Efficient Reduction for Wait-Free Termination Detection in a Crash-Prone Distributed System

Neeraj Mittal[1][⋆], Felix C. Freiling[2], S. Venkatesan[1], and Lucia Draque Penso[2][⋆⋆]

[1] Department of Computer Science, The University of Texas at Dallas, Richardson, TX 75083, USA, {neerajm,venky}@utdallas.edu

[2] Department of Computer Science, RWTH Aachen University, D-52056 Aachen, Germany, {freiling@,lucia@i4.}informatik.rwth-aachen.de

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 at most $O(n + \mu(n, 0))$ messages per failure more than the message complexity of $\mathcal{A}$. Also, its detection latency is at most $O(\delta(n, 0))$ per failure more than that of $\mathcal{A}$. Furthermore, the overhead (that is, the amount of control data piggybacked) on an application message increases by only $O(\log n)$ bits per failure.

The fault-tolerant termination detection algorithm resulting from the transformation satisfies two 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 until one or more processes crash, that is, it is *fault-reactive*. Our transformation can be extended to arbitrary communication topologies provided process crashes do not partition the system.

Key words: distributed system, termination detection, faulty processes, wait-free algorithm, failure detector, algorithm transformation

[⋆] 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".

[⋆⋆] Lucia Draque Penso was supported by Deutsche Forschungsgemeinschaft (DFG) as part of the Graduiertenkolleg "Software for Mobile Communication Systems" at RWTH Aachen University.

1 Introduction

One of the fundamental problems in distributed systems is to detect termination of an ongoing distributed computation. The termination detection problem was independently proposed by Dijkstra and Scholten [1] and Francez [2] more than two decades ago. Since then, many researchers have studied this problem and, as a result, a large number of efficient algorithms have been developed for detecting termination (*e.g.*, [3–14]). Most of the termination 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 [15], which was derived from the fault-sensitive (that is, fault-*in*tolerant) termination detection algorithm by Chandrasekaran and Venkatesan [9]. Venkatesan's algorithm achieves fault-tolerance by *replicating* state information at multiple processes. However, it assumes a *prespecified* bound k on the maximum number of processes that can fail by crashing. Its message complexity is $O(kM + c)$, 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 $k + 1$ (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 $(k + 1)$-connected [15].

Lai and Wu [16] and Tseng [17] modify fault-sensitive termination detection algorithms by Dijkstra and Scholten [1] and Huang [8], respectively, to derive two different fault-tolerant termination detection algorithms. Both algorithms assume that the communication topology is fully connected. 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 f is the actual number of processes that fail during an execution. The algorithm by Lai and Wu [16] has high detection latency of $O(n)$ whereas the algorithm by Tseng [17] has high application and control message overheads of $O(M)$ and $O(f \log n + nM)$, respectively.

Shah and Toueg give a fault-tolerant algorithm for taking a consistent snapshot of a distributed system in [18]. Their algorithm is derived from the fault-sensitive consistent snapshot algorithm by Chandy and Lamport [19]. As a result, each invocation of their snapshot algorithm may generate up to $O(c)$ control messages. It is easy to verify that, when their algorithm is used for termination detection, the message complexity of the resulting algorithm is $O(cM)$ in the worst-case. Similarly, Gärtner and Pleisch [20] give an algorithm for detecting an arbitrary stable predicate in a crash-prone distributed system. (Note that termination is a stable property.) In their algorithm, every relevant local event is *reliably and causally broadcast* to a set of monitors, thereby increasing the message complexity significantly.

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 [1], $\mu(n, 0) = O(n)$ and $\delta(n, 0) = O(1)$ when their algorithm is adopted for a non-diffusing computation.) Further, the application and control message overheads of $\mathcal{B}$ are at most $O(f \log n)$ and $O(f \log n + n \log M)$ more than those for $\mathcal{A}$. (Message overhead refers to the amount of control information piggybacked on a message.) The overhead of $O(n \log M)$ applies only to those control messages that are exchanged whenever a crash is detected. The fault-tolerant termination detection algorithm resulting from the transformation satisfies two desirable properties. First, the 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*.

