Message-Optimal and Latency-Optimal Termination Detection Algorithms for Arbitrary Topologies*

Neeraj Mittal S. Venkatesan Sathya Peri
Department of Computer Science
The University of Texas at Dallas
Richardson, TX 75083
{neerajm,venky}@utdallas.edu sathya.p@student.utdallas.edu

Abstract

An important problem in distributed systems is to detect termination of a distributed computation. A computation is said to have terminated when all processes have become passive and all channels have become empty. In this paper, we present a suite of algorithms for detecting termination of a non-diffusing computation for an arbitrary topology. All our termination detection algorithms have *optimal* message complexity and *optimal* detection latency under varying assumptions.

Key words: monitoring distributed system, termination detection, arbitrary communication topology, optimal algorithm, diffusing and non-diffusing computations, simultaneous and delayed initiations, single-hop and multi-hop application messages

1 Introduction

One of the fundamental problems in distributed systems is to detect termination of an ongoing distributed computation. The problem arises, for example, when computing shortest paths between pairs of nodes in a network. The distributed computation 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 only on

*A preliminary version of the paper first appeared in the 18th Symposium on Distributed Computing (DISC), 2004 [26].

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 [11] and Francez [12] more than two decades ago. Since then, many researchers have worked on this problem and, as a result, a large number of algorithms have been developed for termination detection (*e.g.*, [10, 25, 28, 29, 22, 9, 23, 16, 15, 4, 24, 31, 14, 20]). Note that termination is a stable property. Thus a simple approach for detecting termination is to repeatedly take a consistent snapshot of the underlying computation using any of the algorithms described in [5, 18, 13, 1], and then test the snapshot for termination condition. More efficient algorithms have been developed which do not depend on taking consistent snapshots of the computation. Most of the termination detection algorithms can be broadly classified into four categories, namely *computation tree based*, *invigilator based*, *double wave based* and *single wave based*.

In the computation tree based approach, a dynamic tree is maintained based on the messages exchanged by the underlying computation. A process not currently "participating" in the computation, on receiving an application message, remembers the process that sent the message (and joins the dynamic tree) until it "leaves" the computation. This creates a parent-child relationship among processes that are currently "part" of the computation. A process may join and leave the tree many times. Example of algorithms based on this idea can be found in [11, 29, 4].

In invigilator based approach, a distinguished process called the *coordinator*, is responsible for maintaining current status of all processes either directly or indirectly. The coordinator may either explicitly maintain the number of processes that are currently "participating" in the computation or may only know whether there exists at least one process that is currently "participating" in the computation (ascertained via missing credit/weight [15, 23] or some other mechanism [20]). Many algorithms in this class assume that the topology contains a star and the coordinator is directly connected to every process [15, 23]. These algorithms can be generalized to work for any communication topology at the expense of increased message complexity.

The next two classes of algorithms are based on the notion of *wave* [30]. A wave is a control message or a subset of control messages that sweep through the entire system visiting all processes on the way. As the wave travels through processes, it collects their local snapshots, which are then combined to obtain a snapshot of the entire system.

In double wave based approach, two (possibly inconsistent) snapshots of the computation are taken in such a way that there is *at least one* consistent snapshot lying between the two snapshots.

Termination Detection Algorithm	Message Complexity	Detection Latency	Communication Topology
Computation Tree Based (*e.g.*, [11])	$O(M)^{\ddagger}$	$O(N)$	any
Invigilator Based (*e.g.*, [23])	$O(M)^{\ddagger}$	$O(1)^{\ddagger}$	diameter is constant
Modified Invigilator Based* (*e.g.*, [23])	$O(MD)$	$O(D)^{\ddagger}$	any
Double Wave Based† (*e.g.*, [2])	$O(MN)$	$O(D)^{\ddagger}$	any
Single Wave Based† (*e.g.*, [22])	$O(MN)$	$O(D)^{\ddagger}$	any
Our Algorithm	$O(M)^{\ddagger}$	$O(D)^{\ddagger}$	any
Our Algorithm (non-diffusing computation)	$O(M+N)^{\ddagger}$	$O(D)^{\ddagger}$	any
Our Algorithm (non-diffusing computation and delayed initiation)	$O(\bar{M}+E)^{\ddagger}$	$O(D)^{\ddagger}$	any
Our Algorithm (non-diffusing computation and multi-hop application messages)	$O(MH+N)^{\ddagger}$	$O(D)^{\ddagger}$	any

N: number of processes in the system

E: number of channels in the communication topology

M: number of application messages exchanged by the underlying computation

$\bar{M}$: number of application messages exchanged by the underlying computation after the termination detection algorithm began

D: diameter of the communication topology

H: average number of hops traveled by application messages

$*$: invigilator based adapted for arbitrary communication topology

†: wave is collected using a breadth-first-search spanning tree to ensure optimality of detection latency

‡: complexity expression is optimal

Table 1: Comparison of various termination detection algorithms (assume *diffusing* computation, *simultaneous* initiation and *single-hop* application messages unless indicated otherwise).

The interval between the two snapshots is then tested for any possible activity. In case the interval is *quiescent* (no activity took place), termination can be announced. It can be proved that evaluating the termination condition for either of the snapshots is actually equivalent to evaluating the condition for any consistent snapshot lying between the two snapshots [2]. Various algorithms differ in the manner in which they test for quiescence of an interval and emptiness of channels. Examples of algorithms based on this idea can be found in [22, 9, 24, 14].

In single wave based approach, a snapshot of the computation is first tested for consistency.

If the test evaluates to true, then the snapshot is analyzed for the termination condition. The consistency test is such that if the snapshot is not consistent then the test will definitely evaluate to false. It is possible that the test may evaluate to false even if the snapshot is consistent. However, if the snapshot is taken after the computation has terminated, then the test is guaranteed to evaluate to true. Various algorithm differ in the manner in which they test for consistency of a snapshot and emptiness of channels. Examples of algorithms based on this idea can be found in [28, 22, 16].

In addition, termination detection algorithms can also be classified based on two other attributes: (1) whether the distributed computation starts from a single process or from multiple processes (*diffusing computation* versus *non-diffusing computation*), and (2) 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*). Delayed initiation is useful when the underlying computation is message-intensive and therefore it is preferable to start the termination detection algorithm later when the computation is "close" to termination.

Table 1 shows the (worst-case) message complexity and detection latency for the best algorithm in each of the four classes and for our algorithms. The table also indicates assumption, if any, made about the communication topology. The complexity expressions given in the table are derived under the assumptions that application messages are only exchanged between neighboring processes in the topology and message processing time is negligible compared to message transmission time. Most termination detection algorithms are analyzed under these two assumptions (*e.g.*, [11, 22, 4, 8]). Henceforth, in this paper, all complexity expressions are presented or derived under the two assumptions unless otherwise stated. In [20], Mahapatra and Dutt consider the case when application messages can be exchanged between arbitrary processes. For some algorithms in parallel computing, even non-neighboring processes may be required to exchange application messages with each other [7]. Later, in the paper, we describe how to maintain optimality of our termination detection algorithm when application messages may travel multiple hops.

Chandy and Misra 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 [6]. Also, in the worst-case, the detection latency of any termination detection algorithm measured in terms of message hops is D, where D is the diameter of the communication topology. Algorithms derived from computation tree based approach typically have optimal message complexity but non-optimal detection latency (*e.g.*, [11, 17]). On the other hand, algorithms that use invigilator based approach typically have optimal detection latency but non-optimal message complexity (*e.g.*,

[15, 23, 20]). (The message-complexity is optimal only when the diameter of the communication topology is constant.) To our knowledge, at present, there is no termination detection algorithm that has optimal message complexity as well as optimal detection latency for *all* communication topologies. The message complexity of a termination detection algorithm measures the overhead imposed by the algorithm on the system during its execution. Its detection latency measures the delay incurred between when the computation terminates and when the termination is actually detected (and announced). Clearly, it is desirable to minimize both message complexity and detection latency of a termination detection algorithm.

Note that, for a general non-diffusing computation, 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. Chandrasekaran and Venkatesan [4] prove another lower bound that if the termination detection algorithm is initiated anytime after the computation has started, then the algorithm, in the worst case, must exchange at least E control messages, where E is the number of communication channels in the topology. They also show that delayed initiation is not possible unless all channels are first-in-first-out (FIFO).

Our contributions in the paper are as follows. We present three message-optimal and latency-optimal termination detection algorithms for arbitrary communication topologies under varying assumptions such as (1) whether the initiation is simultaneous or delayed and (2) whether application messages are single-hop or multi-hop. Our first algorithm assumes that the initiation is simultaneous and application messages are single hop. Our second algorithm, which is derived from the first algorithm, assumes that the initiation may be delayed but application messages are single-hop. Our third algorithm, which is again derived from the first algorithm, assumes that the initiation is simultaneous but application messages may be multi-hop. A message-optimal and latency-optimal termination detection algorithm for the case when initiation may be delayed and application messages may be multi-hop can be obtained by combining the modifications used for second and third algorithms. All our termination detection algorithms have very low message overhead as well. Specifically, a message has to carry only one integer whose maximum value is bounded by $2D$, which is independent of the number of messages exchanged by the underlying computation. Intuitively, we achieve optimality with respect to message-complexity and detection-latency *at the same time* by combining computation tree based and invigilator based approaches.

The paper is organized as follows. In Section 2, we discuss the system model and notation used in this paper, and describe the termination detection problem. Section 3 describes an optimal

termination detection algorithm for the case when the detection algorithm has to be initiated along with the computation. Section 4 describes the modifications required to handle the case when the detection algorithm can be initiated at any time after the computation has commenced. Section 5 describes the modifications required to handle the case when application messages can be exchanged between arbitrary processes. Finally, we present our conclusion and outline direction for future research in Section 6.

2 System Model and Problem Statement

2.1 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 exchanging messages over a communication network. There is no common clock or shared memory. Processes are non-faulty and channels are reliable. Message delays are finite but may be unbounded.

We do not assume that the underlying communication topology is fully connected. Two processes can communicate directly with each other only if they are neighbors in the topology. If two processes are neighbors in the topology, then we say that there is a channel between them. We assume that all channels are bidirectional. We use E to refer to the number of channels in the communication topology.

Processes execute *events* and change their states. A *local state* of a process, therefore, is given by the sequence of events it has executed so far starting from the *initial state*. Events are either *internal* or *external*. An external event could be a *send event* or a *receive event*. An event—internal or external—causes the local state of a process to be updated. In addition, an external event causes a message to be sent (send event) or received (receive event).

