# Termination Detection in an Asynchronous Distributed System with Crash-Recovery Failures

## Technical Report
## Department for Mathematics and Computer Science
## University of Mannheim
## TR-2006-008

Felix C. Freiling[1], Matthias Majuntke[2], and Neeraj Mittal[3]

[1] University of Mannheim, D-68131 Mannheim, Germany
[2] RWTH Aachen University, D-52056 Aachen, Germany
[3] The University of Texas at Dallas, Richardson, TX 75083, USA

**Abstract** We revisit the problem of detecting the termination of a distributed application in an asynchronous message-passing model with crash-recovery failures and failure detectors. We derive a suitable definition of termination detection in this model but show that this definition is impossible to implement unless you have a failure detector which can predict the future. We subsequently weaken the problem and strengthen the failure model to allow solvability.

## 1 Introduction

### 1.1 Termination detection

In practice, it is often necessary to know when the computation running in a distributed system has terminated. For example, it is possible to construct an efficient mutual exclusion algorithm in the following way: a first distributed algorithm establishes a spanning tree in the network, while a second algorithm circulates a token in a repeated depth-first traversal of the tree. To ensure the correctness of mutual exclusion, it is vital that the second algorithm is only started once the first algorithm has terminated, resulting in the problem of *termination detection*.

A termination detection algorithm involves a computation of its own which should not interfere with the underlying computation which it observes. Additionally, it should satisfy two properties: (1) it should never announce termination unless the underlying computation has in fact terminated. (2) If the underlying computation has terminated, the termination detection algorithm should eventually announce termination.

But when is a computation in fact terminated? Answering this question means to define an appropriate formal notion of termination. To be general,

the states of processes are mapped to just two distinct states: active and passive. An active process still actively participates in the computation while a passive process does not participate anymore unless it is activated by an active process. Activation can only be done using communication. For message-passing communication, which we also assume in this paper, a widely accepted definition of termination is that (1) all processes are passive and (2) all channels are empty.

### 1.2 Related Work

Many algorithms for termination detection have been proposed in the literature (see the early overview by Mattern [10] for an introduction). Most of them assume a perfect environment in which no faults happen. There is relatively little work on fault-tolerant termination detection [11,14,16,17,15,7,6,12]. All this work assumes the crash-stop failure model meaning that the only failures which may occur are crash faults where processes simply stop to execute steps. A suitable definition of termination is to restrict the fault-free definition to fault-free processes: A computation is terminated if all alive processes are passive and no message is in-transit towards alive processes.

Most papers are versions of fault-free termination detection algorithms extended to the crash-stop model. It is well-known [6] that termination detection is closely related to the problem of *failure detection* [3] in this setting. Intuitively, a failure detector is an abstract device which detects whether a process is crashed or not. One recent paper [12] presents a general transformation which can transform an arbitrary termination detection algorithm for the fault-free setting into a crash-stop-tolerant termination detection algorithm using the concept of a perfect failure detector. A perfect failure detector accurately detects every crash in the system.

In this paper we revisit the termination detection problem in a more severe failure model, namely that of crash-recovery. Roughly speaking, in the crash-recovery model of failures, processes are allowed to crash just like in the crash-stop model but they are also allowed to restart their execution later. The crash-recovery model is much more realistic than the crash-stop model but also much harder to deal with. This has been demonstrated by earlier work in this failure model giving solutions to reliable broadcast [2], consensus using failure detectors [1], or atomic broadcast [13]. We are unaware, however, of any termination detection algorithm for the crash-recovery model.

### 1.3 Problems in the crash-recovery model

Solving the termination detection problem in the crash-recovery model is not an easy task. First of all, it is not clear what a sensible definition of termination is in the crash-recovery model. On the one hand, the classical (fault-free) definition of termination as mentioned above is clearly not suitable: If an active process crashes, there is always the possibility that it recovers later but there is no guarantee that it actually will recover. So an algorithm is in the dilemma to either making a false detection of termination or to possibly waiting infinitely long (see

Figure 1). On the other hand, the definition used in the crash-stop model is also not suitable: An algorithm might announce termination prematurely if an active process which was crashed recovers again (see Figure 2).

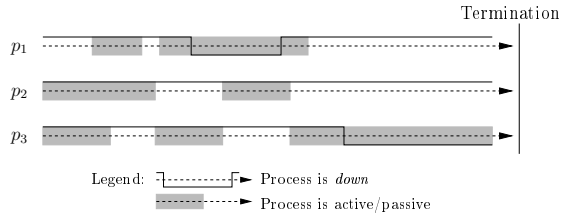


Figure 1: Classical termination in the crash-recovery model

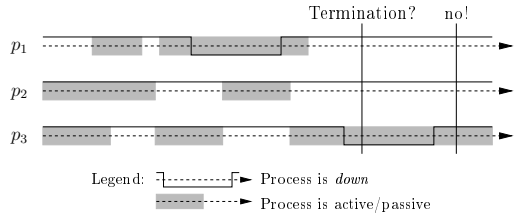


Figure 2: Correct-restricted classical termination in the crash-recovery model

### 1.4 Contributions

The setting of this paper is the crash-recovery model which is equipped with failure detectors. We make the following contributions:

- We give a definition of what it means to detect termination of a distributed computation in the crash-recovery model and argue that it is sensible. Our definition is also a strict generalization of the definitions of termination detection for fault-free and crash-stop models.
- We show that it is impossible to solve the termination detection problem in the crash-recovery model if a process can restart in any state (active or passive) on recovery. We do this by relating the problem of termination detection to the problem of failure detection and show that the necessary failure detector is not implementable.

- We introduce the notion of *stabilizing* termination detection in which termination detection announcements may be revoked a finite number of times. We show that even this weaker variant of termination detection is impossible to solve in the crash-recovery model if the system contains one or more unstable processes—processes that crash and recover infinitely often.
- We introduce the notion of *stabilizing* crash-recovery model in which all processes eventually either stay up or stay down (that is, the crash-recovery model eventually behaves like the crash-stop model). We present an algorithm for solving the stabilizing termination detection problem in the stabilizing crash-recovery model that uses a failure detector which is strictly weaker than the perfect failure detector.

In summary, the results give insight into the additional complexities induced by the crash-recovery model in contrast to the crash-stop model. For lack of space, proofs and our algorithm [9] have been moved to the appendix.

## 2 Model

### 2.1 Distributed System

A *distributed system* consists of a set of processes that are connected using a communication network. The processes communicate with each other by exchanging messages. There is no shared memory. We assume an *asynchronous* distributed system meaning that there is no bound on delays of messages and relative speeds of processes. We use $\Pi$ to denote the set of processes and $\Gamma$ to denote the set of channels in the system.

We assume the existence of a discrete global clock. The global clock is solely a fictional device and the processes do not have access to it. The range $\mathcal{T}$ of the clock's ticks is the set of natural numbers $\mathbb{N}$.

In distributed systems it is usually assumed that communication networks allow communication in duplex mode, that is, messages can be exchanged in both directions between two processes. For our case, it is helpful to assume that both communication directions can be observed independently.

