Available online at www.sciencedirect.com

ScienceDirect

J. Parallel Distrib. Comput. 68 (2008) 855–875

Journal of Parallel and Distributed Computing

www.elsevier.com/locate/jpdc

On termination detection in crash-prone distributed systems with failure detectors[☆]

Neeraj Mittal[a,*], Felix C. Freiling[b], S. Venkatesan[a], Lucia Draque Penso[b]

[a] *Department of Computer Science, The University of Texas at Dallas Richardson, TX 75083, USA*
[b] *Department of Computer Science, University of Mannheim, D-68131 Mannheim, Germany*

Received 18 June 2007; received in revised form 27 November 2007; accepted 4 February 2008
Available online 21 February 2008

Abstract

We investigate the problem of detecting termination of a distributed computation in systems where processes can fail by crashing. Specifically, when the communication topology is fully connected, we describe a way to transform *any* termination detection algorithm $\mathcal{A}$ that has been designed for a failure-free environment into a termination detection algorithm $\mathcal{B}$ that can tolerate process crashes. Our transformation assumes the existence of a *perfect failure detector*. We show that a perfect failure detector is in fact necessary to solve the termination detection problem in a crash-prone distributed system even if *at most one* process can crash.

Let $\mu(n, M)$ and $\delta(n, M)$ denote the message complexity and detection latency, respectively, of $\mathcal{A}$ when the system has n processes and the underlying computation exchanges M application messages. The message complexity of $\mathcal{B}$ is $O(n + \mu(n, 0))$ messages per failure more than the message complexity of $\mathcal{A}$. Also, its detection latency is $O(\delta(n, 0))$ per failure more than that of $\mathcal{A}$. Furthermore, application message size increases by at most $\log(f + 1)$ bits, where f is the actual number of processes that fail during an execution. We show that, when the communication topology is fully connected, under certain realistic assumption, any fault-tolerant termination detection algorithm can be forced to exchange $\Omega(nf)$ control messages in the worst-case even when *at most one* process may be active initially and the underlying computation *does not exchange* any application messages. This implies that our transformation is optimal in terms of message complexity when $\mu(n, 0) = O(n)$.

The fault-tolerant termination detection algorithm resulting from the transformation satisfies three desirable properties. First, it can tolerate the failure of up to $n - 1$ processes. Second, it does not impose any overhead on the fault-sensitive termination detection algorithm until one or more processes crash. Third, it does not block the application at any time. Further, using our transformation, we derive a fault-tolerant termination detection algorithm that is the *most efficient* fault-tolerant termination detection algorithm that has been proposed so far to our knowledge. Our transformation can be extended to arbitrary communication topologies provided process crashes do not partition the system.

Keywords: Distributed system; Termination detection; Faulty processes; Crash failure; Fault-tolerant algorithm; Failure detector; Algorithm transformation; Non-blocking

1. Introduction

One of the important problems in distributed systems is to detect termination of an ongoing distributed computation, which intuitively involves determining whether the computation has ceased all its activities. A process currently involved in some activity is considered to be in active state. A process ceases its activity by changing its state to passive. An active process can make another process active by sending an application message to it. The computation terminates once all processes become passive and stay passive thereafter. The termination detection problem was independently proposed by Dijkstra and Scholten [11] and Francez [13] more than two decades ago. Since then, many researchers have studied this problem and, as a result, a large number of efficient algorithms have been developed for detecting termination (*e.g.*, [31,33,26,10,27,19,4,34,20,37,24,29,25,9]). Most of the termination

[☆] A preliminary version of the paper first appeared in the 19th Symposium on Distributed Computing (DISC), 2005 [N. Mittal, F.C. Freiling, S. Venkatesan, L.D. Penso, Efficient reduction for wait-free termination detection in a crash-prone distributed system, in: Proceedings of the 19th Symposium on Distributed Computing, DISC, September 2005, pp. 93–107].

[*] Corresponding author.

E-mail addresses: neerajm@utdallas.edu (N. Mittal), freiling@informatik.uni-mannheim.de (F.C. Freiling), venky@utdallas.edu (S. Venkatesan), lucia@rumms.uni-mannheim.de (L.D. Penso).

0743-7315/$ - see front matter © 2008 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2008.02.001

detection algorithms in the literature have been developed assuming that both processes and channels stay operational throughout an execution. Real-world systems, however, are often prone to failures. For example, processes may fail by crashing and channels may be lossy. In this paper, we investigate the termination detection problem when processes can fail by crashing. We assume that process crashes do not result in restarting of the primary computation.

One of the earliest fault-tolerant algorithm for termination detection was proposed by Venkatesan [36], which was derived from the fault-sensitive (that is, fault-*in*tolerant) termination detection algorithm by Chandrasekaran and Venkatesan [4]. Venkatesan's algorithm achieves fault-tolerance by *replicating* state information at multiple processes. However, it assumes a *pre-specified* bound $f_{\max}$ on the maximum number of processes that can fail by crashing. Its message complexity is $O(f_{\max} M + c)$, where M is the number of application messages exchanged by the underlying computation and c is the number of channels in the communication topology. As a result, the overhead incurred by the algorithm depends on the *maximum* number of processes that can fail during an execution rather than the *actual* number of processes that fail during an execution. Moreover, the algorithm assumes that it is possible to send up to $f_{\max} + 1$ (possibly different) messages to different processes in an *atomic* manner. Unlike other fault-tolerant termination detection algorithms, however, Venkatesan's algorithm does not assume that the communication topology is fully connected and works as long the topology is $(f_{\max} + 1)$-connected [36]. (A graph is said to be $(k + 1)$-connected if it stays connected even after k or fewer vertices have been removed from the graph.)

Lai and Wu [21] and Tseng [35] modify fault-sensitive termination detection algorithms by Dijkstra and Scholten [11] and Huang [19,27], respectively, to derive two different fault-tolerant termination detection algorithms. Both algorithms assume a fully connected communication topology. However, unlike Venkatesan's algorithm, both have low message complexity of $O(M + fn + n)$, where n is the initial number of processes in the system and f is the actual number of processes that fail during the execution. The algorithm by Lai and Wu [21] has high detection latency of $O(n)$ whereas the algorithm by Tseng [35] has high control message-size complexity of $O(f \log n + nM)$ for control messages exchanged due to process crashes. (We use the term "application message" to describe a message exchanged by the underlying computation and the term "control message" to describe a message exchanged by the termination detection algorithm. Also, message-size complexity refers to the amount of control information piggybacked on a message.)

Shah and Toueg give a fault-tolerant algorithm for taking a consistent snapshot of a distributed system in [32]. Their algorithm is derived from the fault-sensitive consistent snapshot algorithm by Chandy and Lamport [5]. As a result, each invocation of their snapshot algorithm may generate up to $O(c)$ control messages. When their algorithm is used for termination detection, the message complexity of the resulting algorithm is $O(cM)$ in the worst-case. Hélary et al. [18] describe an algorithm for computing a function on a global data in a distributed system where processes may fail by crashing. They also combine their algorithm with Mattern's counter-based termination detection algorithm [26], which is fault-sensitive, to derive a termination detection algorithm that can tolerate process crashes. Informally, their approach involves *repeated* computation of a specific global function until termination is detected [18]. Each instance of the function computation requires $\Omega(n^2)$ control messages and the function may need to be computed $\Omega(M + f)$ times in the worst-case. Gärtner and Pleisch [15] give an algorithm for detecting an arbitrary stable predicate (such as termination) in a crash-prone distributed system. In their algorithm, every relevant local event is *reliably and causally broadcast* to a set of monitors, thereby increasing the message complexity.

In this paper, when the communication topology is fully connected, we describe a way to transform any fault-sensitive termination detection algorithm $\mathcal{A}$ into a fault-tolerant termination detection algorithm $\mathcal{B}$. Our transformation assumes the existence of a perfect failure detector, which we show is necessary to solve the problem. Let $\mu(n, M)$ and $\delta(n, M)$ denote the message complexity and detection latency, respectively, of $\mathcal{A}$ when the system has n processes and the underlying computation exchanges M application messages. The message-complexity of $\mathcal{B}$ is $O(f(n + \mu(n, 0)))$ messages more than the message complexity of $\mathcal{A}$. Also, the detection latency of $\mathcal{B}$ is $O(f\delta(n, 0))$ more than the detection latency of $\mathcal{A}$. For most termination detection algorithms, when the topology is fully connected, $\mu(n, 0)$ is either $O(n)$ or $O(c)$, and $\delta(n, 0)$ is $O(1)$. For example, for the Dijkstra and Scholten's algorithm [11], $\mu(n, 0) = O(n)$ and $\delta(n, 0) = O(1)$ when their algorithm is modified to handle a non-diffusing computation. (A computation is said to be diffusing if at most one process can be active in the beginning and is said to be non-diffusing otherwise.) The application message-size complexity of $\mathcal{B}$ is only $\log(f + 1)$ more than that of $\mathcal{A}$. The fault-tolerant termination detection algorithm $\mathcal{B}$ uses two types of control messages, namely those for termination detection and those for recovering from process failures. The former type of control messages are derived from the control messages of the fault-sensitive termination detection algorithm $\mathcal{A}$. The message-size complexity of termination-detection messages is only $\log(f + 1)$ more than of $\mathcal{A}$. Further, the message-size complexity of failure recovery messages is given by $O(f \log n + n \log M)$. The fault-tolerant termination detection algorithm resulting from the transformation satisfies three desirable properties. First, it can tolerate failure of up to $n - 1$ processes, that is, it is *wait-free*. Second, it does not impose any overhead on the fault-sensitive termination detection algorithm if no process actually crashes during an execution. that is, it is *fault-reactive*. Third, it does not prevent the application from sending and delivering messages at any time, that is, it is *non-blocking*.

The main idea behind our approach is to *restart* the fault-sensitive termination detection algorithm whenever a *new* failure is detected. A separate mechanism is used to account for those application messages that are still in-transit when the termination detection algorithm is restarted. Although the idea behind our transformation is simple, we use it to derive

Table 1
Comparison of various fault-tolerant termination detection algorithms

		Venkatesan [36]	Lai and Wu [21]	Tseng [35]	Our approach [this paper]
Message complexity		$O(f_{\max}M + c)$	$O(M + fn + n)$	$O(M + fn + n)$	$\mu(n, M) + O(f(n + \mu(n, 0)))$
Detection latency		$O(M)$	$O(n)$	$O(f + 1)$	$\delta(n, M) + O(f\delta(n, 0))$
Application message-size complexity		$f_{\max}$-way duplication[b]	–	$O(\log M)$[a]	$\alpha(n, M) + \log(f + 1)$
Control message-size complexity	Termination detection	$O(\log n + \log M)$	$O(f \log n + \log M)$	$O(\log M)$[a]	$\beta(n, M) + \log(f + 1)$
	Failure recovery	–	$O(f \log n + \log M)$	$O(f \log n + nM)$	$O(f \log n) + O(n \log M)$
Assumptions		FIFO channels + atomic multicast	Fully connected topology	Fully connected topology	Fully connected topology

n: initial number of processes in the system. c: number of channels in the communication topology.
M: number of application messages exchanged. f: actual number of processes that crash during the execution.
$f_{\max}$: maximum number of processes that can crash during an execution.
$\mu(n, M)$: message complexity of the fault-sensitive termination detection algorithm with n processes and M application messages. We assume that μ satisfies the following inequality: $\mu(n, X) + \mu(n, Y) \le \mu(n, X + Y) + \mu(n, 0)$.
$\delta(n, M)$: detection latency of the fault-sensitive termination detection algorithm with n processes and M application messages.
$\alpha(n, M)$: application message-size complexity of the fault-sensitive termination detection algorithm with n processes and M application messages.
$\beta(n, M)$: control message-size complexity of the fault-sensitive termination detection algorithm with n processes and M application messages.
[a] Assuming an efficient implementation of weight throwing scheme such as the one described in [27]. The algorithm presented in [35] has much higher message-size complexity of $O(M)$.
[b] Each message is multicast to $f_{\max} + 1$ processes implying that a message is *duplicated* $f_{\max}$ times.

Table 2
The complexity measures when the fault-sensitive termination detection algorithm is acknowledgment-based

Message complexity	Detection latency	Application message-size complexity	Control message-size complexity		Assumptions
			Termination detection	Failure recovery	
$\mu(n, M) + O(f(n + \mu(n, 0)))$	$\delta(n, M) + O(f\delta(n, 0))$	$\log(f + 1)$	$\log(f + 1)$	$O(f \log n)$	Fully connected topology

(For complete notation please refer to Table 1.)

a fault-tolerant termination detection algorithm that is more efficient than all the existing algorithms [21,35,18]. Observe that group communication service (GCS) along with sending view delivery [17] can also be used to devise a fault-tolerant termination detection algorithm. This, however, requires the application to be *blocked* until a new view is installed. We are interested in an approach that does not require the application to be blocked at any time.

