Eventual Leader Election in the Crash-Recovery Failure Model

Cristian Martín
The University of the Basque Country
20018 San Sebastián, Spain
martin2.cristian@gmail.com

Mikel Larrea
The University of the Basque Country
20018 San Sebastián, Spain
mikel.larrea@ehu.es

Abstract

Unreliable failure detectors provide information about process failures. A particular failure detector called Omega has been shown to be the weakest for solving consensus with a majority of correct processes. This work addresses the implementation of Omega in the crash-recovery failure model. Firstly, the definition of Omega is adapted to that model, assuming that processes do not use stable storage. After that, an algorithm implementing Omega under some weak assumptions on communication reliability and synchrony is proposed.

1. Introduction

Unreliable failure detectors [5] are mechanisms that provide (perhaps incorrect) information about process failures. They have been used to solve agreement problems in crash-prone asynchronous distributed systems, e.g., Consensus [15]. In this work, we address the implementation of Omega [4], the weakest failure detector for solving consensus, in the crash-recovery failure model. The Omega failure detector provides an eventual leader election functionality, i.e., eventually all processes agree on a common and non-faulty leader process. Several consensus algorithms based on a leader election mechanism have been proposed [8, 10, 12, 14].

We can find in the literature several algorithms implementing Omega in the crash failure model, in which crashed processes do not recover. In [11] Larrea et al. propose an algorithm that requires all links to be eventually timely. In [2] Aguilera et al. propose an Omega algorithm in a system where some unknown correct process must have all its links eventually timely, while all other links can be lossy and/or asynchronous. Aguilera et al. propose in [3] another Omega algorithm in which only the outgoing links from some unknown correct process to the rest of processes must be eventually timely. More recently, Jiménez et al. propose in [9] an Omega algorithm with unknown membership which requires that eventually all correct processes are reachable timely from some correct process.

Consensus and failure detection have also been studied in the crash-recovery failure model. However, there are few specific algorithms implementing Omega in this failure model. In [1], Aguilera et al. define an adaptation of the $\diamond S$ failure detector to the crash-recovery failure model, and propose an algorithm implementing it in partially synchronous systems [5, 6]. The algorithm requires known membership and assumes a fully connected system. More recently, Martín et al. have proposed in [13] two algorithms implementing Omega in the crash-recovery failure model that rely on the use of stable storage and allow up to $n - 1$ failures, where $n$ is the number of processes in the system.

In this work, we propose an algorithm for Omega in the crash-recovery model in a system where processes do not use any form of stable storage. There is indeed a high cost associated to using stable storage that may severely limit the practicality of earlier protocols. We replace stable storage by the need for a majority of correct processes in order to get all alive processes to eventually agree on the same leader, even if some of them crash and recover infinitely often. A similar trade-off between using stable storage or a majority has been discussed by Wiesmann and Défago in [16] on the implementation of end-to-end communication primitives.

The rest of the paper is organized as follows. In Section 2, we describe the system model, and redefine the property of Omega in the crash-recovery failure model. We present the algorithm implementing Omega in Section 3. In Section 4, we discuss about the eventual timeliness of fair lossy links. In Section 5, we present an adaptation of the algorithm in order to agree on a common set of correct processes. Finally, Section 6 concludes the paper.
2. System Model

We consider a system $S$ composed of a finite and totally ordered set $\Pi = \{p_1, p_2, \ldots, p_n\}$ of $n > 1$ processes that communicate only by sending and receiving messages. Each pair of processes is connected by two unidirectional communication links, one in each direction.

Processes can only fail by crashing. Crashes are not permanent, i.e., crashed processes can recover. In every run, $\Pi$ is composed of the following three disjoint subsets:

1. **Eventually up**, i.e., processes that eventually remain up forever.
2. **Eventually down**, i.e., processes that eventually remain crashed forever.
3. **Unstable**, i.e., processes that crash and recover an infinite number of times.

By definition, processes in (1) are **correct**, while processes in (2) and (3) are **incorrect**. We assume that a majority of processes in the system are correct. We also assume that processes do not have access to any form of stable storage. In particular, when a process crashes all its variables lose their values.

Processes are synchronous, i.e., there are lower and upper bounds on the number of instructions they can execute per unit of time. Each process has a local clock that can accurately measure intervals of time. The clocks of the processes are not synchronized. For simplicity, and without loss of generality, we assume that local processing time is negligible with respect to message communication delays.

We assume that messages are unique, e.g., each message contains the id of the sender and a sequence number. Communication links cannot create or alter messages, but are not assumed to be FIFO. Concerning timeliness or loss properties, we consider the following three types of links [3, 7]:

(a) **Eventually timely links**, where there is an unknown bound $\delta$ on message delays and an unknown (system-wide) global stabilization time $T$, such that if a message is sent at a time $t \geq T$, then this message is received by time $t + \delta$.

(b) **(Typed) Fair lossy links**, where assuming that each message has a type, if for every type infinitely many messages are sent, then infinitely many messages of each type are received (if the receiver process is correct).

(c) **Lossy links**, where the link can lose an arbitrary number of messages (possibly all).

