JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 24, 159-174 (2008)

Improving the Efficacy of a Termination Detection Algorithm*

SATHYA PERI AND NEERAJ MITTAL
Department of Computer Science
The University of Texas at Dallas
Richardson, TX 75083, U.S.A.
E-mail: {sathya.p@student.utdallas.edu; neerajm@utdallas.edu}

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 any time 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 only a small impact on the performance of the given termination detection algorithm.

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

1. 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 any time. 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 [1] and Francez [2] 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.*, [3-17]). 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.*, [14-17]).

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

Received December 5, 2005; revised September 13, 2006; accepted November 1, 2006.
Communicated by Tsan-sheng Hsu.
* A preliminary version of the paper has appeared earlier in the 2004 Proceedings of the ISCA International Conference on Parallel and Distributed Computing Systems (PDCS).

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 any time after the computation has started: *simultaneous initiation* versus *delayed initiation*.

One of the earliest termination detection algorithms, which was proposed by Dijkstra and Scholten [1], assumes that the underlying computation is diffusing. Shavit and Francez [5] 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 [6], and Mattern's credit distribution and recovery algorithm [8]. 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" [17]. 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. 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 at least D message hops, where D is the diameter of the topology. We show that our transformation is efficient. Specifically, we prove that message-complexity and detection latency of the given termination detection algorithm increase by only $O(N)$ messages and $O(D)$ message hops, respectively.

Chandy and Misra [18] 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 [9] give a termination detection algorithm that can be initiated any time *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 [9] 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.

In "strict" delayed initiation, as described by Chandrasekaran and Venkatesan [9], 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. As a result, delayed initiation is possible *only if* all channels are FIFO (first-in-first-out) [9]. Lai *et al.* [19] 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*. An advantage of quasi-delayed initiation over delayed initiation is that channels *are not required* to be FIFO. Further, fewer control messages need to be exchanged with quasi-delayed initiation than with strict delayed initiation. Lai *et al.* [19] modify Dijkstra and Scholten's algorithm, which assumes diffusing computation and simultaneous initiation, to work with quasi-delayed initiation.

Mahapatra and Dutt [14] observe that the approach used by Chandrasekaran and Venkatesan [9] to achieve delayed initiation is applicable to many other termination detection algorithms as well, especially those based on acknowledgment (*e.g.*, [12]) and message-counting (*e.g.*, [6]). 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 substantially and, if yes, how. In this paper, we provide two transformations – for delayed and quasi-delayed initiation – that can be used to initiate *any* termination detection algorithm *later* after the computation has already begun. We show that our transformations for strict and quasi-delayed initiations increase the message-complexity of the given termination detection algorithm by only E and $3N$ messages, respectively without affecting its detection latency. However, we expect the *average* message complexity to be much lower because, with delayed initiation, a termination detection algorithm needs to track fewer application messages.

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 restarted and the earlier initiations of the detection algorithm be simply ignored [20, 21].

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). We believe that our results, while may have limited theoretical significance, have substantial *practical* value.

The paper is organized as follows. In section 2, 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 sections 3 and 4. We discuss the related work in section 5 and outline possible directions for future research in section 6.

2. SYSTEM MODEL AND PROBLEM STATEMENT

2.1 System Model

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. The communication topology is connected but may not be 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.

2.2 The Termination Detection Problem

The termination detection problem involves detecting when an ongoing distributed computation has terminated. A 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 any time.
Rule 4: A passive process becomes active on receiving a message.

A computation *terminates* once all activities it triggered have ceased. This happens once 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 the termination as soon as possible, that is, it has low detection latency. Detection latency is measured in terms of number of message hops. For computing detection latency, we assume 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 any time 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 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*.

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. We say that a message is *in transit* if it has been sent but not yet received.

We now describe the three transformations. We use the following notation when de-

scribing 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 $\mu(M)$ and $\delta(M)$, 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 ease of exposition, 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, $\mu(M) \geq M$ [18]. Moreover, if A can detect termination of a non-diffusing computation, then $\mu(M) \geq N - 1$ and $\delta(M) \geq D$. Finally, in the case of delayed initiation, we use $\overline{M}$ to denote the number of application messages exchanged by the primary computation after the termination detection algorithm began.

