchanged until after the termination detection has actually begun.

In this paper, we transform a given a termination detection algorithm into a termination detection algorithm satisfying certain desirable properties. Our transformation typically involves *simulating* one or more distributed computations based on the events of the underlying distributed computation. We call the underlying computation as *primary computation*, and a simulated computation as *secondary computation*. (We use the terms “underlying computation” and “primary computation" interchangeably.) We call the given termination detection algorithm as the *basic termination detection algorithm*, and the algorithm obtained as a result of the transformation as the *derived termination detection algorithm*.

It is possible for a process p_i to be in different states with respect to different computations. We use $state(\mathcal{F}, C, i)$ to refer to the state of the process p_i for the consistent cut C with respect to the computation $\mathcal{F}$. Specifically, $state(\mathcal{F}, C, i)$ is true if and only if p_i is active for C with respect to $\mathcal{F}$. Formally,

$$state(\mathcal{F}, C, i) \triangleq \begin{cases} \mathsf{true} : \text{process } p_i \text{ is active with respect to computation} \\ \qquad\quad \mathcal{F} \text{ for consistent cut } C \\ \mathsf{false} : \text{otherwise} \end{cases}$$

A secondary computation is "similar" to the primary computation in the sense that it also follows the four rules described earlier. However, the set of messages exchanged by a secondary computation is generally a subset of the set of messages exchanged by the primary computation. Let $messages(\mathcal{F})$ denote the set of messages exchanged by the computation $\mathcal{F}$. We say that a message is in transit with respect to a consistent cut if its send event belongs to the cut but its receive event does not. We use $transit(\mathcal{F}, C)$ to refer to the set of messages of the computation $\mathcal{F}$ that are in transit with respect to the consistent cut C. Formally,

$$transit(\mathcal{F}, C) \triangleq \{ m \in messages(\mathcal{F}) \mid m \text{ is in transit with respect to } C \}$$

Finally, a computation $\mathcal{F}$ has terminated for a consistent cut C if all processes are in passive state with respect to $\mathcal{F}$ for C and there are no messages of $\mathcal{F}$ that are in transit with respect to C. Formally,

$$terminated(\mathcal{F}, C) \triangleq \langle \forall i :: \neg state(\mathcal{F}, C, i) \rangle \wedge (transit(\mathcal{F}, C) = \emptyset)$$

We now describe the three transformations. Unless otherwise stated, we use the following notation when describing the transformations. The basic termination detection algorithm is denoted by A. We use N, E and D to denote the number of processes, the number of channels and the diameter of the communication topology, respectively. For a given distributed system, we assume that the (worst-case) message complexity and detection latency of A are given by $O(\mu(M))$ and $O(\delta(M))$, respectively, where M is the number of application messages exchanged by the computation whose termination A is supposed to detect. Note that the message complexity and detection latency of A may also depend on N, E and D. For clarity, we assume these parameters to be implicit in the definition of μ and δ. Trivially, both μ and δ are monotonically non-decreasing functions of all their parameters—explicit as well as implicit. Also, $\mu(M) \geqslant M$ [21]. Moreover, if A can detect termination of a non-diffusing computation, then $\mu(M) = \Omega(M + N)$ and $\delta(M) = \Omega(D)$.

Finally, we use M to denote the number of application messages exchanged by the primary computation, and, in the case of delayed initiation, $\bar{M}$ to denote the number of application messages exchanged by the primary computation after the termination detection algorithm began.

III. From a Diffusing Computation to a Non-Diffusing Computation

Consider an algorithm specifically designed to detect termination of a diffusing computation. The main idea is to simulate multiple secondary computations, one for each process, such that the secondary computations satisfy the following two conditions. First, each secondary computation is a diffusing computation. Second, detecting termination of all secondary computations is equivalent to detecting termination of the primary computation. We then use an instance of the given termination detection algorithm to detect termination of each secondary computation.

Let $\mathcal{P}$ denote the primary computation and $\mathcal{S}_i$ denote the i^{th} secondary computation with $1 \leqslant i \leqslant N$. Intuitively, process p_i is the initiator of the i^{th} secondary computation $\mathcal{S}_i$. A process participates in all secondary computations (and, of course, the primary computation), and, at any given time, may be in different states with respect to different computations. To ensure that each secondary computation is indeed a diffusing computation, state of a process with respect to different secondary computations is initialized as follows:

$$\begin{aligned} i = j \quad &\Rightarrow \quad state(\mathcal{S}_j, \emptyset, i) = state(\mathcal{P}, \emptyset, i) \\ i \neq j \quad &\Rightarrow \quad state(\mathcal{S}_j, \emptyset, i) = \mathsf{false} \end{aligned}$$

Here, the initial consistent cut is represented using an empty set of events. Clearly, at most one process is initially active with respect to each secondary computation.

To guarantee that detecting termination of all secondary computations is equivalent to detecting termination of the primary computation, we maintain the following two invariants. First, if a process is active with respect to the primary computation, then it is active with respect to at least one secondary computation. Formally,

$$state(\mathcal{P}, C, i) \;\equiv\; \langle \exists j :: state(\mathcal{S}_j, C, i) \rangle \tag{1}$$

Second, every application message exchanged by the primary computation is part of at least one secondary computation. Formally,

$$messages(\mathcal{P}) \;=\; \bigcup_{j=1}^{N} messages(\mathcal{S}_j) \tag{2}$$

For efficiency reasons, it is desirable that a message of the primary computation belongs to at most one secondary computation. Note that, (2), in turn, implies the following:

$$transit(\mathcal{P}, C) \;=\; \bigcup_{j=1}^{N} transit(\mathcal{S}_j, C) \tag{3}$$

We show that the two invariants indeed guarantee that the primary computation terminates when and only when all secondary computations terminate.

