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 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. However, in this case, all channels are required to be first-in-first-out (FIFO). 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, diffusing computation, monitoring distributed system

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 on receiving an application message. A computation is said to have *terminated* when all processes have become passive and all channels have become empty. The problem of termination detection was independently proposed by Dijkstra and Scholten [8] and Francez [9] 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.*, [7, 20, 21, 22, 17, 6, 18, 13, 12, 3, 19, 24, 11, 16]). 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 [4, 14, 10, 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 three 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 [8, 22, 3].

In 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 [16] or may only know whether there exists at least one process that is currently "participating" in the computation (ascertained via missing credit or weight) [18, 12]. 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.

The next two classes of algorithms are based on the notion of *wave* [23]. 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. 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 it 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 [17, 6, 19, 11].

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 [21, 17, 13].

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. 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 [5]. Also, in the worst-case, the detection latency of any termination

Class	Message Complexity	Detection Latency	Communication Topology
Computation Tree Based	$O(M)$	$O(N)$	any
Invigilator Based	$O(M)$	$O(1)$	contains a star
Modified Invigilator Based*	$O(MD)$	$O(D)$	any
Double Wave Based	$O(NM)$	$O(D)$	any
Single Wave Based	$O(NM)$	$O(D)$	any
Our Algorithm	$O(M)$	$O(D)$	any
Our Algorithm (non-diffusing computation)	$O(M+N)$	$O(D)$	any
Our Algorithm (non-diffusing computation and delayed initiation)	$O(M+E)$	$O(D)$	any

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

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 have optimal message complexity but non-optimal detection latency. On the other hand, algorithms that use 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.

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 [3] 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 two message-optimal and latency-optimal termination detection algorithms, which do not make any assumptions about the underlying communication topology. 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.

The paper is organized as follows. In Section 2, we discuss the system model and notation used in this paper. The termination detection problem is described in Section 3. Section 4 and Section 5 describe termination detection algorithms for the two cases when the algorithm has to be initiated along with the computation and when it can be initiated at any time after the computation has commenced. Finally, we present our conclusion and outline direction for future research in Section 6.

2 Model and Notation

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

Processes execute *events* and change their states. 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 causes the local state of a process to be updated. In addition, a send event causes a message to be sent and a receive event causes a message to be received.

Events on a process are totally ordered. However, events on different processes are only partially ordered by the Lamport's happened-before relation [15], 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 it 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.

3 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*. Clearly, it is desirable that the termination detection algorithm exchange as few control messages as possible, that is, it has low message complexity. Also, once the underlying computation terminates, the algorithm should detect it as soon as possible, that is, it has low detection latency. For computing detection latency, it is assumed that each message hop takes one time unit and message processing time is negligible. Detection latency is measured in terms of number of message hops. Furthermore, 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*.

4 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.

4.1 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.

4.1.1 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 [8, 3]. 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 [3].)

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 [16]. 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 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 is currently active, it immediately acknowledges the application messages. 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 BFS 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 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 should 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 $p_0, p_1, \ldots, p_{c_D} = p_i$. 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 $p_{c_{D-2}}$. $\square$

Message-complexity: Our algorithm exchanges the 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 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, 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 $O(M)$.

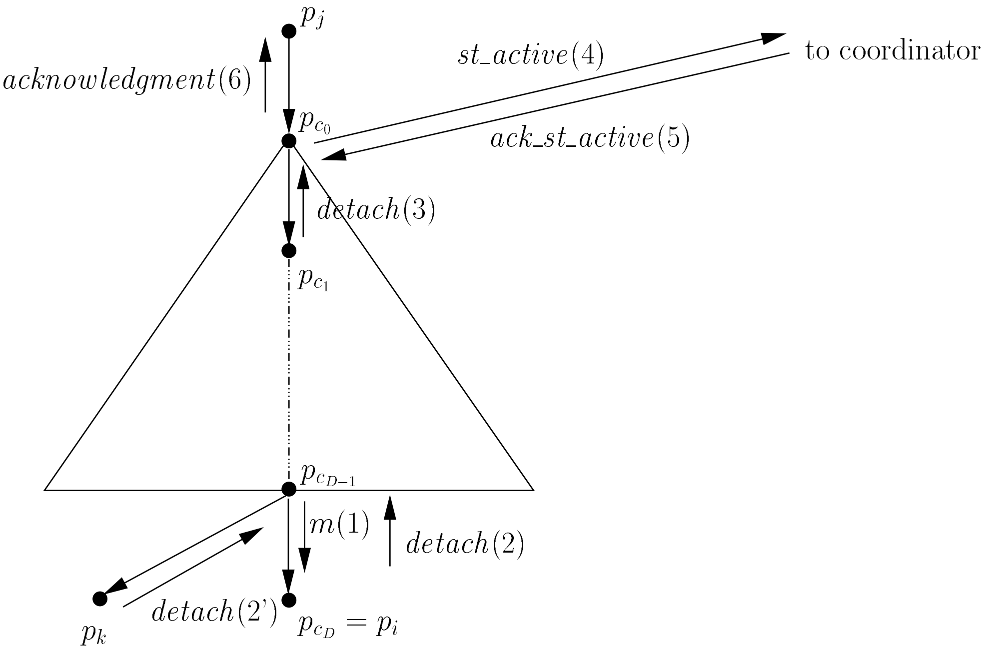

Figure 1: An Illustration of the Termination Detection Algorithm.