3. 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 ith secondary computation with $1 \leq i \leq N$. Intuitively, process p_i is the initiator of the ith 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: The state of process p_i with respect to $\mathcal{S}_i$ is same as its state with respect to $\mathcal{P}$. However, its state with respect to $\mathcal{S}_j$, where $j \neq i$, is passive.

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 (Invariant I1). Second, every application message exchanged by the primary computation is part of at least one secondary computation (Invariant I2). Of course, for efficiency reasons, it is desirable that a message of the primary computation belongs to at most one secondary computation. 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.

Proof: The first invariant I1 implies that no process is active with respect to any secondary computation. The second invariant I2 implies no channel contains a message belonging to any secondary computation. ❑

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 2.2. 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 the two invariants I1 and I2, 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. The first invariant implies 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 all termination detection algorithms for a diffusing computation that we know of, it is the initiator that is responsible for detecting the termination (*e.g.*, [1, 8]. We also assume that there is a distinguished process, called the *coordinator*, that is responsible for detecting termination of the primary computation. Such a distinguished process can be selected initially – when the system starts – using a leader election algorithm [22]. 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 [22]. 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. (It is not necessary for the spanning tree to be a BFS tree. We assume a BFS tree to ensure that the detection latency of the resulting termination detection algorithm is not adversely affected because the detection latency depends on the height of the spanning tree.) 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: To prove safety, note that the *coordinator* announces termination only after all secondary computations have terminated. From Lemma 1, it implies that the primary computation has terminated as well. To show 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. We assume that μ satisfies the following inequality:

$$\mu(X) + \mu(Y) \leq \mu(X + Y) + O(1) \quad (1)$$

This is a realistic assumption because for all termination detection algorithms for a diffusing computation that we know of the message complexity is either $O(M)$ [1, 6] or $O(M\,D)$ [8], both of which satisfy Eq. (1).

Theorem 3 If μ satisfies Eq. (1), then the message complexity of *DTON*(A) is given by $\mu(M) + O(N)$.

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. From Eq. (1), the sum of control messages exchanged by all secondary computations, given by $\mu(M_1) + \mu(M_2) + \ldots + \mu(M_N)$, is bounded by $\mu(M) + O(N)$. Further, there are at most $N - 1$ *terminate* messages. ❑

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 $\delta(M) + O(D)$.

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 $\delta(M_i)$ 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 $\max\{\delta(M_1), \delta(M_2), \ldots, \delta(M_N)\} + O(D)$, which in turn is bounded by $\delta(M) + O(D)$. ❑

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

4. 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 [18]. Therefore, if the underlying computation is message-intensive, then the detection algorithm may exchange a large number of control messages. Chandrasekaran and Venkatesan [10] propose a termination detection algorithm that can be started at any time 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. (Determining when to initiate the termination detection algorithm would require knowledge of the application.)

As discussed earlier, in delayed initiation in the strict sense 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 [10]. 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 by 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.

In quasi-delayed initiation, on the other hand, 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 the 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, fewer control messages need to be exchanged and channels are not required to be FIFO.

In this section, we first provide a transformation that can be used to initiate *any* termination detection algorithm in a delayed manner provided all channels are FIFO. We then describe a transformation that can be used to initiate *any* termination detection algorithm in a quasi-delayed manner even when one or more channels are not FIFO.

4.1 Strict Delayed Initiation (Channels are FIFO)

4.1.1 The main idea

To correctly detect termination with delayed initiation, we use the approach proposed in [9]. (Their approach in turn is similar to Chandy and Lamport's approach for taking a consistent snapshot of a distributed system [23].) 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 any time. 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 channels 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. But, a white application message does not carry any control information.

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.

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. 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 the process is already active with respect to the secondary computation and therefore Rule 4 is not violated. Finally, every red message is part of the secondary computation.

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. We refer to the transformation described in this section as *STOD*.

4.1.2 Proof of correctness and complexity analysis

To prove the correctness, we first 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.

Proof: Since the secondary computation has terminated, no process is active with respect to the secondary computation, and no red application message is in transit. The former implies that no process is active with respect to the primary computation. Moreover, all channels are colored red implying that no white application message is in transit. ❑

We now 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.