Lemma 1: If the primary computation has terminated, then all secondary computations have also terminated, and vice versa. Formally,

$$terminated(\mathcal{P}, C) \;\equiv\; \langle \forall j :: terminated(\mathcal{S}_j, C) \rangle$$

Proof: We have,

$$terminated(\mathcal{P}, C)$$

$\equiv$ { definition of termination }

$$\langle \forall i :: \neg state(\mathcal{P}, C, i) \rangle \wedge (transit(\mathcal{P}, C) = \emptyset)$$

$\equiv$ { from (1) and (3) }

$$\Big\langle \forall i :: \langle \forall j :: \neg state(\mathcal{S}_j, C, i) \rangle \Big\rangle \;\bigwedge\; \Big(\bigcup_{j=1}^{N} transit(\mathcal{S}_j, C) \;=\; \emptyset \Big)$$

$\equiv$ { predicate and set calculus }

$$\Big\langle \forall j :: \langle \forall i :: \neg state(\mathcal{S}_j, C, i) \rangle \Big\rangle \bigwedge \Big\langle \forall j :: transit(\mathcal{S}_j, C) = \emptyset \Big\rangle$$

$\equiv$ { predicate calculus }

$$\Big\langle \forall j :: \langle \forall i :: \neg state(\mathcal{S}_j, C, i) \rangle \wedge (transit(\mathcal{S}_j, C) = \emptyset) \Big\rangle$$

$\equiv$ { definition of termination }

$$\langle \forall j :: terminated(\mathcal{S}_j, C) \rangle$$

This establishes the lemma. ■

Next, we discuss how to simulate each secondary computation. This is important because, in general, a termination detection algorithm can *correctly* detect termination of a computation only if the computation follows the four rules (Rule 1-Rule 4) described in Section II-B. For instance, suppose process p_i is passive with respect to a secondary computation $\mathcal{S}_j$. Clearly, p_i cannot send a message that belongs to $\mathcal{S}_j$ until it becomes active with respect to $\mathcal{S}_j$; otherwise Rule 2 would be violated. Further, p_i can become active with respect to $\mathcal{S}_j$ only on receiving a message that belongs to $\mathcal{S}_j$. In other words, p_i should not become active with respect to $\mathcal{S}_j$ on receiving a message that belongs to some other secondary computation $\mathcal{S}_k$ with $j \neq k$; otherwise Rule 4 would be violated.

To ensure that each secondary computation follows the four rules and, moreover, all secondary computations collectively satisfy (1) and (2), we proceed as follows. Suppose a process is currently active with respect to the primary computation and wants to send a message to another process. From (1), it follows that it is also active with respect to at least one secondary computation. We arbitrarily select one such secondary computation and piggyback the identifier of its initiator on the message, which implies that the message belongs to the selected secondary computation. It also enables the destination process, when it receives a message, to determine the secondary computation to which the message belongs so that it can execute the appropriate control actions, if any. A process, on receiving a message, becomes active with respect to the secondary computation to which the message belongs, in case it is not already active with respect to that computation. Finally, when a process becomes passive with respect to the primary computation, it also becomes passive with respect to all secondary computations with respect to which it was active before.

For detecting termination of the primary computation, from Lemma 1, it is sufficient to detect termination of all secondary computations. We assume that the basic termination detection algorithm is such that there is a unique process that is responsible for detecting termination of the diffusing computation. In fact, for most termination detection algorithms for a diffusing computation, it is the initiator that is responsible for detecting the termination. We also assume that there is a distinguished process, called the *coordinator*, that is responsible for detecting termination of the primary computation. Once a process has detected termination of *all the secondary computations it is responsible for*, it informs the coordinator by sending a *terminate* message to it. To minimize the number of control messages exchanged as a result, we assume that *terminate* messages sent by different processes are propagated to the coordinator in a *convergecast* fashion. This can be accomplished by building a BFS (breadth-first-search) spanning tree of the communication topology rooted at the coordinator in the beginning as a preprocessing step. We refer to the algorithm transformation described in this section as $DTON$. Let A be a termination detection algorithm specifically designed to detect termination of a diffusing computation. We have,

Theorem 2: $DTON(A)$ correctly detects termination of a non-diffusing computation.

Proof: *(safety)* Clearly, the coordinator announces termination only after all secondary computations have terminated. From Lemma 1, it implies that the primary computation has terminated as well.

(liveness) Suppose the primary computation has terminated. From Lemma 1, it follows that all secondary computations have also terminated. Their respective initiators will eventually inform the coordinator about it. As a result, the coordinator will eventually announce termination. ■

We now analyze the message-complexity of the derived termination detection algorithm. Note that every control message exchanged by the derived termination detection algorithm, except a *terminate* message, belongs to one of the instances of the basic termination detection algorithm.

We assume that μ satisfies the following inequality:

$$\mu(X) + \mu(Y) \ \leqslant\ \mu(X+Y) + O(1) \tag{4}$$

This is a realistic assumption because for all termination detection algorithms for a diffusing computation that we know of the message complexity is either $O(M)$ [2], [7] or $O(MD)$ [9], [10], both of which satisfy (4). (In fact, the above inequality holds as long as $\mu(0)$ is $O(1)$ because $\mu(M)$ is $\Omega(M)$.)

Theorem 3: If μ satisfies (4), then the message complexity of $DTON(A)$ is given by $O(\mu(M) + N)$.