Arora and Gouda [1] also provide a mechanism to reset a distributed system. Their reset mechanism involves restarting all processes from their initial states. Messages sent by older instances are simply ignored. Their approach to achieve fault tolerance is quite different from our approach. First, the semantics of their reset operation is different from the semantics of our restart operation. If their reset mechanism is applied to our system, then it will not only reset the termination detection algorithm but will also reset the underlying distributed computation (whose termination is to be detected). Further, application messages exchanged by the underlying computation before it is reset will be discarded. If a failure occurs near the completion of the underlying computation, the entire work needs to be redone. In contrast, in our case, the distributed computation continues to execute without interruption. (We assume that the underlying computation is able to cope with process failures without the need to restart itself.) Therefore, in our case, application messages exchanged before the termination detection algorithm is restarted, especially those exchanged between correct processes, cannot be ignored. Arora and Gouda's approach is more suitable for applications that can be reset on occurrence of a failure whereas our approach is more suitable for applications that continue to execute despite failures. Another difference between the two approaches is that their reset operation, which is self-stabilizing in nature, is designed to tolerate much broader and more severe kinds of faults such as restarts, message losses and arbitrary state perturbations in addition to crash failures. Not surprisingly, their reset operation has higher message and time complexities than our restart operation.

For comparison between various fault-tolerant termination detection algorithms, please refer to Table 1. In the table, we distinguish between two types of control messages: those that are exchanged by the fault-sensitive termination detection algorithm (termination-detection messages) and those that are exchanged to recover from process crashes (failure-recovery messages). The results of applying our transformation to some important termination detection algorithms are given in [28].

Typically, generalized transformations tend to be inefficient compared to customized/specialized transformations. However, when our transformation is applied to fault-sensitive termination detection algorithms by Dijkstra and Scholten [11] and

Huang [19,27], the resulting fault-tolerant algorithms compare very favorably with those by Lai and Wu [21] and Tseng [35]. Specifically, when our transformation is applied to Dijkstra and Scholten's algorithm [11], the resulting algorithm has the same message-complexity and detection latency as the algorithm by Lai and Wu [21]. However, application and failure-recovery messages carry more control information in our algorithm. (Actually, in Lai and Wu's termination detection algorithm, there is no distinction between termination-detection and failure-recovery messages.) On the other hand, when our transformation is applied to Huang's weight throwing algorithm [19], the resulting algorithm has the same message-complexity and detection latency as that of the algorithm by Tseng [35] but has slightly higher message-size complexity for application and termination-detection messages—$O(\log M)+\log(f+1)$ versus $O(\log M)$. Surprisingly, the size of failure-recovery messages, which is given by $O(f \log n+n \log M)$, is *much lower* than that of Tseng's algorithm [35], which is given by $O(f \log n+nM)$. We expect M to be much greater than $f+1$ in practice. Therefore, the overhead of $O(\log M)+\log(f+1)$ for all practical purposes is same as $O(\log M)$.

We show that, when the underlying fault-sensitive termination detection algorithm is acknowledgment-based, the size of failure-recovery messages can be reduced from $O(f \log n + n \log M)$ to only $O(f \log n)$ without affecting other complexity measures. As a corollary, when the optimized transformation is applied to Mittal et al.'s fault-sensitive termination detection algorithm proposed in [29], which is acknowledgment-based, the resulting fault-tolerant termination detection algorithm has message-complexity of $O(M + fn + n)$, detection latency of $O(f + 1)$, application and termination-detection message-size complexity of only $\log(f + 1)$ and failure-recovery message-size complexity of $O(f \log n)$ (see Table 2). Note that the message-size complexity of $\log(f + 1)$ corresponds to piggybacking only one additional integer on a message in practice. As a result, we believe that the above algorithm is the *most efficient* fault-tolerant termination detection algorithm among all fault-tolerant termination detection algorithms that have been proposed so far.

We also show that, when the communication topology is fully connected, our transformation is optimal in the sense that, under certain realistic assumption, any fault-tolerant termination detection algorithm can be forced to exchange $\Omega(nf)$ control messages in the worst-case even when at most one process may be active initially and the underlying computation does not exchange any application messages. Further, the lower bound holds even if all channels are FIFO (first in first out) and a process can atomically multicast a message to multiple processes. Our transformation can also be extended to an arbitrary communication topology provided process crashes do not partition the system. Details of the extension can be found elsewhere [28].

We build upon the work by Wu et al. [38]. We do this in the context of the failure detector hierarchy proposed by Chandra and Toueg [3], a way to compare problems based on the level of *synchrony* required for solving them. We show that termination detection needs the synchrony assumptions of a perfect failure detector to be solvable even if at most one process can crash. This result can be used to further understand the relationship between termination detection and other problems in fault-tolerant distributed computing, such as consensus and atomic broadcast.

Our contributions: To summarize, we make the following contributions in this paper. First, we present a transformation that can be used to convert any fault-sensitive termination detection algorithm for a fully connected communication topology into a fault-tolerant termination detection algorithm able to cope with process crashes. Our transformation uses a perfect failure detector. We also use our transformation to derive the most efficient fault-tolerant termination detection algorithm known so far. Second, we establish that, under certain realistic assumptions, our transformation is optimal in terms of message-complexity when the communication topology is fully connected. Third, we prove that a perfect failure detector is the weakest failure detector for solving the termination detection problem in a crash-prone distributed system. This holds even if at most one process can crash.

Roadmap: This paper is organized as follows. In Section 2, we present our model of a crash-prone distributed system and describe what it means to detect termination in such a system. We discuss our transformation for fully connected topology in Section 3. The lower bound on message complexity of termination detection in a failure-prone environment is derived in Section 4. In Section 5 we investigate the type of failure detector that is necessary for solving the termination detection problem in fault-prone environment. Finally, we present our conclusions and outline directions for future research in Section 6.

2. Model and problem definition

2.1. System model

We assume an asynchronous distributed system consisting of multiple processes, which communicate with each other by exchanging messages over a set of communication channels. There is no global clock or shared memory. Processes are not reliable and may fail by crashing. Once a process crashes, it halts all its operations and never recovers. We use the terms "live process" and "operational process" interchangeably. A process that eventually crashes is called *faulty*. A process that is not faulty is called *correct*. Note that there is a difference between a "live process" and a "correct process". A live process has not crashed yet but may crash in the future. A correct process, on the other hand, never crashes. Let $P = \{p_1, p_2, \ldots, p_n\}$ denote the initial set of processes in the system. We assume that there is at least one correct process in the system.

We assume that all channels are bidirectional but may not be FIFO (first in first out). Channels are assumed to be reliable in the sense that a message sent by a correct process to another correct process is eventually delivered. A message may, however, take an arbitrary amount of time to reach its destination. Unless otherwise stated, we assume that the communication topology is fully connected.

Processes change their states by executing events. When processes are reliable, an execution of a distributed system can be modeled as a *sequence of events* (internal or external) that have been executed so far on different processes. In this paper, we treat crash of a process as a special event on that process. However, unlike other (non-crash) events, a process cannot execute any further events once it has executed the crash event. Therefore, when processes can fail by crashing, a system execution can still be modeled as a sequence of events (internal, external or crash) that have been executed so far on different processes.

2.2. *Failure detection*

We assume the existence of a *perfect failure detector* [3], a mechanism which gives processes reliable information about the operational state of other processes. Upon querying the local failure detector, a process receives a list of *currently suspected* processes. A perfect failure detector satisfies two properties [3]: *strong accuracy* (no correct process is ever suspected) and *strong completeness* (a crashed process is eventually permanently suspected by every correct process). By varying definitions of completeness and accuracy, different types of failure detectors can be defined. For example, an eventually perfect failure detector satisfies *eventually strong accuracy* (eventually no correct process is ever suspected) and strong completeness. Chandra and Toueg [3] define an ordering relation on failure detectors. Failure detector D_1 is *weaker than* failure detector D_2 (denoted $D_1 \leq D_2$) if we can implement D_1 using D_2. Intuitively, if D_1 is weaker than D_2, then D_2 gives at least as much information about failures as D_1. If $D_1 \leq D_2$ and $D_2 \not\leq D_1$ we say that D_1 is *strictly weaker*.

2.3. *Termination detection in a crash-prone system*

We first describe the condition that should hold for a distributed computation to be in terminated state. We then describe the acceptable behavior of a termination detection algorithm, and also discuss various complexity measures we use to evaluate the performance of a termination detection algorithm.

2.3.1. *Termination condition to be evaluated*

Informally, the termination detection problem involves determining when a distributed computation has ceased all its activities. The distributed computation satisfies the following four properties or rules. First, a process is either *active* or *passive*. Second, a process can send a message only if it is active. Third, an active process may become passive at any time. Fourth, a passive process may become active only on receiving a message. Intuitively, an active process is involved in some local activity, whereas a passive process is idle. In case processes never fail and channels are reliable, a distributed computation terminates once all processes become passive and stay passive thereafter. In other words, a distributed computation is said to be *classically-terminated* once all processes become passive and all channels become empty.

In a crash-prone distributed system, once a process crashes, it ceases all its activities. Moreover, any message in-transit towards a crashed process can be ignored because the message cannot initiate any new activity. Therefore, in a crash-prone distributed system, a computation is said to be *strictly-terminated* if all live processes are passive and no channel contains a message in-transit towards a live process. Wu et al. [38] establish that, for the strict-termination detection problem to be solvable in a crash-prone distributed system, it must be possible to *flush* the channel from a crashed process to a live process. A channel can be flushed using either *return-flush* [36] or *fail-flush* [21] primitive. Both primitives allow a live process to ascertain that its incoming channel from the crashed process has become empty.

In case neither return-flush nor fail-flush primitive is available, Tseng suggested *freezing* the channel from a crashed process to a live process [35,18]. When a live process freezes its channel with a crashed process, any message that arrives after the channel has been frozen is ignored. (A process can freeze a channel only after detecting that the process at the other end of the channel has crashed.) We say that a message is *deliverable* if it is destined for a live process along a channel that has not been frozen yet; otherwise it is *undeliverable*. We now say that a distributed computation is *effectively-terminated* if all live processes are passive and there is no deliverable message in-transit towards a live process. Trivially, strict-termination implies effective-termination but not vice versa. Deciding which of the two termination conditions is to be detected depends on the application semantics. In this paper, we focus on detecting whether a computation has effectively-terminated. Our transformation, however, can be easily extended to detect strict-termination as well.

Wu et al. [38] also show that in order for strict-termination detection to be solvable, process faults must be *detectable*. Translated into the terminology of Chandra and Toueg [3], the failure detector used should satisfy strong completeness. We fulfill this requirement by assuming the existence of a perfect failure detector, which additionally satisfies strong accuracy. We justify this assumption later by proving that we need at least a perfect failure detector to solve even effective-termination detection in a crash-prone distributed system. Further, we assume that it is possible to freeze the channel from a crashed process to a live process (that is, application allows messages from crashed processes to be discarded). Hereafter, we focus on effective-termination detection. The transformation results in Section 3, however, remain valid even for strict-termination detection assuming that channels can be flushed instead of frozen. We will return to this point later in Section 5.

2.3.2. *A termination detection algorithm*

An algorithm that solves the termination detection problem should satisfy the following two correctness properties:

- *(liveness)* Once the computation terminates, the algorithm eventually announces termination.

- *(safety)* If the algorithm announces termination, then the computation has indeed terminated.

In this paper, by the phrase "the algorithm announces termination", we mean that some operational process in the system announces termination. In addition to the above properties, it is desirable that the algorithm be *non-blocking*, that is, it should not prevent the application from sending and delivering messages at any time. Later, when proving the lower bound on message-complexity, we consider a weaker form of non-blocking property.

We call a termination detection algorithm *fault-tolerant* if it works correctly even in the presence of faults; otherwise it is called *fault-sensitive* or *fault-intolerant*. In this paper, we use the terms "crash", "fault" and "failure" interchangeably. For convenience, we refer to messages exchanged by the underlying distributed computation as *application messages* and to messages exchanged by the termination detection algorithm as *control messages*.

The performance of a termination detection algorithm is measured in terms of three metrics: message complexity, detection latency and message-size complexity. Message complexity refers to the number of control messages exchanged by the termination detection algorithm in order to detect termination. Detection latency measures the time elapsed between when the underlying computation terminates and when the termination detection algorithm actually announces termination. To compute detection latency of a termination detection algorithm, we assume that message transmission delay as well as failure detection delay is at most one time unit. Further, message processing time is negligible. This is consistent with the assumption made by Lai and Wu [21] and Tseng [35] when analyzing detection latency of their algorithms. Finally, message-size complexity refers to the amount of control data piggybacked on a message by the termination detection algorithm.