### 2.2 Distributed Computation

A distributed system generates a distributed computation by executing a distributed program (sometimes also called algorithm or application). When executing a local program, a process $p$ can produce three types of events: (1) send events for sending a messages $m$ to process $q$, (2) receive events for receiving (or delivering) a message $m$ which was sent by process $q$, and (3) internal events if an instruction is executed and no message is sent or received. Every event executed by a process changes its local state. The state of the entire distributed system—called *configuration*—consists of the set of local states of all processes and the set of all messages in transit.

An *execution* of a distributed algorithm is a maximal sequence $E = (\gamma_0, \gamma_1, \gamma_2, \ldots)$, where, for each $i \geq 0$, $\gamma_i$ is a configuration of the system, and $\gamma_{i+1}$ is obtained from $\gamma_i$ by executing an event on exactly one process. Maximality means that the execution is either infinite or ends in a configuration where no further events can be executed.

**Computations and Executions.** An execution of a distributed algorithm may also be described as a partially ordered set of events that have been executed so far. We define a causality relation $\prec$ as the partial order [8]:

1. If $e$ and $f$ are different events of the same process and $e$ occurs before $f$, then $e \prec f$.
2. If $s$ is a send event and $r$ the corresponding receive event, then $s \prec r$.
3. If $e \prec g$ and $g \prec f$, then $e \prec f$.

In the partial order $\prec$ there may be pairs of events $e$ and $f$ for which neither $e \prec f$ nor $f \prec e$ holds. Such events are called *concurrent*. Executions which differ only in the permutation of concurrent events are called *equivalent*, denoted $E \sim F$. A *computation* of a distributed algorithm is an equivalence class (under $\sim$) of executions of the algorithm. A computation is the class of all interleavings of events which respect to the same causality relation.

### 2.3 Failures

**Crash-Recovery Model.** We assume that processes fail by crashing and may recover subsequently. A process may fail and recover more than once. When a process *crashes*, it stops executing its algorithm and cannot send or receive messages. Processes have two types of storage: volatile and stable. If a process crashes, it looses the entire contents of its volatile storage. Stable storage is not affected by a crash. However, access to stable storage is expensive and should be avoided as much as possible.

A *failure pattern* specifies times at which processes crash and (possibly) recover. It is a function $F$ from the set of global clock ticks $\mathcal{T}$ to the powerset of processes $2^{\Pi}$. If $p \in F(t)$, then process $p$ is crashed at time $t$. Given a failure pattern $F$, a process $p \in \Pi$

- is said to be *up at time* $t$, if $p \notin F(t)$.
- is said to be *down at time* $t$, if $p \in F(t)$.
- *crashes at time* $t \geq 1$ if $p$ is up at time $t-1$ and down at time $t$.
- *recovers at time* $t \geq 1$ if it is down at time $t-1$ and up at time $t$.

As discussed in Aguilera *et al.* [1], a process $p$ in the crash-recovery model belongs to one of the four categories:

- *Always-up.* Process $p$ never crashes.
- *Eventually-up.* Process $p$ crashes at least once, but there is a time after which $p$ is permanently up.

- *Eventually-down.* There is a time after which process $p$ is permanently down.
- *Unstable.* Process $p$ crashes and recovers infinitely many times.

Always-up and eventually-up processes are referred to as *good* processes. Eventually-down and unstable processes are referred to as *bad* processes. We use the phrases *up process* and *live process* synonymously.

In this paper, we assume that communication channels among processes are *eventually-reliable*. A channel from process $p$ to process $q$ is said to be eventually-reliable if it satisfies the following properties:

- *Validity.* If $p$ sends a message to $q$ and neither $p$ nor $q$ crashes, then the message is eventually delivered to $q$.
- *No Duplication.* No message is delivered more than once.
- *No Creation.* No message is delivered unless it was sent.

We also assume that all messages sent by a process are distinct. One way to ensure this is to maintain an *incarnation number* for a process in stable storage. The incarnation number is incremented whenever the process recovers from a crash and written back to stable storage. In addition, there is a sequence number that is stored in volatile storage and is incremented whenever the process sends a message. Each message is piggybacked with the incarnation number and the sequence number, which ensures that all messages sent by a process are distinct.

### 2.4 Failure Detectors in the Crash-Recovery Model

Many important problems in distributed computing such as consensus, atomic broadcast and termination detection are impossible to solve in an asynchronous distributed system when processes are unreliable [5]. To that end, Chandra and Toueg [3] introduced the notion of *failure detector*. A failure detector at a process outputs its current view about the operational state (up or down) of other processes in the system. Depending on the properties that failure detector output has to satisfy, several classes of failure detectors can be defined [3]. With the aid of failure detectors, problems such as consensus, atomic broadcast and termination detection become solvable in unreliable asynchronous distributed systems that are otherwise impossible to solve. A failure detector itself is implemented by making certain synchrony assumptions about the system [3].

The notion of failure detector, which was originally defined for crash-stop failure model, has been extended to crash-recovery failure model as well [1]. The failure detector defined by Aguilera *et al.* [1] for crash-recovery failure model, denoted by $\diamondsuit\mathcal{S}_e$, outputs a list of processes which are deemed to be currently up along with an epoch number for each such process. The epoch number associated with a process roughly counts the number of times the process has crashed and recovered.

In this paper, we use a failure detector with a simpler interface, denoted by $\diamondsuit\mathcal{P}_{cr}$. The failure detector at a process only outputs a list of processes it currently deems to be up (we call this the *trust-list*). Processes which are not on the trust-list are suspected to be down by the failure detector. A suspected

process is unsuspected if it is put on the trust-list. This failure detector satisfies the following properties:

- *Completeness.* Every *eventually-down* process is eventually permanently suspected by all good processes. Every *unstable* process is suspected and unsuspected infinitely often by all good processes.
- *Accuracy.* Every good process is eventually permanently trusted by all good processes.

A failure detector from class $\diamondsuit\mathcal{P}_{\text{cr}}$ is strictly stronger than a failure detector from class $\diamondsuit\mathcal{S}_e$ because, in $\diamondsuit\mathcal{S}_e$, only one good process is required to be permanently trusted by all good processes. Nevertheless $\diamondsuit\mathcal{P}_{\text{cr}}$ can be implemented under the same approach and the same assumptions of partial synchrony made in the original paper of Aguilera *et al.* [1].

## 3 The Termination Detection Problem

In the termination detection problem, the system is executing a distributed program, thereby generating a distributed computation referred to as *underlying computation*. A process in the system can be in two states with respect to the computation: *active* or *passive*. A process can execute an internal event of the computation only if it is active. There are three rules that describe allowed state changes between active and passive states:

- An active process may become passive at any time.
- A passive process can become active only on receiving a message.
- A process can send a message only when it is active.

