

# Safe Termination Detection in an Asynchronous Distributed System when Processes may Crash and Recover

Neeraj Mittal<sup>1</sup>, Kuppahalli L. Phaneesh<sup>1\*</sup>, and Felix C. Freiling<sup>2</sup>

<sup>1</sup> Department of Computer Science, The University of Texas at Dallas, Richardson, TX 75083, USA

<sup>2</sup> Department of Computer Science, University of Mannheim, D-68131 Mannheim, Germany

**Abstract.** The termination detection problem involves detecting whether an ongoing distributed computation has ceased all its activities. We investigate the termination detection problem in an asynchronous distributed system under crash-recovery model. It has been shown that the problem is impossible to solve under crash-recovery model in general. We identify two conditions under which the termination detection problem can be solved in a safe manner. We also propose algorithms to detect termination under the conditions identified.

## 1 Introduction

The termination detection problem arises when a distributed computation terminates *implicitly*, that is, once the computation ceases all its activities, no single process knows about the termination [1]. Therefore a separate algorithm has to be run to detect termination of the computation. To abstract from concrete applications in message-passing systems, the distributed computation is typically modeled using the following four 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. Roughly speaking, a termination detection algorithm must detect termination once the computation which follows these rules has ceased all its activities.

Termination detection has been studied quite extensively for the last few decades, initially under the failure-free model (*e.g.*, [2–6], see [7] for a survey). When both processes and channels are reliable, the termination condition for a distributed computation can be defined as follows [2, 3]: A computation is said to have terminated if all processes have become passive and all channels have become empty.

Termination detection has been studied relatively well in the crash-stop model as well (*e.g.*, [8–12]). In the crash-stop model, once a process crashes,

---

\* The author is currently working at Microsoft Inc., Seattle, Washington, USA.

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, the termination condition for a distributed computation can be defined as follows [8, 9]: A computation is said to have terminated if all up processes have become passive and all channels towards up processes have become empty.

Wu *et al.* [13] establish that, to be able to detect termination in the crash-stop model, it must be possible to *flush* the incoming channel of an up process with a down process. A channel can be flushed using either *return-flush* [8] or *fail-flush* [9] primitive. Both primitives allow an up process to ascertain that its incoming channel with the crashed process has become empty. In the absence of the two primitives, Tseng suggests *freezing* the channel from a down process to an up process [10]. When an up process freezes its channel with a down 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.) In this case, a computation is said to have terminated if all up processes have become passive, all channel between up processes have become empty, all channels from down processes to up processes have been frozen [10–12].

In this paper, we investigate the termination detection problem under a more severe failure model, namely *crash-recovery* model. In the crash-recovery model, processes can crash and later recover from a predefined state. To our knowledge, Majuntke [14] was the first to give a definition of termination in the crash-recovery model. Majuntke [14] shows that, if processes can restart in active state on recovery, then it is impossible to detect termination without the ability to predict future behavior of processes (*e.g.*, whether a crashed process will remain crashed forever or will recover in the future). The impossibility result holds even if a process can restart in an active state only if it crashed in an active state. Majuntke [14] also presents a *stabilizing* termination detection algorithm under the condition that there is no process that crashes and recovers an infinite number of times. The algorithm is stabilizing in the sense that it may falsely announce termination and revoke it later. However, false termination announcements and revocations can happen only a finite number of times even if the underlying computation never terminates [14].

Our focus is on developing non-stabilizing safe termination detection algorithms in the crash-recovery model, that is, unlike in [14], our termination detection algorithms are not allowed to revoke a termination announcement (even if the revocation occurs only a finite number of times). We identify two conditions under which termination of a computation can be detected in a *safe* manner, that is, it is possible to devise a termination detection algorithm that never announces false termination.

1. The first condition requires every process to be *eventually reliable*, that is, every process eventually stays up permanently.
2. The second condition requires a crashed process to *always restart in a passive* state (and rejoin the computation via a recovery operation). Further, a process can deliver an application message only if it is sent to its *current* incarnation, that is, only if the sender is aware of all restarts of the destination

process. We ensure the latter by allowing a process to deliver an application message only if the message is exchanged between current incarnations of the two processes (source and destination).

We present an algorithm to detect termination under each of the conditions. The second algorithm uses a new failure detector suitable to solve the termination detection problem in the crash-recovery model. Due to lack of space, proofs of all lemmas and theorems, and formal descriptions of the two algorithms can be found elsewhere [15].

The paper is organized as follows. We present our system model and notations in Sect. 2, derive a definition of a perfect failure detector for the crash-recovery model in Sect. 3 and formally define the termination detection problem in Sect. 4. We identify the two conditions for safe termination detection in Sect. 5. The algorithms for termination detection under the two conditions are described in Sect. 6 and Sect. 7. Finally, we present our conclusions and outline directions for future research in Sect. 8.