Proof: For convenience, let $M_i = |messages(\mathcal{S}_i)|$. The control messages exchanged by $DTON(A)$ consists of the control messages exchanged by the N instances of A to detect termination of the N secondary computations and *terminate* messages. Therefore the message complexity of $DTON(A)$ is given by:

$$\begin{aligned}
& \sum_{i=1}^{N} O(\mu(M_i)) + O(N) \\
\leqslant\ & \{\ \mu \text{ satisfies (4) }\} \\
& O(\mu(\sum_{i=1}^{N} M_i)) + N \cdot O(1) + O(N) \\
=\ & \{\ \text{each application message belongs to exactly one secondary computation }\} \\
& O(\mu(M)) + O(N)
\end{aligned}$$

This proves the theorem. ■

Finally, we analyze the detection latency of the derived termination detection algorithm. We have,

Theorem 4: The detection latency of $DTON(A)$ is given by $O(\delta(M) + D)$.

Proof: Once the primary computation terminates, all secondary computations terminate as well. For convenience, let $M_i = |messages(\mathcal{S}_i)|$. The termination of a secondary computation $\mathcal{S}_i$ is detected by the appropriate process within $O(\delta(M_i))$ message hops. Once termination of all secondary computations has been detected, the coordinator announces termination of the primary computation within $O(D)$ message hops. Therefore the detection latency of $DTON(A)$ is given by:

$$\begin{aligned} & \max_{1 \leqslant i \leqslant N} \{O(\delta(M_i))\} + O(D) \\ \leqslant \; & \{ \text{ δ is a monotonic function of its argument } \} \\ & O(\delta(M)) + O(D) \end{aligned}$$

This establishes the theorem. ■

Note that it is not necessary for a process to stay active with respect to more than one secondary computation. From (1), it is sufficient for a process to stay active with respect to *at most one* secondary computation as is the case with Shavit and Francez's extension [6] of the Dijkstra and Scholten's algorithm [2]. In other words, if a process is already active with respect to a secondary computation and receives a message that belongs to some other secondary computation, then, after processing the message, it can become passive with respect to the latter secondary computation immediately. The choice when to become passive with respect to a secondary computation depends on the given termination detection algorithm. For example, if, on becoming passive, the detection algorithm requires a control message to be sent to the same process for all secondary computations, then it is preferable for the process to become passive with respect to all secondary computations at the same time and send only a single control message.

Remark 1: In case $\mu(0)$ is $\omega(1)$, it can be verified that μ satisfies the following inequality:

$$\mu(X) + \mu(Y) \leqslant \mu(X+Y) + \mu(0) \tag{5}$$

Using the above inequality, it can be proved that the message complexity of $DTON(A)$ is given by $O(\mu(M) + (1 + \mu(0))N)$. □

We now describe the two transformations for delayed initiation, which use the transformation described in this section.

IV. From Simultaneous Initiation to Delayed Initiation

A termination detection algorithm, in the worst case, may exchange as many control messages as the number of application messages exchanged by the computation [21]. Therefore, if the underlying computation is message-intensive, then the detection algorithm may exchange a large number of control messages. Chandrasekaran and Venkatesan [11] propose a termination detection algorithm that can be started at anytime after the computation has begun. Their algorithm has the advantage that it needs to track only those application messages that are sent after it began executing. As a result, their algorithm is especially suited to detect termination of a message-intensive computation for which it is preferable to initiate the algorithm when the computation is “close” to termination.

In delayed initiation in the strict sense as described by Chandrasekaran and Venkatesan [11], no explicit control information can be maintained about the state of the computation that may aid the termination detection algorithm (for example, the number of application messages that have been sent or received along a channel) before the detection algorithm is started. Consequently, delayed initiation is not possible unless all channels are FIFO [11]. Intuitively, this is because if a channel is non-FIFO then an application message may be delayed arbitrarily along the channel, no process would be aware of its existence, and this message may arrive at the destination after termination has been announced. Therefore the main idea behind delayed initiation is to “flush” those application messages that were sent before the termination detection algorithm started be sending a control message along each channel. Not surprisingly, Chandrasekaran and Venkatesan prove that any termination detection algorithm, when initiated in a delayed manner, must exchange at least E control messages in the worst case, where E is the number of channels in the communication topology. Although delayed initiation actually increases the *worst-case* message complexity of the given termination detection algorithm, but it reduces its *average* message complexity. This is because, in general, much fewer number of application messages need to be tracked by the detection algorithm.

On the other hand, in quasi-delayed initiation, some control information may be maintained about the state of the computation before the termination detection algorithm has started. However, control messages can be exchanged only after termination detection algorithm has actually begun. Quasi-delayed initiation has the same advantage as delayed initiation in the sense that it reduces the *average* number of control messages exchanged by the termination detection algorithm. Moreover, unlike delayed initiation, quasi-delayed initiation does not increase the worst-case message complexity of the given detection algorithm.

In this section, we first provide a transformation that can be used to initiate *any* termination detection algorithm in a delayed manner. However, due to the nature of the delayed initiation, the transformation increases the worst-case message complexity of the given termination detection algorithm by $O(E)$. Next, we describe a transformation that allows *any* termination detection to be started in a quasi-delayed manner without increasing its worst-case message complexity.

A. Delayed Initiation without Maintaining State Information

To correctly detect termination with delayed initiation, we use the scheme proposed in [11]. The main idea is to distinguish between application messages sent by a process *before* it started termination detection and messages sent by it *after* it started termination detection. Clearly, the former messages should be ignored by the termination detection algorithm and the latter messages should be tracked by the termination detection algorithm.

To distinguish between the two kinds of application messages, we use a *marker* message. Specifically, as soon as a process starts the termination detection algorithm, it sends a *marker* message along all its outgoing channels. Therefore, when a process receives a *marker* message along an incoming channel, it knows that any application message received along that channel from now on has to be tracked by the termination detection algorithm. On the other hand, if a process receives an application message on an incoming channel along which it has not yet received a *marker* message, then that message should be ignored by the termination detection algorithm and simply delivered to the application. Intuitively, a *marker* message sent along a channel “flushes” any in-transit application messages on that channel.