We assume that processes have access to stable storage using which they are able to maintain their last saved state during time intervals when they are down. Especially, the state changes between active and passive are written to stable storage (*e.g.*, if an active process crashes it is still active after recovery). This ensures that a process that fails in passive state, on recovery, does not restart in active state. In other words, a process can restart in active state *only if* it failed in active state. Otherwise, even after all processes have become passive and all processes have become empty, which corresponds to termination in failure-free model, the computation can simply restart by a process failing and recovering later. Intuitively, this makes it impossible to detect termination of the computation. As we show later, even under this strong assumption about computation state on recovery, termination detection problem is impossible to solve.

Intuitively, the underlying computation is said to be terminated if no process is currently active or becomes active in the future. To provide a more formal definition of termination in the crash-recovery model, we first examine the traditional definitions of termination in the failure-free and crash-stop models to determine the fundamental properties that various definitions of termination

have in common. This allows us to establish a sensible definition of termination in any environment. To our knowledge, this is the first attempt at defining the termination condition in the crash-recovery model.

### 3.1 Definitions of Termination in Other Failure Models

Termination depends on the events that can occur in the system. We distinguish between two kinds of events: *application events*, which are generated by the application, and *environment events*, which occur according to the failure model (*e.g.*, crashes or recoveries). Let $\mathcal{C}$ denote the set of all possible partial computations in a given failure model. A definition of termination, say **TERM**, can be considered as a predicate on $\mathcal{C}$.

A computation intuitively has come to an end if it does not produce application events anymore. Thus, a computation $C \in \mathcal{C}$ should satisfy the following:

$$\mathbf{TERM}(C) \Longrightarrow C \text{ contains finitely many application events.} \tag{1}$$

A computation containing finitely many application events is said to be in its *final configuration* after the last of the application events has been executed on every process. Note that, in contrast to application events, environment events can occur even in a terminated configuration. For example, a process may crash after a computation has terminated. Let $\mathcal{C}_{\text{finite}}$ denote the subset of computations in $\mathcal{C}$ that contain only finite number of application events.

**Classical Definitions of Termination.** To formally define termination, we consider classical definitions of termination.

**Definition 1** A computation $C \in \mathcal{C}_{\text{finite}}$ is *classically terminated* (notation: $C \in \mathbf{CT}$ or $\mathbf{CT}(C)$) if and only if the final configuration of $C$ satisfies:

$$\mathbf{CT}(C) \iff (\forall p \in \Pi : p \text{ is passive}) \ \wedge \ (\forall (p,q) \in \Gamma : (p,q) \text{ is empty})$$

The above definition of termination is suitable for the failure-free model. In the crash-stop model, one or more processes may fail by crashing. A process is said to be *correct* if it never fails.

**Definition 2** A computation $C \in \mathcal{C}_{\text{finite}}$ is *correct-restricted classically terminated* (notation: $C \in \mathbf{CRCT}$ or $\mathbf{CRCT}(C)$) if and only if the final configuration of $C$ satisfies:

$$\begin{aligned}\mathbf{CRCT}(C) \iff \quad & (\forall p \in \Pi : p \text{ is passive}) \\ & \wedge (\forall (p,q) \in \Gamma, q \text{ is a correct process} : (p,q) \text{ is empty}).\end{aligned}$$

The correct-restricted definition of termination is particularly suitable for the crash-stop model.

**Backward Compatibility.** If we consider the relation between the classical termination and the correct-restricted termination, we see that **CRCT** is "backward compatible": In the crash-stop model, there may be fault-free runs and, in that case, the definition of correct-restricted termination becomes identical to that of classical termination. Formally expressed, let

- $\mathcal{C}_{\text{fault-free}} \subset \mathcal{C}$ be the set of all computations where no failures happen,
- $\mathcal{C}_{\text{crash-stop}} \subset \mathcal{C}$ be the set of all computations where only crash-stop failures happen and
- $\mathcal{C}_{\text{crash-recovery}} \subset \mathcal{C}$ be the set of all computations where crash-recovery failures happen.

So, backward compatibility means that in the fault-free case **CRCT** is equivalent to **CT**. Formally:

$$\forall C \in \mathcal{C}_{\text{fault-free}} : \mathbf{CT}(C) \iff \mathbf{CRCT}(C). \tag{2}$$

To achieve backward compatibility, it is clear that every new definition of termination **TERM** for the crash-recovery model has to satisfy:

$$\forall C \in \mathcal{C}_{\text{crash-stop}} : \mathbf{CRCT}(C) \iff \mathbf{TERM}(C). \tag{3}$$

**A Necessary and Sufficient Condition.** Termination condition has to be "stable". Once the termination condition holds, it should continue to hold. We define a *prefix relation* $\sqsubset$ on set of computations as follows. The expression $C \sqsubset C'$ means that computation $C$ is a strict prefix of computation $C'$. Informally, computation $C$ is a strict prefix of computation $C'$ ($C \sqsubset C'$) if $C'$ differs from $C$ only in some application events "appended" to $C$. Formally

$$\begin{aligned} C \sqsubset C' \iff \quad & (C_{app} \subset C'_{app}) \\ & \wedge \big( (\prec_C \subseteq \prec_{C'}) \wedge (\forall (e, e') \in \prec_{C'} \colon e \in C' \setminus C \Rightarrow e' \in C' \setminus C) \big), \end{aligned}$$

where $\prec_C$ and $\prec_{C'}$ are the causality relations of $C$ and $C'$ respectively, and $C_{app}$ and $C'_{app}$ are the sets of application events in $C$ and $C'$ respectively. It has to be guaranteed that no "new" event in $C'$ causally precedes an event of computation $C$. An example of the strict prefix relation is given in Figure 3.

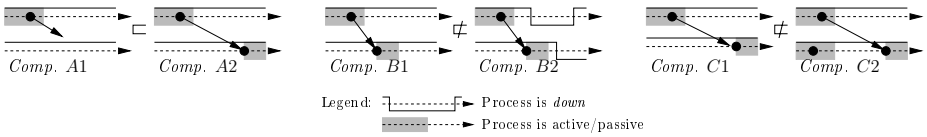


Figure 3: Example of the strict prefix relation

Finally, the *stability property* of termination can be defined as follows:

$$\mathbf{TERM}(C) \iff \nexists C' \in \mathcal{C} : C \sqsubset C' \tag{4}$$

It is important to note that—of course—**CRCT** and **CT** satisfy condition 4 for $\mathcal{C} = \mathcal{C}_{\text{fault-free}}$ and $\mathcal{C} = \mathcal{C}_{\text{crash-stop}}$ respectively.

### 3.2 Definition of Termination in the Crash-Recovery Model

Based on our argument so far, a definition of termination for the crash-recovery model should satisfy the three conditions of finiteness (Eq. 1), backward compatibility (Eq. 3), and stability (Eq. 4). For the crash-recovery model, we can define a process to be correct if it is always-up. Since the termination condition should be stable, it is not sufficient to use the correct-restricted classical termination definition in the crash-recovery model, as explained in the introduction. Note that a process may write its active state to stable storage and therefore recover in an active state.

