On Termination Detection in an Asynchronous Distributed System

Sathya Peri   Neeraj Mittal

Department of Computer Science

The University of Texas at Dallas

Richardson, TX 75083

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

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. We focus on two attributes of a termination detection algorithm. First, whether the distributed computation (whose termination is to be detected) starts from a single process or from multiple processes (*diffusing computation* versus *non-diffusing computation*). Second, whether the detection algorithm should be initiated along with the computation or can be initiated anytime after the computation has started (*simultaneous initiation* versus *delayed initiation*). We show that, given any termination detection algorithm for a diffusing computation, it can be transformed into detecting termination of a non-diffusing computation. Further, we show that, given a termination detection algorithm for simultaneous initiation, it can be transformed into a termination detection algorithm for delayed initiation. We prove the correctness of our transformations and also show that the increase in message complexity is minimal and there is no change in the detection latency.

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

The problem of termination detection was independently proposed by Dijkstra and Scholten [5] and Francez [6] 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.*, [16, 17, 18, 13, 4, 14, 8, 2, 15, 7, 19, 9, 1, 11]). 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.*, [19, 9, 1, 11]).

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

One of the earliest known termination detection algorithms, which was proposed by Dijkstra and Scholten [5], assumes that the underlying computation is diffusing. Shavit and Francez [18] generalize Dijkstra and Scholten's algorithm to work for any non-diffusing computation. Other examples of termination detection algorithms developed for diffusing computation are Mattern's echo algorithm [13], Mattern's credit distribution and recovery algorithm [14] and Huang's weight throwing algorithm [8]. The last two algorithms are very similar and were proposed independently by Mattern and Huang. While modifications have been proposed to the credit distribution and recovery algorithm so that it can be used for a non-diffusing computation, they are ad hoc in nature and are, therefore, specific to the algorithm.

Matocha and Camp, in their paper on taxonomy of termination detection algorithms, write "[termination detection] algorithms that require diffusion place restrictions on the kinds of computations with which they may execute" [12]. In this paper, we show that the restriction is not rigid and can be easily relaxed. Specifically, we give a transformation that can be used to convert *any* termination detection algorithm for a diffusing computation to an algorithm for detecting termination of a non-diffusing computation. Note that, when an arbitrary number of processes can be initially active, 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. We show that if the message-complexity of the given termination detection algorithm is $O(C)$, then the message complexity of the resulting termination detection algorithm is $O(C+N)$.

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 [3]. As a result, if the underlying computation is message-intensive, then a termination detection algorithm may exchange a large number of control messages. Chandrasekaran and Venkatesan

[2] give a termination detection algorithm that can be initiated anytime *after* the computation has started. The advantage of their approach is that the number of control messages exchanged by their algorithm depends on the number of application messages exchanged by the computation *after* the termination detection algorithm began. As a result, their algorithm is especially suited to detect termination of a message-intensive computation for which it is preferable to initiate the algorithm when the computation is "close" to termination. Chandrasekaran and Venkatesan [2] show that, with delayed initiation, any termination detection algorithm must exchange at least E control messages, where E is the number of channels in the communication topology. Moreover, delayed initiation is not possible unless all channels are first-in-first-out (FIFO).

Mahapatra and Dutt [11] observe that the approach used by Chandrasekaran and Venkatesan [2] to achieve delayed initiation is applicable to many other termination detection algorithms as well, especially those based on acknowledgment (*e.g.*, [7]) and message-counting (*e.g.*, [13]). However, from their discussion, it is not clear whether *any* termination detection algorithm designed for simultaneous initiation can be modified to start later without increasing its message complexity in any way (except when required to solve the problem) and, if yes, how. In this paper, we give a transformation that can be used to initiate *any* termination detection algorithm later after the computation has already begun. Moreover, if the message-complexity of the given termination detection algorithm is $O(C)$, then the message complexity of the resulting termination detection algorithm is $O(C + E)$.