## 2 Model and Notation

### 2.1 Distributed System

We assume an asynchronous distributed system consisting of a set of processes, given by  $\Pi = \{p_1, p_2, \dots, p_N\}$ , in which processes communicate by exchanging messages with each other over a communication network. A process changes its state by executing an *event*. The system is asynchronous in the sense that there is no bound on the amount of time a process may take to execute an event or a message may take to arrive at its destination. We do not assume any global clock or shared memory.

There are three kinds of events in the system: *internal event*, *send event* and *receive event*. An event at a process causes the state of the process to be updated. Additionally, a send event causes one or more messages to be sent, whereas a receive event causes a message to be received. Sometimes, we refer to the state of a process as *local state* and the state of a system as *global state*.

We assume that a process executes events sequentially. Therefore events on a process are totally ordered. However, events on different processes are only ordered partially. The partial order between events in the system is given by the *Lamport's happened-before relation* [16] defined as follows. An event  $e$  is said to have happened-before an event  $f$ , denoted by  $e \rightarrow f$ , if

- $e$  and  $f$  are events on the same process and  $e$  was executed before  $f$ , or
- $e$  and  $f$  are send and receive events, respectively, of the same message, or
- there exists an event  $g$  such that  $e \rightarrow g$  and  $g \rightarrow f$ .

We use  $\Rightarrow$  to denote the reflexive closure of  $\rightarrow$ .

## 2.2 Failure Model

We assume that processes are unreliable and may fail by crashing. Further, a crashed process may subsequently recover and resume its operation. While a process is crashed, it does not execute any events. This failure model is referred to as *crash-recovery model*.

In the crash-recovery model, a process may be either *stable* or *unstable*. A process is said to be stable if it crashes (and possibly recovers) only a finite (including zero) number of times; otherwise it is unstable. A stable process can be further classified into two categories: *eventually-up* or *eventually-down* [17]. A process is said to be eventually-up if the process eventually stays up after crashing and recovering a finite number of times; otherwise it is eventually-down. An eventually-up process is said to be *always-up* if it never crashes. Sometimes, eventually-up processes are referred to as *good processes*, and eventually-down and unstable processes are referred to as *bad processes* [17].

A process that is currently operational is called an *up* process, whereas a process that is currently crashed is called a *down* process. We use the phrases “up process” and “live process” interchangeably. Likewise, we use the phrases “down process” and “crashed process” interchangeably.

In the crash-recovery model, in addition to processes, typically, channels are also assumed to be unreliable. We assume *eventually-reliable channels with finite duplication* in this paper. Such channels satisfy the following properties:

- *No creation*:  $p_j$  delivers a message  $m$  only if  $m$  was sent earlier by  $p_i$ ,
- *Finite duplication*:  $p_j$  delivers a message only a finite number of times, and
- *Eventual-reliability*: If  $p_i$  sends a message  $m$  to  $p_j$ , and neither  $p_i$  nor  $p_j$  crashes, then  $p_j$  eventually delivers  $m$ .

An eventually-reliable channel with finite duplication can be implemented on top of a *fair-lossy channel* [18]—a type of unreliable channel providing very weak guarantees—using retransmissions and acknowledgments. We refer to a channel as eventually-reliable if it satisfies no creation, no duplication and eventual-reliability properties. Unless otherwise stated, we assume all channels to be eventually-reliable with finite duplication.

## 2.3 Volatile and Stable Storage

We assume that each process has access to two types of storage mediums: *volatile storage* and *stable storage*. Any data that a process maintains in volatile storage, such as main memory, is lost once the process crashes. On the other hand, data stored in stable storage, such as magnetic disk, is persistent and survives any crashes. However, this persistence comes at the expense of speed. Accessing (reading/writing) stable storage is much slower than accessing volatile storage. As a result, it is desirable to minimize access to stable storage so as to avoid slowing down the system significantly.

## 2.4 Process Incarnations

When a crashed process recovers, we say that the process has a *new incarnation*. At the very least, we use stable storage to distinguish between various incarnations of the same process. Each process maintains an integer in its stable storage that keeps track of its incarnation number, that is, the number of times the process has crashed and recovered. The integer is initially set to 0 for all processes. Whenever a process recovers from a crash, before taking any other action, it reads the value of the integer from its stable storage, increments the value and writes the incremented value back to its stable storage. Observe that it is possible that a process may crash before it is able to write the incremented value back to its stable storage. Clearly, such a recovery is useless for all practical purposes. Therefore we consider a process to be down until it is able to successfully update its incarnation number in its stable storage.