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 [1] and Huang [8], the resulting fault-tolerant algorithms compare very favorably with those by Lai and Wu [16] and Tseng [17]. Specifically, when our transformation is applied to Dijkstra and Scholten's algorithm [1], the resulting algorithm has the same message-complexity and detection latency as the algorithm by Lai and Wu [16]. However, the message overhead—application as well as control—is higher for our algorithm. On the other hand, when our transformation is applied to Huang's weight throwing algorithm [8] (which is similar to Mattern's credit distribution and recovery algorithm [7]), the resulting algorithm has the same message-complexity and detection latency as that of the algorithm by Tseng [17] but has slightly higher application message overhead ($O(\log M + f \log n)$ versus $O(\log M)$). Surprisingly, the overhead of control messages exchanged due to process crashes by our algorithm, which is given by $O(f \log n + n \log M)$, is *much lower* than that of Tseng's algorithm [17], which is given by $O(f \log n + nM)$. For comparison between various fault-tolerant termination detection algorithms, please refer to Table 1. The results of applying our transformation to some important termination detection algorithms are given in [21].

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 in-transit when the termination detection algorithm is restarted. Arora and Gouda [22] also

		Venkatesan [15]	Lai and Wu [16]	Tseng [17]	Our Approach [this paper]
Message Complexity		$O(kM + c)$	$O(M + fn + n)$	$O(M + fn + n)$	$\mu(n, M) + O(f\ (n + \mu(n, 0)))$
Detection Latency		$O(M)$	$O(n)$	$O(f + 1)$	$\delta(n, M) + O(f\ \delta(n, 0))$
Application Message Overhead		k-way duplication	-	$O(\log M)^*$	$\alpha(n, M) + O(f \log n)$
Control Message Overhead	Termination Detection	$O(\log n + \log M)$	$O(f \log n + \log M)$	$O(\log M)^*$	$\beta(n, M) + O(f \log n)$
	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

*Assuming an efficient implementation of weight throwing scheme such as the one described in [7]
n: initial number of processes in the system
c: number of channels in the communication topology
f: actual number of processes that crash during an execution
k: maximum number of processes that can crash during an execution
M: number of application messages exchanged

$\mu(n, M)$:	message complexity of the fault-sensitive termination detection algorithm with n processes and M application messages
$\delta(n, M)$:	detection latency of the fault-sensitive termination detection algorithm with n processes and M application messages
$\alpha(n, M)$:	application message overhead of the fault-sensitive termination detection algorithm with n processes and M application messages
$\beta(n, M)$:	control message overhead of the fault-sensitive termination detection algorithm with n processes and M application messages

Table 1. Comparison of various fault-tolerant termination detection algorithms.

provide a mechanism to reset a distributed system. Their approach is different from our approach in many ways. First, the semantics of their reset operation is different from the semantics of our restart operation. Specifically, 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 if the distributed reset procedure is used. In contrast, in our case, the distributed computation continues to execute without interruption. (When there are process crashes, we assume that the primary computation may be 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. Second, in their approach, the system may be reset more than once for the same failure. This may happen, for example, when multiple processes detect the failure of the same process at different times. Third, 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 than just crash failures. Not surprisingly, their reset operation has higher message and time complexities than our restart operation. Fourth, their approach is *non-masking fault-tolerant*, which implies that the safety specification of the application may be *violated temporarily*, even if there is a single crash fault. When translated to our problem, this means that the termination detection algorithm may falsely announce termination, a case which our approach avoids.

We build upon the work by Wu *et al* [23]. We do this in the context of the failure detector hierarchy proposed by Chandra and Toueg [24], 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 transformation can also be extended to an arbitrary communication topology provided process crashes do not partition the system. (In case partitioning occurs, termination is detected separately in each partition.) For an arbitrary topology, however, the increase in message-complexity and detection latency per failure is higher than that for fully connected topology. Due to lack of space, we only focus on the transformation for fully connected topology in this paper. Details of the transformation for arbitrary topology can be found elsewhere [21].

This paper is organized as follows. In Sect. 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 in Sect. 3. In Sect. 4 we determine the type of failure detector which is necessary for solving termination detection. Finally, we present our conclusions and outline directions for future research in Sect. 5.

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 "non-crashed process", "live process" and "operational process" interchangeably. A process that crashes is called *faulty*. A process that is not faulty is called *correct*. Note that there is a difference between the terms "live process" and "correct process". A live process has not crashed yet but may crash in the future. Let $P = \{p_1, p_n, \ldots, p_n\}$ denote the initial set of processes in the system. We assume that there is at least one correct process in the system at all times.

