On Detecting Termination in the Crash-Recovery Model

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

[1] University of Mannheim, 68131 Mannheim, Germany
[2] Darmstadt University of Technology, 64289 Darmstadt, Germany
[3] The University of Texas at Dallas, Richardson, TX 75083, USA

Abstract. We investigate the problem of detecting termination of a distributed computation in an asynchronous message-passing system where processes may crash and recover. We show that it is impossible to solve the termination detection problem in this model. We identify necessary and sufficient conditions under which it is possible to solve the *stabilizing* version of the problem in which a termination detection algorithm is allowed to make finite number of mistakes. Finally, we present an algorithm to solve the stabilizing termination detection problem under these conditions.

Keywords: asynchronous distributed system, termination detection, crash-recovery failure model, eventual safety, stabilizing algorithm.

1 Introduction

Termination detection involves determining whether a distributed computation has ceased all its activities. It arises when a computation terminates *implicitly* and no process knows about the termination [1]. As a result, a separate algorithm may have to be used to detect termination of the computation. The termination detection problem has been studied quite extensively in fault-free models (*e.g.*, [2,3,4,5,6]). It has also been studied in the crash-stop model (*e.g.*, [7,8,9,10,11,12]). However, little is known about the problem when processes may crash and recover. We believe that the crash-recovery failure model is more realistic than crash-stop failure models because, in practice, to avoid resource exhaustion we must allow crashed processes to recover. However, it is also harder to deal with than the other two failure models as shown by earlier work in this failure model on solving other important distributed computing problems such as consensus [13], reliable broadcast [14] and atomic broadcast [15,14].

In this paper, we investigate the termination detection problem in the *crash-recovery* model. It turns out that it is impossible to solve the problem in this failure model without weakening the problem and/or strengthening the model. The main reason for this impossibility result is that it is not possible to determine the future behavior of a currently down process, that is, whether it will stay down permanently or may recover later.

⋆ Research supported in part by EC IP DECOS, EC NoE ReSIST and DFG Graduiertenkolleg "Cooperative Adaptive and Responsive Monitoring in Mixed Mode Environments" at TU Darmstadt.

⋆⋆ Research supported in part by DFG Graduiertenkolleg "Software for Mobile Communicaton Systems" at RWTH Aachen University.

To circumvent the impossibility result, we weaken the problem by allowing a termination detection algorithm to make mistakes, that is, it may falsely announce termination albeit only a finite number of times. We refer to this problem as the *eventually safe termination detection problem*.

Even the eventually safe termination detection problem cannot be solved without strengthening the model. To that end, we make the following two assumptions about the model. First, there are no unstable processes in the system, *i.e.*, there are no processes that crash and recover infinitely often. Second, processes are equipped with an eventually perfect failure detector. We show that both conditions are necessary.

We finally describe an algorithm for solving the stabilizing termination detection problem under the two assumptions. Such an algorithm is very usefull in scenarios where only performance and not correctness of the application is affected by false termination announcements. 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. The second algorithm is only started once the first algorithm has terminated. Suppose several devices use a mutual exclusion algorithm to select a channel for communication. If due to false termination announcement mutual exclusion property is violated, in the worst case two or more devices choose the same channel and their communication will interfere. Eventually, the algorithm satisfies the safety property and will work properly thereafter. Due to lack of space, for the proofs of some lemmas and theorems we refer to [16].

2 Model and Notation

Distributed System. We assume an asynchronous distributed system consisting of a set of processes, given by $\Pi = \{p_1, p_2, \ldots, p_n\}$, 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.

Failure Model. We assume that processes 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 the *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* [13]. 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* [13]. 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.

We assume *eventually-reliable channels* in this paper. Such channels guarantee reliable communication between good processes that do not crash anymore.

Process Incarnations. We assume that each process has access to *volatile storage* and *stable storage*. 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.