Events on a process are totally ordered. However, events on different processes are only partially ordered by the Lamport's happened-before relation [19], 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 *snapshot* of the system is a collection of local states, one from each process. A local state of a process can be captured by the set of events that have been executed so far on that process. (An empty set of events denotes the initial state.) Therefore, in terms of events, a snapshot, which is also referred to as a *cut*, is a set of events satisfying the following property:

$$S \text{ is a snapshot} \quad \triangleq \quad \langle \forall e, f : e \text{ and } f \text{ are on the same process} : (e \rightarrow f) \wedge (f \in S) \;\Rightarrow\; e \in S \rangle$$

We say that a snapshot *passes through* an event if it is the last event on that process to be contained in the snapshot. A snapshot that contains the receive event of a message but not its send event is not a valid snapshot of the system. Such a snapshot is called an *inconsistent* snapshot. Conversely, we say that a snapshot (or cut) is *consistent* if the following holds:

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

Next, we formally define the termination detection problem.

2.2 The Termination Detection Problem

The termination detection problem involves detecting when an ongoing distributed computation has terminated. The distributed computation is modeled as follows. A process can be either in an *active* state or a *passive* state. A process can send a message only when it is active. An active process can become passive at anytime. 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 the underlying computation as *application messages*, and the messages exchanged by the termination detection algorithm as *control messages*. Unless indicated otherwise, we describe our termination detection algorithms assuming that application messages are only exchanged between neighboring processes, that is, application messages are *single-hop*. This is consistent with the assumption made by most termination detection algorithms (*e.g.*, [11, 22, 4, 17]). Later, in this paper, we discuss the case when application messages may be exchanged between arbitrary processes, that is, when application messages are *multi-hop*.

In this paper, when a process sends a control message, we distinguish between two cases—whether the process has created the message itself or is simply forwarding the message it has received from a neighboring process to another neighboring process. In the former case, we say that the process has *generated* the control message.

It is desirable that the termination detection algorithm exchange as few control messages as possible, that is, the algorithm has low message complexity. The higher the message complexity of a termination detection algorithm, the higher is the overhead imposed by it on the system during execution. Clearly, the overhead imposed by a termination detection algorithm should be minimized. Further, once the underlying computation terminates, the algorithm should detect it as soon as possible, that is, the algorithm has low detection latency [20]. For computing detection latency, it is typically assumed that each message hop takes one time unit and message processing time is negligible compared to message transmission time [30, 3]. Detection latency is measured in terms of number of message hops. Finally, the amount of control information carried by any message—application or control—is minimal, that is, the algorithm has low bit-message complexity.

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*.

3 An Optimal Algorithm for Simultaneous Initiation

In this section, we first describe the main idea behind our algorithm, and then give its formal description. Later, we prove the correctness of our algorithm and also show that it is message-optimal and latency-optimal. Our approach is based on combining computation tree based and invigilator based approaches. This allows us to achieve the best of both approaches, namely optimal message-complexity of computation tree based approach and optimal detection latency of invigilator based approach.

3.1 The Main Idea

We first describe the main idea behind the algorithm assuming that the underlying computation is a diffusing computation. We relax this assumption later.

3.1.1 Detecting Termination of a Diffusing Computation

First, we briefly explain the main idea behind the computation tree based and the invigilator based approaches. Then we discuss how to combine them to obtain the optimal algorithm.

Computation tree based approach: Consider a termination detection algorithm using computation tree based approach [11, 4]. Initially, only one process, referred to as the *initiator*, is active and all other processes are passive. A process, on receiving an application message, sends an *acknowledgment* message to the sender as soon as it knows that all activities triggered by the application message have ceased. The initiator announces termination as soon as it has received an *acknowledgment* message for every application message it has sent so far and is itself passive. The algorithm has optimal message complexity because it exchanges exactly one control message, namely the *acknowledgment* message, for every application message exchanged by the underlying computation. The detection latency, however, is far from optimal. Specifically, a chain of pending *acknowledgment* messages (hereafter, referred to as an *acknowledgment* chain) may grow to a length as long as M, where M is the number of application messages exchanged by the underlying computation. (The reason is that a process may appear multiple times on an *acknowledgment* chain as is the case with the algorithm of [4].)

The detection latency of the algorithm can be reduced from $O(M)$ to $O(N)$ (assuming $M = \Omega(N)$) as follows [11]. Suppose a process has not yet sent an *acknowledgment* message for an application message it received earlier. In case the process receives another application message, it can immediately send an *acknowledgment* message for the latter application message. For termination detection purposes, it is sufficient to assume that all computation activities triggered by the receipt of the latter application message are triggered by the former application message. We refer to the former application message as an engaging application message and to the latter as a non-engaging application message.

Observe that the set of engaging application messages imposes a parent-child relationship among processes "currently participating" in the computation. Specifically, if a process is active or has not yet received an *acknowledgment* message for every application message it has sent so far, then it is "currently a part" of the computation and is referred to as a nonquiescent process. Otherwise, it is "not currently a part" of the computation and is referred to as a quiescent process. At any time, the computation tree, which is dynamic, consists of the set of processes that are nonquiescent at that time.

Invigilator based approach: Now, consider a termination detection algorithm using the invigilator based approach [20]. (The algorithm described here is actually a simplified version of the algorithm presented in [20] but, nevertheless, captures the main idea.) One process is chosen to act

as the coordinator. The coordinator is responsible for maintaining the current status of all processes in the system either directly or indirectly. Suppose a process receives an application message. In case the coordinator does not already know that it is currently active, it sends a control message indicating "I am now active" to the coordinator. Once the process knows that the coordinator has received the control message, it sends an *acknowledgment* message to the sender of the application message. On the other hand, if the process has already informed the coordinator that it is currently active, then it immediately acknowledges the application message. Once a process becomes passive and has received an *acknowledgment* message for every application message it has sent so far, it sends a control message indicating "I am now passive" to the coordinator. Intuitively, if the underlying computation has not terminated, then, as per the coordinator, at least one process is currently active. When the coordinator is directly connected to every process in the system, the algorithm has optimal message complexity (at most three control messages for every application message) and optimal detection latency (which is $O(1)$). When the topology is arbitrary, however, for communication between the coordinator and other processes, a static breadth-first-search (BFS) spanning tree rooted at the coordinator has to be constructed. Every control message that a process sends to the coordinator (along the BFS spanning tree) may cause up to D additional control messages to be exchanged, thereby increasing the message complexity to $O(MD)$.

Achieving the best of the two approaches: As explained above, in the computation-tree based approach, a process reports its status, when it becomes quiescent, to its parent. On the other hand, in the invigilator based approach, a process reports its status, when it becomes quiescent, to the coordinator (directly or indirectly). The main idea is to restrict the number of times processes report their status to the coordinator—to achieve optimal message complexity—and, at the same time, restrict the length of an *acknowledgment* chain—to achieve optimal detection latency.

Whenever a process reports its status to the coordinator, as many as D control messages may have to be exchanged. As a result, to achieve optimal message complexity, the number of times when processes report their quiescent status to the coordinator should be bounded by $O(M/D)$. The rest of the times processes should report their quiescent status to their respective parents in the computation tree. To ensure optimal detection latency, the length of an *acknowledgment* chain should be bounded by $O(D)$. The main problem is to determine, while the computation is executing, when a process should choose the former over the latter. In our algorithm, a process, by default, is supposed to report its status to its parent until it learns that the length of a chain

of pending *acknowledgment* messages, starting from it, has become sufficiently long, that is, the length of the chain has become $\Theta(D)$. At that time, it starts reporting its status to the coordinator. Specifically, it first sends an *st_active* message signifying that "my computation subtree is currently active" to the coordinator. It waits until it has received an acknowledgment from the coordinator in the form of an *ack_st_active* message. The receipt of the *ack_st_active* message implies that the coordinator is aware of some activity in the system and therefore will not announce termination as yet. It then sends an *acknowledgment* message to its parent, thereby breaking the link with its parent and shortening the *acknowledgment* chain. Later, when it becomes quiescent, it sends an *st_passive* message indicating "my computation subtree has now become passive" to the coordinator.

To measure the length of an *acknowledgment* chain, we piggyback an integer counter (referred to as hop counter) on every application message that represents the current length of an *acknowledgment* chain. On receiving an application message, if a process learns that the length of the *acknowledgment* chain has become at least D, then it resets the value of the hop counter to zero. Further, it sends a special control message, referred to as a *detach* message, to the process at a distance of D from it along the *acknowledgment* chain but in the reverse direction. The objective of a *detach* message is to instruct the intended recipient that it should break the link with its parent, become the "head" of the chain and report its status to the coordinator instead of reporting to its parent. (The details of how this happens are discussed in the previous paragraph.) The reason is that the overhead incurred on exchanging control messages with the coordinator, namely *st_active*, *ack_st_active* and *st_passive*, can now be amortized over enough number of processes so as not to affect the message complexity adversely. Note that a process may have multiple chains of *acknowledgment* messages emanating from it. As a result, there may be multiple processes that are at a distance of D from it, all of which generate *detach* messages destined for it. This may increase the message complexity significantly. To that end, we propagate *detach* messages upward along an *acknowledgment* chain in a "modified" convergecast fashion. If a process has already sent a *detach* message to its parent since last becoming non-quiescent, then it ignores any subsequent *detach* message it receives from any of its other children (in the computation tree). Clearly, *at most one* *detach* message is sent in each non-quiescent interval. As a result, the total number of *detach* messages exchanged by the termination detection algorithm is upper-bounded by the total number of application messages exchanged by the underlying computation.