If a process  $p_i$  crashes and the incarnation number of  $p_i$  immediately before the crash was  $x$ , then we say that “incarnation  $x$  of  $p_i$  has crashed”. It is convenient to view process crash and recovery as special kinds of events, namely *crash event* and *recovery event*. We use  $crash_i(x)$  (respectively,  $recovery_i(x)$ ) to denote the crash event (respectively, recovery event) for incarnation  $x$  of process  $p_i$ . We refer to crash and recovery events as *operational events* (as opposed to *program events* that processes execute to change their states). The happened-before relation can be extended to include operational events as well.

We denote the *operational state* of a process (as opposed to *program state* which captures the values of all program variables on the process) using a tuple  $\langle s, x \rangle$  containing two components. The first component, given by  $s$ , indicates the status of the process, that is whether the process is **up** or **down**. The second component, given by  $x$ , indicates the most recent incarnation number of the process. The formal interpretation of the tuple  $\langle s, x \rangle$  is as follows:

- If  $s = \mathbf{up}$ , then the process is currently up and its current incarnation number is  $x$ .
- If  $s = \mathbf{down}$ , then the process is currently down and the most recent incarnation of the process to have crashed is  $x$ .

We assume that  $\mathbf{up} < \mathbf{down}$ . We can now define a less-than relation on operational states of a process as follows:  $\langle s, x \rangle < \langle t, y \rangle$  if either (1)  $x < y$ , or (2)  $x = y$  and  $s < t$ . Observe that the less-than relation as defined totally orders all operational states of a process. As before,  $\leq$  is a reflexive closure of  $<$ . For an operational state  $u = \langle s, x \rangle$ , we use  $u.status$  to refer to the status  $s$  and  $u.number$  to refer to the incarnation number  $x$ . Let  $opstate_i(t)$  denote the operational state of process  $p_i$  at time  $t$ .

Note that it is possible to avoid delivering duplicate messages during an incarnation without using stable storage by logging received messages in volatile storage only.

## 3 Failure Detector for Termination Detection

To solve many important distributed computing problems such as consensus, atomic broadcast and termination detection in an unreliable asynchronous dis-

tributed system, it is sometimes necessary for an up process to know the current status (up or down) of other processes in the system. However, in an asynchronous distributed system, it is not possible to distinguish between a down process and a slow process. To overcome this problem, many solutions to these problems assume the existence of a special device known as *failure detector* [19]. Using a failure detector, a process can maintain its view about the current status (up or down) of other processes in the system. This view might be unreliable and, at any given time, the views at different processes may be different as well. For a failure detector to be useful, these views should eventually be “error-free” and “converge” at good processes. A failure detector can be implemented by making timing assumptions about speeds of processes and delays of messages [19, 20]. The notion of failure detector was originally defined for the crash-stop model (once a process crashes, it never recovers) [19] but has been extended to the crash-recovery model as well (see for example [17]). In this paper, we focus on *realistic failure detectors* which are not capable of predicting the future behavior of a process (*e.g.*, whether a process will stay up forever) [21].

One of the termination detection algorithms we describe in this paper uses a *perfect failure detector* [19] adapted to the crash-recovery model. Informally, a perfect failure detector for the crash-recovery model is responsible for detecting crashes of process incarnations. It satisfies the following properties: (1) *Strong Accuracy*: a process suspects a process incarnation to have crashed only after the incarnation has crashed, and (2) *Strong Completeness*: if a process incarnation has crashed, then eventually every good process permanently suspects the incarnation to have crashed.

The completeness property as stated above is hard to implement in practice. A process may crash immediately after it has updated its incarnation number in the stable storage (but before sending any messages) and no other process in the system will know about the recovery (and hence about the incarnation). Clearly, it is *unreasonable* to expect another process to be able to detect crash of such an incarnation. To address this problem, we define what it means for a process to *know-about* an incarnation. We say a process  $p_i$  knows-about the incarnation  $x$  of process  $p_j$  if there exists an event  $e$  on  $p_i$  such that  $recovery_j(x) \rightarrow e$ . We assume that each process knows-about incarnation 0 of every other process.

Based on the above discussion, we modify the completeness property as follows. It now consists of two parts. First, if some always-up process knows-about a process incarnation and the incarnation has crashed, then eventually every good process permanently suspects the incarnation to have crashed. Second, if some good process permanently suspects a process incarnation to have crashed, then eventually every good process permanently suspects the incarnation to have crashed. We formally model this behavior as follows. The local failure detector at each process  $p_i$  maintains a list, denoted by  $crash-list_i$ , that contains all process incarnations it suspects to have crashed. Each entry in the list is of the form  $\langle i, x \rangle$ , which means that incarnation  $x$  of process  $p_i$  has crashed. Observe that, in practice, it is sufficient for the local failure detector to maintain at most one entry in the list for every process in the system, which corresponds to the *latest* incarnation of the process that it suspects to have crashed. We assume

that  $crash-list_i$  is prefix-closed, that is, if  $\langle j, x \rangle \in crash-list_i$  and  $x \geq 1$ , then  $\langle j, x - 1 \rangle \in crash-list_i$ .