Both our transformations are general in the sense that they do not make any additional assumptions about the communication topology or the ordering of messages (except those made by the given termination detection algorithm or required to solve the problem). Moreover, the worst-case detection latency of the resulting termination detection algorithm is same as that of the given termination detection algorithm. (For delayed initiation, the detection latency is unaffected if the termination detection algorithm is started before the computation terminates.)

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. We describe and analyze the two transformations in Section 4 and Section 5. Due to lack of space, proofs of various lemmas and theorems are given in the appendix. 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. We do not assume that the communication topology is fully connected. There is no common clock or shared memory. Processes are non-faulty and channels are reliable. Message delays are finite but may be unbounded.

Processes execute *events* and change their states. 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 either a *send event* or a *receive event* but not both. 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 [10], which is defined as the smallest transitive relation satisfying the following properties:

1. if events e and f occur on the same process, and e occurred before f in real time then e happened-before f, and

2. if events e and f correspond to the send and receive, respectively, of a message then e happened-before f.

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

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

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

3 The Termination Detection Problem

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

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

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

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

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

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

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

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

In this paper, given a termination detection algorithm, we transform it into a termination detection algorithm satisfying certain desirable properties. Our transformation typically involves *simulating* one or more distributed computations based on the events of the underlying distributed computation. We call the underlying computation as *primary computation*, and a simulated computation as *secondary computation*. (We use the terms "underlying computation" and "primary computation" interchangeably.) It is possible for a process p_i to be in different states with respect to different computations. Therefore we use $state(\mathcal{F}, C, i)$ to refer to the state of the process p_i for the consistent cut C with respect to the computation $\mathcal{F}$. Formally,

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

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

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

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

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

We now describe the two transformations.

4 From Diffusing Computation to Non-Diffusing Computation

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

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

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

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

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

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

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

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

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

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

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

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

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

Next, we discuss how to simulate each secondary computation. This is important because, in general, a termination detection algorithm can *correctly* detect termination of a computation only if the computation follows the four rules (Rule 1-Rule 4) described in Section 3. For instance, suppose process p_i is passive with respect to a secondary computation $\mathcal{S}_j$. Clearly, p_i cannot send a message that belongs to $\mathcal{S}_j$ until it becomes active with respect to $\mathcal{S}_j$; otherwise Rule 2 would be violated. Further, p_i can become active with respect to $\mathcal{S}_j$ only on receiving a message that belongs to $\mathcal{S}_j$. That is, p_i should not become active with respect to $\mathcal{S}_j$ on receiving a message that belongs to some other secondary computation $\mathcal{S}_k$ with $j \neq k$; otherwise Rule 4 would be violated.

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

For detecting termination of the primary computation, from Lemma 1, it is sufficient to detect termination of all secondary computations. We assume that the given algorithm for detecting termination of a diffusing computation is such that, when the computation terminates, the process that detects the termination informs the initiator about it. In fact, for most algorithms, it is the initiator that is responsible for detecting termination of its computation. We also assume that there is a distinguished process, called the *coordinator*, that is responsible for detecting termination of the primary computation. So, when a process detects that the secondary computation initiated by it has terminated, it informs the coordinator by sending a *terminate* message to it. To minimize the number of control messages exchanged as a result, we assume that *terminate* messages sent by different processes are propagated to the coordinator in a *convergecast* fashion. This can be accomplished by building a spanning tree of the communication topology rooted at the coordinator in the beginning as a preprocessing step. We refer to the algorithm transformation described in this section as $DTON$. We now prove the correctness of our transformation.

Theorem 2 *Let A be a termination detection algorithm specifically designed to detect termination of a diffusing computation. Then, $DTON(A)$ correctly detects termination of a non-diffusing computation.*

We next show that our transformation increases the message complexity of the given termination detection algorithm by at most $O(N)$ messages under the following assumption: The message complexity of the given termination detection algorithm is $O(f(M))$ where M is the number of application messages exchanged by the diffusing computation before terminating and f is a function that distributes over addition. This is a realistic assumption because, for all termination detection algorithms for a diffusing computation that we know of, the message complexity is either $O(M)$ [5, 13] or $O(MD)$ [14, 8], where D is the diameter of the communication topology.