For ease of exposition only, we assume that initially all processes and channels are colored *white*. A white process may start executing the termination detection algorithm at anytime. In fact, it is possible for more than one white process to initiate the termination detection algorithm concurrently. On starting termination detection, a process becomes *red*. Further, it sends a *marker* message along all its outgoing channel and also colors all of them red. On receiving a *marker* message along an incoming channel, a process colors the (incoming) channel red. Moreover, if it has not already started termination detection, it initiates the termination detection algorithm. An application message assumes the color of the outgoing channel along which it is sent.

When a white process receives a white application message, it can simply deliver the message to the application because it has not yet started termination detection. Note that it is not possible for a white process to receive a red application message. The reason is that before a process receives a red application message along a channel, it receives a *marker* message along the same channel which causes it to turn red. When a red process receives a red application message, it first executes the action dictated by the termination detection algorithm before delivering the message to the application. The problem arises when a red process, which is passive, receives a white application message. If it simply delivers the message to the application without informing the termination detection algorithm about it, then from the detection algorithm's point of view, the process would become active spontaneously, thereby violating Rule 4. On the other hand, if it informs the termination detection algorithm about receipt of the message, then the detection algorithm may not know what control action to execute. This may happen, for example, when the control action to be executed depends on the control information piggybacked on the application message. Specifically, action may involve sending a control message to some process and the control information piggybacked on the application message may determine the destination process of the control message.

Our approach is to simulate a secondary computation such that the secondary computation and the termination detection algorithm start at the *same time on every process*. The secondary computation is almost identical to the primary computation except possibly in the beginning and satisfies the following two conditions. First, the termination of the secondary computation implies the termination of the primary computation. Second, once the primary computation terminates, the secondary computation terminates eventually. We then use the given termination detection algorithm to detect termination of the secondary computation. For a consistent cut C and a process p_i, let $allRed(C, i)$ be `true` if all incoming channels of p_i are colored red for C.

Let the primary and secondary computations be denoted by $\mathcal{P}$ and $\mathcal{S}$, respectively. A process is active with respect to the secondary computation if either it is active with respect to the primary computation or at least one of its incoming channels is colored white. Formally,

$$state(\mathcal{S}, C, i) \;\equiv\; state(\mathcal{P}, C, i) \vee \neg allRed(C, i) \tag{6}$$

Intuitively, a process stays active with respect to the secondary computation until it knows that it will not receive any white application message in the future. Any white application message is delivered directly to the underlying computation without informing the termination detection algorithm about it. This does not violate any of the four rules from the point of view of the termination detection algorithm. This is because if a process receives a white application message, then, from (6), the process is already active with respect to the secondary computation and therefore Rule 4 is not violated.

Note that our secondary computation is a non-diffusing computation. Therefore we assume that the basic termination detection algorithm can *detect termination of a non-diffusing computation*. This is not a restrictive assumption as implied by our first transformation. Let $white(\mathcal{P})$ and $red(\mathcal{P})$ refer to the set of white and red messages, respectively, exchanged by the underlying computation $\mathcal{P}$. Clearly, $messages(\mathcal{P}) = white(\mathcal{P}) \cup red(\mathcal{P})$. Since all channels are FIFO,

$$\langle \forall i :: allRed(C, i) \rangle \quad \Rightarrow \quad (transit(\mathcal{P}, C) \cap white(\mathcal{P})) = \emptyset \tag{7}$$

Finally, every red message is part of the secondary computation, that is, $messages(\mathcal{S}) = red(\mathcal{P})$. Thus,

$$transit(\mathcal{S}, C) \quad = \quad transit(\mathcal{P}, C) \cap red(\mathcal{P}) \tag{8}$$

We show that termination of the secondary computation implies termination of the primary computation.

Lemma 5: If the secondary computation has terminated, then the primary computation has also terminated. Formally,

$$terminated(\mathcal{S}, C) \;\Rightarrow\; terminated(\mathcal{P}, C)$$

Proof: We have,

$$
\begin{array}{ll}
 & terminated(\mathcal{S}, C) \\
\equiv & \{ \text{ definition of termination } \} \\
 & \langle \forall i :: \neg state(\mathcal{S}, C, i) \rangle \ \wedge \ (transit(\mathcal{S}, C) \ = \ \emptyset) \\
\equiv & \{ \text{ from (6) and (8) } \} \\
 & \langle \forall i :: \neg state(\mathcal{P}, C, i) \wedge allRed(C, i) \rangle \ \wedge \ (transit(\mathcal{P}, C) \cap red(\mathcal{P}) \ = \ \emptyset) \\
\equiv & \{ \text{ predicate calculus } \} \\
 & \langle \forall i :: \neg state(\mathcal{P}, C, i) \rangle \ \wedge \ \langle \forall i :: allRed(C, i) \rangle \ \wedge \ (transit(\mathcal{P}, C) \cap red(\mathcal{P}) \ = \ \emptyset) \\
\Rightarrow & \{ \text{ from (7) } \} \\
 & \langle \forall i :: \neg state(\mathcal{P}, C, i) \rangle \ \wedge \\
 & (transit(\mathcal{P}, C) \cap white(\mathcal{P}) = \emptyset) \ \wedge \ (transit(\mathcal{P}, C) \cap red(\mathcal{P}) = \emptyset) \\
\equiv & \{ \text{ predicate calculus } \} \\
 & \langle \forall i :: \neg state(\mathcal{P}, C, i) \rangle \ \wedge \ \Big(transit(\mathcal{P}, C) \cap (white(\mathcal{P}) \cup red(\mathcal{P})) \ = \ \emptyset\Big) \\
\Rightarrow & \{ \text{ using } transit(\mathcal{P}, C) \subseteq white(\mathcal{P}) \cup red(\mathcal{P}) \ \} \\
 & \langle \forall i :: \neg state(\mathcal{P}, C, i) \rangle \ \wedge \ (transit(\mathcal{P}, C) = \emptyset) \\
\equiv & \{ \text{ definition of termination } \} \\
 & terminated(\mathcal{P}, C)
\end{array}
$$

This establishes the lemma. ■

We also show that once the primary computation terminates, the secondary computation terminates eventually. It suffices to show that the secondary computation terminates once the primary computation terminates and all incoming channels have been colored red.