Let  $crash-list_i(t)$  denote the list at process  $p_i$  at time  $t$ . We assume that if  $p_i$  is down at  $t$ , then  $crash-list_i(t) = \emptyset$ . A perfect failure detector satisfies the following properties:

- *Strong Accuracy*: A process suspects a process incarnation to have crashed only if the incarnation has actually crashed. Formally, for all processes  $p_i$  and  $p_j$ ,

$$\langle j, x \rangle \in crash-list_i(t) \Rightarrow \langle \text{down}, x \rangle \leq opstate_j(t)$$

- *Strong Completeness*: It consists of two parts:

1. If at least one always-up process knows-about a process incarnation, and the incarnation has crashed, then eventually every good process permanently suspects the incarnation to have crashed. Formally, for every good process  $p_i$  and for every process  $p_j$ ,

$$\begin{aligned} & (\text{if some always-up process knows about } recovery_j(x)) \wedge \\ & (\langle \text{down}, x \rangle \leq opstate_j(t)) \\ & \Rightarrow \\ & \langle \exists u :: \langle \forall v : v \geq u : \langle j, x \rangle \in crash-list_i(v) \rangle \rangle \end{aligned}$$

2. If some good process permanently suspects a process incarnation to have crashed, then eventually every good process permanently suspects the incarnation to have crashed. Formally, for all good process  $p_i$  and  $p_k$  and for every process  $p_j$ ,

$$\begin{aligned} & \langle \forall w : w \geq t : \langle j, x \rangle \in crash-list_k(w) \rangle \\ & \Rightarrow \\ & \langle \exists u :: \langle \forall v : v \geq u : \langle j, x \rangle \in crash-list_i(v) \rangle \rangle \end{aligned}$$

The accuracy and completeness properties guarantee that if there are no unstable processes in the system, then eventually all good processes agree on which process incarnations have crashed. Our definition of a perfect failure detector allows a process to “lose” its knowledge about crashes of other processes, especially due to its own crash. We do assume, however, that, once a process suspects a process incarnation to have crashed, it continues to do so until it crashes. This can be easily achieved using volatile storage only.

## 4 The Termination Detection Problem

There are many distributed programs which, when executed, generate distributed computations that do not terminate explicitly but rather terminate *implicitly* [1]. In other words, when the computation terminates, it is possible that no process in the system knows that the computation has terminated. In this case, a separate *termination detection algorithm* has to be run to detect termination of the distributed computation. The distributed computation whose termination has to

be detected is typically modeled using the states *active* and *passive* for processes and the rules mentioned in the introduction.

The termination detection problem involves determining whether the computation has ceased all its activities. In other words, no process is currently involved in any activity, and, moreover, no process can become involved in any activity in the future. Any termination detection algorithm should satisfy the following properties:

- *No false termination announcement (safety)*: If the termination detection algorithm announces termination, then the computation has indeed terminated.
- *Eventual termination announcement (liveness)*: Once the computation terminates, the termination detection algorithm eventually announces termination.

For every failure model, it is necessary to define what it means that a computation has terminated. In the crash-recovery model, a process may recover after crashing and resume its activity. Clearly, if a process, on recovery, can restart in any state—active or passive, then, once the termination condition becomes true, no process can crash thereafter. Otherwise, the termination condition can be simply falsified by a process crash and its subsequent recovery in an active state. This definition is too restrictive. Therefore, we assume that a process can restart in an active state on recovery only if it crashed in active state; otherwise, it restarts in a passive state. A process is said to be *forever-down* if it is currently crashed and never recovers from the crash. The termination condition for a distributed computation can be defined as [14]:

**Definition 1 (termination in crash-recovery model).** *A computation is said to have terminated in the crash-recovery model if every process that is not forever-down has become passive, and every channel towards such a process has become empty.*

Observe that the termination condition in the crash-recovery model, as stated above, requires a failure detector to be able to predict the future behavior of a down process, namely whether a down process will recover in the future or stay down permanently. In fact, Majuntke shows in [14] that it is impossible to detect termination of a computation in the crash-recovery model without using a non-realistic failure detector. However, the definition is still reasonable since Majuntke [14] also proves that the above definition of termination is equivalent to the condition that termination is a *stable property*.

In the next section, we investigate conditions under which it is possible to detect termination using only a realistic failure detector such as the one defined in Sect. 3. In contrast to [14], our focus is on deriving termination detection algorithms that are (perpetually) safe and not eventually safe. Specifically, if the termination detection algorithm announces termination, then the computation has, in fact, terminated.

To avoid confusion, we refer to messages exchanged by a distributed computation as *application messages* and those exchanged by a termination detection algorithm as *control messages*.