Example 1 Figure 1 illustrates the main idea behind our termination detection algorithm. Suppose process p_i, on receiving an engaging application message m, learns that the length of the

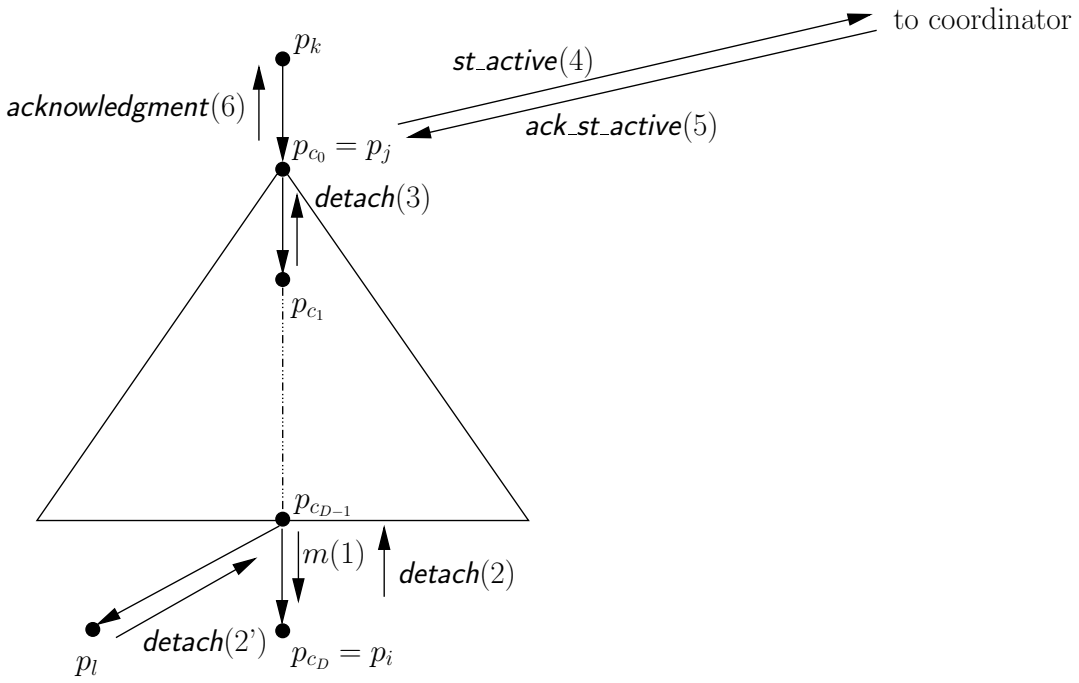

Figure 1: An illustration of the termination detection algorithm.

acknowledgment chain has become at least D. Let the last $D+1$ processes along the chain be denoted by $p_j = p_{c_0}, p_{c_1}, \ldots, p_{c_D} = p_i$. As per our algorithm, p_i generates a *detach* message and sends the message to its parent $p_{c_{D-1}}$. The *detach* message is propagated upward all the way to p_j, which is at a distance of D hops from p_i. Process p_j, on receiving the *detach* message, sends an *st_active* message to the coordinator. The coordinator, on receiving the *st_active* message, sends an *ack_st_active* message to p_j. On receiving the *ack_st_active* message, p_j sends an *acknowledgment* message to its parent, say process p_k, thereby breaking the chain. Numbers in the parentheses show the sequence in which various control messages are exchanged. It is possible that $p_{c_{D-1}}$ has another child, namely process p_l, which also sends a *detach* message to $p_{c_{D-1}}$ destined for p_j. On receiving the second *detach* message, $p_{c_{D-1}}$ simply ignores the message and does not forward it to its parent $p_{c_{D-2}}$.

Note that process p_i is still attached to its parent $p_{c_{D-1}}$. Now, suppose the chain grows further by D more processes and is now given by p_{c_0} $(= p_j)$, p_{c_1}, ..., p_{c_D} $(= p_i)$, $p_{c_{D+1}}$, ..., $p_{c_{2D}}$. As per our algorithm, $p_{c_{2D}}$ generates a *detach* message, which is propagated via processes $p_{c_{2D-1}}$, ..., $p_{c_{D+1}}$ to p_i. Process p_i, on receiving the first *detach* message, breaks the link with its parent $p_{c_{D-1}}$, thereby reducing the length of the chain emanating from p_j. □

Information description: The computation starts from the initially active process. As the computation exchanges application messages, a tree (sometimes referred to as computation tree) is induced on processes by engaging application messages. A tree grows whenever a process in the tree generates an engaging application message and shrinks whenever an engaging application message in the tree is acknowledged. Once the height of a subtree rooted at a process—for which the value of the hop counter is zero—becomes at least D, within $O(D)$ message hops, the process detaches itself from its parent and the subtree rooted at the process becomes a *separate* computation tree. A root process, which has detached itself from its parent, reports its status to the coordinator, and every other process reports its status to its parent in the tree. Whenever a tree becomes empty, its root process informs the coordinator about it. Once all trees have become empty, which happens once all application messages have been acknowledged, the coordinator announces termination. Our termination detection algorithm ensures that the coordinator announces termination if and only if there is no non-empty computation tree in the system.

Message-complexity: Our algorithm exchanges five different types of control messages, namely *acknowledgment*, *detach*, *st_active*, *st_passive* and *ack_st_active*. One *acknowledgment* message is exchanged for every application message. Also, a process sends at most one *detach* message for every engaging application message it receives. Therefore the total number of *acknowledgment* and *detach* messages is upper-bounded by $2M$. The number of *st_active* messages generated by all processes combined is given by $O(M/D)$. This is because a process sends an *st_active* message only when it knows that there are at least $O(D)$ processes in its computation subtree. Each *st_active* message is sent on the BFS spanning tree and, therefore, may result in at most D control messages being exchanged. Finally, the number of *st_passive* messages as well as the number of *ack_st_active* messages is equal to the number of *st_active* messages. Thus the message complexity of our algorithm is $O(M)$.

Detection-latency: Our algorithm ensures that whenever the length of a chain of pending *acknowledgment* messages grows beyond $2D$, within $3D + 1$ message hops (consisting of *detach*, *st_active* and *ack_st_active* messages), the chain is reduced to a length smaller than D. Therefore the detection latency of our algorithm is $O(D)$.

3.1.2 Generalizing to a Non-Diffusing Computation

Assume that two or more processes are active initially, that is, there are multiple initiators of the computation. Intuitively, the coordinator should announce termination only after every initiator has informed it that the computation triggered by it has terminated. The coordinator, however, does not know how many initiators of the computation are there. Therefore, every process, on becoming quiescent for the first time (including the case when it is quiescent to begin with), sends an *initialize* message to the coordinator. The coordinator announces termination only after it has received an *initialize* message from every process (and, of course, a matching *st_passive* message for every *st_active* message). The *initialize* messages are propagated to the coordinator in a convergecast fashion, thereby resulting in only $O(N)$ more messages.

3.2 The Algorithm

A formal description of the termination detection algorithm TDA-SI for simultaneous initiation is given in Figure 2-4. Actions A0-A8 described in Figure 2 and Figure 3 capture the behavior of a process as part of the computation tree. Actions B1-B3 given in Figure 4 describe the behavior of a process as part of the BFS spanning tree. The main function of a process as part of the spanning tree is to propagate messages, namely *initialize*, *st_active*, *ack_st_active* and *st_passive*, back and forth between the coordinator and its descendants in the spanning tree. For ease of exposition of the algorithm, we assume that whenever a process wants to send a control message to the coordinator (for instance, the *initialize* message), it sends that message to itself. The message is then handled either by action B1 or by action B2, and propagated upwards to the coordinator. Likewise, when a process receives an *ack_st_active* message from its parent in the spanning tree, it either propagates that message to one of its children in the spanning tree or sends the message to itself. In the latter case, the message is handled by action A6.

In the formal description our algorithm, whenever a process becomes nonquiescent, we classify it either as a *root process* or a *non-root process*. (The classification for process p_i is captured using variable $root_i$.) The classification depends on how a process becomes nonquiescent. If a process is initially active, then it is classified as root. If a process becomes nonquiescent on receiving an application message with the counter value of $D-1$ (which is reset to zero on incrementing), then the process is classified as root as well. In all other cases, a process is classified as non-root. Note that the same process may be classified as root and non-root at different times during its execution.

However, the classification does not change during a single nonquiescent interval.

When a process becomes nonquiescent as a root, it does not immediately break its link with its parent, if it exists. It breaks the link only after receiving a *detach* message from one of its children in the computation tree. Receipt of a *detach* message implies that the computation subtree rooted at the process contains at least D processes. As a result, the process can start reporting its status to the coordinator instead of its parent. The status of the link—whether it is intact or has been broken—is captured using the variable $independent_i$.

We next prove that the termination detection algorithm TDA-SI described in Figure 2-4 is safe and live.

3.3 Proof of Correctness

Many of our proofs involve induction on either the depth or the height of a vertex in a tree. Recall that the *depth* of a vertex v in a tree, denoted by $depth(v)$, is the length of the path from the root of the tree to v. Also, its *height*, denoted by $height(v)$, is the length of a longest path from v to a leaf. The two definitions can be easily generalized for vertex in a *forest*.

A process, on sending an *st_active* message to the coordinator, expects to receive an *ack_st_active* message eventually. Note that it is easy to route an *st_active*/*st_passive* message from a non-coordinator process to the coordinator. However, routing an *ack_st_active* message from the coordinator to the process that generated the corresponding *st_active* message is non-trivial. One approach to achieve this is by piggybacking the identity of the generating process on the *st_active* message which can then be used to appropriately route the corresponding *ack_st_active* message. This, however, increases the message overhead to $O(\log N)$. Moreover, with this approach, every process needs to know the set of descendants of each of its children in the static spanning tree. Instead, we employ the following mechanism. Every process on the BFS spanning tree propagates the k^{th} *ack_st_active* message to the sender of the k^{th} *st_active* message. This can be accomplished by maintaining a FIFO queue at each process that records the (immediate) sender of every *st_active* message that a process receives. Later, on receiving an *ack_st_active* message, the process uses the queue to forward the *ack_st_active* message to the appropriate process, which is either itself or one of its children. The next lemma can be proved by a simple induction on the depth of a process in the BFS spanning tree. The lemma states that if a process receives a matching *ack_st_active* message for its *st_active* message, then the coordinator "knows" that its subtree is "active"

Termination detection algorithm for process p_i:

Variables:

D: diameter of the topology;
$state_i$:= my initial state; // *whether I am active or passive*
$missing_i := 0$; // *number of unacknowledged application messages*
$hops_i := 0$; // *hop count: my distance from a root process*
$parent_i := \perp$; // *process which made me nonquiescent*
$independent_i$:= true; // *if root, can I detach myself from my parent?*
$pending_i := 0$; // *the number of unacknowledged st_active messages*

// *Actions of process p_i as part of the computation tree*

Useful expressions:

$quiescent_i \triangleq (state_i = passive) \wedge (missing_i = 0)$;
$root_i \triangleq$ **not**$(quiescent_i) \wedge (hops_i = 0)$

(A0) Initial action:

call sendIfQuiescent( ); // *send an initialize message if passive*

(A1) On sending an application message m to process p_j:

send $\langle m, hops_i \rangle$ to process p_j;
$missing_i := missing_i + 1$; // *one more application message to be acknowledged*