Theorem 3 *Let A be a termination detection algorithm specifically designed to detect termination of a diffusing computation. Assume that the message complexity of A for a given distributed system is $O(f(M))$, where M is the number of application messages exchanged by the diffusing computation and f is a function that distributes over addition. Then, the message complexity of $DTON(A)$ is $O(f(\bar{M}) + N)$, where $\bar{M}$ is the number of application messages exchanged by the non-diffusing computation.*

Finally, we show that our transformation does not increase the worst case detection latency of the given termination detection algorithm.

Theorem 4 *Let A be a termination detection algorithm specifically designed to detect termination of a diffusing computation. Then, the worst-case detection latency of $DTON(A)$ is same as that of A.*

Note that it is not necessary for a process to stay active with respect to more than one secondary computation. From (1), it is sufficient for a process to stay active with respect to *at most one* secondary computation as is the case with Shavit and Francez's extension [18] of the Dijkstra and Scholten's algorithm [5]. In other words, if a process is already active with respect to a secondary computation and receives a message that belongs to some other secondary computation, then, after processing the message, it can become passive with respect to the latter secondary computation immediately. The choice when to become passive with respect to a secondary computation depends on the given termination detection algorithm. For example, if, on becoming passive, the detection algorithm requires a control message to be sent to the same process for all secondary computations, then it is preferable for the process to become passive with respect to all secondary computations at the same time and send only a single control message.

5 From Simultaneous Initiation to Delayed Initiation

A termination detection algorithm, in the worst case, may exchange as many control messages as the number of application messages exchanged by the computation [3]. Therefore, if the underlying computation is message-intensive, then the detection algorithm may exchange a large number of control messages. Chandrasekaran and Venkatesan [2] propose a termination detection algorithm that can be started at anytime after the computation has begun. Their algorithm has the advantage that it needs to track only those application messages that are sent after it began executing. As a result, their algorithm is especially suited to detect termination of a message-intensive computation for which it is preferable to initiate the algorithm when the computation is "close" to termination. We describe a transformation that can be used to start any termination detection algorithm later after the computation has commenced.

To correctly detect termination with delayed initiation, we use the scheme proposed in [2]. The main idea is to distinguish between application messages sent by a process *before* it started termination detection and messages sent by it *after* it started termination detection. Clearly, the former messages should be ignored by the termination detection algorithm and the latter messages should be tracked by the termination detection algorithm. 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.

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

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

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

Our approach is to simulate a secondary computation such that the secondary computation and the termination detection algorithm start at the *same time on every process*. The secondary computation is almost identical to the primary computation except possibly in the beginning and satisfies the following two conditions. First, the termination of the secondary computation implies the termination of the primary computation. Second, once the primary computation terminates, the secondary computation terminates eventually. We then use the given termination detection algorithm to detect termination of the secondary computation. For a consistent cut C and a process

p_i, let $allRed(C, i)$ be true if all incoming channels of p_i are colored red for C. Formally,

$$allRed(C, i) \triangleq \begin{cases} \mathsf{true} & : \text{all incoming channels of process } p_i \text{ are colored red in consistent cut } C \\ \mathsf{false} & : \text{otherwise} \end{cases}$$

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

$$state(\mathcal{S}, C, i) \equiv state(\mathcal{P}, C, i) \vee \neg allRed(C, i) \tag{4}$$

Note that our secondary computation is a non-diffusing computation. Therefore we assume that the given termination detection algorithm for simultaenous initiation can detect termination of a non-diffusing computation. This is not a restrictive assumption as implied by the result of Section 4. Let $white(\mathcal{P})$ and $red(\mathcal{P})$ refer to the set of white and red messages, respectively, exchanged by the underlying computation $\mathcal{P}$. Clearly, $messages(\mathcal{P}) = white(\mathcal{P}) \cup red(\mathcal{P})$. Since all channels are FIFO,

$$\langle \forall i :: allRed(C, i) \rangle \Rightarrow (transit(\mathcal{P}, C) \cap white(\mathcal{P})) = \emptyset \tag{5}$$

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

$$transit(\mathcal{S}, C) = transit(\mathcal{P}, C) \cap red(\mathcal{P}) \tag{6}$$

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

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

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

Next, we show that once the primary computation terminates, the secondary computation terminates eventually. To that end, it suffices to show that the secondary computation terminates once the primary computation terminates and all incoming channels have been colored red.