## 5 Conditions for Safe Termination Detection

One of the reasons why detecting termination in the crash-recovery model is hard is because a crashed process, on recovery, may restart in an active state.

**Lemma 1.** *Assume that: (1) a crashed process, on recovery, may restart in an active state provided it failed in an active state, and (2) at most one process in the system is bad. Then there is no termination detection algorithm that can detect termination of every computation in a safe and live manner.*

Lemma 1 implies that, to be able to detect termination of a computation in a safe manner, we have to weaken at least one of two assumptions, that is, either (1) a crashed process, on recovery, always restarts in a passive state, or (2) all processes in the system eventually stay up permanently. We consider the two one by one. First, assume that processes in the system eventually stay up forever, that is, all processes are eventually reliable. In this case, the termination condition for a distributed computation in the crash-recovery model becomes equivalent to that in the failure-free model. Therefore the first condition under which we investigate the termination detection problem is:

**Condition 1 (eventually reliable processes).** *All processes in the system are good processes.*

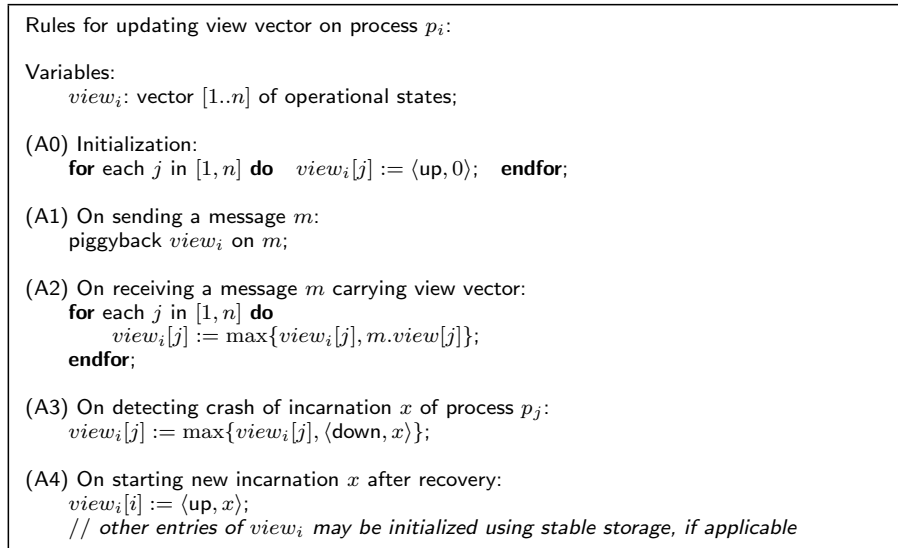
Next, assume that a process, on recovery, always restarts in a passive state. Intuitively, this means that a process, on recovery, cannot start any activity on its own but has to wait to receive an application message from another process. Therefore the second condition under which we investigate the termination detection problem is:

**Condition 2 (passive recovery).** *A crashed process, on recovery, always restarts in a passive state.*

However, the above condition, by itself, does not solve the problem completely. For a message  $m$ , let  $snd(m)$  and  $rcv(m)$  denote the send and receive events, respectively, of  $m$ . Suppose process  $p_i$  sends an application message  $m$  to process  $p_j$ . We say that  $m$  is *old* with respect to incarnation  $x$  of  $p_j$ , where  $x \geq 1$ , if  $recovery_j(x) \not\rightarrow snd(m)$ . In other words, when  $p_i$  sent  $m$ , it did not know about incarnation  $x$  of  $p_j$ . We show that such an old application message may create a problem for a termination detection algorithm.

**Lemma 2.** *Assume that: (1) a crashed process, on recovery, always restarts in a passive state, (2) at most two process in the system are bad, and (3) a process can accept an old application message. Then there is no termination detection algorithm that can detect termination of every computation in a safe and live manner.*

The main idea behind the proof of Lem. 2 is as follows. To tolerate eventually-down processes, it is not sufficient to ensure that all channels towards up processes are empty before announcing termination. It may also be necessary to



**Fig. 1.** Rules for updating view vector on a process.

ensure that all channels between down processes are empty (unless, of course, all down processes stay down permanently which requires knowledge about the future). Clearly, it is reasonable to assume that the channel from  $p_i$  to  $p_j$  can be tested for emptiness only by either  $p_i$  or  $p_j$  and not by any third process. To address this problem, we take an approach that is analogous to freezing of a channel in the crash-stop model.