Let $F$ be a failure pattern in the crash-recovery model. A process $p \in \Pi$ is called *forever-down* at time $t$ if

$$\forall t' \geq t : p \in F(t').$$

Intuitively, a *forever-down* process is a down process that never recovers. Also, a *temporarily-down* process is a process that eventually recovers (but may crash again later). Using the notion of forever-down process, we now give a definition of termination in the crash-recovery model.

**Definition 3** A computation $C \in \mathcal{C}_{\text{finite}}$ is *robust-restricted terminated* (notation: $C \in \mathbf{RRT}$ or $\mathbf{RRT}(C)$) if and only if the final configuration of $C$ satisfies:

$$\begin{aligned}\mathbf{RRT}(C) \iff \quad & (\forall p \in \Pi : p \text{ is passive} \vee p \text{ is } \textit{forever-down}) \\ & \wedge (\forall (p,q) \in \Gamma : \neg(q \text{ is } \textit{forever-down}) \Rightarrow (p,q) \text{ is empty}).\end{aligned}$$

We call processes *robust* if they are so resilient against failures that they, whenever they crash, they eventually recover. The term *robust-restricted terminated* was chosen because we demand that the subset of robust processes should have terminated.

The following two theorems show that robust-restricted termination satisfies the backward compatibility property (property 2) and the stability condition (property 4) that all definitions of termination should satisfy.

**Theorem 4**

$$\forall C \in \mathcal{C}_{\text{crash-stop}} : \mathbf{RRT}(C) \iff \mathbf{CRCT}(C)$$

**Theorem 5**

$$\mathbf{RRT}(C) \iff \nexists C' \in \mathcal{C}_{\text{crash-recovery}} : C \sqsubset C'$$

where $\sqsubset$ is the strict prefix relation on computations.

This, robust-restricted termination is a good definition of termination detection in the crash-recovery model.

### 3.3 Termination Detection Algorithm

Now that we know what it means for a computation to have terminated, we specify formally the properties of the termination detection problem:

- *Liveness.* If the underlying computation satisfies the termination condition, then the termination detection algorithm must announce termination eventually.
- *Safety.* If the termination detection algorithm announces termination, then the underlying computation has indeed terminated.
- *Non-Interference.* The termination detection algorithm must not influence the underlying computation.

To avoid confusion, we refer to messages sent by the underlying computation as *application messages* and messages sent by the termination detection algorithm as *control messages*.

## 4 Impossibility of Termination Detection and its Consequences

We assume that the processes have access to failure detector modules which observe the occurrence of failures in the system. Failure detectors are defined as general functions of the failure pattern, including functions that may provide information about future failures. Of course, such failure detectors cannot be implemented in the real world. Delporte-Gallet *et al.* [4] introduced the notion of *realistic* failure detector. A failure detector is called realistic if it cannot guess the future behavior of the processes. In this work, we restrict ourselves to *realistic failure detectors*. To determine, if an execution is *robust-restricted terminated* it may be necessary to decide whether a currently down process is temporarily-down or forever-down. The two kinds of processes only differ in their future behavior. As a result, we postulate that no realistic failure detector can distinguish between a temporarily-down process and a forever-down process.

**Lemma 6** No *realistic failure detector* can decide at time $t$ whether a process $p$ that is currently down is *temporarily-down* or *forever-down* in the crash-recovery model.

Based on above lemma, it is possible to show that the termination detection problem cannot be solved in the crash-recovery model using only a realistic failure detector. This can be proved by reducing the problem of building a non-realistic failure detector to the problem of solving termination detection.

**Theorem 7** *A non-realistic failure detector is necessary for solving termination detection in the crash-recovery model.*

The next result is a direct consequence of Theorem 7.

**Corollary 8** The termination detection problem cannot be solved in the crash-recovery model using only a realistic failure detector.

### 4.1 From Impossibility to Solvability

Two common ways to circumvent the impossibility result of Corollary 8 are to restrict the failure model and weaken the problem. We argue that both are necessary here. As defined in Section 2.3, there are four classes of processes in the crash-recovery model: always-up, eventually-up, eventually-down and unstable processes. By examining these four classes with regard to their potential to impede the solvability of termination detection the class of unstable processes causes most harm. Always-up, eventually up or down processes do not "cause trouble"—after finite time their behavior becomes predictable. But the only predictable property of unstable processes is that they always stay unpredictable. Thus, we restrict the crash-recovery model by assuming that there are no unstable processes.

**Definition 9** Now we regard the crash-recovery model as defined in Section 2.3 and make the additional assumption that there are *no unstable processes*. We refer to the resulting failure model as *stabilizing crash-recovery model*.

A simple implication of this restriction is that, in the stabilizing crash-recovery model, eventually all crashed processes are forever-down. This can be stated as the following lemma:

**Lemma 10** Eventually the *stabilizing crash-recovery* model changes to *crash-stop* model.

To weaken the problem, we weaken the safety property of the termination detection algorithm. Specifically, a termination detection algorithm is allowed to announce termination falsely albeit only a finite number of times. In other words, a finite number of times the termination detection algorithm may announce termination even through the underlying computation has not yet terminated, and revoke the termination announcement. Eventually the algorithm "stabilizes" and correctly detects termination.

**Definition 11** An algorithm that solves the *stabilizing termination detection problem* has to satisfy the following properties:

- *Liveness.* If the underlying computation has terminated, then eventually the termination detection algorithm announces termination and henceforth termination is not revoked.
- *Eventual Safety.* Eventually, if the termination detection algorithm announces termination, then the underlying computation has indeed terminated.

We use the notion of stabilization at two places: first, to restrict our problem to *stabilizing* termination detection, and second, to restrict the failure model to *stabilizing* crash-recovery model. Both assumptions are necessary as we now show: (1) Termination detector in the crash-recovery model: Not solvable (Corollary 8). (2) Termination detection in the *stabilizing* crash-recovery model: when a process crashes, a realistic failure cannot distinguish between whether the crash is temporary or permanent. If it assumes that crash is temporary (but it actually is permanent), then termination is never announced and the liveness property is violated. On the other hand, if it assumes that crash is permanent (but it actually is temporary), termination is announced prematurely and the safety property is violated. (3) *Stabilizing* termination detection in the *stabilizing* crash-recovery model: we provide an algorithm in the next section. (4) *Stabilizing* termination detection in the crash-recovery model: Assume a computation which never terminates because at least one active process crashes and recovers infinitely often. As a result, the termination detection algorithm will never cease announcing termination erroneously, because it expects the unstable process to "stabilize"—that is, to eventually cease crashing/recovering.

Therefore, our approach to solve *stabilizing* termination detection in the *stabilizing* crash-recovery model is a reasonable approach. In the next section we solve stabilizing termination detection in the stabilizing crash-recovery model using a failure detector from class $\diamondsuit\mathcal{P}_{\text{cr}}$. The next theorem shows that this kind of failure detector is also necessary. Hence, the failure detector $\diamondsuit\mathcal{P}_{\text{cr}}$ is the weakest one for solving stabilizing termination detection in the stabilizing crash-recovery model. This can be shown by reducing the problem of stabilizing termination detection to $\diamondsuit\mathcal{P}_{\text{cr}}$.