Failure Detector for Termination Detection. 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) [17]. The termination detection algorithm described in this paper needs an *eventually perfect failure detector* [18]. Intuitively, an eventually perfect failure detector is responsible for monitoring the operational state of all processes in the system. It may make mistakes in the beginning. For instance, it may believe that a process is down when, in fact, the process is actually up, and vice versa. However, it should eventually have a correct view of the system.

In this paper, we do not distinguish between the local failure detector at a process and the process itself unless necessary. If process p_i believes process p_j to be down, we say that p_i *suspects* p_j. On the other hand, if p_i believes p_j to be up, we say that p_i *trusts* p_j. We model the output of the failure detector using a list of trusted processes. For a failure detector to be eventually perfect, the list at different processes should satisfy the following properties:

- *Completeness* (two parts): (1) Every bad stable process is eventually permanently suspected by all good processes. (2) Every bad unstable process is either eventually permanently suspected by all good processes, or suspected and trusted infinitely often by all good processes.
- *Accuracy*: Every good process is permanently trusted by all good processes.

Note that if there are no unstable processes in the system, then eventually all good processes agree on which processes are currently up.

3 The Termination Detection Problem

The distributed computation whose termination has to be detected is typically modeled using the following four rules. First, a process is either in *active* state or *passive* state. 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. The termination detection problem involves determining whether the computation has ceased all its activities. Formally,

Definition 1 (termination condition). *A computation is said to have terminated if no process is currently active and, further, no process becomes active in the future.*

We consider a process to be in an active state only when it is up. Further, we conclude that a process, on recovery, restarts in an active state only if it failed in an active state. Any termination detection algorithm should satisfy the following two desirable 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.

Note that the termination condition, as formulated in Definition 1, requires reasoning on future states of processes. For failure-free and crash-stop failure models, however, it is possible to reformulate the termination condition so that the condition can be evaluated on a current state of the system and does not require any future knowledge. For instance, for the failure-free model, the termination condition can be redefined as: *all processes are passive and all channels are empty*. Likewise, for the crash-stop model, the termination condition can be redefined as: *all live processes are passive and all channels towards live processes are empty*. However, for the crash-recovery model, in general, it is not possible to formulate the termination condition in a manner that does not require knowledge of future behavior of processes. One of the main reasons is that a termination detection algorithm in general should be able to distinguish between whether a process is only *temporarily down* or is *permanently down*. We now prove this impossibility result:

Theorem 1. *It is impossible to solve the termination detection problem unless it is possible to distinguish between whether a process is temporarily down or permanently down.*

Proof. Consider the initial system state in which only one process, say p_a, is active and all other processes are passive. We construct two possible executions of the system. In the first execution, p_a crashes and never recovers. Clearly, the computation has terminated and any correct termination detection algorithm should eventually announce termination. Assume that some process, say p_c, announces termination at time t. Next, consider the second execution that is identical to the first execution until time t. However, after t, p_a recovers and restarts in an active state. Clearly, the only difference between the two executions is in the behavior of p_a after t. Specifically, in the first execution, p_a is permanently down at t, whereas, in the second execution, it is only temporarily down at t. Since p_c cannot distinguish between the two executions, it announces termination at t in the second execution as well. However, in the second execution, the computation has not terminated at t. □

To circumvent the impossibility result, we weaken the termination detection problem by relaxing the safety property. Specifically, a termination detection algorithm is allowed to announce termination *falsely*. However, only a *finite* number of such false announcements are allowed. Formally, a termination detection algorithm should now satisfy the following properties:

- *Finite number of false termination announcements (eventual safety):* Eventually, if a good process announces termination, then the computation has indeed terminated.
- *Eventual termination announcement (liveness):* Once the computation terminates, eventually every good process announces termination.