The main difference is that instead of freezing channels between processes, we now freeze channels between *process incarnations*. Specifically, if a process suspects a process incarnation to have crashed, then it stops accepting application messages from that incarnation. Further, it only accepts those application messages that are sent to its current incarnation. To implement freezing of a channel between process incarnations, each process has to maintain its view of the most recent incarnation of other processes in the system. This can be accomplished by maintaining a vector analogous to Fidge/Mattern's vector clock [22, 23]. We refer to this vector as *view vector*. The vector for process  $p_i$ , denoted by  $view_i$ , maintains the operational states of all processes in the system *as per  $p_i$ 's view*. The vector is piggybacked on every message (application as well as control) a process sends. As in the case of vector clock, a process, on receiving a message, updates its vector by taking a component-wise maximum of its vector and the vector received. Additionally, a process updates its vector on recovery and on detecting a crash. Like vector clocks, two view vectors are compared component-wise. Figure 1 describes the actions for modifying view vector.

For a program event  $e$  on process  $p_i$ , we use  $e.view$  to denote the view vector value on  $p_i$  immediately after executing  $e$ . Note that, since a process has up-to-date knowledge about its own operational state,  $e.view[i]$  represents the

operational state of  $p_i$  immediately after executing  $e$ . If  $e$  is not a program event (that is, it is a crash or recovery event of  $p_i$ ), we define the  $i^{\text{th}}$  entry of  $e.view$ , given by  $e.view[i]$ , as the operational state of  $p_i$  immediately after executing  $e$ . For instance, if  $e = crash_i(x)$  for some  $x$ , then  $e.view[i] = \langle \text{down}, x \rangle$ . Likewise, if  $e = recovery_i(x)$  for some  $x$ , then  $e.view[i] = \langle \text{up}, x \rangle$ . All other entries of  $e.view$  are assumed to be set to their lowest values. Specifically, the  $j^{\text{th}}$  entry of  $e.view$  with  $j \neq i$  has the value  $\langle \text{up}, 0 \rangle$ . Clearly, the  $i^{\text{th}}$  entry of the view vector of process  $p_i$  is monotonically non-decreasing even across crashes and recoveries.

We assume that the view vector of a process is stored in volatile storage but may be flushed to stable storage periodically while the process is up. Therefore, the view vector of a process is monotonically non-decreasing as long as the process does not crash.

For a message  $m$ , let  $m.view$  denote the vector piggybacked on  $m$ . We say that  $p_i$  believes  $p_j$  to be currently up if  $view_i[j].status = \text{up}$ . We now formally define what it means to freeze a channel between two process incarnations.

**Condition 3 (channel freezing).** *Consider an application message  $m$  sent by process  $p_i$  to process  $p_j$ . Then  $p_j$  accepts  $m$  if and only if both the following conditions hold:*

1.  $view_j[j] = m.view[j]$  and
2.  $view_j[i] \leq m.view[i]$ .

We present two algorithms for safe termination detection. The first algorithm detects termination when Cond. 1 holds. The second algorithm detects termination when Cond. 2 and Cond. 3 hold.

## 6 Termination Detection with Eventually Reliable Processes

In this section, we present a termination detection algorithm assuming eventually reliable processes.

As explained before, when all processes are eventually reliable, detecting termination of a distributed computation becomes equivalent to detecting that all processes are passive and all channels are empty. In the crash-free model, testing whether a channel is empty is relatively easy. To test whether a channel from process  $p_i$  to process  $p_j$  is empty, it is sufficient to test that the number of messages that  $p_i$  has sent to  $p_j$  so far is equal to the number of messages that  $p_j$  has received from  $p_i$  so far. However, in the crash-recovery model, a message that  $p_i$  sends to  $p_j$  may arrive at  $p_j$  while  $p_j$  is down and is, therefore, lost. As a result, when comparing  $p_i$  and  $p_j$ 's states, if  $p_j$  is missing a message sent to it by  $p_i$ , we cannot distinguish between the case when the message has been lost and the case when the message has been simply delayed.

Therefore, to detect termination, we need some other mechanism to test for emptiness of a channel. To that end, we define a special operation on a channel, which we refer to as *flush*. A flush operation is defined using two events: `start_flush` and `end_flush`. A process  $p_i$  initiates a flush operation on its outgoing channel

with another process, say process  $p_j$ , by executing the `start_flush` event. A flush operation initiated by  $p_i$  ends when  $p_i$  executes a matching `end_flush` event. A flush operation should satisfy the following two properties:

- *No old message delivery after flush (safety)*: Once  $p_i$  executes an `end_flush` event,  $p_j$  does not deliver any application message that  $p_i$  sent before executing the corresponding `start_flush` event.
- *Eventual flush completion (liveness)*: If neither  $p_i$  nor  $p_j$  crashes, then eventually  $p_i$  executes a matching `end_flush` event.

