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

Neeraj Mittal, S. Venkatesan, and 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. Detecting termination of a distributed computation is a fundamental problem in distributed systems. We present two optimal algorithms for detecting termination of a non-diffusing distributed computation for an arbitrary topology. Both algorithms are *optimal* in terms of message complexity and detection latency. The first termination detection algorithm has to be initiated along with the underlying computation. The message complexity of this algorithm is $\Theta(N + M)$ and its detection latency is $\Theta(D)$, where N is the number of processes in the system, M is the number of application messages exchanged by the underlying computation, and D is the diameter of the communication topology. The second termination detection algorithm can be initiated at any time *after* the underlying computation has started. The message complexity of this algorithm is $\Theta(E + M)$ and its detection latency is $\Theta(D)$, where E is the number of channels.

Key words: termination detection, quiescence detection, optimal message complexity, optimal detection latency

1 Introduction

One of the fundamental problems in distributed systems is to detect termination of an ongoing distributed computation. The termination detection problem was independently proposed by Dijkstra and Scholten [1] and Francez [2] more than two decades ago. Since then, many researchers have studied this problem and, as a result, a large number of algorithms have been developed for termination detection (*e.g.*, [3–15]). Although most of the research work on termination detection was conducted in 1980s and early 1990s, a few papers on termination detection still appear every year (*e.g.*, [13–15]).

Most termination detection algorithms can be broadly classified into three categories, namely *computation tree based*, *invigilator based*, and *wave based*. (A more detailed survey of termination detection algorithms can be found in [16].)

In the computation tree based approach, a dynamic tree is maintained based on 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 $[1, 4, 10]$.

In the invigilator based approach, a distinguished process, called the *coordinator*, is responsible for maintaining current status of all processes. The coordinator may either explicitly maintain the number of processes that are currently "participating" in the computation [14] or may only know whether there exists at least one process that is currently "participating" in the computation (ascertained via missing credit or weight) $[7, 9]$. Algorithms in this class typically assume that the topology contains a star and the coordinator is directly connected to every process. These algorithms can be generalized to work for any communication topology at the expense of increased message complexity.

In the wave based approach, repeated snapshots of the underlying computation are taken and tested for termination condition. The testing procedure for termination takes into consideration the possibility that the snapshot may not be consistent. Examples of algorithms based on this approach can be found in $[3, 5, 6, 8, 17]$.

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.

A termination detection algorithm should have low message-complexity (that is, it should exchange as few control messages as possible) and low detection latency (that is, it should detect termination as soon as possible). The former is desirable because it minimizes the overhead incurred on executing the termination detection algorithm. The latter is desirable in situations when the results of the computation can be used only after the computation has terminated.

Chandy and Misra [18] prove that any termination detection algorithm, in the worst case, must exchange at least M control messages, where M is the number of application messages exchanged by the underlying computation. Further, for a general non-diffusing computation (when an arbitrary subset of processes can be active initially), any termination detection algorithm must exchange at least $N - 1$ control messages in the worst-case, where N is the number of processes in the system. Chandrasekaran and Venkatesan [10] prove another lower bound that if the termination detection algorithm is initiated 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). Finally, note that, 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.

Table 1. Comparison of various termination detection algorithms (assume diffusing computation and simultaneous initiation unless indicated otherwise).

Category	Message Complexity	Detection Latency	Communication Topology	Representative References
Computation Tree Based	$O(M)$	$O(N)$	any	[1, 4, 10]
Invigilator Based	$O(M)$	$O(1)$	contains a star	[14, 7, 9]
Modified Invigilator Based*	$O(MD)$	$O(D)$	any	
Wave Based	$O(NM)$	$O(D)$	any	[3, 5, 6, 8, 17]
Our Algorithm	$O(M)$	$O(D)$	any	[this paper]
Our Algorithm (non-diffusing computation)	$O(M + N)$	$O(D)$	any	[this paper]
Our Algorithm (non-diffusing computation and delayed initiation)	$O(M + E)$	$O(D)$	any	[this paper]

N: number of processes in the system
M: number of (application) messages exchanged by the underlying computation
D: diameter of the communication topology
E: number of channels in the communication topology
$*$: invigilator based adapted for arbitrary communication topology

Table 1 shows the (worst-case) message complexity and detection latency for the best algorithm in each of the three classes and for our algorithms. (For the wave based approach, we assume that a spanning tree is used to collect a snapshot of the system.) The table also indicates assumption, if any, made about the communication topology. Algorithms based on the computation tree based approach have optimal message complexity but non-optimal detection latency. On the other hand, algorithms that use the invigilator based approach have optimal detection latency but non-optimal message complexity. (The message-complexity is optimal only when the topology contains a star.)