(A2) On receiving an application message $\langle m, count \rangle$ from process p_j:

if not$(quiescent_i)$ **then** // *a non-engaging application message*
send $\langle$*acknowledgment*$\rangle$ message to process p_j;
else // *an engaging application message*
$parent_i := p_j$;
$hops_i := (count + 1)$ mod D;
if $root_i$ **then**
send $\langle$*detach*$\rangle$ message to $parent_i$; // *instruct root of my parent's subtree to detach*
$independent_i$:= false; // *but I am still attached to my parent*
endif;
endif;
$state_i := active$;
deliver m to the application;

(A3) On receiving $\langle$*acknowledgment*$\rangle$ message from process p_j:

$missing_i := missing_i - 1$; // *one more application message has been acknowledged*
call acknowledgeParent( ); // *send acknowledgment to my parent if quiescent*
call sendIfQuiescent( ); // *send initialize/st_passive message if quiescent*

(A4) On changing state from active to passive:

call acknowledgeParent( ); // *send acknowledgment to my parent if quiescent*
call sendIfQuiescent( ); // *send initialize/st_passive message if quiescent*

Figure 2: Termination detection algorithm TDA-SI for simultaneous initiation.

```
Termination detection algorithm for process p_i (continued):

(A5) On receiving ⟨detach⟩ message from process p_j:
     if (root_i ∧ not(independent_i)) then          // should I handle detach message myself?
          independent_i := true;                    // I can now detach myself from my parent
          send ⟨st_active⟩ to myself;               /* send st_active message to the coordinator
                                                       (the message is handled by action B2) */
          pending_i := pending_i + 1;
     else if not(root_i) then                       // detach message is meant for the root of my subtree
          if (have not yet forwarded a detach message
              to parent_i since last becoming nonquiescent) then
               send ⟨detach⟩ message to parent_i;
          endif;
     endif;

(A6) On receiving ⟨ack_st_active⟩ message from myself;
     pending_i := pending_i − 1;                    // one more st_active message has been acknowledged
     call acknowledgeParent( );                     // may need to send acknowledgment to my parent

(A7) On invocation of acknowledgeParent( ):
     if (quiescent_i or
        (root_i ∧ independent_i ∧ (pending_i = 0))) then
          if (parent_i ≠ ⊥) then                    // do I have a parent?
               send ⟨acknowledgment⟩ message to parent_i;
               parent_i := ⊥;
          endif;
     endif;

(A8) On invocation of sendIfQuiescent( ):
     if (root_i ∧ independent_i ∧ quiescent_i) then  // should I send initialize/st_passive message?
          if (have not yet sent an initialize message) then
               send ⟨initialize⟩ message to myself;  /* send initialize message to the coordinator
                                                        (the message is handled by action B1) */
          else send ⟨st_passive⟩ to myself; endif;   /* send st_passive message to the coordinator
                                                        (the message is handled by action B2) */
     endif:
```

Figure 3: Termination detection algorithm TDA-SI for simultaneous initiation (continued).

Lemma 1 *A process receives a matching* ack_st_active *message for its* st_active *message only after the* st_active *message has been received by the coordinator.*

Proof: The lemma can be proved by a simple induction on the depth of a process in the BFS spanning tree using the observation that every process sends the k^{th} *ack_st_active* message to the sender of the k^{th} *st_active* message. □

We say that a process is *quiescent* if it is passive and has received an *acknowledgment* message

Termination detection algorithm for process p_i (continued):

// *Actions of process p_i as part of the BFS spanning tree*

Variables:

$father_i$: parent in the BFS spanning tree;
$sons_i$: number of children in the BFS spanning tree;
$activity_i := 0$; // *activity counter: number of active subtrees*
$whoSent_i :=$ empty queue; /* *records the sender of each st_active message (k^{th} ack_st_active message is sent to the process from which k^{th} st_active message was received)* */

Useful expressions:

$coordinator_i \triangleq (father_i = p_i)$;
$terminated_i \triangleq$ (have received $sons_i + 1$ *initialize* messages) $\wedge (activity_i = 0)$;

(B1) On receiving $\langle$*initialize*$\rangle$ message from process p_j:
if $coordinator_i$ **then**
if $terminated_i$ **then** announce termination; **endif**;
else if (have received $sons_i + 1$ *initialize* messages) **then**
send $\langle$*initialize*$\rangle$ message to $father_i$;
endif;

(B2) On receiving $\langle$*st_status*$\rangle$ message (*st_status* $\in$ {*st_active*, *st_passive*}) from process p_j:
if $coordinator_i$ **then**
if (*st_status* = *st_passive*) **then** $activity_i := activity_i - 1$;
else
$activity_i := activity_i + 1$;
send $\langle$*ack_st_active*$\rangle$ message to process p_j;// *acknowledge the st_active message*
endif;
if $terminated_i$ **then** announce termination; **endif**;
else
if (*st_status* = *st_active*) **then**
enqueue p_j to $whoSent_i$; // *record the sender*
endif;
send $\langle$*st_status*$\rangle$ message to $father_i$; // *forward st_status message to my father*
endif;

(B3) On receiving $\langle$*ack_st_active*$\rangle$ message from $father_i$:
$p :=$ dequeue $whoSent_i$;
send $\langle$*ack_st_active*$\rangle$ message to process p; /* *send k^{th} ack_st_active message to the process from which k^{th} st_active message was received* */

Figure 4: Termination detection algorithm TDA-SI for simultaneous initiation (continued).

for every application message it has sent so far. We partition the events on a process into two categories: *quiescent* and *nonquiescent*. An event is said to be quiescent if the process becomes quiescent immediately after executing the event; otherwise it is nonquiescent. A maximal sequence of contiguous quiescent events on a process is called a *quiescent interval*. The notion of *nonquiescent*

interval can be similarly defined. An interval is created as soon as its starting event is executed, and is completed once its last event is executed. An execution of a process can be viewed as an alternating sequence of quiescent and nonquiescent intervals.

We also partition the set of application messages into two categories: *engaging* and *non-engaging*. An application message is said to be *engaging* if its destination process, on receiving the message, changes its status from quiescent to nonquiescent; otherwise it is non-engaging.

Observe that the set of engaging application messages induces a forest (of trees) on the set of nonquiescent intervals. Specifically, given two nonquiescent intervals x and y, there is an edge from x to y in the forest, denoted by $x \mapsto y$, if an engaging application message sent during x is received during y. Let NQI denote the set of all nonquiescent intervals. It can be verified that $\langle NQI, \mapsto \rangle$ is indeed a forest of trees. For an interval x, let $proc(x)$ refer to the process on which events in x are executed. The next lemma proves that if the computation terminates eventually, then the execution of every process ends with a quiescent interval.

Lemma 2 *Assume that the underlying computation eventually terminates. Then, every nonquiescent process eventually becomes quiescent.*

Proof: Assume that the underlying computation has terminated. Therefore once a process becomes quiescent it stays quiescent. This implies that the set of nonquiescent intervals NQI is finite. The proof is by induction on the height of a nonquiescent interval in the forest $\langle NQI, \mapsto \rangle$. A process acknowledges a non-engaging application message immediately. Thus it is sufficient to show that every engaging application message is eventually acknowledged. Consider a nonquiescent interval $x \in NQI$ with $proc(x) = p_i$.

Base Case $[height(x) = 0]$**:** In this case, all application messages sent in x are non-engaging. Therefore process p_i eventually becomes quiescent.

Induction Step $[height(x) > 0]$**:** Consider a nonquiescent interval y with $x \mapsto y$ Clearly, $height(y) < height(x)$. Therefore, using induction hypothesis, $proc(y)$ eventually becomes quiescent. This, in turn, implies that p_i eventually receives an *acknowledgment* message for the engaging application message it sends during x to $proc(y)$. Since y is chosen arbitrarily, we can infer that p_i eventually receives an *acknowledgment* message for every engaging application message it sends during x. Therefore p_i eventually becomes quiescent. □

From the algorithm, a process sends an *initialize* message when it becomes quiescent for the first time (including the case when it is quiescent to begin with). The following proposition can be easily verified:

Proposition 3 *Assume that the underlying computation eventually terminates. Then, every process eventually sends an* initialize *message. Moreover, a process sends an* initialize *message only when it is quiescent for the first time.*

It is important for the correctness of our algorithm that the coordinator receives *st_active* and *st_passive* messages in correct order. If channels are FIFO, then this can be achieved easily. If one or more channels are non-FIFO, then the algorithm has to be slightly modified. Details of the modifications required are described in Section 3.5. For now, assume that all channels are FIFO. We have,

Proposition 4 *The* st_active *and* st_passive *messages sent by a process are received by the coordinator in the order in which they are sent.*

The following lemma establishes that if the computation terminates then every process sends an equal number of *st_active* and *st_passive* messages in alternate order.

Lemma 5 *Each process sends a possibly empty sequence of* st_active *and* st_passive *messages in an alternate fashion, starting with an* st_active *message. Furthermore, if the underlying computation eventually terminates, then every* st_active *message is eventually followed by an* st_passive *message.*

Proof: The execution of a process can be viewed as an alternating sequence of quiescent and nonquiescent intervals. If a process is initially passive, then the execution starts with a quiescent interval; otherwise it starts with a nonquiescent interval. Also, if the underlying computation eventually terminates, then, from Lemma 2, the execution of every process ends with a quiescent interval.

From Proposition 3, every process sends an *initialize* message in the first quiescent interval. But the first interval for an initially active process is a nonquiescent interval. It can be verified that an initially active process does not send any *st_active* message in the first nonquiescent interval. This is because a process sends an *st_active* message only when it changes it detaches itself from its parent in the computation tree (action A5).

Finally, it can be verified that after the first quiescent interval, if a process sends an *st_active* message in a nonquiescent interval, then it sends an *st_passive* message in the following quiescent interval. □

We refer to the difference between the number of *st_active* and *st_passive* messages received by the coordinator as the *activity counter*. Using Proposition 4 and Lemma 5, it follows that:

Corollary 6 *The activity counter at the coordinator always has a non-negative value. Moreover, immediately after processing an* st_active *message, the value of the activity counter is positive.*

Also, from Lemma 5, it follows that:

Corollary 7 *Assume that the underlying computation eventually terminates. Then, for every* st_active *message the coordinator receives, it eventually receives a matching* st_passive *message.*

We are now ready to prove the correctness of our algorithm. First, we prove that our algorithm is live.

Theorem 8 (TDA-SI is live) *Assume that the underlying computation eventually terminates. Then, the coordinator eventually announces termination.*

Proof: To establish the liveness property, it suffices to show that the following two conditions hold eventually. First, the coordinator receives all *initialize* messages it is waiting for. Second, the activity counter at the coordinator becomes zero permanently.