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 length smaller than D. Therefore the detection latency of our algorithm is $O(D)$.

4.1.2 Generalizing for 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.

4.2 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 behaviour of a process as part of the computation tree. Actions B1-B3 given in Figure 4 describe the behaviour 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 upward 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 it to itself. In case it sends the message to itself, it is handled by action A6.

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

4.3 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. For now, assume that the coordinator continues to process *st_active* messages even after it has announced termination. Later, we show that the coordinator does not receive any *st_active* message after it has announced termination. 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 $O(\log N)$. Instead, we employ the following mechanism. Every process on the BFS spanning tree sends 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 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. The next two lemmas can be proved by a simple induction on the depth of a process in the BFS spanning tree. The first lemma states that a process eventually receives a matching *ack_st_active* message for every *st_active* message it sends.

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

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.*

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 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*

```
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
```

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).

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 $\langle NQI, \mapsto \rangle$

```
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 ≜ (father_i = p_i);
    terminated_i ≜ (have received sons_i + 1 initialize messages) ∧(activity_i = 0);

(B1) On receiving ⟨initialize⟩ 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 ⟨initialize⟩ message to father_i;
    endif;

(B2) On receiving ⟨st_status⟩ message (st_status ∈ {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 ⟨ack_st_active⟩ 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;            // record the sender
        endif;
        send ⟨st_status⟩ message to father_i;  // forward st_status message to my father
    endif;

(B3) On receiving ⟨ack_st_active⟩ message from father_i:
    p := dequeue whoSent;
    send ⟨ack_st_active⟩ 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).

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 3 *Assume that the underlying computation eventually terminates. Then, every non-quiescent 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, clearly, p_i eventually receives an *acknowledgment* message for every engaging application message it sends during x. Therefore p_i eventually becomes quiescent. $\square$

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.*

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 required are described in Section 4.5. For now, assume that all channels are FIFO. We have,

Proposition 5 *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 6 *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 3, the execution of every process ends with a quiescent interval.

From Proposition 4, every process sends an *initialize* message in the first quiescent interval. But the first interval for an initially active process is a nonquiescent interval. However, it can be verified that an initially active process does not send any *st_active* message in the first nonquiescent interval.

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. □

From Lemma 6, 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 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 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 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 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,

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

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

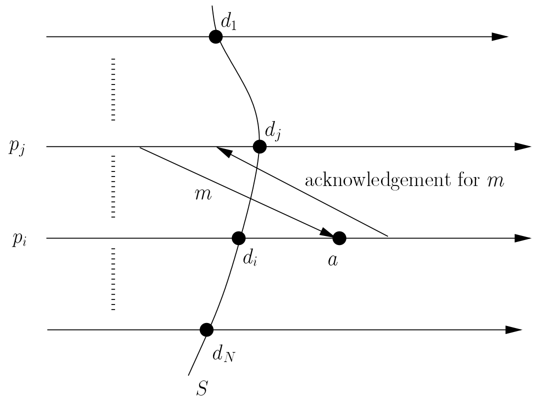

Figure 5: Proving the safety of TDA-SI.

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 Figure 5. 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,

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

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 rcv. Using Lemma 2, $rcv \rightarrow d_j$. Since $d_j \rightarrow \mathit{announce}$, $rcv \rightarrow \mathit{announce}$. Using Proposition 5 and Lemma 6, immediately after executing rcv, clearly, the activity counter of 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 a. This again contradicts our choice of d_i. □

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

4.4 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),

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

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 modification required are described in Section 4.5. 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* $\Theta(M+N)$, *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 generate up to D additional messages. Likewise, an *ack_st_active* message may also generate up to D more messages. From Lemma 6, 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 sent 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 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:

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

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 the algorithm required in case one of more channels are non-FIFO.

4.5 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 send (Proposition 5). 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.

5 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.

```
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 ≜ (state_i = passive) ∧ (missing_i = 0) ∧ (all incoming channels have been colored);

(C1) On receiving ⟨marker⟩ message from process p_j:
      if not(startDetection_i) then
            send ⟨marker⟩ message along all outgoing channels;
            startDetection_i := true;
      endif;
      if (p_i ≠ 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.

To correctly detect termination with delayed initiation, we use the scheme proposed in [3]. 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. 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 so far, 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.

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 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 $O(D)$.

6 Conclusion and Future Work

In this paper, we provide 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. However, the coordinator does not need to know the initial state of the system (which processes are initially active). 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. Although both our algorithms have optimal message complexity and optimal detection latency, they are sub-optimal in bit-message complexity. Specifically, each application message has to be modified to carry an integer whose maximum value is D. Thus the bit-message complexity of our algorithms is $O(M \log D + N)$ and $O(M \log D + E)$, respectively. An interesting research question is: "Is there a termination detection algorithm that is optimal in terms of detection latency as well as bit-message complexity?"

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)*, La Martinique, France, December 2003.

[3] 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.

[4] 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.

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

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

[7] 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.

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

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

[10] 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.

[11] 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.

[12] 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.

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

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

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

[16] 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.

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

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

[19] 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.

[20] 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.

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

[22] 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.

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

[24] 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.