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. We present two message-optimal and latency-optimal termination detection algorithms, which do not make any assumptions about the underlying communication topology. Furthermore, the amount of control information carried by each message is small ($\approx O(\log D)$). The first algorithm has to be started along with the computation. The second algorithm can be started anytime after the commencement of the computation. Our approach is a combination of computation tree based and invigilator based approaches.

2   System Model and Problem Description

2.1   System Model

We assume an asynchronous distributed system consisting of N processes $P = \{p_1, p_2, \ldots, p_N\}$, which communicate with each other by sending messages over a set of bidirectional channels. There is no common clock or shared memory. Processes are non-faulty and channels are reliable. Message delays are finite but may be unbounded. Processes change their states by executing events.

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.

A computation is *diffusing* if only one process is active initially; otherwise it is *non-diffusing*. If the termination detection algorithm is initiated along with the computation, then we refer to it as *simultaneous initiation*. On the other hand, if the termination detection algorithm is initiated after the computation has started, then we refer to it as *delayed initiation*.

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*.

3   An Optimal Algorithm for Simultaneous Initiation

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.

Detecting Termination of a Diffusing Computation: First, we explain the main idea behind computation tree based and 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 [1, 10]. 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 (sometimes referred to as an *acknowledgment* chain) may grow to a length as long as M, where M is the number of application messages exchanged. (The reason is that a process may appear multiple times on an *acknowledgment* chain as is the case with the algorithm of [10].)

The detection latency of the algorithm can be reduced from $O(M)$ to $O(N)$ (assuming $M = \Omega(N)$) as follows. 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 has not yet sent an *acknowledgment* message for every application message it has received 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 [14]. One process is chosen to act as the coordinator. The coordinator is responsible for maintaining the current status of every process in the system. Suppose a process receives an application message. In case the coordinator does not already know that it (the process) 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, in case the coordinator already knows that it (the process) is currently active, 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 other process in the system, the algorithm has optimal message complexity (at most three control message 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 spanning tree) may generate up to D additional messages, 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. 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, the number of times when
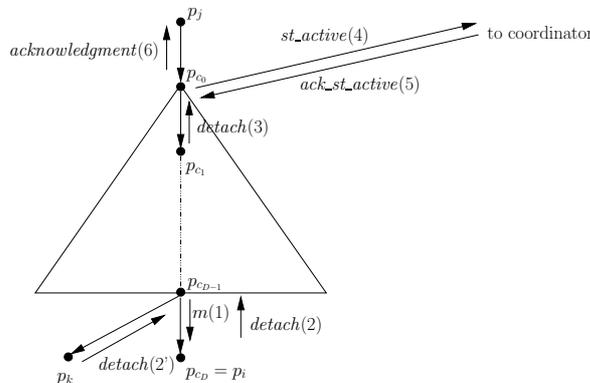
Fig. 1. An Illustration of the Termination Detection Algorithm.

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, its length 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 now announce termination as yet. It then sends an *acknowledgment* message to its parent, thereby breaking its 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 on every application message that refers to 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 integer 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 its intended recipient that it should become the head of the chain and report its status to the coordinator instead of its parent (the details of how it 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 will send *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 propagated a *detach* message to its parent, then it ignores any subsequent *detach* messages it receives.

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

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 application message it receives. Therefore the total number of *acknowledgment* and *detach* messages is 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, which may generate at most D additional messages. 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 an *acknowledgment* chain grows beyond $2D$, within $3D + 1$ message hops (consisting of *detach*, *st_active* and *ack_st_active* messages), the chain is reduced to length smaller than D. Thus the detection latency of our algorithm is $O(D)$.

Generalizing for a Non-Diffusing Computation: Assume that there may be 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 Formal Description

A formal description of the termination detection algorithm TDA-SI for simultaneous initiation is given in Fig. 2 and Fig. 3. Actions A0-A8 described in the two figures capture the behavior of a process as part of the computation tree. Due to lack of space, actions of processes as part of the BFS spanning tree have been omitted and can be found in [19]. 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.

3.3 Proof of Correctness and Optimality

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* (or *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 $O(\log N)$, and also requires each process to know all its descendants. Instead, we employ the following mechanism. Every process on the BFS spanning tree sends the kth *ack_st_active* message to the sender of the kth *st_active* message. This can be accomplished by maintaining a FIFO queue that records the 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. We have,

Lemma 1. *A process eventually receives an ack_st_active message for every st_active message it sends.*

Due to lack of space, proofs of lemmas and some theorems have been omitted, and can be found in [19]. The second 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"

Lemma 2. *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.*

```
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 := ⊥;                            // 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  ≜  (state_i = passive) ∧ (missing_i = 0);
    root_i  ≜  (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 ⟨m, hops_i⟩ to process p_j;
    missing_i := missing_i + 1;               // one more application message to be acknowledged

(A2) On receiving an application message ⟨m, count⟩ from process p_j:
    if not(quiescent_i) then                  // a non-engaging application message
        send ⟨acknowledgment⟩ 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 ⟨detach⟩ 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 ⟨acknowledgment⟩ 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
```

Fig. 2. Termination detection algorithm TDA-SI for simultaneous initiation.