Note that *initialize* messages are propagated to the coordinator in a convergecast fashion. From Proposition 3, eventually every process sends an *initialize* message. It can be easily verified that every process on the BFS spanning tree will eventually send an *initialize* message to its parent in the spanning tree. As a result, the first condition holds eventually.

For the second condition, assume that the underlying computation has terminated. Then, from Lemma 2, every process eventually becomes quiescent and stays quiescent thereafter. This implies that every process sends only a finite number of *st_active* and *st_passive* messages. Therefore the coordinator also receives only a finite number of *st_active* and *st_passive* messages. Furthermore, from Corollary 7, the coordinator receives an equal number of *st_active* and *st_passive* messages. □

Finally, we prove that our algorithm is safe, that is, it never announces false termination.

Theorem 9 (TDA-SI is safe) *The coordinator announces termination only after the underlying computation has terminated.*

Proof: Consider only those processes that become active at least once. Let *announce* denote the event on executing which the coordinator announces termination, and let lqe_i denote the *last* quiescent event on process p_i that happened-before *announce*. Such an event exists for every process. This is because the coordinator announces termination only after it has received all *initialize* messages it is waiting for. This, in turn, happens only after every process has sent an *initialize* message, which a process does only when it is quiescent.

Consider the snapshot S of the computation consisting of all lqe events. Assume, on the contrary, that the computation has not terminated for S and that some process becomes active after S. Let NQE denote the set of nonquiescent events executed in the future of S. Consider a *minimal* event mqe in NQE—minimal with respect to the happened-before relation. Formally,

$$\langle \forall x : x \in NQE : x \not\rightarrow mqe \rangle$$

Clearly, mqe occurred on receiving an engaging application message, say m. Moreover, m is a message sent from the past of S to the future of S. Otherwise, it can be shown that mqe is not a minimal event in NQE—a contradiction. Let m be sent by process p_j to process p_i. Also, let $snd(m)$ and $rcv(m)$ correspond to the send and receive events of m, respectively. Then, $snd(m) \rightarrow lqe_j$. This implies that p_j becomes quiescent after sending m. Therefore it receives the *acknowledgment* message for m, denoted by $ack(m)$, before executing lqe_j. This is depicted in Figure 5(a). There are two cases to consider:

Case 1: Process p_i sends the *acknowledgment* message for m on executing a quiescent event, say qe (see Figure 5(b)). Clearly, the *acknowledgment* message creates a causal path from qe to lqe_j. We have,

$$\begin{array}{ll} & (qe \text{ is a quiescent event on } p_i) \wedge (lqe_i \rightarrow qe) \wedge (qe \rightarrow lqe_j) \wedge (lqe_j \rightarrow announce) \\ \Rightarrow & \{ \rightarrow \text{is transitive} \} \\ & (qe \text{ is a quiescent event on } p_i) \wedge (lqe_i \rightarrow qe) \wedge (qe \rightarrow announce) \end{array}$$

In other words, qe is a quiescent event on p_i that happened-before *announce* and is executed after lqe_i. This contradicts our choice of lqe_i.

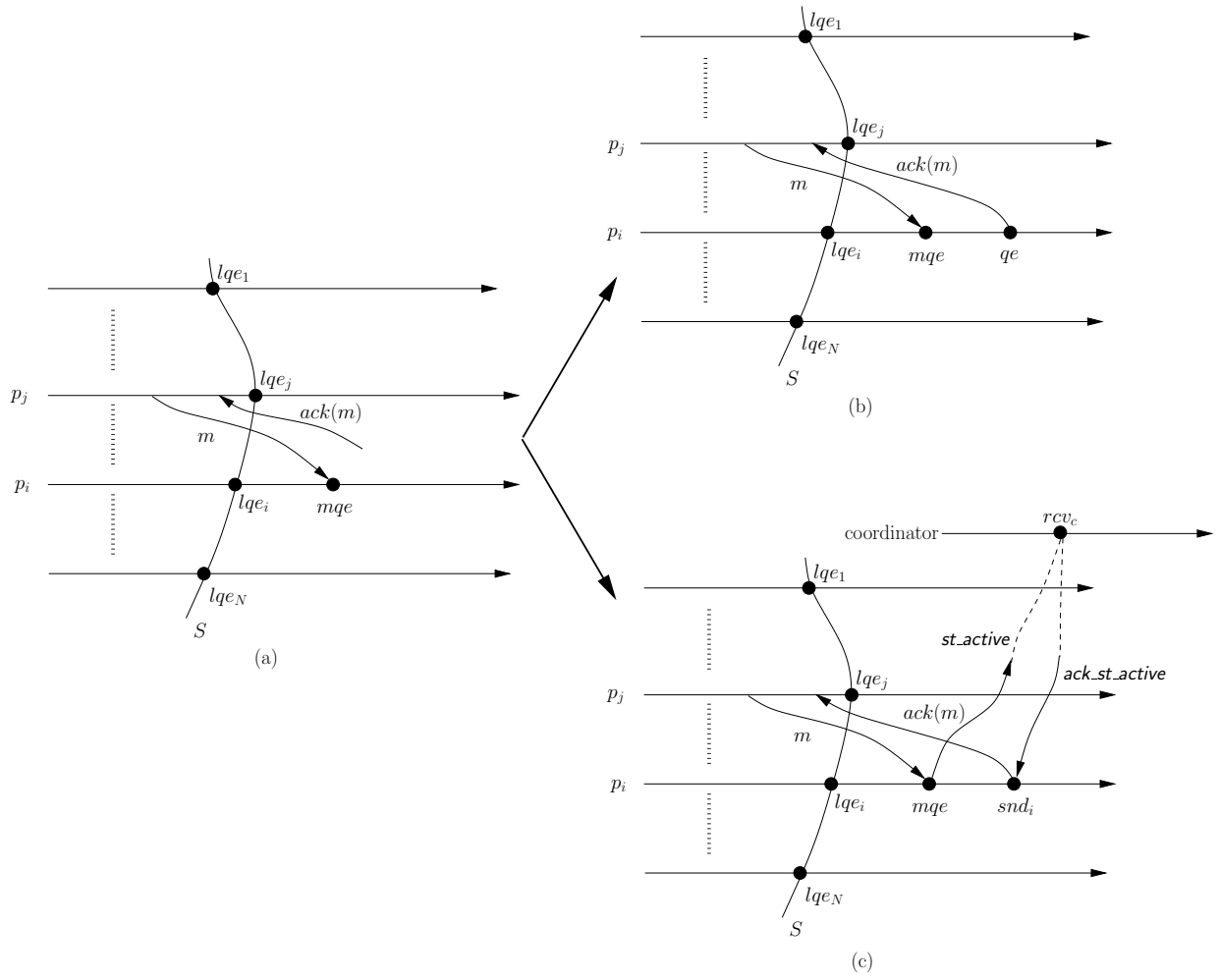

Figure 5: Proving the safety of TDA-SI.

Case 2: Process p_i sends an *acknowledgment* message for m before becoming quiescent. This happens only when p_i receives an *ack_st_active* message for the *st_active* message it sends in the current nonquiescent interval (which starts with mqe). Let the receive event of the *st_active* message on the coordinator be denoted by rcv_c (see Figure 5(c)). Also, let the send event of $ack(m)$ on process p_i be denoted by snd_i. Using Lemma 1, $rcv_c \rightarrow snd_i$. Therefore we have,

$$
\begin{aligned}
& (rcv_c \rightarrow snd_i) \wedge (snd_i \rightarrow lqe_j) \wedge (lqe_j \rightarrow announce) \\
\Rightarrow \quad & \{ \rightarrow \text{is transitive } \} \\
& rcv_c \rightarrow announce
\end{aligned}
$$

From Corollary 6, immediately after executing rcv_c, the value of the activity counter at the coordinator is greater than zero. For the coordinator to announce termination, its activity counter should be zero. This implies that the coordinator receives a matching *st_passive* message from p_i

later but before announcing termination. Clearly, p_i sends this *st_passive* message only on executing some quiescent event after mqe. This again contradicts our choice of lqe_i. □

We next prove that TDA-SI is both message-optimal and latency-optimal.

3.4 Proof of Optimality

For a nonquiescent interval x with $proc(x) = p_i$, let $hops(x)$ denote the value of the variable $hops_i$ during the interval x. From the algorithm (action A2),

$$x \mapsto y \quad \Rightarrow \quad hops(y) = (hops(x) + 1) \bmod D \tag{1}$$

To prove the optimality of TDA-SI, the following proposition comes in useful.

Proposition 10 *For a nonquiescent interval $x \in NQI$ with $hops(x) = 0$, if $height(x) \geqslant D$, then $proc(x)$ eventually receives a detach message during x (that is, before the interval x ends) and vice versa.*

The above proposition holds as long as the *acknowledgment* message for an engaging application message does not "overtake" any *detach* message sent earlier. Clearly, no "overtaking" occurs if all channels are FIFO. In case one or more channels are non-FIFO, the algorithm TDA-SI has to be modified slightly to ensure that Proposition 10 holds. Details of the modifications required are described in Section 3.5. We now show that our algorithm is message-optimal.

Theorem 11 (TDA-SI is message-optimal) *Assume that the underlying computation eventually terminates. Then, the number of control messages exchanged by the algorithm is given by $\Theta(M + N)$, where N is the number of processes in the system and M is the number of application messages exchanged by the underlying computation.*

Proof: Our algorithm exchanges six different types of control messages, namely *acknowledgment*, *detach*, *initialize*, *st_active*, *st_passive* and *ack_st_active*. We now bound each of the six types of control messages.

The number of *acknowledgment* messages is same as the number of application messages M. A process sends at most one *detach* message per engaging application message. Therefore the total number of *detach* messages is upper-bounded by M. Every process sends at most one *initialize* message. Further, *initialize* messages are propagated to the coordinator in a convergecast fashion. Hence the total number of *initialize* messages exchanged by processes is given by $O(N)$.

Every *st_active* and *st_passive* message has to be propagated to the coordinator along the BFS spanning tree and, therefore, may cause up to D additional control messages to be exchanged. Likewise, an *ack_st_active* message may also cause up to D additional control messages to be exchanged. From Lemma 5, every process generates an equal number of *st_active* and *st_passive* messages. Moreover, the number of *ack_st_active* messages a process receives is equal to the number of *st_active* messages it sends. Thus it is sufficient to show that the total number of *st_active* messages generated by all processes combined is bounded by M/D.