3. From fault-sensitive algorithm to fault-tolerant algorithm

We assume that the given fault-sensitive termination detection algorithm is able to detect termination of a non-diffusing computation, when any subset of processes can be initially active. This is not a restrictive assumption as it is proved in [30] that any termination detection algorithm for a diffusing computation, when at most one process is initially active, can be efficiently transformed into a termination detection algorithm for a non-diffusing computation. The transformation increases the message complexity of the underlying termination detection algorithm by only $O(n)$ messages and, moreover, does not increase its detection latency [30]. We also assume that, as soon as a process learns about the failure of its neighboring process, it freezes its incoming channel with the process.

3.1. The main idea

The main idea behind our transformation is to *restart* the fault-sensitive termination detection algorithm on the set of *currently operational processes* whenever a new failure is detected. We denote the fault-sensitive termination detection algorithm – an input to our transformation – by $\mathcal{A}$, and to the fault-tolerant termination detection algorithm – the output of our transformation – by $\mathcal{B}$. Before restarting $\mathcal{A}$, we ensure that all operational processes agree on the set of processes that have failed. The strong accuracy property of a perfect failure detector ensures that a new instance of $\mathcal{A}$ is started on *all* operational processes, that is, no operational process is "left out". This is necessary to ensure the safety of the termination detection algorithm. On the other hand, the strong completeness property of a perfect failure detector ensures that every crash is eventually detected and, as a result, a new instance of $\mathcal{A}$ is eventually restarted on a set of processes none of which fails thereafter. This is necessary to ensure the liveness of $\mathcal{A}$.

3.1.1. A safe subset of processes

Consider a subset of processes Q. We say that a computation has *terminated with respect to* Q (classically or strictly or effectively) if the respective termination condition holds when evaluated only on processes and channels in the subsystem induced by Q (that is, when the system consists of only processes in Q and channels between them). Also, we say that Q has become *safe* if (1) all processes in $P \setminus Q$ have failed, and (2) every process in Q has learned about the failure of all processes in $P \setminus Q$ (and has, therefore, frozen its incoming channels with processes in $P \setminus Q$). We now show that a safe subset satisfies a form of monotonicity.

Theorem 1. *Consider a safe subset of processes Q. Assume that all processes in Q are live. Then a distributed computation has effectively-terminated with respect to P if and only if it has classically-terminated with respect to Q.*

Proof. *(if)* Assume that the distributed computation has classically-terminated with respect to Q. Thus all processes in Q are passive and all channels among processes in Q are empty. Since Q is a safe subset of processes, all processes in $P \setminus Q$ have crashed. In other words, all live processes in the system, namely the processes in Q, are passive. Further, since every process in Q knows that all processes in $P \setminus Q$ have crashed, all channels from processes in $P \setminus Q$ to processes in Q have been frozen. As a result, there is no deliverable message in transit to any live process (that is, a process in Q). Thus the distributed computation has effectively-terminated with respect to P.
(only if) Now, assume that the distributed computation has effectively-terminated with respect to P. Therefore all live processes are passive, which implies that all processes in Q are passive. Further, there is no deliverable message in transit towards any live process. Specifically, since all processes in Q are live and all processes in $P \setminus Q$ have crashed, none of the channels among processes in Q contain a deliverable message in transit. This, in turn, implies that all channels among processes in Q are actually empty. In other words, the distributed computation has classically-terminated with respect to Q. □

3.1.2. A fault-sensitive algorithm is safe

The above theorem implies that if all live processes agree on the set of failed processes and there are no further crashes, then it is sufficient to ascertain that the underlying computation has classically-terminated with respect to the set of operational processes. An advantage of detecting classical termination is that we can use $\mathcal{A}$, a fault-sensitive termination detection algorithm, to detect termination. We next show that, even if one or more processes crash, $\mathcal{A}$ does not announce false termination. In other words, a fault-sensitive termination detection algorithm always satisfies the safety property but may not satisfy the liveness property.

Theorem 2. *When a fault-sensitive termination detection algorithm is executed on a distributed system prone to process crashes then the algorithm still satisfies the safety property, that is, it never announces false termination.*

Proof. Let σ be an execution of the system in which one or more processes crash and the fault-sensitive termination detection algorithm announces termination. Consider a prefix τ of σ in which the last event corresponds to termination announcement by some process, say p_t. We show that the underlying computation has indeed terminated when p_t announces termination.

Consider a sub-execution κ of τ obtained after removing all crash events from τ. Intuitively, this means that, as far as processes that stay operational in τ are concerned, they execute exactly the same sequence of events in τ and κ. On the other hand, as far as processes that crash in τ are concerned, they execute exactly the same sequence of events in τ and κ until they crash (in τ). Moreover, once a process crashes in τ, it does not execute any events in κ until p_t has announced termination but, nevertheless, stays operational. Clearly, κ is a valid execution of the system. This is because it is possible to delay execution of any process for an arbitrary but finite amount of time due to the asynchronous nature of the system.

Note that process p_t executes exactly the same sequence of events in τ and κ and, therefore, cannot distinguish between them without using a failure detector. As a result, if it announces termination in τ, then it should also announce termination in κ. Unlike τ, however, κ is a failure free execution of the system. Since the termination detection algorithm works correctly for κ, the underlying computation has actually terminated in κ when p_t announces termination. This in turn implies that the underlying computation has terminated in τ as well when p_t announces termination. $\square$

The strong completeness property of a perfect failure detector ensures that an instance of $\mathcal{A}$ is eventually initiated involving only the set of *correct* processes. This intuitively ensures the liveness of $\mathcal{B}$. Now, when $\mathcal{A}$ is restarted, a mechanism is needed to deal with application messages that were sent before $\mathcal{A}$ is restarted but are received after $\mathcal{A}$ has been restarted. Such application messages are referred to as stale or *old application messages*. The current instance of $\mathcal{A}$ may not be able to handle an old application message correctly. One simple approach is to "hide" an old application message from the current instance of $\mathcal{A}$ and deliver it directly to the underlying distributed computation. However, on receiving an old application message, if the destination process changes its state from passive to active, then, to the current instance of $\mathcal{A}$, it would appear as if the process became active spontaneously. This violates one of the four rules of the distributed computation. Clearly, the current instance of $\mathcal{A}$ may not work correctly in the presence of old application messages and therefore *cannot be directly* used to detect termination of the underlying computation.

3.1.3. Handling old application messages using a secondary computation

We use the following approach to deal with old application messages. We *superimpose* another computation on top of the underlying computation. We refer to the superimposed computation as the *secondary computation* and to the underlying computation as the *primary computation*. As far as live processes are concerned, the secondary computation is almost identical to the primary computation except possibly in the beginning. Whenever a process crashes and all live processes agree on the set of failed processes, we *simulate a new instance of the secondary computation* in the subsystem induced by the set of operational processes. The processes in the subsystem are referred to as the *base set* of the simulated secondary computation. We then use a *new instance of the fault-sensitive termination detection algorithm* to detect termination of the secondary computation. The older instances of the secondary computation and the fault-sensitive termination detection algorithm are simply aborted. We maintain the following invariants. First, if the secondary computation has classically terminated then the primary computation has classically terminated as well. Second, if the primary computation has classically terminated, then the secondary computation classically terminates eventually. Note that an operational process starts new instances of the secondary computation and the fault-sensitive termination detection algorithm *at the same time*.

We now describe the behavior of a live process with respect to the secondary computation. Intuitively, a process stays active with respect to the secondary computation at least until it knows that it cannot receive any old application message (from another live process) in the future. Consider a safe subset of processes Q. Suppose an instance of the secondary computation is initiated in the subsystem induced by Q. A process $p_i \in Q$ is passive with respect to the current instance of the secondary computation if both of the following conditions hold:

1. it is passive with respect to the primary computation, and
2. it knows that there is no old application message in transit towards it from any process in Q.

An old application message is delivered directly to the primary computation and is hidden from the current instance of the secondary computation as well as the current instance of the fault-sensitive termination detection algorithm. Specifically, only those application messages that are sent by the current instance of the secondary computation are tracked by the corresponding instance of the fault-sensitive termination detection algorithm. (In other words, all application messages are exchanged *through the current instance* of the termination detection algorithm except for old application messages.) It can be verified that the secondary computation is "legal" in the sense that it satisfies all the four rules of the distributed computation. Therefore the fault-sensitive termination detection algorithm $\mathcal{A}$ can be safely used to detect (classical) termination of the secondary computation even in the presence of old application messages. First, we show that, to detect termination of the primary computation, it is safe to detect termination of the secondary computation.

Theorem 3. *Consider a secondary computation initiated in the subsystem induced by a safe subset of processes Q. Then, if the secondary computation has classically terminated with respect to Q, then the primary computation has classically terminated with respect to Q.*

Proof. Assume that the secondary computation has classically terminated with respect to Q. Therefore all processes in Q are passive with respect to the secondary computation and no channel between processes in Q contains an application message belonging to the current instance of the secondary computation. This, in turn, implies that all processes in Q are passive with respect to the primary computation and no channel between processes in Q contains an application message belonging to the current or an older instance of the secondary computation. Moreover, since all processes in Q are passive, no process in Q has crashed, which implies that no new instance of the secondary computation has been started. Therefore the primary computation has classically terminated with respect to Q. □

Next, we prove that, to detect termination of the primary computation, it is sufficient to detect the termination of the secondary computation under certain conditions.

Theorem 4. *Consider a secondary computation initiated in the subsystem induced by a safe subset of processes Q. Assume that the primary computation has classically terminated with respect to Q and each process in Q eventually learns that there are no old application messages in transit towards it sent by other processes in Q. If all processes in Q stay operational, then the secondary computation eventually classically terminates with respect to Q.*

Proof. Assume that the primary computation has classically terminated with respect to Q and each process in Q eventually learns that there are no old application messages in transit towards it sent by other processes in Q. Clearly, since no process in Q crashes, all processes in Q eventually turn passive with respect to the secondary computation initiated on Q. Further, none of the channels among processes in Q contains an application message belonging to the secondary computation initiated on Q. Thus the secondary computation eventually classically terminates with respect to Q. □

We next describe how to ensure that all operational processes agree on the set of failed processes before restarting the secondary computation and the fault-sensitive termination detection algorithm. Later, we describe how to ascertain that there are no relevant old application messages in transit. There are many ways for a live process to determine the number of old application messages that are in transit towards it from other live processes. We describe one such mechanism in Section 3.1.5. Another mechanism is described in Section 3.5. We assume that both application and control messages are piggybacked with the complement of the base set of the current instance (of the secondary computation in progress), which can be used to identify the specific instance of the secondary computation. We refer to this complement set as *instance identifier*.

3.1.4. Achieving agreement on the set of failed processes

Whenever a crash is detected, one of the live processes is chosen to act as the *coordinator*. Specifically, the process with the smallest identifier among all live processes acts as the coordinator. Every process, on detecting a new failure, sends a NOTIFY message to the coordinator containing the set of all processes that it knows have failed. The coordinator maintains, for each operational process p_i, processes that have failed according to p_i. On determining that all operational processes agree on the set of failed processes, the coordinator sends a RESTART message to each operational process. A RESTART message instructs a process to initiate a new instance of the secondary computation on the appropriate set of processes and also to start a new instance of the fault-sensitive termination detection algorithm to detect its termination.

It is possible that, before receiving the RESTART message for a new instance, a process receives an application message or some other control message that is sent by a more recent instance of the secondary computation than the one currently in progress at that process. In that case, before processing the message received, it behaves as if it has also received a RESTART message and acts accordingly.

The strong accuracy property of a perfect failure detector ensures that no operational process is erroneously suspected by other operational processes to have crashed. Without this property, the new instance of the secondary computation and the fault-tolerant termination detection algorithm may not involve all operational processes. This may result in false detection of termination, especially if the processes that have been "left out" are active.

3.1.5. Tracking old application messages

A process stays active with respect to the current instance of the secondary computation at least until it knows that it cannot receive any old application message from one of the processes in the relevant subsystem. To that end, each process maintains a count of the number of application messages it has sent to each process so far and, also, a count of the number of application messages it has received from each process so far.

A process, on starting a new instance of the secondary computation, sends an OUTSTATE message to the coordinator; the message contains the number of application messages it sent to each process just before restarting the secondary computation. The coordinator, on receiving an OUTSTATE message from every operational process, sends an INSTATE message to all live processes. An INSTATE message sent to process p_i contains the number of application messages that each process has sent to p_i before starting the current instance of the secondary computation. This information can be easily computed by the coordinator after it has received an OUTSTATE message from all live processes.

Clearly, once a process has received an INSTATE message from the coordinator, it can determine how many old application messages are in transit towards it and wait until it has received all those messages before becoming passive for the first time with respect to the current instance of the secondary computation.