We assume that all channels are bidirectional but may not be FIFO (first-in-first-out). Channels are reliable in the sense that if a process never crashes, then every message destined for it 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, that is, every pair of operational processes can directly communicate with each other.

We assume the existence of a *perfect failure detector* [24], a device 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 [24]: *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, the eventually perfect failure detector satisfies *eventually strong accuracy* (eventually no correct process is ever suspected) and strong completeness.

2.2 Termination Detection in a Crash-Prone System

Informally, the termination detection problem involves determining when a distributed computation has ceased all its activity. 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 both processes 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, a crash-prone distributed system 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* [23] 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* [15] or *fail-flush* [16] 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 [17]. 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 say that the system 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. We believe that detecting effective-termination is sufficient in most cases.

Wu *et al* [23] 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 [24], 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.

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 overhead. 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. Finally, message overhead refers to the amount of control data piggybacked on a message by the termination detection algorithm.

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.

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 [25] 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. We also assume that, as soon as a process learns about the failure of its neighbouring process, it freezes its incoming channel with the process.

Due to lack of space, we only describe the main idea behind our transformation. Further, we only state the main lemmas and theorems that are used to prove its correctness and analyze its performance. The formal description of the algorithm and omitted proofs can be found in [21].

3.1 The Main Idea

The main idea behind our transformation is to *restart* the fault-sensitive termination detection algorithm algorithm on the set of *currently operational processes* whenever a new failure is detected. We refer to 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. This is useful as explained further.

Consider a subset of processes Q. We say that a distributed 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. 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 $P \setminus Q$. We have,

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

The above theorem implies that if all alive 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.

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.*

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*. Clearly, 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.

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 the new instances of both the secondary computation and the fault-sensitive termination detection algorithm start at the same time on the same set of processes.

We now describe the behavior of a 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 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 processes in Q. Then, if the secondary computation has classically terminated with respect to Q, then 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 processes in Q. Assume that the primary computation has classically terminated with respect to Q and each process in Q eventually learns that there are 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.*

We next describe how to ensure that all operational processes agree on the set of failed processes before restarting the secondary computation the fault-sensitive termination detection algorithm. Later, we describe how to ascertain that there are no relevant old application messages in transit. 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.

Achieving Agreement on the Set of Failed Processes: Whenever a process crashes, 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, start a new instance of the fault-sensitive termination detection algorithm to detect its termination.

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

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 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 at least 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.

3.2 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 1. *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.

Theorem 5 (safety property). *If $\mathcal{B}$ announces termination, then the underlying computation has effectively terminated.*

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

Theorem 6 (liveness property). *Once the underlying computation effectively terminates, $\mathcal{B}$ eventually announces termination.*

3.3 Performance 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. We now analyze the message complexity and detection latency of the fault-tolerant termination detection algorithm $\mathcal{B}$. Let f denote the actual number of processes that fail during an execution of $\mathcal{B}$.

Lemma 1. *The number of times $\mathcal{A}$ is restarted is bounded by f.*

To compute the message complexity of $\mathcal{B}$, we assume that $\mu(n, M)$ satisfies the following constraint for $k \geq 1$:

$$\sum_{i=1}^{k} \mu(n, M_i) \leq \mu(n, \sum_{i=1}^{k} M_i) + (k-1)\, \mu(n, 0) \tag{6.1}$$

For all existing termination detection algorithms that we are aware of, $\mu(n, M)$ is linear in M. It can be verified that if $\mu(n, M)$ is a linear function in M, then the inequality (6.1) indeed holds.

Theorem 7 (message complexity). *The message complexity of $\mathcal{B}$ is given by $\mu(n, M) + O(f\ (n + \mu(n, 0)))$.*

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 latency is bounded by one time unit as well.

Theorem 8 (detection latency). *The detection latency of $\mathcal{B}$ is given by $\delta(n, M) + O(f\delta(n, 0))$.*

We next bound the message overhead of $\mathcal{B}$. Let $\alpha(n, M)$ and $\beta(n, M)$ denote the application and control message overhead, respectively, of $\mathcal{A}$ when the system has n processes and the underlying computation exchanges M application messages.