Observe that a process sends an *st_active* message only when it is nonquiescent and, moreover, it sends at most one *st_active* message per nonquiescent interval (action A5). We, therefore, bound the number of nonquiescent intervals in which an *st_active* message is sent. Let $I \subseteq NQI$ denote the set of those nonquiescent intervals during which an *st_active* message is sent. Also, observe that a process sends an *st_active* message during a nonquiescent interval only if the interval is created on receiving an engaging application message (action A5). In other words, if a process is nonquiescent to begin with, it does not send any *st_active* message during the (initial nonquiescent) interval. Let $NI \subseteq NQI$ denote the set of nonquiescent intervals created on receiving an engaging application message. We have,

$$I \subseteq NI \subseteq NQI \quad \text{and} \quad |NI| \leqslant M \tag{2}$$

Consider a nonquiescent interval $x \in NQI$. From the algorithm (action A5),

$$x \in I \quad \Rightarrow \quad hops(x) = 0 \tag{3}$$

We define $childset(x)$ as the set of those nonquiescent intervals which are at a distance of at most $D-1$ message hops from x in the forest $\langle NQI, \mapsto \rangle$, and refer to it as the *childset* of x. Clearly, $childset(x) \subseteq NI$. Note that a process sends an *st_active* message during a nonquiescent interval only after it has received at least one *detach* message. Thus, from Proposition 10, (3) and the definition of I, $height(x)$ is at least D which implies that:

$$x \in I \quad \Rightarrow \quad |childset(x)| \geqslant D \tag{4}$$

Since $\langle NQI, \mapsto \rangle$ is a forest, from (3), (1) and the definition of childset,

$$(\{x, y\} \subseteq I) \wedge (x \neq y) \quad \Rightarrow \quad childset(x) \cap childset(y) = \emptyset \tag{5}$$

We have,

$$\Big(\bigcup_{x \in I} childset(x) \Big) \subseteq NI$$

$\Rightarrow$ { using (5) }

$$\Big(\sum_{x \in I} |childset(x)| \Big) \leqslant |NI|$$

$\Rightarrow$ { using (4) and (2) }

$$D \times |I| \leqslant \Big(\sum_{x \in I} |childset(x)| \Big) \leqslant M$$

$\Rightarrow$ { algebra }

$$|I| \leqslant M/D$$

This establishes that TDA-SI is message-optimal. □

We now show that our algorithm is latency-optimal. The next lemma states one the underlying computation has terminated, no process stays nonquiescent for a "long" time.

Lemma 12 *Once the underlying computation terminates, every process becomes quiescent within $O(D)$ message hops, where D is the diameter of the communication topology.*

Proof: Assume that the underlying computation has terminated. Consider two processes p_i and p_j that are still nonquiescent just after the computation terminates. We say that p_i is *waiting on* p_j if p_i has sent an engaging application message to p_j but p_j has not yet sent an acknowledgment for that message. Now, consider any chain formed using "waiting on" relationships that starts from process p_i and whose length is at least $2D$. Clearly, the chain consists of a process p_k such that p_k is at a distance of at most D from p_i in the chain and $hops_k = 0$. From the algorithm, p_k receives a *detach* message within D message hops of termination. After receiving the *detach* message, p_k sends an *st_active* message to the coordinator for which it receives a matching *ack_st_active* message within $2D$ message hops. After receiving the *ack_st_active* message, p_k sends an *acknowledgment* message to its parent, if it has not already done so, causing the chain to break and become shorter. In other words, within $3D+1$ message hops of termination, all chains of "waiting on" relationships are reduced to length smaller than D. Clearly, once that happens, all processes become quiescent within D message hops. □

Finally, we have,

Theorem 13 (TDA-SI is latency-optimal) *Once the underlying computation terminates, the coordinator announces termination within $O(D)$ message hops.*

Proof: From Lemma 12, every process becomes quiescent within $O(D)$ message hops after the computation has terminated and stay quiescent thereafter. Therefore all *initialize*, *st_active* and *st_passive* messages are generated within $O(D)$ message hops of termination and no more messages are generated after that. Since the coordinator is at most D message hops away from any process, the coordinator receives all *initialize*, *st_active* and *st_passive* messages within $O(D)$ message hops of termination soon after which it announces termination (action B2). □

In the next section, we discuss modifications to our termination detection algorithm required in case one or more channels are non-FIFO.

3.5 Dealing with Non-FIFO Channels

To prove the correctness and optimality of the algorithm TDA-SI, we make the assumption that all channels are FIFO at two places. First, to ensure that the coordinator receives *st_active* and *st_passive* messages in the order in which they are sent (Proposition 4). Second, to ensure that the *acknowledgment* message for an engaging application messages does not "overtake" any *detach* message sent earlier (Proposition 10). In case one or more channels are non-FIFO, the following modifications to the algorithm can be used to ensure that both propositions still hold.

Ensuring that the coordinator receives st_active and st_passive messages in order: For convenience, we use *st_status* message to refer to an *st_active* message as well as an *st_passive* message, when it is not necessary to distinguish between the two. In the modified algorithm, the coordinator acknowledges all *st_status* messages, that is, both *st_active* and *st_passive* messages. Further, a process does not send the next *st_status* message until it has received an acknowledgment for its previous *st_status* message. This can be accomplished by maintaining a FIFO queue at each process. When a a new *st_status* message is generated by a process, the message is buffered until the process has sent all previous *st_status* messages and, moreover, has received acknowledgments for all of them. It can be verified that the above two modifications do not affect the correctness and message-optimality of our algorithm. However, they may increase the detection latency. Specifically, it is possible that when the underlying computation terminates the queue still contains a large number of *st_status* messages. To prevent the queue from becoming too long, we can proceed as follows. A process, on generating an *st_active* message, checks to see if the queue contains an *st_status* message. If the queue is non-empty, then the process simply discards the (new) *st_active*

message and also deletes the last *st_status* message, which will be an *st_passive* message, from the queue. Intuitively, the "new" *st_active* message "cancels" the "old" *st_passive* message. A similar optimization can be performed when an *st_passive* message is generated. This ensures that the queue never contains more than one pending *st_status* message.

Ensuring that the acknowledgment message for an engaging application message does not overtake any detach message sent earlier: In the modified algorithm, every *detach* message is acknowledged. Specifically, a process, after sending a *detach* message to its parent, waits until it has received an acknowledgment (for the *detach* message) from its parent before sending the *acknowledgment* message for the engaging application message. The notion of quiescence is now redefined as: a process is quiescent if it is passive, has received an *acknowledgment* message for every application message it has sent so far, *and has received an acknowledgment for every* detach *message it has sent so far*. It can be verified that the aforesaid modifications do not affect the correctness and optimality of our algorithm.

4 An Optimal Algorithm for Delayed Initiation

If the underlying computation is message-intensive, then it is desirable not to initiate the termination detection algorithm along with the computation. It is preferable, instead, to initiate it later, when the underlying computation is "close" to termination. This is because, in the latter case, the (worst-case) message-complexity of the termination detection algorithm would depend on the number of application messages exchanged by the computation *after* the termination detection algorithm has commenced. As a result, with delayed initiation, the termination detection algorithm generally exchanges fewer number of control messages than with simultaneous initiation.

To correctly detect termination with delayed initiation, we use the scheme proposed in [4]. 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 not be "tracked" by the termination detection algorithm and the latter messages should be "tracked" by the termination detection algorithm. Note that delayed initiation is not possible unless all channels are FIFO. This is because if one or more channels are non-FIFO then an application message may be delayed arbitrarily on a channel, no process would be aware of its existence, and this message may arrive at the destination after termination has been announced. Therefore we assume that all channels are FIFO. We also assume that each process knows all its

Termination detection algorithm for process p_i:

// Modification of TDA-SI: the algorithm for simultaneous initiation.
// Changes: one new action C1 and new definition for quiescence. The actions A1-A8 and B1-B3 remain the
// same and are executed only after the commencement of the termination detection algorithm.
// Application messages received along an uncolored channel are not acknowledged and are simply delivered to
// the application, whereas those received along a colored channel are handled using action A2.

Variables:
    $startDetection_i$:= false; *// am I executing the termination detection algorithm?*

Useful expression:
    // we have to redefine what it means for a process to be quiescent
    $quiescent_i \triangleq (state_i = passive) \wedge (missing_i = 0) \wedge$ (all incoming channels have been colored);

(C1) On receiving $\langle$*marker*$\rangle$ message from process p_j:
    if not$(startDetection_i)$ **then**
        send $\langle$*marker*$\rangle$ message along all outgoing channels;
        $startDetection_i$:= true;
    endif;
    if $(p_i \neq p_j)$ **then**
        color the incoming channel from process p_j;
    endif;
    if $quiescent_i$ **then**
        call sendIfQuiescent( );
    endif;

Figure 6: Termination detection algorithm TDA-DI for delayed initiation.

neighboring processes (that is, outgoing channels).

In order 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 acknowledged as per 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 not be acknowledged and should be 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, we assume that initially all incoming channels are *uncolored*. Further, a process, on receiving a *marker* message along an incoming channel, *colors* the channel along which it has received the *marker* message.

To initiate the termination detection algorithm, the coordinator sends a *marker* message to itself.

When a process receives a *marker* message, as explained before, it colors the incoming channel along which the *marker* message is received. Additionally, if it is the first *marker* message to be received, the process starts executing the termination detection algorithm and also sends a *marker* message along all its outgoing channels. Note that the coordinator should not announce termination at least until every process has received a *marker* message along all its incoming channels and therefore has colored all its incoming channels. Otherwise, some uncolored channel may contain an application message that neither the sender nor the receiver is aware of and the message may arrive after the termination is announced. This will violate the safety of the detection algorithm. To that end, we redefine the notion of quiescence as follows: a process is quiescent if it is passive, has received an *acknowledgment* message for every application message it has sent since it started executing the termination detection algorithm, and *all its incoming channels have been colored*. A formal description of the termination detection algorithm TDA-DI for delayed initiation is given in Figure 6.