**Theorem 12** *A failure detector from class $\diamondsuit\mathcal{P}_{\text{cr}}$ is necessary for solving stabilizing termination detection in the stabilizing crash-recovery model.*

## 5 Solving Stabilizing Termination Detection

In this section, we develop an algorithm for solving the stabilizing termination detection problem in the stabilizing crash-recovery model. It turns out that since we are solving a weaker version of the termination detection problem in more restricted crash-recovery model, we can *weaken* some of the assumptions we made earlier. First, we assume that a process, on recovery, may start in active state only if it crashed in active state. This assumption may be weakened to: a process, on recovery, may start in active or passive state irrespective of the state in which it crashed. The actual state on recovery depends on the the application-specific recovery mechanism including the extent to which the application utilizes stable storage to log its state during execution. Second, we assume that channel does not duplicate any message. This assumption may be weakened to: a channel may duplicate a message finite number of times. With this weakened assumption, our channel can easily implemented on top of a fair-lossy channel using retransmissions and acknowledgments.

### 5.1 Algorithm Ideas

The main idea of our termination detection algorithm is that every process saves information about all messages it sends and receives. If the knowledge of all processes is combined, then the total set of messages in transit may be computed. Messages which are in transit towards a crashed process are assumed to be lost. If such a message is delivered anyhow, then the corresponding channel may have been incorrectly assumed to be empty as a result of which termination may have announced prematurely. The termination announcement is then revoked. Finally, the properties of the *stabilizing* crash-recovery model guarantee that eventually erroneous announcements of termination will end.

The second idea is that every process is responsible for its own state and the state of all its incoming channels. If a process becomes passive and it believes all its incoming channels to be empty, then it proposes—by using a broadcast primitive—the announcement of termination. If a process has received such a termination announcement proposal from all live process, it announces termination. For the termination detection algorithm, we use a *best-effort* broadcast primitive [2]. Informally, best-effort broadcast guarantees that a broadcast message is correctly delivered to all processes which are currently up and do not crash while executing the broadcast protocol. All currently up processes agree on the delivered message. Of course, all messages are delivered only once and no message is created by the best-effort broadcast primitive.

The whole system consists of (1) the underlying computation $C$ which is observed with respect to termination, (2) the superimposed and crash-recovery tolerant termination detection algorithm $A$ we develop now, and (3) a failure detector $D$ of class $\Diamond\mathcal{P}_{\text{cr}}$. If a passive process delivers a message, then it executes an $\langle$becoming active$\rangle$ event. We assume that all events that are executed satisfy the following: the corresponding instructions are executed within one time unit (atomically).

### 5.2 Algorithm in Words

Every processes saves all its sent and received messages in volatile storage. (It is not necessary to store an entire message but rather it is sufficient to store some uniquely identifying information about the message.) For this purpose a process $p_i$ maintains two vector variables $sent_i[1..n]$ and $rcvd_i[1..n]$ of type "message set". Messages sent by process $p_i$ to process $p_j$ are stored in $sent_i[j]$, whereas messages received by $p_i$ from $p_j$ are stored in $rcvd_i[j]$. To minimize access to stable storage, every process saves messages only up to its next crash. Upon recovery it initializes all its variables.

If a process $p_i$ detects the crash of a process $p_j$, then $p_i$ clears the content of $sent_i[j]$. Upon becoming passive, a process $p_i$ broadcasts its vector $sent_i$ to all processes. Thereafter the vector $sent_i$ does not change as long as process $p_i$ remains passive (because passive processes do not send messages).

Whenever a process detects recovery of another process (via its failure detector), it again broadcasts its vector *sent* to all processes if it is passive. If a

process $p_i$ receives such a vector $sent_j$ broadcast by process $p_j$, then $p_i$ saves the messages contained in $sent_j[i]$ in $sent\_to\_me_i[j]$, overwriting the previous contents of $sent\_to\_me_i[j]$. This ensures that $p_i$ does not wait for messages sent by $p_j$ that may have been lost while $p_i$ was down. By comparing the variables $sent\_to\_me_i$ and $rcvd_i$, process $p_i$ determines whether there are any messages in transit towards it. Once a passive process believes that all its incoming channels with live processes have become empty, it broadcasts a TERM signal to all processes. By sending such a TERM signal, a process proposes to all other processes that termination should be announced.

Finally, a process announces termination once it has received TERM signals from all live processes through a termination event ⟨TERMINATION⟩. On becoming active, a process broadcasts a NO_TERM signal if it has proposed TERM earlier. Because this algorithm solves *stabilizing* termination detection, prematurely announced terminations have to be revoked. The revocation event ⟨NO_TERMINATION⟩ is announced if the recovery of a process is detected or if an application message is received. At the delivery of a NO_TERM signal, termination is also revoked. A formal description of the algorithm is given in the appendix (Algorithm B.1).

**Some Issues.** The basic algorithm described above (Algorithm B.1) has some shortcomings which have to be addressed for it to work correctly. First, we have to ensure that an entry of vector *sent_to_me* is overwritten on receiving a *sent* vector only if the entry in *sent* vector is "newer" than the corresponding entry in *sent_to_me* vector. Second, a process may not detect all crashes of another process. As a result, it may not reset the entry for that process in its *sent* vector. To understand this better, consider the following scenario. Process $p_i$ sends a message $m$ to process $p_j$. Before $p_j$ receives $m$, it crashes and $m$ arrives while $p_j$ is down. If $p_i$ does not reset $sent_i[j]$ (which purges $m$), $p_j$ will continue to believe that $m$ is in transit towards it. Third, proposal to announce termination by various processes should be *consistent* with each other. Specifically, let $e_i$ and $e_j$ be the events when $p_i$ and $p_j$, respectively, propose to announce termination. Let $a_i$ and $b_i$ be the *last* passive events on $p_i$ satisfying $a_i \prec e_i$ and $b_i \prec e_j$, respectively. Termination should be announced only if $a_i = b_i$. If $a_i \prec b_i$, then $p_i$'s proposal is based on "older" state of $p_i$ than $p_j$'s proposal. A similar inconsistency exists if $b_i \prec a_i$. Testing for consistency among termination proposals ensures that the algorithm satisfies the following desirable properties: (1) if no process fails during an execution and there are no false suspicions, our algorithm satisfies the safety property, that is, it never announces false termination, and (2) even if the computation never terminates, our algorithm announces and revokes termination only a finite number of times.

These issues can be addressed by maintaining a vector $current_i$ on each process $p_i$ whose each entry is a tuple. The first entry of $current_i[j]$ denotes $p_i$'s estimate of $p_j$ current incarnation number. The second entry of $current_i[j]$ denotes $p_i$'s estimate of the number of application messages that $p_j$ has sent so far. The vector is stored in a process' volatile storage. Every message sent by a

process is timestamped with this vector. It can be shown that, with this modification, all the issues mentioned above can be addressed using this vector. For example, whenever $p_i$'s estimate of $p_j$'s incarnation number changes, $p_i$ assumes that it may have missed $p_j$'s crash and resets $sent_i[j]$. This vector can also be used to appropriately timestamp a proposal by a process for termination announcement so that consistency among termination announcement proposals by various processes can be tested.