3.1.6. *Reducing the message-size complexity*

Observe that all messages except for NOTIFY messages carry an instance identifier that basically corresponds to the set of processes that all operational processes have *agreed on* to have crashed when new instances of the secondary computation and the fault-sensitive termination detection algorithm are started. This instance identifier is used by the receiving process for two purposes. First, it is used to identify the instance of the secondary computation to which the message belongs (that is, older, current or newer). Second, if the receiving process decides to restart the secondary computation (along with the fault-sensitive termination detection algorithm), then it is used to determine the base set of the new instances.

To determine whether a message belongs to an older, the current or a newer instance of the secondary computation, it is sufficient to compare the cardinality of the set of failed processes comprising the instance identifier. We do not need to compare the sets themselves. For example, instead of using $\{p_i, p_j\}$ as an instance identifier, we use its cardinality, namely two. Now, as far as determining the base set is concerned, we proceed as follows. Each process keeps track of the *order* in which it detects process crashes. Suppose a process p_i currently executing an instance x of the secondary computation receives a message belonging to a newer instance y, that is, $x < y$. To determine the base set for instance y, p_i determines the *first y processes* it detected to have crashed, which we claim should be same as the base set for instance y of the secondary computation. Clearly, a process in the system can start instance y of the secondary computation only after receiving a RESTART message for that instance from the coordinator. This implies that, when the coordinator sent RESTART message for instance y to all processes, all operational processes, say Q, agreed on the set of failed processes and, moreover, the set contained exactly y processes. Note that the set Q should include p_i because of strong accuracy property of a perfect failure detector. This, in turn, implies that, for all operational processes, the set of first y processes they detected to have crashed should be identical (although their crashes may have been detected in different orders by different processes).

Note that reducing the size of termination-detection messages is significant because, as shown in [6], as many as M termination-detection messages may be exchanged in the worst-case. On the other hand, the number of failure-recovery messages generated is bounded by $O(nf)$.

3.2. *A formal description of the transformation*

A formal description of the transformation is given in Figs. 1–4. The transformation is described using twelve atomic actions. Actions A0–A8 are executed by every process. Action A0 initializes the state of a process, and also starts the first instances of the secondary computation $\mathcal{SC}$ and the fault-sensitive termination detection algorithm $\mathcal{A}$ by invoking action A8. Actions B0–B2 are executed by a process when it becomes the coordinator. Action B0 initializes the state of a process on becoming the coordinator for the first time. Actions A1, A2 and A3 describe the hooks that interface the underlying computation with the termination detection algorithm: Action A1 is invoked when an application message is sent, action A2 is invoked when an application message is received, and action A3 is invoked when a control message is received. Note that every application message sent by the underlying computation is first intercepted by the current instance of $\mathcal{A}$, which may piggyback some control data on the message. It is then intercepted by the transformation (action A1) which piggybacks the instance identifier on the message. Conversely, every application and control message received is first intercepted by the transformation (actions A2 and A3) which strips off the instance identifier from the message and hands it over to either the underlying computation directly (if an old application message) or the current instance of $\mathcal{A}$ (otherwise). Action A4 is invoked when the current instance of $\mathcal{A}$ announces termination.

Actions A5–A8 handle process crashes. Action A5 is invoked whenever crash of a process is detected by the failure detector. As part of executing action A5, a new coordinator is selected (if required) and incoming channel with the crashed process is frozen. Action A6 is invoked whenever a RESTART message is received from the coordinator instructing the process to start new instances of the $\mathcal{SC}$ and $\mathcal{A}$. Action A7 is invoked on receiving an INSTATE message from the coordinator which carries information about the number of old application messages that have been sent to the process from other live processes so far.

Actions B1 and B2 are executed by a process when it is acting as the coordinator. Action B1 is invoked when the coordinator receives a NOTIFY message from a process. As part of executing B1, the coordinator checks if all processes have reached an agreement on the set of failed processes. If yes, it instructs all currently operational processes to start new instances of $\mathcal{SC}$ and $\mathcal{A}$ by broadcasting a RESTART message. Finally, action B2 is invoked when the coordinator receives an OUTSTATE message. Once it has received an OUTSTATE message from all live processes, it computes the number of old application messages in transit towards each live process from other live processes. It then sends this information to each live process by sending an INSTATE message to that process.

```
Transformation for process $p_i$:

Variables:
    $failed_i$: set of processes that have failed;
    $coordinator_i$: process acting as the coordinator;
    $current_i$: the current instance of the secondary computation;
    $view_i$: set of processes involved in the current instance of the secondary computation;

    $sent_i$: vector $[1..n]$ of number of application messages that have been sent to each process
            in the current instance so far;
    $received_i$: vector $[1..n]$ of number of application messages that have been received from each process
            so far that belong to the current instance;
    $oldSent_i$: vector $[1..n]$ of number of old application messages that were sent to each process in all
            previous instances combined;
    $oldReceived_i$: vector $[1..n]$ of total number of old application messages that have been
            received from each process so far;
    $allKnown_i$: whether $p_i$ knows the number of old application messages sent to it by other live processes;
            // $othersOldSent_i$ has a valid value only if $allKnown_i$ is true
    $othersOldSent_i$: vector $[1..n]$ of total number of old application messages that were sent to process $p_i$
            by each process;
            // $othersOldSent_i[j] - oldReceived_i[j]$ captures the number of old application messages
            // sent by process $p_j$ in transit towards process $p_i$

Notation:
    $\mathcal{A}$: fault-sensitive termination detection algorithm;
    $\mathcal{SC}$: secondary computation;
        process $p_i$ is passive in $\mathcal{SC}$ if and only if:
            (1) $p_i$ is passive in the underlying computation,
            (2) $allKnown_i$ is set to true, and
            (3) for each $p_k \in view_i$, $othersOldSent_i[k] = oldReceived_i[k]$;

(A0) Initial action:
    // initialize all variables
    $current_i := 0$;
    $failed_i := \emptyset$;
    $view_i := P$;
    $\forall k : sent_i[k] := 0$;
    $\forall k : received_i[k] := 0$;
    $\forall k : oldSent_i[k] := 0$:
    $\forall k : oldReceived_i[k] := 0$;
    $\forall k : othersOldSent_i[k] := 0$;
    $allKnown_i :=$ true;
    call startNewInstance($current_i$);

(A1) On sending an application message $m$ to process $p_j$:
    ++$sent_i[j]$;
    // piggyback the instance identifier on the message and send it to the process
    send message $m(current_i)$ to process $p_j$;
```

Fig. 1. Transforming a fault-sensitive termination detection algorithm into a fault-tolerant termination detection algorithm.

3.3. *Proof of correctness*

We now prove that our transformation produces an algorithm $\mathcal{B}$ that solves the effective-termination detection problem given that $\mathcal{A}$ is a correct fault-sensitive algorithm for solving the classical termination detection problem. The following proposition can be easily verified:

Proposition 5. *Whenever an instance of $\mathcal{A}$ is initiated on a process set Q, all processes in $P \setminus Q$ have in fact crashed and all channels from processes in $P \setminus Q$ to Q have been frozen.*

First, we prove the safety property.

```
Transformation for process $p_i$ (continued):

(A2) On receiving an application message $m(instance)$ from process $p_j$:
    if ($instance < current_i$) then
        // it is an old application message
        ++$oldReceived_i[j]$;
        deliver $m$ to the underlying computation after removing any control data piggybacked on $m$;
        // may have become passive with respect to the secondary computation
    else
        if ($current_i < instance$) then
            // process $p_j$ has already started a new instance of the secondary computation
            call startNewInstance($instance$);
        endif;
        ++$received_i[j]$;
        // tell the fault-sensitive termination detection algorithm about the application message
        inform $\mathcal{A}(current_i)$ that the application message $m$ has been received from process $p_j$;
    endif;

(A3) On receiving a control message $m(instance)$ from process $p_j$:
    if ($current_i \leq instance$) then
        if ($current_i < instance$) then
            // process $p_i$ has already started a new instance of the secondary computation
            call startNewInstance($instance$);
        endif;
        // tell the fault-sensitive termination detection algorithm about the control message
        inform $\mathcal{A}(current_i)$ that the control message $m$ has been received from process $p_j$;
    endif;

(A4) On the current instance of $\mathcal{A}$ announcing termination:
    announce termination;

(A5) On detecting the failure of process $p_j$:
    // update the list of failed processes
    $failed_i := failed_i \cup \{p_j\}$;
    // select a new coordinator if required
    $coordinator_i := \min\{p_k \mid p_k \in P$ and $p_k \notin failed_i\}$;
    send NOTIFY($failed_i$) message to $coordinator_i$;
    // all subsequent messages received from process $p_j$ will be dropped
    freeze the incoming channel from process $p_j$;

(A6) On receiving RESTART($instance$) message from process $p_j$:
    if ($current_i < instance$) then
        // start a new instance of the secondary computation and
        // the fault-sensitive termination detection algorithm
        call startNewInstance($instance$);
    endif;
```

Fig. 2. Transforming a fault-sensitive termination detection algorithm into a fault-tolerant termination detection algorithm (continued).

Theorem 6 (*Safety Property*). *If $\mathcal{B}$ announces termination, then the underlying computation has effectively terminated.*

Proof. Assume that $\mathcal{B}$ announces termination. This implies that some instance of $\mathcal{A}$ detected classical termination of the corresponding instance of the secondary computation run by some subset Q of processes. From Theorem 3, it follows that the underlying computation has also classically terminated with respect to Q. Finally, from Theorem 1, it follows that the underlying computation has effectively terminated with respect to P. □

Next, we show that $\mathcal{B}$ is live. That is,

Theorem 7 (*Liveness Property*). *Once the underlying computation effectively terminates, $\mathcal{B}$ eventually announces termination.*

```
Transformation for process $p_i$ (continued):

(A7) On receiving INSTATE($instance$, $othersOldSent$) message from process $p_j$:
    if ($instance = current_i$) then
        // can now initialize $othersOldsent_i$
        $othersOldSent_i := othersOldSent$;
        $allKnown_i :=$ true;
        // may have become passive with respect to the secondary computation
    endif;

(A8) On invocation of startNewInstance($instance$):
    abort $\mathcal{A}(current_i)$ and $\mathcal{SC}(current_i)$, if any;
    $current_i := instance$;
    let $failedSet_i$ denote the set of first $instance$ processes that process $p_i$ suspected to have crashed
    $view_i := P \setminus failedSet_i$;
    $\forall k \in view_i : oldSent_i[k] := oldSent_i[k] + sent_i[k]$;
    $\forall k \in view_i : sent_i[k] := 0$;
    $\forall k \in view_i : oldReceived_i[k] := oldReceived_i[k] + received_i[k]$;
    $\forall k \in view_i : received_i[k] := 0$;
    if ($current_i > 0$) then
        $allKnown_i :=$ false;
        send OUTSTATE($current_i$,$oldSent_i$) message to the coordinator;
    endif;
    // note that $failed_i$ should contain at least $instance$ processes
    start a new instance of $\mathcal{SC}$ and $\mathcal{A}$ on $view_i$:
```

Fig. 3. Transforming a fault-sensitive termination detection algorithm into a fault-tolerant termination detection algorithm (continued).

Proof. Assume that the underlying computation is effectively terminated and consider the point in time when the last process crashes. Our algorithm ensures that eventually a new instance of the secondary computation is initiated on the set Q of remaining live processes. Further, each operational process eventually learns, via an INSTATE message, the number of old application messages in transit towards it. Since the underlying computation has effectively terminated, from Theorem 1, it follows that the underlying computation has classically terminated with respect to Q. Further, using Proposition 5 and Theorem 4, it implies that the secondary computation initiated on Q classically terminates eventually. As a result, the corresponding instance of $\mathcal{A}$ eventually announces termination of the secondary computation on Q. □

3.4. The complexity analysis

Let $\mu(n, M)$ and $\delta(n, M)$ denote the message complexity and detection latency, respectively, of $\mathcal{A}$ when the system has n processes and the underlying computation exchanges M application messages. To compute the message complexity of $\mathcal{B}$, we assume that $\mu(n, M)$ satisfies the following constraint for $k \geq 1$:

$$\sum_{i=1}^{k} \mu(n, M_i) \leq \mu\left(n, \sum_{i=1}^{k} M_i\right) + (k-1)\mu(n, 0). \qquad (1)$$

For all existing termination detection algorithms that we are aware of, $\mu(n, M)$ is of the form $a + b \cdot M$, where a and b are some system dependent constants. It can be verified that the above inequality indeed holds if $\mu(n, M)$ is of the above form. In fact, it can be verified that the inequality holds as long as $\mu(n, M)$ does not contain any term that is *sub-linear* in M except for a constant (*e.g.*, $\sqrt{M}$, $\log M$).