Once the coordinator starts the termination detection algorithm, all incoming channels are colored within $O(D)$ message hops. The following theorem can be proved in a similar manner as Lemma 2:

Lemma 14 *Once the underlying computation terminates, all processes eventually become quiescent.*

Moreover, using the definition of quiescence and the fact that all channels are FIFO, it follows that:

Lemma 15 *If all processes are quiescent, then no channel contains an application message that was sent by a process* before *starting the termination detection algorithm.*

From the above lemma, we can infer that:

Lemma 16 *If all processes are quiescent, then the underlying computation has terminated.*

Proof: Assume that all processes are quiescent. Therefore every application message that was sent by a process *after* starting the termination detection algorithm has been acknowledged. This implies that no channel contains an application message that was sent by a process *after* starting the termination detection algorithm. Moreover, from Lemma 15, no channel contains an application message that was sent by a process *before* starting the termination detection algorithm. In other

words, all channels are empty (of application messages). Moreover, since all processes are quiescent, they are passive as well. □

Intuitively, TDA-DI announces termination once it detects that all processes have become quiescent, and vice versa. Therefore its liveness follows from Lemma 14 and its safety follows from Lemma 16.

The only additional messages exchanged by TDA-DI are *marker* messages. Therefore the message-complexity of TDA-DI is $O(\bar{M} + E)$, where $\bar{M}$ is number of application messages exchanged by the distributed computation *after* the termination detection algorithm has started and E is the number of channels in the communication topology. Note that $\bar{M}$ may be as large as M in the worst case. Therefore the *worst-case* message complexity of the termination detection algorithm with delayed initiation is actually more than that of the algorithm with simultaneous initiation. However, we expect the *average* message complexity to be much lower in the case of delayed initiation because much fewer application messages will need to be tracked on average.

Also, assuming that the termination detection algorithm is started before the underlying computation terminates, the detection latency of TDA-DI is $O(D)$. This is because, once the coordinator starts the termination detection algorithm, within $O(D)$ message hops, all processes start the termination detection algorithm and all incoming channels become colored. After this, similar to Lemma 12 and Theorem 13, it can be proved that all processes become quiescent and termination is detected within $O(D)$ message hops. Elsewhere, we show that any termination detection algorithm designed for simultaneous initiation can be transformed into a termination detection algorithm for delayed initiation with minimal impact on its performance [27].

5 An Optimal Algorithm for Multi-Hop Application Messages

In this section, we describe modifications required to our algorithm to ensure optimality with respect to message-complexity and detection-latency when application messages can be exchanged between arbitrary processes. We describe the modifications assuming simultaneous initiation. The ideas in this section can be easily combined with the ideas in the previous section to maintain optimality with delayed initiation as well.

We assume that an application message sent by process p_i to process p_j travels along a shortest path from p_i to p_j. We also assume that any *acknowledgment* or *detach* message that p_j sends to

p_i travels along the path taken by the application message in reverse direction.

5.1 Modifications to Our Algorithm

When application messages can be exchanged between arbitrary processes (and not just neighboring processes), each link in an *acknowledgment* chain may consist of $\Theta(D)$ message hops in the worst-case. As a result, the length of the *acknowledgment* chain in terms of number of message hops may be as large as $\Theta(D^2)$. This means that the worst-case detection latency of our algorithm to $\Theta(D^2)$, which is clearly suboptimal.

To achieve latency-optimality, instead of incrementing the hop counter by one every time an application message is exchanged, we increment the counter by the number of hops in the path taken by the application message. Observe that, with this modification, the value of the hop counter can actually become more than D. The counter is reset to zero as soon as its value becomes greater than or equal to D. This ensures that the value of the counter never exceeds $2D$, which, in turn, implies that the length of the *acknowledgment* chain in terms of number of message hops never exceeds $2D$.

As before, when a process resets the hop counter, its behavior is similar to that of a root process in TDA-SI. Specifically, it generates a *detach* message that is propagated upwards to the closest root process in the computation tree, which is at a distance of at least D message hops from it. Further, it maintains its link with its parent and reports its status to it until it receives a *detach* message from one of its children in the computation tree.

Clearly, our modifications ensure that the detection latency of the resulting algorithm is $O(D)$ in the worst-case. We now show that the modified algorithm has optimal message-complexity as well. We refer to the modified algorithm as TDA-SI-MH.

5.2 Proof of Message-Optimality

Note that, in TDA-SI-MH, a *st_active* or *st_passive* message generated by a process may not be amortized over at least D application messages. This is because the *acknowledgment* chain is cut as soon as its length becomes at least D in term of number of message hops and not in terms of number of application messages. In fact, in the worst case, the length of the *acknowledgment* chain may consist of only one application message (that travels a distance of D message hops). As a result, it may appear that TDA-SI-MH is no longer message-optimal. We show that our algorithm is still message-optimal if we count a control message (such as an *acknowledgment* message or a

detach message) that travels a distance of d message hops as d different control messages. This is also consistent with the way we count *st_active*, *st_passive* and *ack_st_active* messages in the analysis of TDA-SI-MH.

Let H denote the *average* number of hops traveled by an application message. It is given by the ratio $\frac{\text{total number of hops traveled by all application messages}}{\text{total number of application messages}}$. Note that $1 \leqslant H \leqslant D$. When application messages are only exchanged between neighboring processes, clearly, $H = 1$. We show that the message complexity of the modified algorithm is $\Theta(MH + N)$, which we prove is optimal.

First, we show that the worst-case message complexity of any termination detection algorithm is given by $\Omega(MH + N)$, thereby proving the optimality of TDA-SI-MH. It suffices to show that the worst-case message complexity of any termination detection algorithm is $\Omega(MH)$ when the computation is diffusing. Note that one may be tempted to think that the lower bound trivially follows from Chandy and Misra's lower bound proof [6] by replacing each application message that travels a distance of d hops with d application messages, each of which travels a distance of one hop. However the transformation is not correct for the following reason. Suppose an application message from process p_i to process p_j travels via processes $p_{k_1}, p_{k_2}, \ldots, p_{k_{d-1}}$. Then, before the transformation, each process p_{k_l}, where $1 \leqslant l < d$, basically acts as a relay; it does not become active on receiving the application message. However, after the transformation, each process p_{k_l}, where $1 \leqslant l < d$, has to become active on receiving the corresponding application message to satisfy the rules of the computation. Another approach is to assume that an "intermediate" application message does not really spawn any activity in the system in the sense that a passive process on receiving such a message stays active for a very short while during which it simply forwards the message to the next process. However, Chandy and Misra's lower bound proof [6] assumes that each application message is capable of spawning independent activity in the system and therefore the proof does not carry over to the system obtained after the transformation.

Note that, if there exists a computation state after which the termination detection algorithm exchanges an infinite number of control messages, then the lower bound trivially holds for that algorithm. Therefore assume that, after each computation state, the termination detection algorithm exchanges only a finite number of control messages after which it does not exchange any control message until the computation executes an event. Our lower bound proof uses the following lemma.

Lemma 17 *Consider two processes p_i and p_j that are at distance of H hops from each other. Assume that the system is in a state in which only p_i and p_j are active, all other processes are passive and all channels are empty. Then there exist an execution σ of the system (starting from*

the given state) and a process $p_k \in \{p_i, p_j\}$ such that (1) after σ only p_k is active, all other processes are passive and all channels are empty, (2) no application message is exchanged during σ, and (3) at least $\lceil H/2 \rceil$ control messages are exchanged during σ.

Proof: We consider two executions of the computation from the given state: one in which both processes become passive and one in which only one of them becomes passive.

In the first execution of the computation, denoted by κ_1, both processes p_i and p_j become passive without generating any application message. Clearly, once that happens, the computation terminates. Consider an execution τ_1 of the termination detection algorithm from the state resulting after executing κ_1 such that (1) after τ_1, no more control messages are exchanged, and (2) some process announces termination in τ_1. Such an execution exists because, by assumption, the termination detection algorithm exchanges only a finite number of messages in any computation state and, moreover, it is live. Let p_t be a process that announces termination in τ_1. Process p_t can announce termination only after "learning" that both p_i and p_j have become passive. Otherwise, it can be shown that the termination detection algorithm is not safe. Note that, in an asynchronous distributed system, the knowledge that some process has become passive can only be acquired through a (possibly empty) causal chain of messages [6]. Let K denote the set of processes that "learn" during the execution τ_1 of the algorithm that both p_i and p_j have become passive. The set K is non-empty because it at least contains p_t. For a process $p_x \in K$, let $first(p_x)$ denote the *earliest* event on p_x when p_x acquired that knowledge during τ_1. Consider a process $p_{\min} \in K$ such that $first(p_{\min})$ is a *minimal* event, with respect to $\rightarrow$, among all events in $\{first(p_x) \mid p_x \in K\}$. Note that $p_{\min}$ is at a distance of at least $\lceil H/2 \rceil$ hops from either p_i or p_j. Without loss of generality, assume that $p_{\min}$ is at a distance of at least $\lceil H/2 \rceil$ hops from p_i. For $p_{\min}$ to "learn" that p_i has become passive, there should be causal chain of messages starting from when p_i becomes passive and ending at $first(p_{\min})$ such that all messages in the chain are sent during τ_1. Let C denote the set of processes through which this chain passes. (C includes p_i but does not include $p_{\min}$.) Clearly, C contains at least $\lceil H/2 \rceil$ processes. Moreover, from the way $p_{\min}$ is chosen, each process in C sends its *first* control message during τ_1 without knowing that the other process p_j has become passive.

Now, consider the second execution of the computation, denoted by κ_2, in which only process p_i becomes passive; process p_j remains active. Processes in C clearly cannot distinguish between executions κ_1 and κ_2 of the computation when they send their first control message during τ_1. Therefore there exists an execution τ_2 of the termination detection algorithm from the state resulting

after κ_2 such that (1) at least $\lceil H/2 \rceil$ processes send a control message during τ_2, and (2) no control message is sent after τ_2 until the computation executes an event. The required execution σ of the system is given by κ_2 followed by τ_2. □

We are now ready to prove the lower bound on message complexity of a termination detection algorithm. Our proof is based on the proof of lower bound on message complexity of a termination detection algorithm given in Tel [30] for the case when $H = 1$.

Theorem 18 (lower bound on message-complexity) *Consider a diffusing computation and assume that the computation eventually terminates. Then, the worst-case message-complexity of any termination detection algorithm is given by $\Theta(MH)$, where M is the number of application messages exchanged by the underlying computation and H is the average number of hops traveled by the application messages.*

Proof: The proof is constructive by nature. We construct a system execution in steps. In each step, the underlying computation exchanges one application message that travels a distance of H hops because of which the termination detection algorithm is forced to exchange $\Omega(H)$ control messages.