We say that a process is *quiescent* if it is passive and has received an *acknowl-edgment* message 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. We also partition the set of

```
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 the coordinator (via the BFS spanning tree);
            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 the coordinator (via the BFS spanning tree);
        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 the coordinator (via the BFS spanning tree);
            else  send ⟨st_passive⟩ to the coordinator (via the BFS spanning tree); endif;
        endif:
```

Fig. 3. Termination detection algorithm TDA-SI for simultaneous initiation (contd.).

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. We have,

Lemma 3. *Assume that the underlying computation eventually terminates. Then, every nonquiescent process 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 4. *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.*

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. *Every 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.*

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 modification are described in [19]. For now, assume that all channels are FIFO. We are now ready to prove the correctness of our algorithm. First, we prove that our algorithm is live.

Theorem 6 (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, given by *"number of st_active messages received - number of st_passive messages received"*, becomes zero permanently.

Note that *initialize* messages are propagated to the coordinator in a convergecast fashion. From Proposition 4, 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 on 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 3, 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 Lemma 5, the coordinator receives an equal number of *st_active* and *st_passive* messages. □

Now, we prove that our algorithm is safe, that is, it never announces false termination. The proof uses the well-known Lamport's happened-before relation, which we denote by →.

Theorem 7 (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 d_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 passing through all d_is. Assume, on the contrary, that the computation has not terminated for S and that some
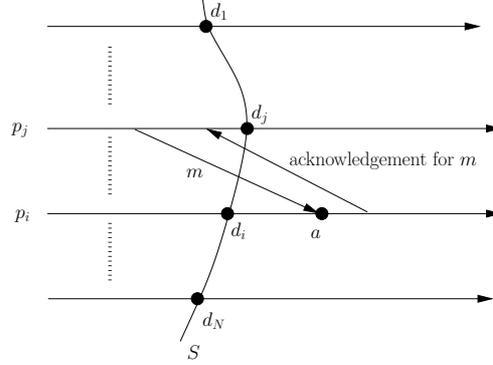
Fig. 4. Proving the safety of TDA-SI.

process becomes active after S. Let NQE denote the set of nonquiescent events executed in the future of S. Consider a *minimal* event a in NQE—minimal with respect to the happened-before relation. Formally,

$$\langle \forall x : x \in NQE : x \nrightarrow a \rangle$$

Clearly, a 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 a 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, $snd(m) \rightarrow d_j$. This implies that p_j becomes quiescent after sending m. Therefore it receives the *acknowledgment* message for m before executing d_j. This is depicted in Fig. 4. There are two cases to consider:

Case 1: Process p_i sends an *acknowledgment* message for m on executing a quiescent event, say b. Clearly, the *acknowledgment* message creates a causal path from b to d_j. We have,

$$(b \text{ is a quiescent event on } p_i) \wedge (d_i \rightarrow b) \wedge (b \rightarrow d_j) \wedge (d_j \rightarrow announce)$$

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

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 a). Let the receive event of the *st_active* message on the coordinator be denoted by r. Using Lemma 2, $r \rightarrow d_j$. Since $d_j \rightarrow announce$, $r \rightarrow announce$. For the coordinator to announce termination, it should receive 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 a. This again contradicts our choice of d_i. □

We next show that our termination detection algorithm is optimal in terms of message complexity and detection latency.

Theorem 8 (TDA-SI is message-optimal). *Assume that the underlying computation eventually terminates. Then, the number of control messages exchanged by the algorithm is $\Theta(M + N)$, where N is the number of processes in the system and M is the number of application messages exchanged by the underlying computation.*

Also, we have,

Theorem 9 (TDA-SI is latency-optimal). *Once the underlying computation terminates, the coordinator announces termination within $O(D)$ message hops.*

We now show how to modify our termination detection algorithm so that it can be started later anytime after the computation has begun.

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 a fewer number of control messages than with simultaneous initiation.

To correctly detect termination with delayed initiation, we use the scheme proposed in [10]. 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. Henceforth, we assume that all channels are FIFO.

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. To enable delayed initiation, we just 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 so far, and *all its incoming channels have been colored*. A formal description of the termination detection algorithm for delayed initiation, which we refer to as TDA-DI, is given in [19].