Proof: Since the primary computation has terminated, no process is active with respect to the primary computation. Further, no channel is colored white. This implies that no process is active with respect to the secondary computation. Again, since the primary computation has terminated, no application message is in transit. This implies that no application message tracked by the secondary computation is in transit. Combining the two, it follows that the secondary computation has also terminated. ❑

From Lemmas 5 and 6, to detect termination of the primary computation, it is safe

and live to detect termination of the secondary computation. Let A be a termination detection algorithm for a non-diffusing computation that requires simultaneous initiation. We have:

Theorem 7 Assume all channels are FIFO. Then *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. We first prove that *STOD*(A) is safe. Suppose *STOD*(A) announces termination. This implies that the secondary computation has terminated. From Lemma 5, the primary computation has also terminated. We now show that *STOD*(A) is live. Suppose the primary computation has terminated. Then, from Lemma 6, the secondary computation also terminates eventually. 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 $\mu(\bar{M}) + E$.

Proof: The message complexity of *STOD*(A) is given by the sum of Eq. (1) the number of *marker* messages exchanged and Eq. (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 $\mu(\bar{M}) + E$. ❑

Note that, in the worst case $\bar{M}$ may be same as M. Therefore the worst case message complexity of *STOD*(A) is $\mu(M) + E$. However, depending on when the termination detection is initiated, $\bar{M}$ may be much smaller than M. 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 $\delta(\bar{M})$.

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 D message hops. Moreover, once the secondary computation terminates, A detects its termination within $\delta(\bar{M})$ message hops. As a result, the detection latency of *STOD*(A) is $\max\{\delta(\bar{M}), D\}$. Since A can detect termination of a non-diffusing computation, its detection latency is at least D. Therefore the detection latency of *STOD*(A) is $\delta(\bar{M})$. ❑

4.2 Quasi-Delayed Initiation (Channels may be Non-FIFO)

We first describe modifications to our approach for strict delayed initiation to

achieve quasi-delayed initiation. We then describe modifications to reduce message complexity of the resulting termination detection algorithm from $\mu(\bar{M}) + E$ to $\mu(\bar{M}) + 3N$.

4.2.1 Modifications to the transformation for strict delayed initiation

Every process maintains the number of white application messages it has sent and received along each of its channels. Moreover, every application message is piggybacked with a bit indicating its color – white or red. This allows a process to distinguish between white and red application messages. When a process sends a *marker* message to its neighboring process, it piggybacks the number of white application messages it has sent to that neighbor along with the *marker* message. Note that, unlike in the previous case, it is now possible for a process to receive a red application message along a channel before receiving a *marker* message along that channel. For convenience, assume that a *marker* message is colored red. We require that a process turn red (and start the termination detection algorithm) just before it receives any red colored message – control or application – along any channel. Once a process has received *marker* messages from all its neighboring processes, it can ascertain whether there are any white application messages in transit towards it. This can be accomplished by comparing the number of white application messages sent to it (sum of the numbers received along with *marker* messages) with the number of white application messages it has received so far (available locally). As soon as a process has determined that it has received all white application messages that have been sent to it (by its neighboring processes), its secondary computation becomes identical to its primary computation. Until then, the process stays active with respect to its secondary computation.

4.2.2 Reducing the message complexity

Note that *marker* messages serve two purposes. First, they ensure that, eventually, all processes become red and start the termination detection algorithm. Second, using the information piggybacked on *marker* messages, each process can determine the number of white application messages in transit towards it. To reduce the message-complexity, we propose a different scheme for solving the above two problems. As in the case of transformation for handling a non-diffusing computation, we assume the existence of a distinguished process referred to as the *coordinator* and a BFS spanning tree of the communication topology rooted at the *coordinator*. The *coordinator* starts the termination detection algorithm by changing its color from white to red. It next instructs other processes to start the termination detection algorithm by broadcasting a *marker* message via the spanning tree. As before, a process turns red (and starts the termination detection algorithm) just before receiving a red colored message. The *coordinator* then collects the number of white application messages that have been sent to each process by its neighboring processes using a convergecast operation over the spanning tree. Finally, the *coordinator* informs each process about the total number of white application messages sent to it by its neighboring processes using a broadcast operation over the spanning tree. Clearly, at most $3N$ messages are exchanged by the two broadcast and one convergecast operations over a spanning tree. However, a message involved in the operations may carry a vector of size N, thereby increasing the message size to $O(N \log M)$. We refer to the transforma-

tion obtained after the above-described modifications as *STOQD*.