Let f denote the actual number of processes that fail during an execution of $\mathcal{B}$. We next prove a lemma that is useful in analyzing the message complexity and detection latency.

Lemma 8. *The number of times $\mathcal{A}$ is restarted is bounded by f.*

Proof. A new instance of $\mathcal{A}$ is started only when a new failure occurs and, moreover, all operational processes have detected the failure. Since at most f processes can fail, $\mathcal{A}$ can be restarted at most f times. □

We categorize control messages into two groups. The first group consists of control messages exchanged by different instances of $\mathcal{A}$. The second group consists of control messages exchanged as a result of process crash, namely NOTIFY, RESTART, OUTSTATE and INSTATE. We refer to the messages in the first group as *termination-detection messages* and to the messages in the second group as *failure-recovery messages*.

Theorem 9 (*Message Complexity*). *The message complexity of $\mathcal{B}$ is $\mu(n, M) + O(f(n + \mu(n, 0)))$.*

```
Actions when process $p_i$ becomes the coordinator:

Variables:
    $othersFailed_i$: vector $[1..n]$ of set of failed processes according to each process;
    $allFailed_i$: set of processes suspected by at least one process;
    $instance_i$: the current instance of the secondary computation;
    $toReceive_i$: number of OUTSTATE messages still to be received;
    $outState_i$: vector $[1..n][1..n]$ of the number of old application messages that a process has sent
                  to other processes;
                  // $outState_i[k][l]$ denotes the number of old application messages that process $p_k$ has
                  // sent to process $p_l$
    $instate_i$: vector $[1..n][1..n]$ of the number of old application messages that other processes have sent
                 to a process;
                 // $inState_i[k][l]$ denotes the number of old application messages that have been sent to
                 // process $p_k$ by process $p_l$

(B0) On becoming the coordinator for the first time:
    // initialize all variables
    $\forall k : k \neq i : othersFailed_i[k] := \emptyset$;
    $othersFailed_i[i] := failed_i$;
    $instance_i := |failed_i|$;
    $allFailed_i := failed_i$;

(B1) On receiving NOTIFY(rcv_failed) message from process $p_j$:
    // is it a new notification message?
    if $(othersFailed_i[j] \subset$ rcv_failed$)$ then
        $othersFailed_i[j] :=$ rcv_failed;
        $allFailed_i := allFailed_i \cup$ rcv_failed;
        // do all operational processes agree on the set of failed processes?
        if $\langle \forall k : p_k \in P \setminus allFailed_i : othersFailed_i[k] = allFailed_i \rangle$ then
            $instance_i := |allFailed_i|$;
            send RESTART($instance_i$) message to each process in $P \setminus allFailed_i$;
            // initialize the number of OUTSTATE messages that are expected to be received
            $toReceive_i := n - |allFailed_i|$;
        endif;
    endif;

(B2) On receiving OUTSTATE($instance$,$oldSent$) message from process $p_j$:
    if $(instance = |allFailed_i|)$ then
        $--toReceive_i$;
        $outState_i[j] := oldSent$;
        // have all OUTSTATE messages been received?
        if $(toReceive_i = 0)$ then
            for each $k$ such that $p_k \in P \setminus allFailed_i$ do
                // compute the number of old application messages sent to process $p_k$
                $\forall l : p_l \in P \setminus allFailed_i : inState_i[k][l] := outState_i[l][k]$;
                send INSTATE($instance_i$,$inState_i[k]$) message to process $p_k$;
            endfor;
        endif;
    endif;
```

Fig. 4. Transforming a fault-sensitive termination detection algorithm into a fault-tolerant termination detection algorithm (continued).

Proof. Let M_i denote the number of application messages associated with the ith instance of $\mathcal{A}$. From Lemma 8, there are at most $f+1$ instances of $\mathcal{A}$. Therefore,

$$\sum_{i=1}^{f+1} M_i = M.$$

From (1), the number of termination-detection messages is given by:

$$\sum_{i=1}^{f+1} \mu(n, M_i) \leq \mu\left(n, \sum_{i=1}^{f+1} M_i\right) + f\mu(n,0)$$
$$= \mu(n, M) + f\mu(n,0).$$

Also, the number of failure-recovery messages is at most $4n$ per failure (n NOTIFY, n RESTART, n OUTSTATE and n INSTATE). □

We now bound the detection latency of $\mathcal{B}$. To compute detection latency in an asynchronous distributed system, it is typically assumed that message delay is at most one time unit. Moreover, we assume that the failure detection delay is bounded by one time unit as well.

Theorem 10 (*Detection Latency*). *The detection latency of $\mathcal{B}$ is given by $\delta(n, M) + O(f\delta(n, 0))$.*

Proof. Assume that the underlying computation has terminated. The worst-case scenario occurs when a process crashes just before the current instance of $\mathcal{A}$ is able to detect termination. Clearly, when a process fails, a new instance of the secondary computation is started on all operational processes within $O(1)$ time units assuming that there are no more failures—one time unit for failure detection, one time unit for the coordinator to receive all NOTIFY messages and one time unit for all live processes to receive RESTART messages. Once an instance of the secondary computation is initiated, it terminates with $O(1)$ time units as soon as every live process has received an INSTATE message from the coordinator. Once an instance of the secondary computation terminates, its termination is detected within $O(\delta(n, 0))$ time units. Note that $\delta(n, 0) = \Omega(1)$. Therefore, after a process fails its termination is detected within $O(\delta(n, 0))$ time units unless some other process has failed. It can be proved by induction that the termination detection can be delayed by only $O(f\delta(n, 0))$ time units. □

In general, if message delay is bounded by d_m time units and failure detection latency is bounded by d_f time units, then the termination detection latency of our algorithm is bounded by $\delta(n, M) + O(f(\delta(n, 0) + d_m + d_f))$ time units. (Note that $\delta(n, 0)$ is also a function of d_m in this case.) We next bound the message-size complexity of $\mathcal{B}$. Let $\alpha(n, M)$ and $\beta(n, M)$ denote the application and control message-size complexity, respectively, of $\mathcal{A}$ when the system has n processes and the underlying computation exchanges M application messages.

Theorem 11 (*Application Message-Size Complexity*). *The application message-size complexity of $\mathcal{B}$ is $\alpha(n, M) + \log(f + 1)$.*

Proof. The additional information piggybacked on an application message is the number of failed processes, which is bounded by $\log(f + 1)$. □

Finally, we bound the control message-size complexity of $\mathcal{B}$.

Theorem 12 (*Control Message-Size Complexity*). *The control message-size complexity of $\mathcal{B}$ for termination-detection messages is given by $\beta(n, M) + \log(f + 1)$ and for failure-recovery messages is given by $O(f \log n + n \log M)$.*

Proof. The additional information piggybacked on all termination-detection messages is the number of failed processes, which is bounded by $\log(f + 1)$. A failure-recovery message contains the following information: (1) set of failed processes, and (2) count of the number of application messages sent so far to each process. The overhead due to the two is bounded by $O(f \log n)$ and $O(n \log M)$, respectively. □

3.5. Optimizing for the special case: When all application messages are acknowledged

Our basic transformation makes minimal assumptions about the termination detection algorithm. When the fault-sensitive termination detection algorithm is acknowledgment-based, it is possible to reduce the size of application and control messages significantly without affecting the message complexity and detection latency at all. In the basic transformation, failure recovery messages, specifically INSTATE and OUTSTATE messages, have to carry a vector of size n, where each entry in the vector is bounded by $O(\log M)$. Vectors exchanged through these messages help processes determine whether there are any old application messages in transit towards them. In case the underlying fault-sensitive termination detection algorithm uses acknowledgment messages to test for emptiness of channels, we can avoid exchanging vectors altogether. The main idea is as follows. A process, on restarting the termination detection algorithm, waits until all its old application messages sent to currently live processes have been acknowledged. It then sends a QUIESCENT message to the coordinator. Once the coordinator has received a QUIESCENT message from all operational processes, it knows that there are no old application messages in transit towards any operational process and sends an ALL_QUIESCENT message to all operational processes. On receiving the ALL_QUIESCENT message, a process can infer that it will not receive any old application messages hereafter and the secondary computation running at the process becomes identical to its primary computation. Observe that, for the modification to work correctly, all application messages have to be acknowledged. Specifically, a process has to send an acknowledgment message even for an old application message.

With the above modification, the size of failure-recovery messages decreases from $O(f \log n + n \log M)$ to $O(f \log n)$. All other complexity measures remain the same. Further, our transformation satisfies another desirable property, namely size of all messages actually becomes *bounded*.

3.6. Discussion

If return-flush primitive is available, then our transformation can be easily adapted to detect strict termination as follows. A process, on detecting the crash of another process, first executes return-flush primitive to flush all messages in transit towards it from the crashed process before sending the NOTIFY message to the coordinator. Our transformation can also be generalized to derive a fault-tolerant termination detection algorithm for an arbitrary communication topology as follows. Whenever a process crashes, remaining processes in the system first elect a new leader (or a coordinator) using Awerbuch's spanning tree construction algorithm [2]. The newly elected leader is then responsible for starting a new instance of the secondary computation and the fault-sensitive termination detection algorithm. Details of the transformation for arbitrary communication topology can be found elsewhere [28].

4. Establishing the lower bound on message-complexity

Observe that if the computation exchanges $\Omega(nf)$ application messages, then any termination detection algorithm – fault-sensitive or fault-tolerant – has to exchange $\Omega(nf)$ control messages in the worst-case [6]. We, however, show that if the communication topology is fully connected, then, under certain realistic assumption, any fault-tolerant termination detection algorithm can be forced to exchange $\Omega(nf)$ control messages even if the underlying computation *does not generate any application messages*. Specifically, we assume that the termination detection algorithm is non-inhibitory in nature as defined next:

Definition 1 (*Non-Inhibitory Property*). A termination detection algorithm is said be *non-inhibitory* if it satisfies the following property: whenever the computation at process p_i wants to send an application message to the computation at process p_j and neither p_i nor p_j crashes, the message can be delivered to the computation at p_j within a finite number of steps of p_i and p_j.

Note that if a termination detection algorithm immediately delivers an application message on arrival to the underlying computation, then it trivially satisfies the non-inhibitory property. In general, a termination detection algorithm satisfies the non-inhibitory property only if it delays delivery of an application message to the computation because of messages sent *before* the application message was sent. In almost all termination detection algorithms – fault-sensitive as well as fault-tolerant – that we are aware of, the termination detection algorithm transmits the application message immediately possibly after piggybacking some control information on it. Further, an application message is delivered to the computation immediately on arrival possibly after processing the control information piggybacked on the message. Therefore most termination detection algorithms in the literature are non-inhibitory in nature. A termination detection algorithm that does not satisfy the non-inhibitory property is the algorithm by Francez [13] which uses partial freezing of the computation to detect termination.

The main idea behind the lower bound proof is as follows. Consider an initial state of the system in which all but one processes are passive. The one remaining process may be passive or active—the exact state is not known to other processes. We show that if this process fails, then there is an execution of the system in which all operational processes except one send at least one control message. Moreover, after the control messages have been sent, the system reaches a state in which these operational processes, which sent control messages, do not know the exact state of the one remaining operational process. By repeating this argument, the desired lower bound can be proved. To formally prove the lower bound, we first define some notation.

4.1. System states, executions and sub-executions

A *state of a distributed system* can be modeled using the *initial state* of the system and the *sequence of events* that have been executed so far. An initial state of the system basically specifies the state of each process with respect to the computation and the termination detection algorithm. The state of the system obtained after executing events in σ starting from the initial state I is denoted by $\langle I, \sigma\rangle$. We use ϵ to denote an empty sequence of events. Trivially, $I = \langle I, \epsilon\rangle$. We use the terms "sequence of events" and "system execution" interchangeably.

For two system executions σ and τ, we use $\sigma \circ \tau$ to denote the system execution obtained after appending τ to σ. For a system state $\langle I, \sigma\rangle$ and a system execution τ, we use $\langle I, \sigma\rangle \models \tau$ to denote the fact that τ is a *valid* execution of the system starting from state $\langle I, \sigma\rangle$, that is, it is possible to extend the execution σ to the execution $\sigma \circ \tau$ by executing events in τ.

For a system execution σ and a process p_i, $\sigma\langle p_i\rangle$ refers to the sub-execution of σ that contains those events of σ that belong to p_i. Also, let $\sigma\langle \bar{p}_i\rangle$ denote the sub-execution of σ that contains those events of σ that do not belong to p_i. Note that an event in σ either belongs to $\sigma\langle p_i\rangle$ or $\sigma\langle \bar{p}_i\rangle$. For a channel from a process p_i to a process p_j, $\sigma\langle p_i \to p_j\rangle$ denotes the sub-execution of σ that contains those events of σ that affect the state of the channel. Observe that $\sigma\langle p_i \to p_j\rangle$ basically consists of all events of σ that either involve sending of a message by p_i to p_j or receiving of a message by p_j from p_i. The following proposition follows from the "asynchronous nature" of the system.