Note that, even after weakening the safety property, the termination detection problem remains impossible to solve as long as the system contains unstable processes. An unstable process can repeatedly crash and recover in an active state causing the termination detection algorithm to make infinite number of mistakes. Therefore we assume that the system does not contain any unstable process, that is, every process eventually stays either up or down. We denote the resulting model as *crash/finite-recovery model.*

Suppose the underlying distributed system is such that failures are expected to be rare. Further, even the failure detector makes mistakes only rarely. In this case, we would like the termination detection algorithm to also make mistakes only rarely. Formally,

- *No false termination announcements in the absence of failures and false suspicions (zero degradation):* Assume that no process crashes during an execution and no process is falsely suspected of having crashed by the failure detector. Then, a process announces termination only if the computation has terminated.

In the next section, we describe a termination detection algorithm that satisfies eventual safety, liveness and zero degradation properties.

4 An Eventually Safe Termination Detection Algorithm

Our termination detection algorithm uses an eventually perfect failure detector defined in Sect. 2. It turns out that such a failure detector is actually necessary for solving the eventually safe termination detection problem in the stabilizing crash-recovery model. Specifically, we can implement an eventually perfect failure detector using an eventually safe termination detection algorithm as follows. There are n computations in the system, one for each process; the computation for process p_i is denoted by $\mathcal{C}_i$. In $\mathcal{C}_i$, p_i is always active while it is up and all other processes are always passive. Further, no process sends any application message in any computation. Additionally, there are n instance of the termination detection algorithm, and the instance $\mathcal{A}_i$ of the algorithm is responsible for observing the computation $\mathcal{C}_i$. When instance $\mathcal{A}_i$ announces (respectively, revokes) termination at a process p_j, p_j starts suspecting (respectively, trusting) p_i. It can be easily verified that this transformation correctly implements an eventually perfect failure detector.

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

Each process maintains three different vectors, namely *incarnation vector*, *sent vector* and *received vector*. A process uses the first vector to keep track of the latest incarnation of other processes in the system. It uses the sent vector to keep track of the number of application messages it has sent to the latest incarnation of other processes.

```
Rules for updating incarnation vector on process $p_i$:
Variables:
    $iv_i$: vector $[1..n]$ of incarnation numbers, initially $[0, 0, \ldots, 0]$;
(A1) On sending a message $m$:
    piggyback $iv_i$ on $m$;
(A2) On receiving a message $m$ carrying incarnation vector $m.iv$:
    for each $j$ in $[1, n]$ do
        $iv_i[j] := \max\{iv_i[j], m.iv[j]\}$;
    endfor;
(A3) On starting new incarnation $x$ after recovery:
    $iv_i[i] := x$;
    // other entries of $iv_i$ may be initialized using stable storage, if applicable
```

Fig. 1. Rules for maintaining incarnation vector on a process

Finally, it uses the received vector to keep track of the number of application messages it has received from the latest incarnation of other processes *that were sent to its current incarnation*. The incarnation, sent and received vectors for process p_i are denoted by iv_i, $sent_i$ and $received_i$, respectively.

The rules for maintaining the incarnation vector are similar to those for maintaining a Fidge/Mattern vector [19,20]. A process, on recovery, sets its own entry in the incarnation vector to its incarnation number. A process piggybacks the incarnation vector on every message it sends. Further, a process, on receiving a message, updates its incarnation vector by taking the component-wise maximum of each entry in the current vector and the vector received along with the message. Figure 1 describes the actions A1-A3 for maintaining the incarnation vector. Action A1 is executed whenever a process sends a message. Action A2 is executed whenever a process receives a message. Finally, action A3 is executed whenever a process recovers and starts a new incarnation.

We now describe a scheme that a process periodically uses to test if the computation has terminated. As part of the scheme, process requests all processes in the system to send their current local states to it. The local state of a process includes: (1) the incarnation vector, (2) the state with respect to the application, (3) the sent vector, and (4) the received vector. The process waits until it has received a local state from all processes that it currently trusts. It first ascertains that all trusted processes have identical incarnation vectors. If not, the process aborts the current instance of the testing scheme. If yes, the process checks whether the computation has terminated by evaluating the following two conditions:

1. all trusted processes are passive, and
2. sent and received vectors of all trusted processes "match" with each other when restricted to only entries for trusted processes.