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

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

We refer to the algorithm transformation described in this section as $STOD$. We now prove the correctness of our transformation.

Theorem 7 *Let A be a termination detection algorithm that requires simultaneous initiation. Then, $STOD(A)$ correctly detects termination with delayed initiation.*

We next show that our transformation increases the message complexity of the given termination detection algorithm by at most $O(E)$, where E is the number of channels in the communication topology.

Theorem 8 *Let A be a termination detection algorithm that requires simultaneous initiation. Assume that the message complexity of A for a given distributed system is $O(f(M))$, where M is the number of application messages exchanged by the computation. Then, the message complexity of $STOD(A)$ is $O(f(\bar{M}) + E)$, where $\bar{M}$ is the number of application messages exchanged by the computation after the $STOD(A)$ has started.*

Finally, we show that our transformation does not increase the worst case detection latency of the given termination detection algorithm assuming that the detection algorithm is initiated before the computation has terminated.

Theorem 9 *Let A be a termination detection algorithm that requires simultaneous initiation. If $STOD(A)$ is initiated before the underlying computation terminates, then the worst-case detection latency of $STOD(A)$ is same as that of A*

Besides efficiency, another advantage of delayed initiation is that it can be used to make a termination detection algorithm fault tolerant. Consider a distributed system in which processes can fail by crashing. On detecting failure of a process, the termination detection algorithm can be simply restarted and the earlier initiations of the detection algorithm can be ignored [2].

6 Conclusion

In this paper, we show that, given an algorithm to detect termination of a diffusing computation, it can be efficiently transformed into an algorithm to detect termination of a non-diffusing computation. Further, given a termination detection algorithm that has to be initiated along with the computation, it can be efficiently transformed into a termination detection algorithm that can be initiated anytime after the computation has started.

As future work, we plan to investigate whether the idea of delayed initiation can be applied to algorithms for detecting other stable properties.

References

[1] 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.

[2] 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.

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

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

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

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

[7] 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.

[8] 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.

[9] A. A. Khokhar, S. E. Hambrusch, and E. Kocalar. Termination Detection in Data-Driven Parallel Computations/Applications. *Journal of Parallel and Distributed Computing (JPDC)*, 63(3):312–326, March 2003.

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

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

[12] J. Matocha and T. Camp. A Taxonomy of Distributed Termination Detection Algorithms. *The Journal of Systems and Software*, 43(3):207–221, November 1999.

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

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

[15] 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.

[16] 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.

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

[18] 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.

[19] G. Stupp. Stateless Termination Detection. In *Proceedings of the 16th Symposium on Distributed Computing (DISC)*, pages 163–172, Toulouse, France, 2002.

A Omitted Proofs

A.1 Omitted Proofs of Section 4

Proof for Lemma 1: We have,

$$
\begin{aligned}
& \quad terminated(\mathcal{P}, C) \\
\equiv & \quad \{ \text{ definition of termination } \} \\
& \quad \langle \forall i :: \neg state(\mathcal{P}, C, i) \rangle \wedge (transit(\mathcal{P}, C) = \emptyset) \\
\equiv & \quad \{ \text{ from (1) and (3) } \} \\
& \quad \Big\langle \forall i :: \langle \forall j :: \neg state(\mathcal{S}_j, C, i) \rangle \Big\rangle \ \bigwedge \ \Big(\bigcup_{j=1}^{N} transit(\mathcal{S}_j, C) \ = \ \emptyset \Big) \\
\equiv & \quad \{ \text{ predicate and set calculus } \} \\
& \quad \Big\langle \forall j :: \langle \forall i :: \neg state(\mathcal{S}_j, C, i) \rangle \Big\rangle \ \bigwedge \ \Big\langle \forall j :: transit(\mathcal{S}_j, C) = \emptyset \Big\rangle \\
\equiv & \quad \{ \text{ predicate calculus } \} \\
& \quad \Big\langle \forall j :: \langle \forall i :: \neg state(\mathcal{S}_j, C, i) \rangle \wedge (transit(\mathcal{S}_j, C) = \emptyset) \Big\rangle \\
\equiv & \quad \{ \text{ definition of termination } \} \\
& \quad \langle \forall j :: terminated(\mathcal{S}_j, C) \rangle
\end{aligned}
$$

This establishes the lemma. □

Proof for Theorem 2: *(safety)* Clearly, the coordinator announces termination only after all secondary computations have terminated. From Lemma 1, it implies that the primary computation has terminated as well.