We provide an implementation of the flush operation later in this section. We now describe a scheme that enables a process to test if the underlying computation has terminated. The scheme consists of two phases. In the first phase, the process, which is testing for termination, requests all processes to flush their outgoing channels and also send their local states to it. A process sends a local state of passive if it is passive at the time of receiving the request and stays passive until all its outgoing channels have been flushed; otherwise it sends a local state of active. If local states of all processes indicate that all processes are passive, then the scheme proceeds to the second phase. In the second phase, the process again contacts all processes to determine if any one of them became active since sending its previous response. If no such process exists and no process fails during the entire execution of the scheme, then the process infers that the computation has terminated. We prove that the scheme is safe, that is, a process detects termination only if the computation has terminated.

To ensure liveness, a process uses an instance of the scheme to test whether the computation has terminated whenever it becomes passive or recovers from a crash. We show that once the computation terminates, some process eventually detects termination. Different instances of the scheme are differentiated using an *instance identifier*, which consists of (1) the identifier of the initiating process, (2) its incarnation number and (3) a sequence number. The sequence number helps differentiate between various instances of the scheme initiated by the same incarnation of a process. The sequence number can be stored in the volatile storage. We refer to the termination detection algorithm described in this section as TDA-ER. We show that:

**Theorem 1 (TDA-ER is safe and live).** *If TDA-ER announces termination, then the computation has already terminated. Further, once the computation terminates, TDA-ER eventually announces termination.*

Let  $R$  denote the sum of (1) the number of active-to-passive transitions in the computation and (2) the number of recovery events in the execution. Then there are at most  $R$  invocations of the testing scheme in total.

## 6.1 Implementing Flush Operation

To implement flush operation, we assume that all channels are eventually reliable (no duplication) and, moreover, satisfy FIFO property. On initiating a flush operation on an outgoing channel (that is, on executing a `start_flush` event), a

process sends a flush message to the neighbor of the channel. The neighbor, on receiving the flush message, sends an acknowledgment message back to the process. On receiving the acknowledgment message, the process executes the `end_flush` event.

Another way to implement the flush operation is to use stable storage. A process logs every application message it sends and receives in stable storage. Further, it periodically retransmits every message in stable storage until it receives an acknowledgment for it. When a flush operation is initiated, it executes the `end_flush` event once all messages sent before the `start_flush` event have been acknowledged.

## 7 Termination Detection with Passive Recovery and Channel Freezing

In this section, we present a termination detection algorithm assuming passive recovery and channel freezing. Unlike in the previous case, in this case, a process may eventually crash and never recover. Therefore, as is usually the case, we need some kind of a failure detector to aid processes in determining the current status of other processes in the system. Specifically, we use a perfect failure detector defined in Sect. 3 to solve the termination detection problem. We do assume, however, that there is *at least one always-up process* in the system.

Due to passive recovery and channel freezing, when a crashed process recovers, it has to execute a recovery operation to rejoin the computation. Otherwise, it can never become active again. Intuitively, as part of the recovery operation, a process informs other operational processes in the system about its recovery. This serves two purposes. First, other processes can start sending it application messages which can now be accepted by the process since they will carry its latest incarnation number. Second, if the process crashes again, then the failure detector is obligated to detect its crash due to the strong completeness property.

We use the following recovery operation. A crashed process, on recovery, broadcasts a restart message to all processes in the system. It then waits to receive an acknowledgment from all those processes that it believes have not crashed even once. This ensures that at least one always-up process knows about the recovery. Note that all messages exchanged in the recovery operation (namely, restart and acknowledgment) are piggybacked with the incarnation vector of the sending process. Any application message received before the recovery operation has completed is buffered and processed later.

As in the previous algorithm, we now describe a scheme that enables a process to test if the underlying computation has terminated. The process, which is testing for termination, requests all processes in the system to send their current local states to it. The local state of a process includes: (1) the view vector, (2) the state with respect to the application, (3) the number of application messages it has sent to the latest incarnation of each process, and (4) the number of application messages it has received from the latest incarnation of each process. The process waits until it has received a local state from each process that it

believes to be currently up. It then infers that the computation has terminated if both the following conditions hold:

1. all processes currently up in its view have identical view vectors, and
2. all processes currently up in its view are passive and all channels between them are empty.

We show that the scheme is safe, that is, a process detects termination only if the computation has terminated. To ensure liveness, a process uses an instance of the scheme to test whether the computation has terminated whenever it becomes passive or its view vector changes. We show that once the computation terminates, some process eventually detects termination. As before, different instances of the scheme can be differentiated using an appropriate instance identifier.

**Theorem 2 (TDA-CF is safe and live).** *If TDA-CF announces termination, then the computation has already terminated. Further, once the computation terminates, TDA-CF eventually announces termination.*

Let  $R_c$  denote the number of active-to-passive transitions in the computation and  $R_o$  denote the number of crash and recovery events in the execution. Then there are at most  $R_c + NR_o$  invocations of the testing scheme in total.

## 8 Conclusions

We have identified two conditions under which the termination detection problem can be solved in a safe manner when processes can crash and recover. We have also proposed a termination detection algorithm to solve the problem under each of the two conditions.