Proposition 13. *Consider a system state $\langle I, \sigma\rangle$ and a system execution τ such that τ is a valid execution of the system starting from state $\langle I, \sigma\rangle$. If a process p_i does not send any message during τ, then $\tau\langle \bar{p}_i\rangle$ is also a valid execution of the system starting from state $\langle I, \sigma\rangle$. Formally,*

$$(\langle I, \sigma\rangle \models \tau) \wedge (p_i \text{ does not send any message during } \tau) \Rightarrow \langle I, \sigma\rangle \models \tau\langle \bar{p}_i\rangle.$$

Two events are causally related if they are related by the Lamport's happened-before relation [22]. Specifically, an event e *happened-before* an event f, if one of the following conditions holds:

1. events e and f are events on the same process and e occurred-before f in real-time, or
2. events e and f are send and receive events, respectively, of the same message, or
3. there exists an event g such that e happened-before g and g happened-before f.

If e happened-before f, then we say that f *causally depends* on e. Intuitively, if f causally depends on e, then it may not be possible to execute f until e has been executed. On the other hand, if f does not causally depend on e, then f can be executed even if e has not been executed. We use $\sigma^*\langle \bar{p}_i\rangle$ to denote the sub-execution of σ obtained after removing all events in $\sigma\langle p_i\rangle$ and events that causally depend on some event in $\sigma\langle p_i\rangle$ from σ. The next proposition is a generalization of the previous proposition.

Proposition 14. *Consider a system state $\langle I, \sigma\rangle$ and a finite system execution τ such that τ is a valid execution of the system starting from state $\langle I, \sigma\rangle$. For any process p_i, $\tau^*\langle \bar{p_i}\rangle$ is a valid execution of the system starting from state $\langle I, \sigma\rangle$. Formally,*

$$\langle I, \sigma\rangle \models \tau \Rightarrow \langle I, \sigma\rangle \models \tau^*\langle \bar{p_i}\rangle.$$

For a system execution σ, let crashed(σ) denote the set of processes that failed during σ. The future of a system state $\langle I, \sigma\rangle$ with respect to processes in Q is said to be *contained* in the future of a system state $\langle J, \tau\rangle$, denoted by $\langle I, \sigma\rangle \overset{Q}{\sqsubseteq} \langle J, \tau\rangle$, if the following conditions are satisfied:

- processes in Q have identical states in I and J,
- crashed(σ) = crashed(τ),
- for each process p_i in Q, $\sigma\langle p_i\rangle = \tau\langle p_i\rangle$, and
- for each channel from process p_j to process p_i with $p_i \in Q$, $\sigma\langle p_j \to p_i\rangle$ is a prefix of $\tau\langle p_j \to p_i\rangle$.

Intuitively, when the above conditions are satisfied, any execution of the system starting from state $\langle I, \sigma\rangle$, which only contains steps belonging to processes in Q, is also a valid execution of the system starting from state $\langle J, \tau\rangle$. Note that the converse may not hold. This is because some channels in $\langle J, \tau\rangle$ may contain messages that are not present in the corresponding channels in $\langle I, \sigma\rangle$. (These channels are from processes in $P \setminus Q$ to processes in Q.) Formally, a system execution σ is said to be *restricted to processes in* Q, abbreviated as Q-*restricted*, if it only contains events belonging to processes in Q. We have,

Proposition 15. *Consider two system states $\langle I, \sigma\rangle$ and $\langle J, \tau\rangle$, and a subset of processes Q such that $\langle I, \sigma\rangle \overset{Q}{\sqsubseteq} \langle J, \tau\rangle$. If κ is a Q-restricted execution of the system starting from $\langle I, \sigma\rangle$, then κ is also a valid execution of the system starting from $\langle J, \tau\rangle$. Formally,*

$$(\langle I, \sigma\rangle \overset{Q}{\sqsubseteq} \langle J, \tau\rangle) \wedge (\langle I, \sigma\rangle \models \kappa) \wedge (\kappa \textit{ is } Q\textit{-restricted}) \Rightarrow \langle J, \tau\rangle \models \kappa.$$

The above proposition follows from the fact that, in an asynchronous distributed system, a process only has "local knowledge" of the system and, therefore, only has limited ability to distinguish between various system states [6]. For a system state $\langle I, \sigma\rangle$, let active$\langle I, \sigma\rangle$ (respectively passive$\langle I, \sigma\rangle$) denote the set of active (respectively passive) processes in $\langle I, \sigma\rangle$. Let the crash event of process p_i be denoted by crash(p_i). We are now ready to prove the lower bound.

4.2. Lower bound proof

Our lower bound proof uses two special system states, namely elementary and critical. A system state $\langle I, \sigma\rangle$ is said to be *elementary* if all processes are passive in the initial state I. A system state is said to be *critical* if, in the state, exactly one operational process is active and no channel contains any application message.

Consider an elementary system state $\langle I, \sigma\rangle$ and a critical system state $\langle J, \tau\rangle$ that are "indistinguishable" to the processes that are passive in the two states. We show that, if the process that is active in the critical state fails, then a large number of passive processes have to send control messages before termination can be announced. Specifically, we show that if the active process of $\langle J, \tau\rangle$ crashes, then all processes in Q except possibly one have to send one or more control messages before some process can announce termination.

Lemma 16. *Consider two system states $\langle I, \sigma\rangle$ and $\langle J, \tau\rangle$ such that:*

- *$\langle I, \sigma\rangle$ is an elementary state,*
- *$\langle J, \tau\rangle$ is a critical state with active$\langle J, \tau\rangle = \{p_i\}$ and passive$\langle J, \tau\rangle = Q$, and*
- *$\langle I, \sigma\rangle \overset{Q}{\sqsubseteq} \langle J, \tau\rangle$.*

If $Q \neq \emptyset$, then any execution κ of the system starting from $\langle I, \sigma \circ crash(p_i)\rangle$ in which no process crashes and some process eventually announces termination satisfies the following:

- *κ is Q-restricted, and*
- *each process in Q except possibly one sends at least one control message during κ.*

Proof. Consider an execution κ of the system starting from $\langle I, \sigma \circ \text{crash}(p_i)\rangle$ in which no process crashes and some process, say p_t, eventually announces termination. Such an execution exists since the termination detection algorithm is live. Suppose there are two processes in Q, say p_j and p_k, that do not send any control message during κ. Clearly, either $p_j \neq p_t$ or $p_k \neq p_t$. Without loss of generality, assume the former. Let $R = Q \setminus \{p_j\}$. Consider the system execution $\rho = \kappa\langle \bar{p_j}\rangle$. Note that ρ is R-*restricted* (and therefore Q-*restricted*). Since p_j does not send any message (application or control) during κ, we can apply Proposition 13 and thus have:

$$\rho \text{ is a valid execution of the system starting from state } \langle I, \sigma \circ \text{crash}(p_i)\rangle. \tag{2}$$

Also, since p_t announces termination during κ and $\kappa\langle p_t\rangle = \rho\langle p_t\rangle$, we have,

$$p_t \text{ announces termination during } \rho. \tag{3}$$

Further, we have,

$$\begin{aligned}
& (\langle I, \sigma\rangle \overset{Q}{\sqsubseteq} \langle J, \tau\rangle) \wedge (\langle I, \sigma \circ \text{crash}(p_i)\rangle \models \rho) \\
\Rightarrow\ & \{\text{adding crash}(p_i) \text{ to } \sigma \text{ and } \tau\} \\
& (\langle I, \sigma \circ \text{crash}(p_i)\rangle \overset{Q}{\sqsubseteq} \langle J, \tau \circ \text{crash}(p_i)\rangle) \\
& \wedge(\langle I, \sigma \circ \text{crash}(p_i)\rangle \models \rho) \\
\Rightarrow\ & \{\rho \text{ is } Q\textit{-restricted} \text{ and using Proposition 15}\} \\
& \langle J, \tau \circ \text{crash}(p_i)\rangle \models \rho.
\end{aligned}$$

In other words,

ρ is a valid execution of the system starting from state
$\langle J, \tau \circ \text{crash}(p_i)\rangle$. (4)

Next, consider the system state $\langle J, \tau\rangle$. Suppose, in this state, the computation at p_i sends an application message to the computation at p_j. Using the non-inhibitory property of the termination detection algorithm, there exists a finite system execution λ consisting only of events belonging to p_i and p_j such that p_j receives the application message from p_i and becomes active. We have,

$$\begin{aligned}
&\langle J, \tau\rangle \overset{R}{\sqsubseteq} \langle J, \tau\rangle\\
\Rightarrow\ &\{\lambda \text{ is } \{p_i, p_j\}\text{-}\textit{restricted}, \text{crashed}(\lambda) = \emptyset,\\
&\ \text{and } R \cap \{p_i, p_j\} = \emptyset\}\\
&\langle J, \tau\rangle \overset{R}{\sqsubseteq} \langle J, \tau \circ \lambda\rangle\\
\Rightarrow\ &\{\text{adding crash}(p_i) \text{ to both } \tau \text{ and } \tau \circ \lambda\}\\
&\langle J, \tau \circ \text{crash}(p_i)\rangle \overset{R}{\sqsubseteq} \langle J, \tau \circ \lambda \circ \text{crash}(p_i)\rangle\\
\Rightarrow\ &\{\rho \text{ is } R\text{-}\textit{restricted}, \text{using (4) and Proposition 15}\}\\
&\langle J, \tau \circ \lambda \circ \text{crash}(p_i)\rangle \models \rho.
\end{aligned}$$

Note that the computation has not terminated in state $\langle J, \tau \circ \lambda \circ \text{crash}(p_i)\rangle$ because p_j is active. Therefore the computation has not terminated in state $\langle J, \tau \circ \lambda \circ \text{crash}(p_i) \circ \rho\rangle$ as well. However, using (3), p_t announces termination during ρ, which violates the safety property of the termination detection algorithm. Therefore it follows that each process in Q except possibly one sends at least one control message during κ. □

Next, we show that it is possible to go from a pair of "indistinguishable" elementary and critical system states to another pair of "indistinguishable" elementary and critical systems states while forcing the termination detection algorithm to exchange a relatively large number of control messages. The next lemma allows us to repeat the argument in the previous lemma by showing that it is possible to go from state $\langle I, \sigma\rangle$ (respectively $\langle J, \tau\rangle$) to state $\langle I, \sigma \circ \phi\rangle$) (respectively $\langle J, \sigma \circ \psi\rangle$) such that $\langle J, \tau \circ \psi\rangle$ is a critical state and $\langle I, \sigma \circ \phi\rangle \overset{R}{\sqsubseteq} \langle J, \tau \circ \psi\rangle$ where $R = \text{passive}\langle J, \tau \circ \psi\rangle$.