If both conditions evaluate to true and the last action by the process was to revoke termination announcement, then the process announces termination. On the other hand, if one of the conditions fails and the last action was to announce termination, then the process revokes that last termination announcement. A process also aborts the current instance of the testing scheme if either its incarnation vector or the output of its failure detector changes. Note that the two conditions for termination are similar to those used in Mattern's channel-counting algorithm [4], which is a fault-intolerant termination detection algorithm.

We say that a system has *stabilized* if (1) each process has stabilized (that is, no more process crashes and recoveries), (2) the failure detector at each process has stabilized (that is, no more changes in the output of the failure detector), and (3) any application message delivered hereafter is sent by the current incarnation of the sender to the current incarnation of the receiver. We show that, once the system has stabilized, the testing scheme satisfies the safety property. Specifically, the two conditions described above evaluate to true only if the computation has terminated.

To ensure liveness, a process *periodically* uses an instance of the above-described scheme to test for termination. Different instances of the scheme are distinguished 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 differentiating between various instances of the scheme initiated by the same incarnation of a process. The sequence number can be stored in the volatile storage.

```
Actions of the termination detection algorithm TDA-ES for process $p_i$:
Variables:
    $state_i$: state of $p_i$ with respect to the application (maintained by the application);
    $sent_i$: vector $[1..n]$ of number of application messages send to each process, initially $[0, 0, \ldots, 0]$ ;
    $received_i$: vector $[1..n]$ of number of application messages received from each process, initially $[0, 0, \ldots, 0]$ ;
    $announcement_i$: whether termination has occurred, initially false;

(B1) Whenever the $j^{th}$ entry in $iv_i$ advances:
    $sent_i[j] := 0$;
    $received_i[j] := 0$;

(B2) On starting a new incarnation after recovery:
    for $j \in [1..n]$ do
        $sent_i[j] := 0$;
        $received_i[j] := 0$;
        $announcement_i :=$ false;
    endfor;

(B3) On sending an application message $m$ to process $p_j$:
    $++\, sent_i[j]$;
    send $m$ to process $p_j$;

(B4) On receiving an application message $m$ from process $p_j$:
    let $m.iv$ denote the incarnation vector piggybacked on $m$;
    if ($iv_i[i] = m.iv[i]$) and ($iv_i[j] = m.iv[j]$) then
        $++\, received_i[j]$;
    endif;
    deliver $m$ to the application;

(B5) On invocation of testForTermination( ):
    send REQUEST message to all processes;

(B6) On receiving REQUEST message from process $p_j$:
    send RESPONSE($iv_i, state_i, sent_i, received_i$) message to process $p_j$;

(B7) On receiving RESPONSE($iv_j, state_j, sent_j, received_j$) from process $p_j$:
    let $T_i$ denote the set of currently trusted processes;
    if (a RESPONSE message has been received from all processes in $T_i$) then
        if $\langle \forall\, p_x, p_y : \{p_x, p_y\} \subseteq T_i : iv_x = iv_y \rangle$ then
            // all processes in $T_i$ have identical incarnation vectors
            $test_i := \langle \forall\, p_x : p_x \in T_i : state_x =$ passive$\rangle \wedge$
                    $\langle \forall\, p_x, p_y : \{p_x, p_y\} \subseteq T_i : sent_x[y] = received_y[x] \rangle$;
            if ($announcement_i \wedge \neg test_i$) then
                // revoke termination announcement
                $announcement_i :=$ false;
            else if ($\neg announcement_i \wedge test_i$) then
                // announce termination
                $announcement :=$ true;
            endif;
        endif;
    endif;
```

Fig. 2. The termination detection algorithm TDA-ES

We refer to our termination detection algorithm as TDA-ES. A formal description of the algorithm is given in Fig. 2. It consists of seven actions B1-B7. Action B1 is executed whenever the incarnation vector of a process changes and is invoked from action A2 or action A3. Action B2 is executed when a process recovers and is invoked from action A3. Action B3 (respectively, B4) is executed whenever a process sends (respectively, receives) an application message. Note that action A1 has to be executed after executing action B3. Further, action B4 is invoked after action A2 has been executed. Actions B5, B6 and B7 are executed as part of the testing scheme. Note that REQUEST and RESPONSE messages carry instance identifier which is not shown in the description.