Lemma 6: If the primary computation has terminated and all incoming channels have been colored red, then the secondary computation has also terminated. Formally,

$$terminated(\mathcal{P}, C) \wedge \langle \forall i :: allRed(C, i) \rangle \ \Rightarrow terminated(\mathcal{S}, C)$$

Proof: We have,

$$terminated(\mathcal{P}, C) \wedge \langle \forall i :: allRed(C, i) \rangle$$

$\equiv$ { definition of termination }

$$\langle \forall i :: \neg state(\mathcal{P}, C, i) \rangle \;\wedge\; (transit(\mathcal{P}, C) = \emptyset) \;\wedge\; \langle \forall i :: allRed(C, i) \rangle$$

$\equiv$ { predicate and set calculus }

$$\langle \forall i :: \neg state(\mathcal{P}, C, i) \wedge allRed(C, i) \rangle \;\wedge\; (transit(\mathcal{P}, C) \cap red(\mathcal{P}) \;=\; \emptyset)$$

$\Rightarrow$ { from (6) and (8) }

$$\langle \forall i :: \neg state(\mathcal{S}, C, i) \rangle \;\wedge\; (transit(\mathcal{S}, C) = \emptyset)$$

$\equiv$ { definition of termination }

$$terminated(\mathcal{S}, C)$$

This establishes the lemma. ■

From Lemma 5 and Lemma 6, to detect termination of the primary computation, it is safe and live to detect termination of the secondary computation. We refer to the algorithm transformation described in this section as $STOD$. Let A be a termination detection algorithm for a non-diffusing computation that requires simultaneous initiation. We have,

Theorem 7: $STOD(A)$ correctly detects termination with delayed initiation.

Proof: Our transformation ensures that the given termination detection algorithm A and the simulated secondary computation $\mathcal{S}$ start simultaneously on every process.

(safety) Suppose $STOD(A)$ announces termination. This implies that the secondary computation has terminated. From Lemma 5, it follows that the primary computation has also terminated.

(liveness) Suppose the primary computation has terminated. Then, from Lemma 6, the secondary computation also terminates eventually. Clearly, once that happens, A eventually announces termination. ■

We now analyze the message complexity of the derived termination detection algorithm. We have, *Theorem 8:* The message complexity of $STOD(A)$ is given by $O(\mu(\bar{M}) + E)$.

Proof: The message complexity of $STOD(A)$ is given by the sum of (1) the number of *marker* messages exchanged and (2) the number of control messages exchanged by A when used to detect termination of the secondary computation. Clearly, at most one *marker* message is generated for every channel. Also, the number of messages exchanged by the secondary computation is same as the number of red application messages. Thus the message complexity of $STOD(A)$ is given by $O(\mu(\bar{M}) + E)$. ■

Note that, in the worst case $\bar{M}$ may be same as M. Therefore the worst case message complexity of $STOD(A)$ is $O(\mu(M)+E)$. We now analyze the detection latency of the derived termination detection algorithm.

Theorem 9: If $STOD(A)$ is initiated before the underlying computation terminates, then the detection latency of $STOD(A)$ is $O(\delta(\bar{M}))$.

Proof: From Lemma 6, once the primary computation has terminated, the secondary computation terminates as soon as all incoming channels become red. Clearly, once $STOD(A)$ is initiated, all incoming channels become red within $O(D)$ message hops. Moreover, once the secondary computation terminates, A detects its termination within $O(\delta(\bar{M})$ message hops. As a result, the detection latency of $STOD(A)$ is $O(\delta(\bar{M}) + D)$. Since A can detect termination of a non-diffusing computation, its detection latency is $\Omega(D)$. Therefore the detection latency of $STOD(A)$ is $O(\delta(\bar{M}))$. ■

B. Delayed Initiation using State Information: Quasi-Delayed Initiation

As in the case of delayed initiation, we assume that the given termination detection algorithm is such that it can be used to detect termination of a non-diffusing computation. Also, for ease of exposition, we assume that all channels are FIFO and initially all processes are colored white.

One of the processes acts as the *coordinator* and a BFS spanning tree is rooted at the coordinator. The coordinator is responsible for initiating the termination detection algorithm. When the coordinator wants to initiate the termination detection algorithm, it broadcasts a *start* message to all processes (including itself) via the spanning tree. On receiving the *start* message, a white process changes its color to red and starts executing the termination detection algorithm. As in the case of delayed initiation, we simulate a secondary computation that satisfies the following properties. First, the secondary computation and the termination detection algorithm start at the same time on every process. Initially, all processes are active with respect to the secondary computation. A white application message is delivered directly to the computation without informing the termination detection algorithm about it. Second, the termination of the secondary computation implies the termination of the primary computation. Third, once the primary computation terminates, the secondary computation terminates eventually.

In the transformation for delayed initiation, *marker* messages serve two distinct purposes. First, a *marker* message flushes the channel of any in-transit white application message. Second, once a process has received a *marker* message along all its incoming channels, it knows that it will not receive any white application message in the future. Once a process knows that there are no white application messages in-transit towards it, the secondary computation on that process becomes identical to the primary computation. In the transformation for quasi-delayed initiation, we still use a *marker* message to flush a channel. (We use the term *flush* instead of *marker* now.) However, we use a different mechanism to enable a process to know that there are no more white application messages in-transit towards it.