**Theorem 13** *Our algorithm (Algorithm B.1) solves the stabilizing termination detection problem in a stabilizing crash-recovery system using a failure detector from class* $\diamondsuit\mathcal{P}_{\mathrm{cr}}$.

## 6 Conclusion

In this paper, we have studied the termination detection problem under the crash-recovery model. We have identified a necessary and sufficient condition that a termination condition should satisfy and have shown that no termination detection algorithm can solve the problem without knowing the future failure pattern. We have also developed a stabilizing algorithm for solving the termination detection problem under the stabilizing crash-recovery model.

## References


1. Marcos Kawazoe Aguilera, Wei Chen, and Sam Toueg. Failure Detection and Consensus in the Crash-Recovery Model. In *DISC '98*, pages 231–245, London, UK, 1998. Springer-Verlag.
2. Romain Boichat and Rachid Guerraoui. Reliable Broadcast in the Crash-Recovery Model. In *SRDS'00*, pages 32–41, Nürnberg, Germany, October 2000. IEEE Computer Society.
3. Tushar Deepak Chandra and Sam Toueg. Unreliable Failure Detectors for Reliable Distributed Systems. *J. ACM*, 43(2):225–267, 1996.
4. Carole Delporte-Gallet, Hugues Fauconnier, and Rachid Guerraoui. A Realistic Look at Failure Detectors. In *DSN '02: Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pages 345–353, Washington, DC, USA, 2002. IEEE Computer Society.
5. Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, 32(2):374–382, 1985.
6. Felix C. Gärtner and Stefan Pleisch. (Im)Possibilities of Predicate Detection in Crash-Affected Systems. In *WSS '01: Proceedings of the 5th International Workshop on Self-Stabilizing Systems*, pages 98–113, London, UK, 2001. Springer-Verlag.
7. Ten-Hwang Lai and Li-Fen Wu. An $(n-1)$-Resilient Algorithm for Distributed Termination Detection. *IEEE Transactions on Parallel and Distributed Systems*, 6(1):63–78, 1995.
8. Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

9. Matthias Majuntke. Termination Detection in Systems Where Processes May Crash and Recover. Diploma Thesis, RWTH Aachen University, 2006. `http://pi1.informatik.uni-mannheim.de/diploma/pdf/16/TDiCRM_final.pdf`.
10. Friedemann Mattern. Algorithms for Distributed Termination Detection. *Distributed Computing*, 2(3):161–175, 1987.
11. Jayadev Misra. Detecting Termination of Distributed Computations Using Markers. In *Proceeding of the Second Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 290–294, 1983.
12. N. Mittal, F. Freiling, S. Venkatesan, and L. D. Penso. Efficient Reduction for Wait-Free Termination Detection in a Crash-Prone Distributed System. In *19th International Symposium on Distributed Computing (DISC)*, pages 93–107, Cracow, Poland, September 2005.
13. L. Rodrigues and M. Raynal. Atomic Broadcast in Asynchronous Crash-Recovery Distributed Systems. In *ICDCS 2000*, page 288, Washington, DC, USA, 2000. IEEE Computer Society.
14. A. Shah and S. Toueg. Distributed Snapshots in Spite of Failures. Technical Report TR84-624, Department of Computer Science, Cornell University, Ithaca, NY, USA, July 1984.
15. Yu-Chee Tseng. Detecting Termination by Weight-Throwing in a Faulty Distributed System. *Journal of Parallel and Distributed Computing*, 25(1):7–15, 1995.
16. S. Venkatesan. Reliable Protocols for Distributed Termination Detection. *IEEE Transactions on Reliability*, 38(1):103–110, April 1989.
17. L.-F. Wu, T.-H. Lai, and Y.-C. Tseng. Consensus and Termination Detection in the Presence of Faulty Processes. In *Proceedings of the International Conference on Parallel and Distributed Systems*, pages 267–274, Hsinchu, Taiwan, December 1992.

# A Omitted Proofs

### Proof of Theorem 4

PROOF SKETCH: At first, we show that in the crash-stop model down processes are equivalent to forever-down processes. Then, we consider the definitions of **RRT** and **CRCT**. These definitions can be transformed into each other if "down" is replaced by "forever-down" and vice versa.

1. $p$ down in $C \iff p$ forever-down in $C$
   PROOF: Follows from $C \in \mathcal{C}_{\text{crash-stop}}$. □
2. $\mathbf{RRT}(C) \implies \mathbf{CRCT}(C)$
   PROOF:
   2.1. All processes in $C$ are passive or forever-down and all channels towards processes which are not forever-down are empty.
   PROOF: Follows from the definition of $\mathbf{RRT}(C)$ (Definition 3). □
   2.2. Q.E.D.
   PROOF: Follows from step 1 and step 2.1 ("forever-down" can be replaced by "down" in step 2.1). □
3. $\mathbf{CRCT}(C) \implies \mathbf{RRT}(C)$
   PROOF:
   3.1. All processes in $C$ are passive or down and all channels towards processes which are not down are empty.
   PROOF: Follows from the definition of $\mathbf{CRCT}(C)$ (Definition 2). □
   3.2. Q.E.D.
   PROOF: Follows from step 1 and step 3.1 ("down" can be replaced by "forever-down" in step 3.1). □
4. Q.E.D.
   PROOF: Follows from steps 2 and 3. □

### Proof of Theorem 5

PROOF SKETCH: This proof is done by two indirect proofs. If there is a computation $C'$ where $C$ is a strict prefix of $C'$, then $C$ is not robust-restricted terminated. This is because application events may be appended to computation $C$. Conversely, if computation $C$ is not robust-restrited terminated, then it may be extended to a computation $C'$ with $C \sqsubset C'$.

1. $\mathbf{RRT}(C) \implies \nexists C' \in \mathcal{C}_{\text{crash-recovery}} : C \sqsubset C'$
   PROOF:
   1.1. ASSUME: $\exists C' \in \mathcal{C}_{\text{crash-recovery}} : C \sqsubset C'$
   PROVE: $\neg\mathbf{RRT}(C)$
   PROOF:
   1.1.1. $C_{app} \subset C'_{app}$
   PROOF: Follows from the definition of strict prefix. □
   1.1.2. $\exists \tilde{e} \in C'_{app} \setminus C_{app}$
   PROOF: Follows from step 1.1.1. □
   1.1.3. $\exists (e, \tilde{e}) \in \prec_{C'} : e \in C$

PROOF: Follows from step 1.1.2 and the definition of strict prefix. □

1.1.4. CASE: $\exists p \in \Pi : p$ is active-up in the final configuration of $C$.
PROOF: Follows from step 1.1.3 and $e$ and $\tilde{e}$ occur on the same process $p$. □