Lemma 17. *Consider two system states $\langle I, \sigma\rangle$ and $\langle J, \tau\rangle$ such that:*

- *$\langle I, \sigma\rangle$ is an elementary state,*
- *$\langle J, \tau\rangle$ is a critical state with active$\langle J, \tau\rangle = \{p_i\}$ and passive$\langle J, \tau\rangle = Q$, and*
- *$\langle I, \sigma\rangle \overset{Q}{\sqsubseteq} \langle J, \tau\rangle$.*

If $Q \neq \emptyset$, then there exists two system executions ϕ and ψ starting from states $\langle I, \sigma\rangle$ and $\langle J, \tau\rangle$, respectively, and a subset of processes R such that:

- *crashed(ϕ) = crashed$(\psi) = \{p_i\}$,*
- *$\langle J, \tau \circ \psi\rangle$ is a critical state with active$\langle J, \tau \circ \psi\rangle = \{p_j\}$ and passive$\langle J, \tau \circ \psi\rangle = R$,*
- *$\langle I, \sigma \circ \phi\rangle \overset{R}{\sqsubseteq} \langle J, \tau \circ \psi\rangle$, and*
- *at least $n - |\text{crashed}(\sigma \circ \phi)| - 1$ control messages are sent during ϕ.*

Proof. Consider any execution κ of the system starting from state $\langle I, \sigma \circ \text{crash}(p_i)\rangle$ in which no process crashes and some process eventually announces termination during κ. Using Lemma 16, κ is *Q-restricted* and all processes in Q except possibly one sends one or more control messages during κ. Let $S \subseteq Q$ be the set of processes that send at least one control message during κ. Clearly, $Q \setminus S$ contains at most one process. In case $Q \setminus S \neq \emptyset$, let p_j be the process in the singleton set $Q \setminus S$. On the other hand, if $Q \setminus S = \emptyset$, then we choose p_j to be a process in Q such that each process in $Q \setminus \{p_j\}$ sends its *first* control message without receiving a message that p_j sent during κ. Process p_j exists because otherwise it can be shown that the Lamport's happened-before relation contains a cycle—a contradiction. Let $R = Q \setminus \{p_j\}$. Consider the system execution $\rho = \kappa^*\langle \bar{p_j}\rangle$. Note that ρ is *R-restricted* (and therefore *Q-restricted*). Using Proposition 14, we have:

ρ is a valid execution of the system starting from state
$\langle I, \sigma \circ \text{crash}(p_i)\rangle$. (5)

Clearly, the manner in which p_j is chosen, we have,

$\langle \forall p_k : p_k \in R : p_k$ sends at least one control message during $\rho\rangle$. (6)

Also, we have,

$$\begin{aligned}
&(\langle I, \sigma\rangle \overset{Q}{\sqsubseteq} \langle J, \tau\rangle) \wedge (\langle I, \sigma \circ \text{crash}(p_i)\rangle \models \rho)\\
\Rightarrow\ &\{\text{adding crash}(p_i) \text{ to } \sigma \text{ and } \tau\}\\
&(\langle I, \sigma \circ \text{crash}(p_i)\rangle \overset{Q}{\sqsubseteq} \langle J, \tau \circ \text{crash}(p_i)\rangle)\\
&\wedge(\langle I, \sigma \circ \text{crash}(p_i)\rangle \models \rho)\\
\Rightarrow\ &\{\rho \text{ is } Q\text{-}\textit{restricted} \text{ and using Proposition 15}\}\\
&\langle J, \tau \circ \text{crash}(p_i)\rangle \models \rho.
\end{aligned}$$

In other words,

ρ is a valid execution of the system starting from state
$\langle J, \tau \circ \text{crash}(p_i)\rangle$. (7)

Next, consider the system state $\langle J, \tau\rangle$. Suppose, in this state, the computation at p_i sends an application message to the computation at p_j. Using the non-inhibitory property of the termination detection algorithm, there exists a finite system execution λ consisting only of events belonging to p_i and p_j such that p_j receives the application message from p_i and becomes active. We have,

$$\begin{aligned}
&\langle J, \tau\rangle \overset{R}{\sqsubseteq} \langle J, \tau\rangle\\
\Rightarrow\ &\{\lambda \text{ is } \{p_i, p_j\}\text{-}\textit{restricted}, \text{crashed}(\lambda) = \emptyset, \text{ and}\\
&\ R \cap \{p_i, p_j\} = \emptyset\}\\
&\langle J, \tau\rangle \overset{R}{\sqsubseteq} \langle J, \tau \circ \lambda\rangle\\
\Rightarrow\ &\{\text{adding crash}(p_i) \text{ to both } \tau \text{ and } \tau \circ \lambda\}\\
&\langle J, \tau \circ \text{crash}(p_i)\rangle \overset{R}{\sqsubseteq} \langle J, \tau \circ \lambda \circ \text{crash}(p_i)\rangle
\end{aligned}$$

$\Rightarrow$ {ρ is *R-restricted*, using (7) and Proposition 15}
$\langle J, \tau \circ \lambda \circ \text{crash}(p_i)\rangle \models \rho$.

The required system executions are given by $\phi = \text{crash}(p_i) \circ \rho$ and $\psi = \lambda \circ \text{crash}(p_i) \circ \rho$. Clearly, $\text{crashed}(\phi) = \text{crashed}(\psi) = \{p_i\}$. Also, in state $\langle J, \tau \circ \psi\rangle$, only p_j is active, all other operational processes are passive and no channel contains any application message. Therefore $\langle J, \tau \circ \psi\rangle$ is a critical state with $\text{active}\langle J, \tau \circ \psi\rangle = p_i$ and $\text{passive}\langle J, \tau \circ \psi\rangle = R$. It can be verified that $\langle I, \sigma \circ \phi\rangle \overset{R}{\sqsubseteq} \langle J, \tau \circ \psi\rangle$. Finally, since $R = Q \setminus \{p_j\} = (P \setminus \text{crashed}(\sigma \circ \phi)) \setminus \{p_j\}$, $|R| = n - |\text{crashed}(\sigma \circ \phi)| - 1$. Therefore at least $n - |\text{crashed}(\sigma \circ \phi)| - 1$ control messages are sent during ϕ. □

The lower bound can now be proved using the previous two lemmas.

Theorem 18 (*Lower Bound*). *Any termination detection algorithm must exchange $\Omega(nf)$ control messages in the worst case even if at most one process is active initially and the computation does not exchange any application messages, where n, with $n > 2$, is the number of processes in the system and f, with $f < n$, is the number of processes that fail during the execution.*

Proof. The proof is by construction. Consider two initial system states I and J such that all processes are passive in I and all but one are passive in J. Let $\sigma_0 = \epsilon$, $\tau_0 = \epsilon$ and $Q_0 = \text{passive}\langle J, \epsilon\rangle$. Clearly, both are possible initial states of the computation when at most one process can be active. Further, (1) $\langle I, \sigma_0\rangle$ is an elementary state, (2) $\langle J, \tau_0\rangle$ is a critical state with $\text{passive}\langle J, \tau_0\rangle = Q_0$, and (3) $\langle I, \sigma_0\rangle \overset{Q_0}{\sqsubseteq} \langle J, \tau_0\rangle$.

We repeatedly apply Lemma 17 starting from system states $\langle I, \sigma_0\rangle$ and $\langle J, \tau_0\rangle$. Consider the kth application of the lemma with $1 \le k \le f$. Let the two system executions obtained by applying the lemma be ϕ_k and ψ_k. Also, let $\sigma_k = \sigma_{k-1} \circ \phi_k$ and $\tau_k = \tau_{k-1} \circ \psi_k$. Since exactly one process crashes after each application of the lemma, exactly k processes have crashed during σ_k. Further, since at least $n - k - 1$ control messages are sent during ϕ_k, $\Omega(nk)$ control messages are sent during σ_k. Clearly, no application messages are exchanged during σ_k because the system has already terminated in the initial state I. Finally, (1) $\langle I, \sigma_k\rangle$ is an elementary state, (2) $\langle J, \tau_k\rangle$ is a critical state with $\text{passive}\langle J, \tau_k\rangle = Q_k$, and (3) $\langle I, \sigma_k\rangle \overset{Q_k}{\sqsubseteq} \langle J, \tau_k\rangle$. Therefore Lemma 17 can be applied again to system states $\langle I, \sigma_k\rangle$ and $\langle J, \tau_k\rangle$ provided $k < f$ which, in turn, implies that $k < n - 1$, that is, $Q_k \neq \emptyset$.

The lower bound is obtained by applying Lemma 17 up to f times. □

Note that the lower bound proof depends on Propositions 13–15. These propositions hold even if all channels are FIFO and a process can atomically multicast a message to several processes. Therefore the lower bound holds under these assumptions as well.

5. The weakest failure detector for termination detection

Failure detectors are not only an abstraction to yield information about the operational state of processes, they can also be regarded as *synchrony abstractions* since they are usually implemented using heartbeat messages and timeouts [23]. For example, an eventually perfect failure detector is strictly weaker than a perfect failure detector, and, therefore, can be implemented with weaker synchrony assumptions (namely those of *partial synchrony* [12] instead of full synchrony). Proving that a certain type of failure detector is *necessary* for solving a problem provides an insight into the minimal amount of synchrony needed to solve that problem. In this section, unless otherwise stated, "termination" refers to "effective-termination".

We now show that a perfect failure detector is necessary for solving the termination detection problem in a crash-prone distributed system. To that end, we transform an instance of any fault-tolerant termination detection algorithm into a perfect failure detector at one process p_i, that is, p_i is able to reliably detect process crashes. A full perfect failure detector can then be implemented by using n parallel instances of the transformation algorithm, one for each process p_i.

Assume that we are given an algorithm $\mathcal{A}$ that can detect termination of any computation involving an arbitrary set of processes even in the presence of process crashes. Recall again that the phrase "eventually, $\mathcal{A}$ announces termination" to mean that eventually some live process announces termination. To implement a perfect failure detector at process p_i, we set up n independent computations $\mathcal{C}_{ij}$, one for each process p_j. The computation $\mathcal{C}_{ij}$ consists of only two processes p_i and p_j. Initially, p_i is passive and p_j is active. Further, p_j never becomes passive and the computation does not exchange any application messages. By construction, $\mathcal{C}_{ij}$ terminates when and only when p_j crashes. An instance of the termination detection algorithm $\mathcal{A}$, say $\mathcal{A}_{ij}$, is used to detect termination of the computation $\mathcal{C}_{ij}$. We now emulate a perfect failure detector as follows: Whenever p_i detects termination of $\mathcal{C}_{ij}$, it starts suspecting p_j permanently. We now show that this algorithm implements a perfect failure detector at process p_i if $\mathcal{A}$ correctly solves the effective-termination detection problem.

First, consider the strong accuracy property, which says that a process is never suspected before it crashes. Assume that p_i suspects p_j. It follows from our transformation that the instance $\mathcal{A}_{ij}$ of the termination detection algorithm has announced termination of the computation $\mathcal{C}_{ij}$. Since $\mathcal{A}$ satisfies the safety property, $\mathcal{C}_{ij}$ has indeed terminated. This, in turn, implies that p_j has crashed.

Now, consider the strong completeness property, which says that eventually every crashed process is permanently suspected by every correct process. Assume that p_i is correct and p_j has crashed. Once p_j crashes, clearly, the termination condition holds for the computation $\mathcal{C}_{ij}$. Since $\mathcal{A}$ satisfies the liveness property, its instance $\mathcal{A}_{ij}$ eventually announces termination of $\mathcal{C}_{ij}$. This, in turn, implies that p_i eventually announces termination of $\mathcal{C}_{ij}$. From the construction, upon detecting termination of $\mathcal{C}_{ij}$, p_i starts suspecting p_j permanently. This concludes the proof.

Overall, this shows that if we can solve termination detection in a crash-prone distributed system, then we can also implement a perfect failure detector in such a system. In other words, a perfect failure detector is *necessary* for solving the effective-termination detection problem.

Remark 1. Our transformation for implementing a perfect failure detector using a fault-tolerant termination detection algorithm assumes that a process can stay active *indefinitely*. In case this is not acceptable, we provide the following alternative transformation. Specifically, the computation $\mathcal{C}_{ij}$ is now defined as follows. Initially, process p_j is active but process p_i is passive. While a process (p_i or p_j) is active, it sends an application message to the other process and becomes passive. Clearly, as the computation $\mathcal{C}_{ij}$ proceeds, p_i and p_j continuously make each other active – starting from p_j – until one of them crashes. It can be verified that if the instance $\mathcal{A}_{ij}$ announces termination at p_i, then p_j has crashed (strong accuracy). Further, once p_j crashes, the instance $\mathcal{A}_{ij}$ eventually announces termination at p_i if p_i is correct (strong completeness). □

The *weakest failure detector* for a problem is a failure detector that is necessary and sufficient to solve that problem. We show above that a perfect failure detector is necessary. Our transformation in Section 3 shows that a perfect failure detector is also sufficient. Combining the two, we can conclude that a perfect failure detector is the weakest failure detector for solving the effective-termination detection problem. The result holds as long as at least one process can crash and assuming that channels can be frozen. Therefore, it generalizes the result of Wu et al. [38], which shows that a failure detector has to be complete. Our result also further clarifies the relationship between the termination detection problem and the *consensus problem*: Wu et al. [38] show that *strict-termination* detection is harder than consensus because the former cannot be solved without the ability to flush incoming channels with failed processes. By relating termination detection to the failure detector hierarchy of Chandra and Toueg [3], our result has two interesting corollaries. First, even *effective-termination* detection is strictly harder than consensus in environments where a majority of processes remains correct. This follows from the result that in such cases the weakest failure detector for solving consensus (namely, eventually weak failure detector) is strictly weaker than a perfect failure detector [3]. Second, when any number of processes can crash (and we restrict ourselves to using only *realistic* failure detectors which cannot predict the future behavior of processes), effective-termination detection is actually equivalent to consensus [8].

By using the same line of argument as above, it can be shown that the perfect failure detector is also the weakest failure detector for *strict* termination detection if channels can be flushed instead of frozen. In such a setting and after detecting the crash of a neighbor, a process may invoke a *return-flush* or *fail-flush* [36,21] primitive on the connecting channel to make certain that the channel is empty. Does this mean that a perfect failure detector is the weakest failure detector for strict termination detection as well?