We now prove that our algorithm satisfies eventual safety, liveness and zero degradation properties. Note that there is no unstable process in the system and the failure detector is eventually perfect. Therefore it follows that:

Proposition 1. *The system eventually becomes stable.*

We say that the incarnation vector at a good process has *stabilized* if its value has stopped changing. Note that the i^{th} entry of the incarnation vector at process p_i is incremented only when p_i recovers. Any other process p_j, with $j \neq i$, simply copies a (new) value into the i^{th} entry of its incarnation vector from the vector it has received along with a message. Therefore, we have:

Proposition 2. *Once the system has become stable, the incarnation vector at a good process eventually becomes stable.*

Every good process periodically uses the testing scheme to test for termination. Once the system becomes stable, by our assumption, all channels between good processes become reliable. Therefore every good process receives infinite number of messages from every other good process. It can be easily verified that:

Proposition 3. *If the incarnation vector at every good process has become stable, then all good processes have identical incarnation vectors.*

We say that a system has become *strongly stable* if the system has become stable and the incarnation vector at every good process has become stable. We refer to the incarnation of a good process that never crashes as the *final incarnation*. Note that an instance of the testing scheme initiated after the system has become strongly stable always completes successfully (that is, is not aborted by its initiator). Also, once the system has become stable, every good process permanently trusts all good processes and permanently suspects all bad processes. This implies that if an instance of the testing scheme is initiated after the system has become strongly stable then its termination conditions are evaluated on local states of all good and only good processes. We now show that our testing scheme is safe and live if it is initiated after the system has become strongly stable.

Lemma 1. *Any instance of the testing scheme initiated after the system has become strongly stable indicates termination only if the computation has terminated.*

Lemma 2. *Any instance of the testing scheme initiated after (1) the system has become strongly stable and (2) the computation has terminated indeed indicates termination.*

The liveness of our algorithm follows from the fact that every good process periodically initiates an instance of the testing scheme to test for termination.

Proposition 4. *If no process crashes during an execution and no process is falsely suspected of having crashed by the failure detector, then the system is strongly stable in the initial state.*

Theorem 2. *TDA-ES satisfies eventual safety, liveness and zero degradation properties.*

Proof. Eventual safety follows from Proposition 1, Proposition 2, Proposition 3 and Lemma 1. Liveness follows from Proposition 1, Proposition 2, Proposition 3 and Lemma 2. Zero degradation follows from Lemma 1 and Proposition 4. □

5 Discussion

In our algorithm, as described above, each message has to carry a vector consisting of n entries. It is possible to optimize our algorithm so that only RESPONSE messages are required to carry a vector. An application message only needs to carry two entries from the incarnation vector of the sender, namely entries corresponding to the sender and the receiver. Specifically, an application message sent by process p_i to process p_j carries entries $iv_i[i]$ and $iv_i[j]$. It can be verified that all propositions and lemmas in the previous section still hold with this modification.

An interesting question to ask is when can the termination detection problem be solved in a *safe* manner under crash-recovery model. We answer this question in [21] where we identify two conditions under which the safe termination detection problem can indeed be solved. These conditions are rather strong compared to the conditions identified in this paper. For example, one of the conditions requires the availability of a *perfect failure detector*, processes to *always restart in passive state after recovery* and processes to *reject old application messages*.