Theorem 9 (application message overhead). *The application message overhead of $\mathcal{B}$ is $\alpha(n, M) + O(f \log n)$.*

Finally, we bound the control message overhead of $\mathcal{B}$. Note that control messages can be categorized 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 10 (control message overhead). *The control message overhead of $\mathcal{B}$ for termination detection messages is given by $\beta(n, M) + O(f \log n)$ and for failure recovery messages is given by $O(f \log n + n \log M)$.*

4 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 [26]. 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* [27] instead of full synchrony). Proving that a certain type of failure detector is *necessary* for solving a problem gives an indication about 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 termination detection in a crash-prone distributed system. To that end, we transform an instance of a fault-tolerant termination detection algorithm into a perfect failure detector at one process q, that is, q is able to reliably detect process crashes. A perfect failure detector can then be implemented by using n parallel instances of the transformation algorithm, one per process.

Assume that we are given an algorithm $\mathcal{A}$ that can detect termination of an arbitrary computation among n processes even in the presence of process crashes. We now set up n independent computations C_i, one for each process p_i. The computation C_i is such that process p_i is initially active and all processes apart from p_i are passive. In the computation no messages are sent and received and p_i never becomes passive. Now consider some process $q \neq p_i$ and the corresponding computation C_i. Process q starts an instance of the termination detection algorithm $\mathcal{A}$ with respect to the computation C_i. Whenever $\mathcal{A}$ announces the termination of C_i, q henceforth permanently suspects p_i. The same actions are performed for every other process in the system, that is, q invokes n parallel instances of $\mathcal{A}$, one for each computation C_i.

We now show that this algorithm implements a perfect failure detector if $\mathcal{A}$ correctly solves the effective-termination detection problem. First consider strong accuracy (a process is never suspected before it crashes) and assume that q suspects p_i. It follows from our transformation that $\mathcal{A}$ has announced termination of the computation C_i. This means that all processes in C_i are either crashed or passive. Since C_i is such that p_i is never passive, it implies that p_i has crashed.

Now consider strong completeness (eventually every crashed process is suspected by every correct process) and assume that p_i has crashed and q is correct. Once p_i crashes, clearly the termination condition holds for the computation C_i. Since $\mathcal{A}$ is a correct termination detection algorithm, $\mathcal{A}$ eventually announces termination of C_i at q. Upon announcing termination, q starts suspecting p_i, concluding 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. Hence, it is impossible to solve termination detection when one or more processes can crash assuming only a failure detector that is strictly weaker than a perfect failure detector. In other words, a perfect failure detector is *necessary* for solving the effective-termination detection problem.

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 Sect. 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* [23], which shows that a failure detector must be complete. Our result also further clarifies the relationship between the termination detection problem and the *consensus problem*: Wu *et al* [23] show that termination detection is at least as hard to solve as consensus. By relating termination detection to the failure detector hierarchy of Chandra and Toueg [24], our result has two interesting corollaries. First, 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 consensus is strictly weaker than a perfect failure detector [24]. Second, when any number of processes can crash, termination detection is actually equivalent to consensus [28].

5 Conclusions and Future Work

In this paper, we presented a transformation using a perfect failure detector that can be used to convert any termination detection algorithm for a fully connected communication topology that 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. Moreover, when applied to fault-sensitive termination detection algorithms by Dijkstra and Scholten [1] and Huang [8], the resulting fault-tolerant termination detection algorithms compare very favorably with those by Lai and Wu [16] and Tseng [17]. Our transformation can be generalized to an arbitrary communication topology provided process crashes do not partition the system. We also 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.

As part of future work, we plan to investigate the termination detection problem when crashed processes may recover and channels may be lossy. We also plan to apply ideas proposed in this paper to transform other fault-sensitive algorithms—such as for detecting other stable properties—into fault-tolerant algorithms.

References

1. Dijkstra, E.W., Scholten, C.S.: Termination Detection for Diffusing Computations. Information Processing Letters (IPL) **11** (1980) 1–4
2. Francez, N.: Distributed Termination. ACM Transactions on Programming Languages and Systems (TOPLAS) **2** (1980) 42–55
3. Rana, S.P.: A Distributed Solution of the Distributed Termination Problem. Information Processing Letters (IPL) **17** (1983) 43–46
4. Shavit, N., Francez, N.: A New Approach to Detection of Locally Indicative Stability. In: Proceedings of the International Colloquium on Automata, Languages and Systems (ICALP), Rennes, France (1986) 344–358
5. Mattern, F.: Algorithms for Distributed Termination Detection. Distributed Computing (DC) **2** (1987) 161–175
6. Dijkstra, E.W.: Shmuel Safra's Version of Termination Detection. EWD Manuscript 998. Available at `http://www.cs.utexas.edu/users/EWD` (1987)
7. Mattern, F.: Global Quiescence Detection based on Credit Distribution and Recovery. Information Processing Letters (IPL) **30** (1989) 195–200
8. Huang, S.T.: Detecting Termination of Distributed Computations by External Agents. In: Proceedings of the IEEE International Conference on Distributed Computing Systems (ICDCS). (1989) 79–84
9. Chandrasekaran, S., Venkatesan, S.: A Message-Optimal Algorithm for Distributed Termination Detection. Journal of Parallel and Distributed Computing (JPDC) **8** (1990) 245–252