We say that a red process is *safe* if all white application messages it sent have been received. When a process starts the termination detection algorithm, it sends a *flush* message along those outgoing channels along which it has sent *at least one* white application message. A process, on receiving a *flush* message, acknowledges the message by sending an *ack_flush* message. Clearly, since all channels are FIFO, once a process has received an *ack_flush* message for every *flush* message it sent, it knows that it has become safe.

Note that a process cannot become passive with respect to the secondary computation once it becomes safe. In fact, it has to stay active with respect to the secondary computation at least until it knows that all its *neighbors have become safe*. This is accomplished as follows. Once a process knows that it has become safe, it sends a *safe* message to the coordinator. To minimize the number of control messages generated as a result, *safe* messages are propagated to the coordinator in a *convergecast* fashion. After the coordinator has received a *safe* message from all its children in the spanning tree, it broadcasts an *all_safe* message to all processes (including itself) via the spanning tree. Once a process has received an *all_safe* message, the secondary computation on that process becomes identical to the primary computation. Let the primary and secondary computations be denoted by $\mathcal{P}$ and $\mathcal{S}$, respectively. Formally,

$$state(\mathcal{S}, C, i) \;\equiv\; state(\mathcal{P}, C, i) \vee (p_i \text{ has not received an } \textit{all\_safe} \text{ message}) \tag{9}$$

For a consistent cut C, let $\mathit{safe}(C, i)$ be true if process p_i has received an *ack_flush* message for every *flush* message it sent at C. We have,

$$\langle \forall\, i :: \mathit{safe}(C, i) \rangle \quad \Rightarrow \quad (transit(\mathcal{P}, C) \cap white(\mathcal{P})) = \emptyset$$

Thus, we have,

$$p_i \text{ has received an } \textit{all\_safe} \text{ message} \quad \Rightarrow \quad (transit(\mathcal{P}, C) \cap white(\mathcal{P})) = \emptyset \tag{10}$$

As in the case of delayed initiation, the messages of the secondary computation are given by red application messages, that is, $messages(\mathcal{S}) = red(\mathcal{P})$. The transformation assumes that a process, on receiving an application message, is able to determine whether the color of the message is white or red. Clearly, if a process receives an application message along a channel after it has received a *flush* message along that channel, then it knows that the color of the message is red. However, if a process receives an application message along a channel along which it has not received a *flush* message, then the color of the message may be white or red. The latter may happen when a red process sends an application message along a channel along which it has not sent any white application message. In this case, as per the transformation, it will not send any *flush* message on starting the termination detection algorithm. To that end, before sending the *first* red application message along a channel, a process sends a special control message along that channel. We refer to this message as *inform* message. With this approach, a process can receive a red application message along a channel only after it has received an *inform* message along that channel.

Finally, before a process receives the *start* message, it may receive a control message from one of its neighbors. This implies that the neighboring process from which the message is received has already started the termination detection algorithm. Therefore the process can also start the termination detection algorithm without waiting for the *start* message to arrive.

To summarize, the transformation for quasi-delayed initiation is as follows:

- When the coordinator wants to start the termination detection algorithm, it sends a *start* message to itself.
- Before receiving a control message of any type, a white process changes its color to red. This ensures that a process delivers (receives and processes) a control message only after it has turned red. A process, on changing its color, initiates the termination detection algorithm and sends a *start* message to all its children in the spanning tree. It also sends a *flush* message to every neighboring process to which it has sent at least one white application message.
- A process, on receiving a *flush* message, sends an *ack_flush* message to the sender of the *flush* message.
- Before sending an application message along a channel, a red process first sends an *inform* message along the channel, if it has not already done so.
- If a red process receives an application message along a channel before receiving an *inform* message along the channel (that is, the color of the message is white), then it delivers the message directly to the underlying computation. Otherwise, it delivers the message to the termination detection algorithm which in turn is responsible for delivering it to the computation.
- Once a process has received an *ack_flush* message for every *flush* message it sent, it becomes safe and sends a *safe* message to the coordinator. All *safe* messages are propagated to the coordinator in a convergecast fashion.
- Once the coordinator has received a *safe* message from every process (through a convergecast), it broadcasts an *all_safe* message to all processes via the spanning tree.
- Once a process has received an *all_safe* message, the secondary computation on the process

becomes identical to the primary computation.

- All other control messages, namely those generated by the given termination detection algorithm, are handled by the detection algorithm itself.

We refer to the algorithm transformation described in this section as $STOQD$. Using (9) and (10), it can be proved that to detect the termination of the primary computation, it is safe and live to detect the termination of the secondary computation by proving lemmas similar to Lemma 5 and Lemma 6. The proofs have been omitted to avoid repetition. We first analyze the message complexity of $STOQD(A)$.

Theorem 10: The message complexity of $STOQD(A)$ is given by $O(\mu(\bar{M}) + F)$, where F denotes the number of *flush* messages exchanged.

Proof: In addition to control messages exchanged by A, $STOQD(A)$ exchanges *start*, *flush*, *ack_flush*, *inform*, *safe* and *all_safe* messages. The total number of *start*, *safe* and *all_safe* messages is bounded by $3N$. The number of *inform* messages is bounded by $\bar{M}$. Therefore the message complexity of $STOQD(A)$ is given by:

$$\begin{aligned} & O(\mu(\bar{M}) + N + \bar{M} + F) \\ = \quad & \{ \text{ secondary computation is a non-diffusing computation} \Rightarrow O(\mu(\bar{M})) = \Omega(\bar{M} + N) \ \} \\ & O(\mu(\bar{M}) + F) \end{aligned}$$

This establishes the theorem. ■

Since $\bar{M}$ may be M and F may be E in the worst case, it may appear that the worst case message complexity of $STOQD(A)$ is $O(\mu(M) + E)$, which is same as that of $STOD(A)$. However, since $\mu(M) \geqslant M$, μ satisfies the following inequality:

$$\mu(X) + Y \ \leqslant \ \mu(X + Y) \tag{11}$$

Therefore, we have,

Corollary 11: The worst case message complexity of $STOQD(A)$ is $O(\mu(M))$.