1.1.5. CASE: Message is in transit.
PROOF: Follows from step 1.1.3 and $\tilde{e}$ is a receive event. □

1.1.6. Q.E.D.
PROOF: Steps 1.1.4 and 1.1.5 cover all cases. □

1.2. Q.E.D.
PROOF: Follows from step 1.1 and indirect proof $((A \Rightarrow B) \Leftrightarrow (\neg B \Rightarrow \neg A))$. □

2. $\nexists C' \in \mathcal{C}_{\text{crash-recovery}} : C \sqsubset C' \Longrightarrow \mathbf{RRT}(C)$
PROOF:
2.1. ASSUME: $\neg\mathbf{RRT}(C)$
PROVE: $\exists C' \in \mathcal{C}_{\text{crash-recovery}} : C \sqsubset C'$
PROOF:
2.1.1. CASE: Exists process $p$ in the final configuration of $C$ which is active-up.
PROOF: Follows from the definition of $\neg\mathbf{RRT}(C)$. □

2.1.2. CASE: Exists non-empty channel towards a non forever-down process in the final configuration of $C$.
PROOF: Follows from the definition of $\neg\mathbf{RRT}(C)$. □

2.1.3. An active-up process produces events.
PROOF: Follows from the definition of active-up process. □

2.1.4. A message in transit towards a non forever-down process is possibly delivered and produces a receive event.
PROOF: Follows from the definition of a reliable channel. □

2.1.5. Q.E.D.
PROOF: Follows from steps 2.1.1, 2.1.2, 2.1.3, 2.1.4 and the definition of strict prefix. □

2.2. Q.E.D.
PROOF: Follows from step 2.1 and indirect proof. □

3. Q.E.D.
PROOF: Follows from steps 1 and 2. □

**Proof of Lemma 6**

PROOF SKETCH: We assume a realistic failure detector that may decide whether a process $p$ is down or forever-down. Then we suppose two failure patterns which are the same up to a particular time $t$. At time $t$ process $p$ is down in the first failure pattern and forever-down in the second one. Because the failure detector distinguishes down from forever-down processes the failure detector histories are different at time $t$. This contradicts the assumption that the failure detector is realistic.

1. ASSUME: (1) $D$ is a realistic failure detector.

(2) $D$ can decide at time $t$ whether process $p$ is down or forever-down.

PROVE: false

PROOF:

1.1. There exist two failure patterns $F_1$ and $F_2$ with the following properties regarding process $p$:
   (a) Up to time $t$, $F_1$ and $F_2$ are the same, formally $\forall t' \leq t : F_1(t') = F_2(t')$.
   (b) $p$ crashes at time $t$ in $F_1$ and $F_2$, formally $p \in F_1(t) \wedge p \in F_2(t)$.
   (c) $p$ eventually recovers in $F_1$, formally $\exists \tilde{t} > t : p \notin F_1(\tilde{t})$.
   (d) $p$ is forever-down in $F_2$, formally $\forall \tilde{t} \geq t : p \in F_2(\tilde{t})$.

   PROOF: $F_1$ and $F_2$ are valid failure patterns in the crash-recovery model. □

1.2. $D$ will declare $p$ as forever-down in $F_2$ but not in $F_1$ at time $t$.

   PROOF: Follows from assumption 1.(2) and step 1.1. □

1.3. There exist failure detector histories $H_1 \in D(F_1)$ and $H_2 \in D(F_2)$ such that $H_1 \in D(F_1) \neq H_2 \in D(F_2)$.

   PROOF: Follows from step 1.2 and the definition of a failure detector. □

1.4. $D$ is not a realistic failure detector.

   PROOF: From step 1.1.(a) follows that $F_1$ and $F_2$ are the same up to time $t$. From step 1.3 follows that there exist two failure detector histories which are different at time $t$. The step follows from the definition of a realistic failure detector. □

1.5. Q.E.D.

   PROOF: Step 1.4 contradicts assumption 1.(1). □

2. Q.E.D.

   PROOF: Follows from step 1 and proof by contradiction. □

**Proof of Theorem 7**

PROOF SKETCH: We implement a non-realistic failure detector by using a termination detection algorithm for the crash-recovery model. Every process runs a computation with only one process which is always active. Additionally, every process executes $n$ instances of the termination detection algorithm, where every instance observes one computation of every other process. If the termination detection algorithm announces one computation to be terminated, then the corresponding process is forever-down. Namely, the computation of a process $p_i$ only terminates if $p_i$ is forever-down, because all processes are supposed to be always active. By Lemma 6, a failure detector that recognizes forever-down processes is not realistic.

1. Any termination detection algorithm can be transformed into a non-realistic failure detector.

   PROOF:

   1.1. Let $T_{CRM}$ be any termination detection algorithm in the crash-recovery model.

      PROOF: Exists from step 1. □

1.2. Consider the following construction:
- Every process $p_i$ runs a computation $C_i$ consisting of just process $p_i$ which is always active.
- All processes start $n$ instances $T^i_{CRM}$ of $T_{CRM}$, one for every computation $C_i$.
- Upon announcement of termination of $T^i_{CRM}$, the process declares $p_i$ as forever-down.

If the construction announces $p_i$ as forever-down, then $p_i$ in fact is forever-down.

PROOF: Follows from the safety property of $T_{CRM}$ which is a correct algorithm (step 1.1). □

1.3. If $p_i$ is forever-down, then eventually the construction in step 1.2 will announce $p_i$ as forever-down.

PROOF: Follows from the liveness property of $T_{CRM}$ and step 1.1. □

1.4. The failure detector implemented in step 1.2 is non-realistic.

PROOF: The failure detector can decide at time $t$ whether $p$ is down or forever-down in the crash-recovery model (steps 1.2 and 1.3). The step follows from Lemma 6. □

1.5. Q.E.D.

PROOF: Follows from step 1.4. □

2. Q.E.D.

PROOF: Follows from step 1 and definition of "necessary". □

**Proof of Theorem 13**

PROOF SKETCH: We will show that the liveness and the eventual safety property of stabilizing termination detection are satisfied. The liveness proof is straightforward. To prove eventual safety, we will construct a finite point in time where the safety property holds. Naturally non-interference is satisfied because the processes do not influence the underlying computation.

1. ASSUME: **RRT**($C$) holds.
   PROVE: Eventually, all live processes announce ⟨TERMINATION⟩ and henceforth never revoke termination.

   PROOF:

   1.1. All non-forever-down processes are passive and all channels towards non-forever-down processes are empty.

   PROOF: Follows from the definition of **RRT**. □

   1.2. ∃ point in time $t_1$ when the last process $p_l$ performs its final recovery.

   PROOF: Follows from the definition of the stabilizing crash-recovery model. □

   1.3. After $t_1$ all down processes are forever-down.

   PROOF: Follows from step 1.2. □

   1.4. At time $t_1$ process $p_l$ broadcasts its $sent_l$ variable.

   PROOF: Follows from steps 1.1 and 1.2 and lines 25-28. □

   1.5. ∃ time $t_2 > t_1$: After $t_2$ all other live processes broadcast their *sent* variables.