10. Tel, G., Mattern, F.: The Derivation of Distributed Termination Detection Algorithms from Garbage Collection Schemes. ACM Transactions on Programming Languages and Systems (TOPLAS) **15** (1993) 1–35
11. Khokhar, A.A., Hambrusch, S.E., Kocalar, E.: Termination Detection in Data-Driven Parallel Computations/Applications. Journal of Parallel and Distributed Computing (JPDC) **63** (2003) 312–326
12. Mahapatra, N.R., Dutt, S.: An Efficient Delay-Optimal Distributed Termination Detection Algorithm. To Appear in Journal of Parallel and Distributed Computing (JPDC) (2004)
13. Wang, X., Mayo, J.: A General Model for Detecting Termination in Dynamic Systems. In: Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS), Santa Fe, New Mexico (2004)
14. Mittal, N., Venkatesan, S., Peri, S.: Message-Optimal and Latency-Optimal Termination Detection Algorithms for Arbitrary Topologies. In: Proceedings of the Symposium on Distributed Computing (DISC), The Netherlands (2004) 290–304
15. Venkatesan, S.: Reliable Protocols for Distributed Termination Detection. IEEE Transactions on Reliability **38** (1989) 103–110
16. Lai, T.H., Wu, L.F.: An $(N-1)$-Resilient Algorithm for Distributed Termination Detection. IEEE Transactions on Parallel and Distributed Systems (TPDS) **6** (1995) 63–78
17. Tseng, Y.C.: Detecting Termination by Weight-Throwing in a Faulty Distributed System. Journal of Parallel and Distributed Computing (JPDC) **25** (1995) 7–15
18. Shah, A., Toueg, S.: Distributed Snapshots in spite of Failures. Technical Report TR84-624, Department of Computer Science, Cornell University, Ithaca, NY (1984) (Revised February 1985).
19. Chandy, K.M., Lamport, L.: Distributed Snapshots: Determining Global States of Distributed Systems. ACM Transactions on Computer Systems **3** (1985) 63–75
20. Gärtner, F.C., Pleisch, S.: (Im)Possibilities of Predicate Detection in Crash-Affected Systems. In: Proceedings of the 5th Workshop on Self-Stabilizing Systems (WSS), Lisbon, Portugal, Springer-Verlag (2001) 98–113
21. Mittal, N., Freiling, F.C., Venkatesan, S., Penso, L.D.: 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 (2005)
22. Arora, A., Gouda, M.G.: Distributed Reset. IEEE Transactions on Computers **43** (1994) 1026–1038
23. Wu, L.F., Lai, T.H., Tseng, Y.C.: Consensus and Termination Detection in the Presence of Faulty Processes. In: Proceedings of the International Conference on Parallel and Distributed Systems (ICPADS), Hsinchu, Taiwan (1992) 267–274
24. Chandra, T.D., Toueg, S.: Unreliable Failure Detectors for Reliable Distributed Systems. Journal of the ACM **43** (1996) 225–267
25. Peri, S., Mittal, N.: On Termination Detection in an Asynchronous Distributed System. In: Proceedings of the ISCA International Conference on Parallel and Distributed Computing Systems (PDCS), California (2004) 209–215
26. Larrea, M., Fernández, A., Arévalo, S.: On the Implementation of Unreliable Failure Detectors in Partially Synchronous Systems. IEEE Transactions on Computers **53** (2004) 815–828
27. Dwork, C., Lynch, N., Stockmeyer, L.: Consensus in the Presence of Partial Synchrony. Journal of the ACM **35** (1988) 288–323
28. Delporte-Gallet, C., Fauconnier, H., Guerraoui, R.: A Realistic Look At Failure Detectors. In: Proceedings of the International Conference on Dependable Systems and Networks (DSN), Washington, DC, USA, IEEE Computer Society (2002) 345–353