4.2.3 Proof of correctness and complexity analysis

The transformation *STOQD* ensures that a process stays active with respect to its secondary computation as long as there is at least one white application message in transit towards it. As a result, when the secondary computation terminates, we can conclude the following. First, there are no white application messages in transit (in any channel). Second, all processes are passive with respect to the primary computation. Third, there are no red application messages in transit in any channel. This in turn implies that the primary computation has terminated as well. Also, clearly, once the primary computation terminates, the secondary computation terminates eventually. It can be proved, in a manner similar to Lemmas 5, 6 and Theorem 7, that to detect the termination of the primary computation, it is safe and live to detect the termination of the secondary computation. The proofs have been omitted to avoid repetition.

Theorem 10 *STOQD*(*A*) correctly detects termination with quasi-delayed initiation when all channels may not be FIFO.

A broadcast or a convergecast operation over a spanning tree uses at most *N* messages. Further, the height of a BFS tree is at most *D*. Therefore, we have:

Theorem 11 The message complexity of *STOQD*(*A*) is given by $\mu(\bar{M}) + 3N$. Moreover, if *STOQD*(*A*) is initiated before the underlying computation terminates, then the detection latency of *STOQD*(*A*) is given by $\delta(\bar{M})$.

5. DISCUSSION

In [5], Shavit and Francez extend Dijkstra and Scholten's termination detection algorithm (DSTDA) for 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.

Lai *et al.* [19] 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 [5]. Intuitively, they use $N - 1$ 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 $\mu(M + N)$. In contrast, the message complexity with our transformation is $\mu(M) + O(N)$ in case $\mu(0) = O(1)$ and is $\mu(M) + O(N\mu(0) + N)$ in general. For all termination detection algorithms that we are aware of, $\mu(M)$ is of the form $\alpha M + \beta$, where α and β are functions of N, E and D. It can be proved that our transformation has comparable or better message complexity than their transformation when α is $\Omega(\beta)$. In fact, to the best of our knowledge, for all termination detection algorithms for a diffusing computation, α is $\Omega(1)$ and β is 0. For example, for Huang's weight throwing algorithm [9], α is $\Theta(D)$ (when adapted for arbitrary communication topology) and β is 0.

Lai *et al.* [20] 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.

A common approach for detecting termination is to repeatedly take snapshots of the system and test each snapshot for termination condition until the condition evaluates to true (*e.g.*, [31-33]). It is possible to delay taking snapshots until the computation is "closed" to termination. (In other words, the first snapshot is taken when the computation is believed to be closed to termination.) With this approach, as many as $O(N\overline{M} + N)$ control messages may be exchanged, where $\overline{M}$ is the number of application messages exchanged since the first instance of the snapshot algorithm is initiated. With our approach, depending on the given termination detection algorithm, only $O(\overline{M} + N)$ control messages may be exchanged [1, 9, 16].

6. CONCLUSION AND FUTURE WORK

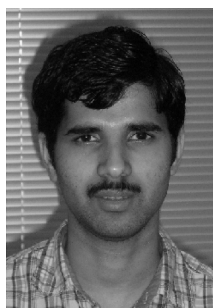
In this paper, we have presented general transformations for improving the efficacy of any termination detection algorithm. Specifically, by postponing the initiation of the termination detection algorithm, we expect to reduce the number of application messages that need to be tracked by a termination detection algorithm. This may significantly improve the *average* message complexity of the algorithm. Moreover, our transformations do not adversely affect other complexity measures of a termination detection algorithm including detection latency and message overhead.

As a future work, it would be interesting to investigate whether the notion of delayed initiation – strict and quasi – be applied to detecting other stable properties besides termination. This is important because the complexity of many stable property detection algorithms depends on the number of events that are tracked by the detection algorithm. As a result, the technique of delayed initiation can be used to significantly reduce the average number of events that needs to be tracked by the detection algorithm, thereby reducing the overhead imposed on the system by the detection algorithm.