PROOF: Follows from steps 1.1, 1.2 and the correctness of the failure detector and lines 11-17. □

1.6. $\exists$ time $t_3 > t_2$: After $t_3$ all live processes have received *sent* variables from all other live processes.
PROOF: Follows from the termination property of broadcast and steps 1.4 and 1.5. □

1.7. After $t_3$: Every live process knows which messages have been sent to it and which messages it has received since its final recovery.
PROOF: Follows from step 1.6 and lines 36-38 and 22-24. □

1.8. After time $t_3$: Every live process broadcasts a TERM signal.
PROOF:

1.8.1. After $t_3$: Every live process is passive.
PROOF: Follows from step 1.1. □

1.8.2. After $t_3$: All messages sent to a live process after its final recovery have been received.
PROOF: Follows from step 1.1. □

1.8.3. Q.E.D.
PROOF: Follows from steps 1.8.1, 1.8.2 and 1.7 and lines 43-48. □

1.9. $\exists$ time $t_4 > t_3$: After $t_4$ every live process has delivered a TERM signal from every live process.
PROOF: Follows from step 1.8 and properties of broadcast. □

1.10. After $t_4$: Every live process announces ⟨TERMINATION⟩.
PROOF: Follows from step 1.9 and lines 49-51 and 52-54. □

1.11. After $t_4$: No live process revokes termination.
PROOF: Follows from step 1.2 and lines 11-12, step 1.1 and lines 18-19, 29-35 and 39-42. □

1.12. Q.E.D.
PROOF: Follows from steps 1.10 and 1.11 and the definition of stabilizing termination detection. □

2. PROVE: $\exists$ finite point in time where henceforward holds: If a live process announces ⟨TERMINATION⟩, then **RRT** holds.
PROOF:

2.1. $\exists$ point in time $t_1$: After $t_1$ all live processes have performed their final recovery.
PROOF: Follows from the definition of the stabilizing crash-recovery model. □

2.2. After $t_1$: For every process $p_i$ there is only a finite number of messages sent to $p_i$ before $p_i$'s final recovery, which have not yet been delivered to $p_i$ ($p_i$-spurious messages).
PROOF: Follows from step 2.1 and only a finite number of messages may be sent in finite time. □

2.3. $\exists$ time $t_2 > t_1$: After $t_2$ no spurious messages are in transit.
PROOF: Follows from step 2.2 and the channel property that eventually all messages are delivered or lost. □

2.4. For any time after $t_2$: If $p_l$ announces ⟨TERMINATION⟩, then **RRT** holds.

PROOF:

2.4.1. $p_l$ has delivered a TERM signal from every live process.

PROOF: Follows from lines 49-54. □

2.4.2. All live processes have broadcast TERM.

PROOF: Follows from step 2.4.1 and the properties of broadcast and line 45. □

2.4.3. All live processes are passive.

PROOF: Follows from line 44 and step 2.4.2. □

2.4.4. If $sent\_to\_me_l[j] \backslash rec_l[j] = \emptyset$, then all channels towards $p_l$ are empty.

PROOF:

2.4.4.1. $p_l$ has received variable $sent_j$ from every live process $p_j$ after $p_j$'s final recovery.

PROOF: Follows from step 2.1 and lines 36-38 and 25-28 and the initial value of *sent_to_me*. □

2.4.4.2. $sent\_to\_me_l[j]$ contains all messages sent to $p_l$ by $p_j$ after the final recovery of $p_l$.

PROOF: Follows from steps 2.4.4.1 and 2.3. □

2.4.4.3. Q.E.D.

PROOF: Follows from steps 2.4.4.1 and 2.4.4.2 and the definition of empty. □

2.4.5. Q.E.D.

Follows from steps 2.4.3 and 2.4.4. □

2.5. Q.E.D.

PROOF: Follows from step 2.4 and the eventual safety property. □

3. Q.E.D.

PROOF: Follows from steps 1 and 2 and the definition of stabilizing termination detection. □

# B Formal Description of the Algorithm

**Algorithm B.1** Superimposed Termination Detection

On every monitor process $p_i$:
**Variables:**
1: $failed_i$ **set** of processes **init** $\emptyset$
2: $rcvd_i[1..n]$ **array** of message sets **init** empty {*received messages*}
3: $sent_i[1..n]$ **array** of message sets **init** empty {*sent messages*}
4: $passive_i$ **boolean init** defined by accordant event
5: $term_i[1..n]$ **array** of **boolean init** false
6: $sent\_to\_me_i[1..n]$ **array** of message sets **init** $\perp$

**Process** $p_i$:
7: **upon** $\langle crash_i(p_j)\rangle$ **do**
8: $failed_i := failed_i \cup p_j$
9: $sent_i[j] := \emptyset$
10: **end do**

11: **upon** $\langle recover_i(p_j)\rangle$ **do**
12: $\langle$NO_TERMINATION$\rangle$
13: $term_i[j] :=$ false
14: **if** $(passive_i =$ true$)\wedge(i \neq j)$ **then**
15: $beb\_Broadcast(sent_i)$
16: **end if**
17: **end do**

18: **upon** $\langle receive\_app_i(m), p_j\rangle$ **do**
19: $\langle$NO_TERMINATION$\rangle$
20: $rcvd_i[j] := rcvd_i[j] \cup m$
21: **end do**

22: **upon** $\langle send\_app_i(m), p_j\rangle$ **do**
23: $sent_i[j] := sent_i[j] \cup m$
24: **end do**

continued on next page

**Algorithm B.1** Superimposed Termination Detection (cont.)

```
25: upon ⟨p_i becoming passive⟩ do
26:    passive_i := true
27:    beb_Broadcast(sent_i)
28: end do

29: upon ⟨p_i becoming active⟩ do
30:    passive_i := false
31:    if (term_i[i] = true) then
32:       term_i[i] := false
33:       beb_Broadcast(NO_TERM)
34:    end if
35: end do

36: upon ⟨beb_Deliver_i(s), p_j⟩ do
37:    sent_to_me_i[j] := s[i]
38: end do

39: upon ⟨beb_Deliver_i(NO_TERM), p_j⟩ do
40:    term_i[j] := false
41:    ⟨NO_TERMINATION⟩
42: end do

43: upon ⟨∀p_j ∉failed_i : sent_to_me_i[j]\rcvd_i[j] = ∅⟩ do
44:    if (passive_i = true) then
45:       beb_Broadcast(TERM)
46:       term_i[i] := true
47:    end if
48: end do

49: upon ⟨beb_Deliver_i(TERM), p_j⟩ do
50:    term_i[j] := true
51: end do

52: upon ⟨∀p_j ∉failed_i : term_i[j] = true⟩ do
53:    ⟨TERMINATION⟩
54: end do

55: upon ⟨p_i itself recovers⟩ do
56:    for k = 1...n do
57:       rcvd_i[k] := ∅
58:       sent_i[k] := ∅
59:       term_i[k] := false
60:       sent_to_me_i[k] := ⊥
61:    end for
62:    trigger either ⟨p_i becoming passive⟩ or ⟨p_i becoming active⟩
63:    initialize failed_i according to current failure detector output
64: end do
```