Consider two processes p_i and p_j that are at a distance of H hops from each other. Assume that the system is in a state X in which only one process, say p_i, is active, all other processes are passive and all channels are empty. (This state may be the initial state of the system.) Now, suppose p_i sends an application message to p_j which makes p_j active. Since the termination detection algorithm eventually stops exchanging control messages, the system eventually reaches a state Y—via an execution σ_1—in which both p_i and p_j are active, all other processes are passive and all channels are empty. We can now apply Lemma 17 to system state Y. Thus there exists an execution σ_2 of the system that takes the system to a state Z such that (1) in Z only one process $p_k \in \{p_i, p_j\}$ is active, all other processes are passive and all channels are empty, (2) no application message is exchanged during σ_2 and (3) at least $\Omega(H)$ control messages are exchanged during σ_2.

Combining the two executions, we can conclude that there exists an execution σ of the system, which is given by σ_1 followed by σ_2, such that (1) exactly one application message is exchanged during σ, (2) the application message travels a distance of H hops, (3) at least $\Omega(H)$ control messages are exchanged during σ, and (4) the system state after σ is *isomorphic* to the system state before σ.

The last property implies that the above-described construction can be repeated *ad infinitum*, thereby proving the lower bound. □

Next, we show that TDA-SI-MH has optimal message-complexity.

Theorem 19 (TDA-SI-MH is message-optimal) *Assume that the underlying computation eventually terminates. Then, the number of control messages exchanged by the modified algorithm is given by $\Theta(MH + N)$, where N is the number of processes in the system, M is the number of application messages exchanged by the underlying computation and H is the average number of hops traveled by the application messages.*

Proof: The structure of the proof is quite similar to the structure of the proof for Theorem 11. We present it anyway for the sake of completeness.

Our algorithm exchanges six different types of control messages, namely *acknowledgment*, *detach*, *initialize*, *st_active*, *st_passive* and *ack_st_active*. We now bound each of the six types of control messages.

Clearly, the number of *acknowledgment* messages is equal to the total number of hops traveled by all application messages collectively, which is given by MH. A process sends at most one *detach* message per engaging application message. Therefore the number of *detach* messages is upper-bounded by MH. Every process sends at most one *initialize* message. Further, *initialize* messages are propagated to the coordinator in a convergecast fashion. Hence the total number of *initialize* messages exchanged is given by $O(N)$.

Every *st_active* and *st_passive* message has to be propagated to the coordinator along the BFS spanning tree and, therefore, may have to travel a distance of up to D message hops. Likewise, an *ack_st_active* message may have to travel a distance of up to D message hops. From Lemma 5, every process sends equal number of *st_active* and *st_passive* messages. Moreover, the number of *ack_st_active* messages a process receives is equal to the number of *st_active* messages it sends. Thus it is sufficient to show that the total number of *st_active* messages generated by all processes combined is bounded by MH/D. This in turn would imply that total number of *st_active*, *st_passive* and *ack_st_active* messages exchanged are given by $O(MH)$.

Observe that a process sends an *st_active* message only when it is nonquiescent and, moreover, it sends at most one *st_active* message per nonquiescent interval. We, therefore, bound the number of nonquiescent intervals in which an *st_active* message is sent. Let $I \subseteq NQI$ denote the set of those

nonquiescent intervals during which an *st_active* message is sent. For two nonquiescent intervals x and y with $x \mapsto y$, the distance between x and y is given by the *number of hops* traveled by the engaging application message that created y (and was sent during x). Our proof uses the notion of childset of a nonquiescent interval, which was defined in the proof of Theorem 11. It can be verified that:

$$(\{x, y\} \subseteq I) \wedge (x \neq y) \quad \Rightarrow \quad childset(x) \cap childset(y) = \emptyset \tag{6}$$

Also, let $totalhops(x)$ denote the total number of hops traveled by application messages that created nonquiescent intervals in $childset(x)$. From the definition of I,

$$x \in I \quad \Rightarrow \quad totalhops(x) \geqslant D \tag{7}$$

We have,

$$\begin{array}{ll} & \{ \text{ using (6) } \} \\ & \sum_{x \in I} totalhops(x) \leqslant MH \\ \Rightarrow & \{ \text{ using (7) } \} \\ & D \times |I| \leqslant \sum_{x \in I} totalhops(x) \leqslant MH \\ \Rightarrow & \{ \text{ algebra } \} \\ & |I| \leqslant MH/D \end{array}$$

This establishes the theorem. □

Note that, to achieve optimality, our algorithm does not require the knowledge of H, the average number of hops traveled by application messages. Also, as opposed to our algorithm, Mahapatra and Dutt's algorithm [20], which is also latency-optimal, has worst-case message complexity of $O(MD+N)$ *irrespective* of the average number of hops traveled by application messages. Therefore our algorithm is *always as efficient (asymptotically) as* Mahapatra and Dutt's algorithm [20] and is *more efficient (asymptotically)* than their algorithm when H is $o(D)$.

6 Conclusion and Future Work

In this paper, we have presented three optimal algorithms for termination detection when processes and channels are reliable, and all channels are bidirectional. All three of the algorithms have optimal message complexity and optimal detection latency under varying assumptions. Algorithms TDA-SI

and TDA-SI-MH have to be initiated along with the computation. The former algorithm is optimal when application messages are only exchanged between neighboring processes, whereas the latter is optimal when application messages can be exchanged between arbitrary processes. Algorithm TDA-DI can be initiated at any time *after* the computation has started. However, all channels are required to be FIFO for the algorithm to work correctly, which is also necessary to solve the problem.

All of our algorithms currently have two limitations. First, all processes need to know the diameter of the communication topology within a constant factor. (It is not necessary to know the exact value of the diameter as long as the estimate is within a constant factor of the actual value.) Second, they are asymmetric in the sense that one of the processes acts as a coordinator and is responsible for maintaining the state of the system. An interesting research direction would be to design a termination detection algorithm with optimal message complexity and detection latency that is fully symmetric and in which the amount of knowledge a process needs to have about the system is minimized [21].

References

[1] S. Alagar and S. Venkatesan. An Optimal Algorithm for Recording Snapshots using Casual Message Delivery. *Information Processing Letters (IPL)*, 50:311–316, 1994.

[2] 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)*, pages 20–33, La Martinique, France, December 2003.

[3] H. Attiya and J. Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. John Wiley and Sons Limited, 2nd edition, 2004.

[4] S. Chandrasekaran and S. Venkatesan. A Message-Optimal Algorithm for Distributed Termination Detection. *Journal of Parallel and Distributed Computing (JPDC)*, 8(3):245–252, March 1990.

[5] K. M. Chandy and L. Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

[6] K. M. Chandy and J. Misra. How Processes Learn. *Distributed Computing (DC)*, 1(1):40–52, 1986.

[7] W. J. Dally. Performance Analysis of k-ary-n cube Interconnection Networks. *IEEE Transactions on Computers*, 39(6):775–785, June 1990.

[8] M. Demirbas and A. Arora. An Optimal Termination Detection Algorithm for Rings. Technical Report OSU-CISRC-2/00-TR05, The Ohio State University, February 2000.

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

[10] E. W. Dijkstra, W. H. J. Feijen, and A. J. M. van Gasteren. Derivation of a Termination Detection Algorithm for Distributed Computations. *Information Processing Letters (IPL)*, 16:217–219, 1983.

[11] E. W. Dijkstra and C. S. Scholten. Termination Detection for Diffusing Computations. *Information Processing Letters (IPL)*, 11(1):1–4, 1980.

[12] N. Francez. Distributed Termination. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2(1):42–55, January 1980.

[13] J.-M. Hélary, C. Jard, N. Plouzeau, and M. Raynal. Detection of Stable Properties in Distributed Applications. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 125–136, 1987.

[14] J.-M. Hélary and M. Raynal. Towards the Construction of Distributed Detection Programs, with an Application to Distributed Termination. *Distributed Computing (DC)*, 7(3):137–147, 1994.

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

[16] S.-T. Huang. Termination Detection by using Distributed Snapshots. *Information Processing Letters (IPL)*, 32:113–119, August 1989.

[17] 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)*, pages 646–649. IEEE Computer Society, 1992.

[18] T.-H. Lai and T. H. Yang. On Distributed Snapshots. *Information Processing Letters (IPL)*, 25(3):153–158, May 1987.

[19] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM (CACM)*, 21(7):558–565, July 1978.

[20] N. R. Mahapatra and S. Dutt. An Efficient Delay-Optimal Distributed Termination Detection Algorithm. To Appear in Journal of Parallel and Distributed Computing (JPDC), 2004.

[21] J. Matocha and T. Camp. A Taxonomy of Distributed Termination Detection Algorithms. *The Journal of Systems and Software*, 43(3):207–221, November 1998.

[22] F. Mattern. Algorithms for Distributed Termination Detection. *Distributed Computing (DC)*, 2(3):161–175, 1987.

[23] F. Mattern. Global Quiescence Detection based on Credit Distribution and Recovery. *Information Processing Letters (IPL)*, 30(4):195–200, 1989.

[24] F. Mattern, H. Mehl, A. Schoone, and G. Tel. Global Virtual Time Approximation with Distributed Termination Detection Algorithms. Technical Report RUU-CS-91-32, University of Utrecht, The Netherlands, 1991.

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

[26] 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 (DISC)*, pages 290–304, Amsterdam, The Netherlands, October 2004.

[27] 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)*, pages 209–215, San Francisco, California, USA, September 2004.

[28] S. P. Rana. A Distributed Solution of the Distributed Termination Problem. *Information Processing Letters (IPL)*, 17(1):43–46, 1983.

[29] 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)*, pages 344–358, Rennes, France, 1986.

[30] G. Tel. *Introduction to Distributed Algorithms*. Cambridge University Press (US Server), second edition, September 2000.

[31] G. Tel and F. Mattern. The Derivation of Distributed Termination Detection Algorithms from Garbage Collection Schemes. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(1):1–35, January 1993.

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.

S. Venkatesan received his B.Tech. degree in Civil Engineering and M.Tech. degree in Computer Science from the Indian Institute of Technology, Madras in 1981 and 1983, respectively. He completed his Ph.D. degree in Computer Science from the University of Pittsburgh in December

1988. In January 1989, he joined the University of Texas at Dallas where he is currently an Associate Professor of Computer Science and Head of Telecommunications Engineering Program. His research interests are in Distributed Systems, Mobile Computing, Wireless Networks and Testing and Debugging Distributed Programs. He has been the Chief Architect of IPmobile (a startup company acquired by Cisco Systems in September 2000), a member of the advisory board of Jahi Networks (a startup company acquired by Cisco Systems in December 2004), and a consultant to many high technology companies in the Dallas-Fort Worth region.

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 the University of Texas at Dallas and is a member of the Advanced Networking and Dependable Systems Laboratory (ANDES). His research interests include peer-to-peer computing and dynamic distributed systems.