Proof: From Theorem 10, the worst case message complexity of $STOQD(A)$ is given by:

$$O(\mu(\bar{M}) + F)$$

$$\leqslant \quad \{ \text{ using (11) } \}$$

$$O(\mu(\bar{M} + F))$$

$$\leqslant \quad \{ \ \bar{M} + F \leqslant M \ \}$$

$$O(\mu(M))$$

This establishes the corollary. ■

Note that, once the coordinator starts the termination detection algorithm, all processes receive an *all_safe* message within $O(D)$ message hops. As a result, once the primary computation terminates, the secondary computation terminates within $O(D)$ message hops. The following theorem can be proved in a similar way as Theorem 9.

Theorem 12: If $STOQD(A)$ is initiated before the underlying computation terminates, then the detection latency of $STOQD(A)$ is given by $O(\delta(\bar{M}))$.

Optimization: If a process has sent a *start* message along a channel, then it is not necessary to send a *flush* message along that channel. Likewise, if a process has sent a *flush* message along a channel, then it is not necessary to send an *inform* message along that channel. In other words, the functionalities of the three control messages can be combined into a single control message with a bit piggybacked on the message indicating whether the recipient has to send an acknowledgment message to the sender.

1) Dealing with Non-FIFO Channels: When one or more channels are non-FIFO, the main problem is to distinguish between white and red application messages. This is because white application messages are invisible to the termination detection algorithm and are delivered directly to the computation, whereas red application messages are tracked by the termination detection algorithm. To that end, every application message is piggybacked with its color—white or red. (With this modification, *inform* messages become redundant.) A process, on sending a *flush* message along a channel, piggybacks the number of white application messages that it has sent along that channel. A process, on receiving a *flush* message along a channel, responds with an *ack_flush* message only after it has received all the white application messages along that channel. This can be ascertained using the information piggybacked on the *flush* message and by counting the number of white messages that have been received so far.

V. Discussion

In [2], Shavit and Francez extend Dijkstra and Scholten's termination detection algorithm (DSTDA), which is designed to detect termination of a diffusing computation, to work for any non-diffusing computation. DSTDA detects termination by maintaining a dynamic tree of processes currently participating in the computation. Termination is announced once the tree becomes empty. Shavit and Francez generalize DSTDA to maintain a *dynamic forest* of trees. Termination is announced once the forest becomes empty. Our approach can be viewed as a generalization of their approach in the sense that each tree in the forest corresponds to a diffusing computation with the root of the tree as the initiator. However, modifications made by Shavit and Francez to DSTDA are highly customized in nature. On the other hand, our approach works with any termination detection algorithm for a diffusing computation. Moreover, in Shavit and Francez's approach, a process cannot be active with respect to more than one secondary computation. There is no such restriction with our approach. Finally, in our approach, different termination detection algorithms can be used to detect termination of different secondary computations at least in theory.

Lai *et al* [22] also extend DSTDA to enable it to detect termination of a non-diffusing computation. However, their approach is different from that of Shavit and Francez's approach [6]. Intuitively, they use $O(N)$ fictitious application messages in the beginning before the start of the computation to convert a non-diffusing computation into a diffusing computation. Their approach can also be used to transform any termination detection algorithm for a diffusing computation to a termination detection algorithm for a non-diffusing computation. However, since fictitious application messages are also tracked by the termination detection algorithm, the message complexity of the termination detection algorithm obtained using their transformation is given by $O(\mu(M + N))$. In contrast, the message complexity with our transformation is $O(\mu(M) + N)$ in case $\mu(0) = O(1)$ and $O(\mu(M) + (1 + \mu(0))N)$ in general. It can be proved that their transformation is more efficient provided the ratio $\frac{\mu(X)-\mu(Y)}{X-Y}$ is $o(1+\mu(0))$ in the worst-case; otherwise, our transformation is more efficient. (For example, if $\mu(M) = MD$, then $\mu(0) = 0$ and the value of the ratio is D.) Moreover, using (11) and the fact that $\mu(M) = \Omega(M)$, our transformation works *at least as well as* their transformation in case $\mu(0) = O(1)$. This is indeed the case for all termination detection algorithms for a diffusing computation that we are aware of.

Lai *et al* [22] also modify DSTDA so that it can be started in a quasi-delayed manner. To that end, they modify DSTDA so that instead of acknowledging each application message individually, a process may acknowledge more than one application message by sending a single control message. As a result, the modifications made by Lai *et al* to DSTDA are highly specialized in nature. On the other hand, our transformations (for delayed as well as quasi-delayed initiation) do not assume *any* specific termination detection algorithm and in fact work for any termination detection algorithm. Further, for quasi-delayed initiation, the only control information we need to maintain is whether a white application message was sent along a channel, which is independent of the specific termination detection algorithm. In their approach, in contrast, the control information maintained is specific to DSTDA.

An interesting question that arises is: “Is it possible to initiate other stable property detection algorithms in a delayed manner?” The question makes sense because the complexity of many stable property detection algorithms depends on the number of events that are tracked by the detection algorithm. Using delayed initiation, the average number of events that need to be tracked by the detection algorithm can be substantially reduced. As a future work, we plan to identify classes of computations for which it is possible to initiate detection algorithms in a delayed manner.

REFERENCES

[1] S. Peri and N. Mittal, “On Termination Detection in an Asynchronous Distributed System,” in *Proceedings of the ISCA International Conference on Parallel and Distributed Computing Systems (PDCS)*, San Francisco, California, Sept. 2004, pp. 209–215.