References

1. Tel, G.: Distributed Control for AI. Technical Report UU-CS-1998-17, Information and Computing Sciences, Utrecht University, The Netherlands Technical Report UU-CS–17, Information and Computing Sciences (1998)
2. Dijkstra, E.W., Scholten, C.S.: Termination Detection for Diffusing Computations. Information Processing Letters (IPL) 11(1), 1–4 (1980)
3. Francez, N.: Distributed Termination. ACM Transactions on Programming Languages and Systems (TOPLAS) 2(1), 42–55 (1980)
4. Mattern, F.: Algorithms for Distributed Termination Detection. Distributed Computing (DC) 2(3), 161–175 (1987)
5. Mattern, F.: Global Quiescence Detection based on Credit Distribution and Recovery. Information Processing Letters (IPL) 30(4), 195–200 (1989)
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, pp. 290–304 (October 2004)

7. Venkatesan, S.: Reliable Protocols for Distributed Termination Detection. IEEE Transactions on Reliability 38(1), 103–110 (1989)
8. 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), 63–78 (1995)
9. Tseng, Y.C.: Detecting Termination by Weight-Throwing in a Faulty Distributed System. Journal of Parallel and Distributed Computing (JPDC) 25(1), 7–15 (1995)
10. Hélary, J.M., Murfin, M., Mostefaoui, A., Raynal, M., Tronel, F.: Computing Global Functions in Asynchronous Distributed Systems with Perfect Failure Detectors. In: IEEE Transactions on Parallel and Distributed Systems (TPDS), September 2000, vol. 11(9), pp. 897–909 (2000)
11. Gärtner, F.C., Pleisch, S. (Im)Possibilities of Predicate Detection in Crash-Affected Systems. In: Proceedings of the 5th Workshop on Self-Stabilizing Systems (WSS),, Lisbon, Portugal, October 2001, pp. 98–113 (2001)
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: Fraigniaud, P. (ed.) DISC 2005. LNCS, vol. 3724, pp. 93–107. Springer, Heidelberg (2005)
13. Aguilera, M.K., Chen, W., Toueg, S.: Failure Detection and Consensus in the Crash Recovery Model. Distributed Computing (DC) 13(2), 99–125 (2000)
14. Boichat, R., Guerraoui, R.: Reliable and Total Order Broadcast in the Crash-Recovery Model. Journal of Parallel and Distributed Computing (JPDC) 65(4), 397–413 (2005)
15. Rodrigues, L., Raynal, M.: Atomic Broadcast in Asynchronous Crash-Recovery Distributed Systems and its Use in Quorum-Based Replication. IEEE Transactions on Knowledge and Data Engineering 15(5), 1206–1217 (2003)
16. Freiling, F., Majuntke, M., Mittal, N.: Termination Detection in an Asynchronous Distributed System with Crash-Recovery Failures. Technical report, TR-2006-008, University of Mannheim (2006)
17. Delporte-Gallet, C., Fauconnier, H., Guerraoui, R.: A Realistic Look At Failure Detectors. In: Proceedings of the IEEE International Conference on Dependable Systems and Networks (DSN), Washington, DC, USA, pp. 345–353 (2002)
18. Chandra, T.D., Toueg, S.: Unreliable Failure Detectors for Reliable Distributed Systems. Journal of the ACM 43(2), 225–267 (1996)
19. Mattern, F.: Virtual Time and Global States of Distributed Systems. In: Parallel and Distributed Algorithms: Proceedings of the Workshop on Distributed Algorithms (WDAG), pp. 215–226 (1989)
20. Fidge, C.J.: Logical Time in Distributed Computing Systems. IEEE Computer 24(8), 28–33 (1991)
21. Mittal, N., Phaneesh, K.L., Freiling, F.C.: Safe Termination Detection in an Asynchronous Distributed System when Processes may Crash and Recover. In: Shvartsman, A.A. (ed.) OPODIS 2006. LNCS, vol. 4305, pp. 126–141. Springer, Heidelberg (2006)