(liveness) Suppose the primary computation has terminated. From Lemma 1, all secondary com-

putations have also terminated. Their respective initiators will eventually inform the coordinator about it. As a result, the coordinator will eventually announce termination. □

Proof for Theorem 3: For convenience, let $M_i = |messages(\mathcal{S}_i)|$. The control messages exchanged by $DTON(A)$ consists of the control messages exchanged by the N instances of A to detect termination of the N secondary computations and *terminate* messages. Therefore the message complexity of $DTON(A)$ is given by:

$$
\begin{aligned}
& \Big(\sum_{i=1}^{N} O(f(M_i))\Big) + O(N) \\
= \quad & \{\ f \text{ distributes over addition } \} \\
& O(f(\sum_{i=1}^{N} M_i)) + O(N) \\
= \quad & \left\{ \begin{array}{l} \text{each application message is part of at most one secondary computation} \\ \text{and using (2)} \end{array} \right\} \\
& O(f(\bar{M})) + O(N) = O(f(\bar{M}) + N)
\end{aligned}
$$

This proves the theorem. □

Proof for Theorem 4: Let the worst case detection latency of A be $O(L)$ message hops. Clearly, L is $\Omega(D)$, where D is the diameter of the communication topology. From Lemma 1, when the primary computation terminates, all secondary computations terminate as well. Once all secondary computations have terminated, the coordinator learns about their termination within $O(D)$ message hops. As a result, the worst case detection latency of $DTON(A)$ is $O(L + D)$ which is same as $O(L)$. □

A.2 Omitted Proofs of Section 5

Proof for Lemma 5: We have,

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

$\equiv$ { definition of termination }

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

$\equiv$ { from (4) and (6) }

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

$\equiv$ { predicate calculus }

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

$\Rightarrow$ { from (5) }

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

$\equiv$ { predicate calculus }

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

$\Rightarrow$ { using $transit(\mathcal{P}, C) \subseteq white(\mathcal{P}) \cup red(\mathcal{P})$ }

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

$\equiv$ { definition of termination }

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

This establishes the lemma. □

Proof for Lemma 6: We have,

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

$\equiv$ { definition of termination }

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

$\equiv$ { predicate and set calculus }

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

$\Rightarrow$ { from (4) and (6) }

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

$\equiv$ { definition of termination }

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

This establishes the lemma. $\square$

Proof for Theorem 7: Our transformation ensures that the given termination detection algorithm A and the simulated secondary computation $\mathcal{S}$ start simultaneously on every process.

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

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

Proof for Theorem 8: The message complexity of $STOD(A)$ is given by the sum of (1) the number of *marker* messages exchanged and (2) the number of control messages exchanged by A when used to detect termination of the secondary computation. Clearly, at most one *marker* message is generated for every channel. Also, the number of messages exchanged by the secondary computation is same as the number of red application messages, and a red application is generated only after $STOD(A)$ has started. $\Box$

Proof for Theorem 9: Let the worst case detection latency of A be $O(L)$ message hops. Clearly, L is $\Omega(D)$, where D is the diameter of the communication topology. From Lemma 6, when the primary computation terminates, the secondary computation terminates as soon as all incoming channels become red. Clearly, once $STOD(A)$ is initiated, all incoming channels become red within $O(D)$ message hops. Moreover, once the secondary computation terminates, A detects its termination within $O(L)$ message hops. As a result, the worst case detection latency of $STOD(A)$ is $O(L + D)$ which is same as $O(L)$. $\Box$