Our algorithm for the second condition uses a perfect failure detector which is strictly stronger than the failure detector used to solve consensus in the crash-recovery model [17]. When processes do not recover after crashing, the set of assumptions for our second algorithm become *identical* to those under crash-stop model. Since a perfect failure detector is necessary to detect termination in the crash-stop model [13, 12], we believe that a perfect failure detector is necessary to detect termination in the crash-recovery model as well when two or more processes may be bad. We plan to prove this rigorously in the future.

## References

1. Tel, G.: Distributed Control for AI. Technical Report UU-CS-1998-17, Information and Computing Sciences, Utrecht University, The Netherlands (1998)
2. Dijkstra, E.W., Scholten, C.S.: Termination Detection for Diffusing Computations. Information Processing Letters (IPL) **11**(1) (1980) 1–4
3. Francez, N.: Distributed Termination. ACM Transactions on Programming Languages and Systems (TOPLAS) **2**(1) (1980) 42–55
4. Stupp, G.: Stateless Termination Detection. In: Proceedings of the 16th Symposium on Distributed Computing (DISC), Toulouse, France (2002) 163–172

5. Khokhar, A.A., Hambruch, S.E., Kocalar, E.: Termination Detection in Data-Driven Parallel Computations/Applications. *Journal of Parallel and Distributed Computing (JPDC)* **63**(3) (2003) 312–326
6. Mittal, N., Venkatesan, S., Peri, S.: Message-Optimal and Latency-Optimal Termination Detection Algorithms for Arbitrary Topologies. In: *Proceedings of the 18th Symposium on Distributed Computing (DISC)*, Amsterdam, The Netherlands (2004) 290–304
7. Matocha, J., Camp, T.: A Taxonomy of Distributed Termination Detection Algorithms. *The Journal of Systems and Software* **43**(3) (1998) 207–221
8. Venkatesan, S.: Reliable Protocols for Distributed Termination Detection. *IEEE Transactions on Reliability* **38**(1) (1989) 103–110
9. Lai, T.H., Wu, L.F.: An  $(N - 1)$ -Resilient Algorithm for Distributed Termination Detection. *IEEE Transactions on Parallel and Distributed Systems (TPDS)* **6**(1) (1995) 63–78
10. Tseng, Y.C.: Detecting Termination by Weight-Throwing in a Faulty Distributed System. *Journal of Parallel and Distributed Computing (JPDC)* **25**(1) (1995) 7–15
11. H elary, J.M., Murfin, M., Mostefaoui, A., Raynal, M., Tronel, F.: Computing Global Functions in Asynchronous Distributed Systems with Perfect Failure Detectors. *IEEE Transactions on Parallel and Distributed Systems (TPDS)* **11**(9) (2000) 897–909
12. Mittal, N., Freiling, F.C., Venkatesan, S., Penso, L.D.: Efficient Reduction for Wait-Free Termination Detection in a Crash-Prone Distributed System. In: *Proceedings of the 19th Symposium on Distributed Computing (DISC)*. (2005) 93–107
13. 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
14. Majuntke, M.: Termination Detection in Systems Where Processes May Crash and Recover. Master’s thesis, RWTH Aachen University (2006)
15. Mittal, N., Phaneesh, K.L., Freiling, F.C.: Safe Termination Detection in an Asynchronous Distributed System when Processes may Crash and Recover. Technical Report UTDCS-41-06, Department of Computer Science, The University of Texas at Dallas, Richardson, TX 75083, USA (2006)
16. Lamport, L.: Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM (CACM)* **21**(7) (1978) 558–565
17. Aguilera, M.K., Chen, W., Toueg, S.: Failure Detection and Consensus in the Crash Recovery Model. *Distributed Computing (DC)* **13**(2) (2000) 99–125
18. Basu, A., Charron-Bost, B., Toueg, S.: Simulating Reliable Links with Unreliable Links in the Presence of Process Crashes. In: *Proceedings of the Workshop on Distributed Algorithms (WDAG)*, Bologna, Italy (1996) 105–122
19. Chandra, T.D., Toueg, S.: Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM* **43**(2) (1996) 225–267
20. Larrea, M., Fern andez, A., Ar evalo, S.: On the Implementation of Unreliable Failure Detectors in Partially Synchronous Systems. *IEEE Transactions on Computers* **53**(7) (2004) 815–828
21. 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 (2002) 345–353
22. Mattern, F.: Virtual Time and Global States of Distributed Systems. In: *Parallel and Distributed Algorithms: Proceedings of the Workshop on Distributed Algorithms (WDAG)*. (1989) 215–226
23. Fidge, C.J.: Logical Time in Distributed Computing Systems. *IEEE Computer* **24**(8) (1991) 28–33