The correctness and optimality proof of TDA-DI is similar to that of TDA-SI. Intuitively, it can be verified that once the underlying computation terminates, TDA-DI eventually announces termination after all incoming channels have been colored (that is, TDA-DI is live). Also, TDA-DI announces termination only after all processes have become quiescent. This in turn implies that the underlying computation has terminated (that is, TDA-DI is safe). The message-complexity of TDA-DI is at most E more than the message complexity of TDA-SI, where E is the number of channels in the communication topology. Also, assuming that the termination detection algorithm is started before the underlying computation terminates, the detection latency of TDA-DI is $O(D)$.

5 Conclusion and Future Work

In this paper, we present two optimal algorithms for termination detection when processes and channels are reliable, and all channels are bidirectional. Both our algorithms have optimal message complexity and optimal detection latency. The first termination detection algorithm has to be initiated along with the computation. The second termination detection algorithm can be initiated at any time after the computation has started. However, in this case, all channels are required to be FIFO. Our algorithms do not make any assumptions about the communication topology (whether it contains a star or is fully connected) or the distributed computation (which processes are active initially).

Our termination detection algorithms have the following limitations. First, we assume that every process knows the diameter D of the communication topology. Second, each application message carries an integer whose maximum value is D. As a result, the bit-message complexity of our algorithms is given by $O(M \log D + N)$ and $O(M \log D + E)$, respectively, which is sub-optimal. Third, we use a coordinator, which is responsible for processing *st_passive* and *st_active* messages sent by other processes. The coordinator may become a bottleneck. As future work, we plan to develop a termination detection algorithm that does not suffer from the above-mentioned limitations.

References

1. Dijkstra, E.W., Scholten, C.S.: Termination Detection for Diffusing Computations. Information Processing Letters (IPL) **11** (1980) 1–4
2. Francez, N.: Distributed Termination. ACM Transactions on Programming Languages and Systems (TOPLAS) **2** (1980) 42–55
3. Rana, S.P.: A Distributed Solution of the Distributed Termination Problem. Information Processing Letters (IPL) **17** (1983) 43–46
4. Shavit, N., Francez, N.: A New Approach to Detection of Locally Indicative Stability. In: Proceedings of the International Colloquium on Automata, Languages and Systems (ICALP), Rennes, France (1986) 344–358
5. Mattern, F.: Algorithms for Distributed Termination Detection. Distributed Computing (DC) **2** (1987) 161–175
6. Dijkstra, E.W.: Shmuel Safra's Version of Termination Detection. EWD Manuscript 998. Available at `http://www.cs.utexas.edu/users/EWD` (1987)
7. Mattern, F.: Global Quiescence Detection based on Credit Distribution and Recovery. Information Processing Letters (IPL) **30** (1989) 195–200
8. Huang, S.T.: Termination Detection by using Distributed Snapshots. Information Processing Letters (IPL) **32** (1989) 113–119
9. Huang, S.T.: Detecting Termination of Distributed Computations by External Agents. In: Proceedings of the IEEE International Conference on Distributed Computing Systems (ICDCS). (1989) 79–84
10. Chandrasekaran, S., Venkatesan, S.: A Message-Optimal Algorithm for Distributed Termination Detection. Journal of Parallel and Distributed Computing (JPDC) **8** (1990) 245–252
11. Tel, G., Mattern, F.: The Derivation of Distributed Termination Detection Algorithms from Garbage Collection Schemes. ACM Transactions on Programming Languages and Systems (TOPLAS) **15** (1993) 1–35
12. Stupp, G.: Stateless Termination Detection. In: Proceedings of the 16th Symposium on Distributed Computing (DISC), Toulouse, France (2002) 163–172
13. Khokhar, A.A., Hambrusch, S.E., Kocalar, E.: Termination Detection in Data-Driven Parallel Computations/Applications. Journal of Parallel and Distributed Computing (JPDC) **63** (2003) 312–326
14. Mahapatra, N.R., Dutt, S.: An Efficient Delay-Optimal Distributed Termination Detection Algorithm. To Appear in Journal of Parallel and Distributed Computing (JPDC) (2004)
15. Wang, X., Mayo, J.: A General Model for Detecting Termination in Dynamic Systems. In: Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS), Santa Fe, New Mexico (2004)
16. Matocha, J., Camp, T.: A Taxonomy of Distributed Termination Detection Algorithms. The Journal of Systems and Software **43** (1999) 207–221
17. Atreya, R., Mittal, N., Garg, V.K.: Detecting Locally Stable Predicates without Modifying Application Messages. In: Proceedings of the 7th International Conference on Principles of Distributed Systems (OPODIS), La Martinique, France (2003)
18. Chandy, K.M., Misra, J.: How Processes Learn. Distributed Computing (DC) **1** (1986) 40–52
19. Mittal, N., Venkatesan, S., Peri, S.: Message-Optimal and Latency-Optimal Termination Detection Algorithms for Arbitrary Topologies. Technical Report UTDCS-08-04, The University of Texas at Dallas (2004) Available at `http://www.utdallas.edu/~neerajm/`.