Interestingly, the answer is no. Clearly, the ability to flush a channel is necessary to solve strict termination detection. So it remains to compare the power of a flush primitive with the power of a freeze primitive. Freezing the channel can be implemented in asynchronous crash-prone systems very easily: incoming messages from the frozen channel are simply ignored. Finding out whether an incoming channel from a crashed process is empty requires some form of synchrony. Charron-Bost et al. [7,16] prove that, in an asynchronous crash-prone distributed system, it is impossible to reliably detect whether there still exists a message in transit on an incoming channel from a crashed process with any form of failure detector, even a perfect one. Intuitively, this is because a failure detector in the formal sense of Chandra and Toueg [3] is defined as a function of process failures, that is, a function of operational states of processes. It does not offer any information about the final state of a crashed process or the state of a communication channel. This in turn implies that it is impossible to implement a channel flush primitive using a failure detector [16,7]. This impossibility result carries over to strict termination detection, since such a primitive is necessary to solve it. Fortunately, strict termination detection still can be solved in a fully *synchronous* distributed system. If processes can crash, then implementing a synchronous system on top of an asynchronous system requires a *failure detection sequencer*—a device which is strictly stronger than a perfect failure detector [16]. With such a device, it is also possible to implement a flush primitive in an asynchronous distributed system.

6. Conclusions and future work

In this paper, we have presented a transformation using a perfect failure detector that can be used to convert any termination detection algorithm for a fully connected communication topology which has been designed for a failure-free environment into a termination detection algorithm that can tolerate process crashes. Our transformation does not impose any additional overhead on the system (besides that imposed by the underlying termination detection algorithm) if no process actually crashes during an execution. One of the advantages of our transformation is that it can be used to derive the most efficient fault-tolerant termination detection algorithm known so far. Our transformation can be generalized to an arbitrary communication topology provided process crashes do not partition the system. We have proved that a perfect failure detector is the weakest failure detector for solving the termination detection problem in a crash-prone distributed system. This holds even if at most one process can crash. We have also proved that, under certain realistic assumptions, our transformation is optimal in terms of message-complexity when the communication topology is fully connected. As a future work, we plan to investigate the optimality of our transformation with respect to other metrics including detection latency and message-size complexity.

Our focus in this paper has been on studying the termination detection problem under crash-stop failure model, that is, once a process crashes, it never recovers again. An interesting direction of research is to investigate the termination detection problem under crash-recovery model in which crashed processes may recover after some time. Our preliminary work indicates that it is impossible to solve the termination detection problem in crash-recovery model using only a realistic failure detector, that is, a failure detector that cannot predict future behavior of processes [14]. We are currently working on developing a *stabilizing* termination detection algorithm when processes can crash and recover [14]. As a future work, we plan to investigate the conditions under which the termination of a computation can be detected in a *safe* manner under crash-recovery model using only a realistic failure detector.

Acknowledgments

Work by Neeraj Mittal was performed in part while visiting RWTH Aachen University and supported by Deutsche Forschungsgemeinschaft (DFG) within the Graduiertenkolleg “Software for Mobile Communication Systems”. Work by Felix C. Freiling was performed while being affiliated to RWTH Aachen University, Germany. Lucia Draque Penso was supported by Deutsche Forschungsgemeinschaft (DFG) as part of the Graduiertenkolleg “Software for Mobile Communication Systems” at RWTH Aachen University.

References

[1] A. Arora, M.G. Gouda, Distributed reset, IEEE Transactions on Computers 43 (9) (1994) 1026–1038.

[2] B. Awerbuch, Optimal distributed algorithms for minimum weight spanning tree, counting, leader election, and related problems, in: Proceedings of the 19th Annual ACM Symposium on Theory of Computing, STOC, New York, NY, United States, ACM Press, 1987, pp. 230–240.

[3] T.D. Chandra, S. Toueg, Unreliable failure detectors for reliable distributed systems, Journal of the ACM 43 (2) (1996) 225–267.

[4] S. Chandrasekaran, S. Venkatesan, A message-optimal algorithm for distributed termination detection, Journal of Parallel and Distributed Computing (JPDC) 8 (3) (1990) 245–252.

[5] K.M. Chandy, L. Lamport, Distributed snapshots: Determining global states of distributed systems, ACM Transactions on Computer Systems 3 (1) (1985) 63–75.

[6] K.M. Chandy, J. Misra, How processes learn, Distributed Computing (DC) 1 (1) (1986) 40–52.

[7] B. Charron-Bost, R. Guerraoui, A. Schiper, Synchronous system and perfect failure detector: Solvability and efficiency issues, in: Proceedings of the IEEE International Conference on Dependable Systems and Networks, DSN, New York, NY, USA, June 2000, pp. 523–532.

[8] C. Delporte-Gallet, H. Fauconnier, R. Guerraoui, A realistic look at failure detectors, in: Proceedings of the IEEE International Conference on Dependable Systems and Networks, DSN, Washington, DC, USA, 2002, pp. 345–353.

[9] R.F. DeMara, Y. Tseng, A. Ejnioui, Tiered algorithm for distributed process quiescence and termination detection, IEEE Transactions on Parallel and Distributed Systems (TPDS) 18 (11) (2007) 1539–1550.

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

[11] E.W. Dijkstra, C.S. Scholten, Termination detection for diffusing computations, Information Processing Letters (IPL) 11 (1) (1980) 1–4.

[12] C. Dwork, N. Lynch, L. Stockmeyer, Consensus in the presence of partial synchrony, Journal of the ACM 35 (2) (1988) 288–323.

[13] N. Francez, Distributed termination, ACM Transactions on Programming Languages and Systems (TOPLAS) 2 (1) (1980) 42–55.

[14] F. Freiling, M. Majuntke, N. Mittal, On detecting termination in the crash-recovery model, in: Proceedings of the 13th European Conference on Parallel and Distributed Computing (Euro-Par), Rennes, France, August 2007, pp. 629–638.

[15] F.C. Gärtner, S. Pleisch, (Im)Possibilities of predicate detection in crash-affected systems, in: Proceedings of the 5th Workshop on Self-Stabilizing Systems, WSS, Lisbon, Portugal, October 2001, pp. 98–113.

[16] F.C. Gärtner, S. Pleisch, Failure detection sequencers: Necessary and sufficient information about failures to solve predicate detection, in: Proceedings of the 16th Symposium on Distributed Computing, DISC, 2002, pp. 280–294.

[17] R. Guerraoui, L. Rodrigues, Introduction to Reliable Distributed Programming, Springer-Verlag, 2006.

[18] J.-M. Hélary, M. Murfin, A. Mostefaoui, M. Raynal, F. Tronel, Computing global functions in asynchronous distributed systems with perfect failure detectors, IEEE Transactions on Parallel and Distributed Systems (TPDS) 11 (9) (2000) 897–909.

[19] S.-T. Huang, Detecting termination of distributed computations by external agents, in: Proceedings of the IEEE International Conference on Distributed Computing Systems, ICDCS, 1989, pp. 79–84.

[20] A.A. Khokhar, S.E. Hambrusch, E. Kocalar, Termination detection in data-driven parallel computations/applications, Journal of Parallel and Distributed Computing (JPDC) 63 (3) (2003) 312–326.

[21] T.-H. Lai, L.-F. Wu, An $(N-1)$-resilient algorithm for distributed termination detection, IEEE Transactions on Parallel and Distributed Systems (TPDS) 6 (1) (1995) 63–78.

[22] L. Lamport, Time, clocks, and the ordering of events in a distributed system, Communications of the ACM (CACM) 21 (7) (1978) 558–565.

[23] M. Larrea, A. Fernández, S. Arévalo, On the implementation of unreliable failure detectors in partially synchronous systems, IEEE Transactions on Computers 53 (7) (2004) 815–828.

[24] M.J. Livesey, R. Morrison, D.S. Munro, The Doomsday distributed termination detection protocol, Distributed Computing (DC) 19 (5–6) (2007) 419–431.

[25] N.R. Mahapatra, S. Dutt, An efficient delay-optimal distributed termination detection algorithm, Journal of Parallel and Distributed Computing (JPDC) 67 (10) (2007) 1047–1066.

[26] F. Mattern, Algorithms for distributed termination detection, Distributed Computing (DC) 2 (3) (1987) 161–175.

[27] F. Mattern, Global quiescence detection based on credit distribution and recovery, Information Processing Letters (IPL) 30 (4) (1989) 195–200.

[28] N. Mittal, F.C. Freiling, S. Venkatesan, L.D. Penso, Efficient reductions for wait-free termination detection in crash-prone systems, Technical Report AIB-2005-12, Department of Computer Science, Rheinisch-Westfälische Technische Hochschule (RWTH), Aachen, Germany, June 2005.

[29] N. Mittal, S. Venkatesan, S. Peri, A family of optimal termination detection algorithms, Distributed Computing (DC) 20 (2) (2007) 141–162.

[30] S. Peri, N. Mittal, Improving the efficacy of a termination detection algorithm, Journal of Information Science and Engineering (JISE) 24 (1) (2008) 159–174.

[31] S.P. Rana, A distributed solution of the distributed termination problem, Information Processing Letters (IPL) 17 (1) (1983) 43–46.

[32] A. Shah, S. Toueg, Distributed snapshots in spite of failures, Technical Report TR84-624, Department of Computer Science, Cornell University, Ithaca, NY, July 1984 (Revised February 1985).

[33] N. Shavit, N. Francez, A new approach to detection of locally indicative stability, in: Proceedings of the International Colloquium on Automata, Languages and Systems, ICALP, Rennes, France, 1986, pp. 344–358.

[34] G. Tel, F. Mattern, The derivation of distributed termination detection algorithms from garbage collection schemes, ACM Transactions on Programming Languages and Systems (TOPLAS) 15 (1) (1993) 1–35.

[35] Y.-C. Tseng, Detecting termination by weight-throwing in a faulty distributed system, Journal of Parallel and Distributed Computing (JPDC) 25 (1) (1995) 7–15.

[36] S. Venkatesan, Reliable protocols for distributed termination detection, IEEE Transactions on Reliability 38 (1) (1989) 103–110.

[37] X. Wang, J. Mayo, A general model for detecting termination in dynamic systems, in: Proceedings of the 18th International Parallel and Distributed Processing Symposium, IPDPS, Santa Fe, NM, April 2004.

[38] L.-F. Wu, T.-H. Lai, Y.-C. Tseng, Consensus and termination detection in the presence of faulty processes, in: Proceedings of the International Conference on Parallel and Distributed Systems, ICPADS, Hsinchu, Taiwan, December 1992, pp. 267–274.

Neeraj Mittal received his B.Tech. degree in computer science and engineering from the Indian Institute of Technology, Delhi in 1995 and the M.S. and Ph.D. degrees in computer science from the University of Texas at Austin in 1997 and 2002, respectively. He is currently an assistant professor in the Department of Computer Science at the University of Texas at Dallas and a co-director of the Advanced Networking and Dependable System Laboratory (ANDES). His research interests include distributed systems, wireless networks, and security.

Felix C. Freiling is a full professor of computer science at University of Mannheim, Germany. He holds a diploma and a Ph.D. degree in computer science from Technical University of Darmstadt, Germany. Before joining Mannheim, he was a postdoctoral student in the Distributed Programming Laboratory at the Swiss Federal Institute of Technology, Lausanne, Switzerland, and then an associate professor at RWTH Aachen University, Germany. His research in Switzerland was sponsored by a prestigious Emmy Noether Scholarship from the Deutsche Forschungsgemeinschaft. He is interested in the areas of systems safety and systems security and the intersection between both areas.

S. Venkatesan received his B.Tech. degree in Civil Engineering and M.Tech. degree in Computer Science from the Indian Institute of Technology, Madras in 1981 and 1983, respectively. He completed his Ph.D. degree in Computer Science from the University of Pittsburgh in December 1988. In January 1989, he joined the University of Texas at Dallas where he is currently an Associate Professor of Computer Science and Head of Telecommunications Engineering Program. His research interests are in Distributed Systems, Sensor Networks, Cognitive Radio Networks, Mobile and Wireless Networks and Testing and Debugging Distributed Programs. He has been the Chief Architect of IPmobile (a startup acquired by Cisco Systems in September 2000) and a member of the advisory board of Jahi Networks (a startup acquired by Cisco Systems in December 2004).

Dr. Lucia Draque Penso received her Ph.D. degree from Brown University in 2008 and her A.B. degree from the Federal University of Rio de Janeiro in 1998. Her advisors were Prof. Dr. Maurice Herlihy and Prof. Dr. Felix Freiling, the later having hosted her in Germany, initially at RWTH Aachen and later at the University of Manheimm. From 1998 to 2001, she also worked at BNDES, the Economical and Social Development National Bank, a brazilian government agency responsible for nurturing the economy and promoting progress in Brazil.

Dr. Penso research focus on theoretical and practical aspects of distributed computation and network security, with a special regard to wait-free agreement protocols. Nevertheless, she is also interested in problems stemming from the areas of economics, finance, cryptography and combinatorial optimization.