REFERENCES

1. E. W. Dijkstra and C. S. Scholten, "Termination detection for diffusing computations," *Information Processing Letters*, Vol. 11, 1980, pp. 1-4.
2. N. Francez, "Distributed termination," *ACM Transactions on Programming Languages and Systems*, Vol. 2, 1980, pp. 42-55.
3. J. Misra, "Detecting termination of distributed computations using markers," in *Proceedings of the ACM Symposium on Principles of Distributed Computing*, 1983, pp. 290-294.
4. S. P. Rana, "A distributed solution of the distributed termination problem," *Information Processing Letters*, Vol. 17, 1983, pp. 43-46.
5. 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*, 1986, pp. 344-358.
6. F. Mattern, "Algorithms for distributed termination detection," *Distributed Computing*, Vol. 2, 1987, pp. 161-175.
7. E. W. Dijkstra, "Shmuel Safra's version of termination detection," EWD Manuscript 998, http://www.cs.utexas.edu/users/EWD/, 1987.
8. F. Mattern, "Global quiescence detection based on credit distribution and recovery," *Information Processing Letters*, Vol. 30, 1989, pp. 195-200.
9. S. Chandrasekaran and S. Venkatesan, "A message-optimal algorithm for distributed termination detection," *Journal of Parallel and Distributed Computing*, Vol. 8, 1990, pp. 245-252.
10. F. Mattern, H. Mehl, A. Schoone, and G. Tel, "Global virtual time approximation with distributed termination detection algorithms," Technical Report No. RUU-CS-91-32, University of Utrecht, The Netherlands, 1991.
11. G. Tel and F. Mattern, "The derivation of distributed termination detection algorithms from garbage collection schemes," *ACM Transactions on Programming Languages and Systems*, Vol. 15, 1993, pp. 1-35.
12. J. M. Hélary and M. Raynal, "Towards the construction of distributed detection programs, with an application to distributed termination," *Distributed Computing*, Vol. 7, 1994, pp. 137-147.
13. A. A. Khokhar, S. E. Hambrusch, and E. Kocalar, "Termination detection in data-driven parallel computations/applications," *Journal of Parallel and Distributed Computing*, Vol. 63, 2003, pp. 312-326.
14. N. R. Mahapatra and S. Dutt, "An efficient delay-optimal distributed termination detection algorithm," to appear in *Journal of Parallel and Distributed Computing*, 2004.
15. X. Wang and J. Mayo, "A general model for detecting distributed termination in dynamic systems," in *Proceedings of the 18th International Parallel and Distributed Processing Symposium*, 2004, pp. 84b.
16. N. Mittal, S. Venkatesan, and S. Peri, "Message-optimal and latency-optimal termination detection algorithms for arbitrary topologies," in *Proceedings of the 18th Symposium on Distributed Computing*, 2004, pp. 290-304.
17. J. Matocha and T. Camp, "A taxonomy of distributed termination detection algorithms," *Journal of Systems and Software*, Vol. 43, 1998, pp. 207-221.
18. K. M. Chandy and J. Misra, "How processes learn," *Distributed Computing*, Vol. 1,

1986, pp. 40-52.
19. 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*, 1992, pp. 646-649.
20. S. Venkatesan, "Reliable protocols for distributed termination detection," *IEEE Transactions on Reliability*, Vol. 38, 1989, pp. 103-110.
21. N. Mittal, F. C. Freiling, S. Venkatesan, and L. D. Penso, "Efficient reduction for wait-free termination detection in a crash-prone distributed system," in *Proceedings of the 19th Symposium on Distributed Computing*, 2005, pp. 93-107.
22. L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, Vol. 21, 1978, pp. 558-565.
23. G. Tel, *Introduction to Distributed Algorithms*, Cambridge University Press (US Server), 2nd ed., 2000.
24. H. Attiya and J. Welch, *Distributed Computing: Fundamentals*, *Simulations and Advanced Topics*, John Wiley and Sons Limited, 2nd ed., 2004.
25. V. K. Garg, *Elements of Distributed Computing*, John Wiley and Sons, Incorp., New York, 2002.
26. K. M. Chandy and L. Lamport, "Distributed snapshots: Determining global states of distributed systems," *ACM Transactions on Computer Systems*, Vol. 3, 1985, pp. 63- 75.
27. L. Bouge, "Repeated snapshots in distributed systems with synchronous communication and their implementation in CSP," *Theoretical Computer Science*, Vol. 49, 1987, pp. 145-169.
28. T. H. Lai and T. H. Yang, "On distributed snapshots," *Information Processing Letters*, Vol. 25, 1987, pp. 153-158.



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.