[2] E. W. Dijkstra and C. S. Scholten, "Termination Detection for Diffusing Computations," *Information Processing Letters (IPL)*, vol. 11, no. 1, pp. 1–4, 1980.

[3] N. Francez, "Distributed Termination," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 2, no. 1, pp. 42–55, Jan. 1980.

[4] J. Misra, "Detecting Termination of Distributed Computations using Markers," in *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, 1983, pp. 290–294.

[5] S. P. Rana, "A Distributed Solution of the Distributed Termination Problem," *Information Processing Letters (IPL)*, vol. 17, no. 1, pp. 43–46, 1983.

[6] N. Shavit and N. Francez, "A New Approach to Detection of Locally Indicative Stability," in *Proceedings of the International Colloquium on Automata, Languages and Systems (ICALP)*, Rennes, France, 1986, pp. 344–358.

[7] F. Mattern, "Algorithms for Distributed Termination Detection," *Distributed Computing (DC)*, vol. 2, no. 3, pp. 161–175, 1987.

[8] E. W. Dijkstra, "Shmuel Safra's Version of Termination Detection," 1987, eWD Manuscript 998. Available at `http://www.cs.utexas.edu/users/EWD`.

[9] F. Mattern, "Global Quiescence Detection based on Credit Distribution and Recovery," *Information Processing Letters (IPL)*, vol. 30, no. 4, pp. 195–200, 1989.

[10] S.-T. Huang, "Detecting Termination of Distributed Computations by External Agents," in *Proceedings of the IEEE International Conference on Distributed Computing Systems (ICDCS)*, 1989, pp. 79–84.

[11] S. Chandrasekaran and S. Venkatesan, "A Message-Optimal Algorithm for Distributed Termination Detection," *Journal of Parallel and Distributed Computing (JPDC)*, vol. 8, no. 3, pp. 245–252, Mar. 1990.

[12] F. Mattern, H. Mehl, A. Schoone, and G. Tel, "Global Virtual Time Approximation with Distributed Termination Detection Algorithms," University of Utrecht, The Netherlands, Tech. Rep. RUU-CS-91-32, 1991. [Online]. Available: http://citeseer.nj.nec.com/mattern91global.html

[13] G. Tel and F. Mattern, "The Derivation of Distributed Termination Detection Algorithms from Garbage Collection Schemes," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 15, no. 1, pp. 1–35, Jan. 1993.

[14] J.-M. Hélary and M. Raynal, "Towards the Construction of Distributed Detection Programs, with an Application to Distributed Termination," *Distributed Computing (DC)*, vol. 7, no. 3, pp. 137–147, 1994.

[15] A. A. Khokhar, S. E. Hambrusch, and E. Kocalar, "Termination Detection in Data-Driven Parallel Computations/Applications," *Journal of Parallel and Distributed Computing (JPDC)*, vol. 63, no. 3, pp. 312–326, Mar. 2003.

[16] R. Atreya, N. Mittal, and V. K. Garg, "Detecting Locally Stable Predicates without Modifying Application Messages," in *Proceedings of the 7th International Conference on Principles of Distributed Systems (OPODIS)*, La Martinique, France, Dec. 2003, pp. 20–33.

[17] N. R. Mahapatra and S. Dutt, "An Effi cient Delay-Optimal Distributed Termination Detection Algorithm," 2004, to Appear in Journal of Parallel and Distributed Computing (JPDC).

[18] X. Wang and J. Mayo, "A General Model for Detecting Termination in Dynamic Systems," in *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS)*, Santa Fe, New Mexico, Apr. 2004.

[19] N. Mittal, S. Venkatesan, and S. Peri, "Message-Optimal and Latency-Optimal Termination Detection Algorithms for

Arbitrary Topologies," in *Proceedings of the Symposium on Distributed Computing (DISC)*, Amsterdam, The Netherlands, Oct. 2004, pp. 290–304.

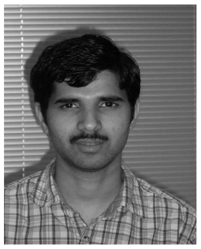
[20] J. Matocha and T. Camp, "A Taxonomy of Distributed Termination Detection Algorithms," *The Journal of Systems and Software*, vol. 43, no. 3, pp. 207–221, Nov. 1999.

[21] K. M. Chandy and J. Misra, "How Processes Learn," *Distributed Computing (DC)*, vol. 1, no. 1, pp. 40–52, 1986.

[22] T.-H. Lai, Y.-C. Tseng, and X. Dong, "A More Efficient Message-Optimal Algorithm for Distributed Termination Detection," in *Proceedings of the 6th International Parallel and Processing Symposium (IPPS)*. IEEE Computer Society, 1992, pp. 646–649.

[23] S. Venkatesan, "Reliable Protocols for Distributed Termination Detection," *IEEE Transactions on Reliability*, vol. 38, no. 1, pp. 103–110, Apr. 1989.

[24] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Communications of the ACM (CACM)*, vol. 21, no. 7, pp. 558–565, July 1978.



Sathya Peri received his M.C.S.A degree in computer science from Madurai Kamaraj University, India in 2001. He worked as a software engineer at HCL Technologies, India for one year from 2001 to 2002. He is currently pursuing his Ph.D. degree in computer science at Advanced Networking and Dependable Systems Laboratory (ANDES) at the University of Texas at Dallas. His research interests include peer-to-peer computing and dynamic distributed systems.

Neeraj Mittal received his B.Tech. degree in computer science and engineering from the Indian Institute of Technology, Delhi in 1995 and M.S. and Ph.D. degrees in computer science from the University of Texas at Austin in 1997 and 2002, respectively. He is currently an assistant professor in the Department of Computer Science and a co-director of the Advanced Networking and Dependable Systems Laboratory (ANDES) at the University of Texas at Dallas. His research interests include distributed systems, mobile computing, networking and databases.