2.1. The Omega failure detector

Chandra et al. defined in [4] a failure detector for the crash failure model called Omega. The output of the failure detector module of Omega at a process $p$ is a single process $q$ that $p$ currently considers to be correct (we say that $p$ **trusts** $q$). Omega satisfies the following property:

**Property 1** There is a time after which every correct process always trusts the same correct process.

Observe that this definition does not say anything about unstable processes. Hence, if we keep it as is for the crash-recovery failure model, unstable processes are allowed to disagree with correct processes, which can be a drawback, e.g., for solving consensus. In practice, it could be interesting that eventually all the processes that are up, either correct or unstable, agree on a common (correct) leader process. Hence, we redefine the property that Omega must satisfy, adapted to the crash-recovery failure model without stable storage.

**Property 2** There is a time after which (1) every correct process always trusts the same correct process $l$, and (2) every unstable process, when up, always trusts either $\bot$ (i.e., it does not trust any process) or $l$. More precisely, upon recovery it trusts first $\bot$, and—if it remains up for sufficiently long—then $l$ until it crashes.

Compared to the definition of Omega proposed in [13], this definition does not force unstable processes to eventually agree **permanently** (when up) on $l$. This is achieved in [13] by the use of stable storage. More precisely, in [13] whenever a process recovers from a crash it reads the identity of the leader from stable storage.

3. The Algorithm

Figure 1 presents an algorithm implementing Omega in system $S$ under the following weak assumptions on communication reliability and synchrony:

i) There is a correct process $p$ such that there is an eventually timely link/path (formed by eventually timely links and correct processes) from $p$ to every correct and every unstable process.\(^1\)

ii) For every correct process $q \neq p$, there is a fair lossy link/path from $q$ to $p$.

iii) For every unstable process $u$, there is a fair lossy link from $u$ to some correct process.

\(^1\)For unstable processes, it is only required whenever they are up.
Every process $p$ executes the following:

\begin{verbatim}
procedure updateLeader()
   (p1) leader$_p$ ← ⊥ such that punish$_p$[l] = min{punish$_p$[q]}, ∀q ∈ candidates$_p$
end procedure

Initialization:
( 1) leader$_p$ ← ⊥
( 2) candidates$_p$ ← II
( 3) ∀q ≠ p : Timeout$_p$[q] ← default time-out interval
( 4) ∀q : punish$_p$[q] ← 0
( 5) send (RECOVERED, p) to all processes
( 6) timers active ← FALSE
( 7) start tasks 1, 2, 3 and 4

Task 1:
( 8) repeat forever every $η$ time units
( 9) send (ALIVE, p, punish$_p$) to all processes

Task 2:
(10) upon reception of message (RECOVERED, q) do
(11) punish$_p$[q] ← punish$_p$[q] + 1

Task 3:
(12) upon reception of message (ALIVE, q, punish$_p$) with q ≠ p for the first time do
(13) send (ALIVE, q, punish$_q$) to all processes
(14) ∀r : punish$_p$[r] ← max{punish$_p$[r], punish$_q$[r]}
(15) ∀r : Timeout$_p$[r] ← max{Timeout$_p$[r], Timeout$_q$[r], punish$_p$[p]}
(16) if $p$ has received so far ALIVE from a majority of processes then
(17) if timers active = FALSE then
(18) ∀q ≠ p : reset timer$_p$(q) to Timeout$_p$[q]
(19) timers active ← TRUE
(20) if q ∈ candidates$_p$ then
(21) candidates$_p$ ← candidates$_p$ \{q\}
(22) Timeout$_p$[q] ← Timeout$_p$[q] + 1
(23) reset timer$_p$(q) to Timeout$_p$[q]
(24) updateLeader()

Task 4:
(25) upon expiration of timer$_p$(q) do
(26) punish$_p$[q] ← punish$_p$[q] + 1
(27) candidates$_p$ ← candidates$_p$ \{q\}
(28) updateLeader()
\end{verbatim}

Figure 1. Algorithm implementing Omega in system $S$.

Every process $p$ has a leader$_p$ variable containing its trusted process (initialized to ⊥), and a candidates$_p$ set containing the processes among which $p$ will choose leader$_p$ (initialized to II). Also, $p$ has a Timeout$_p$[q] time-out with respect to every other process $q$ (initialized to a default value), and a punish$_p$[q] counter of the number of times that every process $q$ has recovered or has been suspected (initialized to 0).

During initialization (and upon recovery), $p$ sends a RECOVERED message to all processes, and starts the four tasks of the algorithm. Then, $p$ sets timers active to FALSE. Note that all the timers of $p$ are inactive. If $p$ does not crash, the reception of such ALIVE messages is guaranteed by the assumption that a majority of processes are correct in $S$. In Task 1, $p$ periodically sends an ALIVE message containing punish$_p$ to all processes. In Task 2, when $p$ receives a RECOVERED message from $q$, $p$ increments punish$_p$.

In Task 3, when $p$ receives an ALIVE message from $q ≠ p$ which was not received previously, $p$ re-sends the message to all processes and updates punish$_p$ with punish$_q$ (taking the highest value for each component of the vector). Then, $p$ also updates all its time-outs, taking the maximum between the current time-outs and punish$_p$[p]. Finally, if $p$ has received so far ALIVE messages from a majority of processes, if the timers are not active yet, $p$ resets all its timers (for the first time after the recovery), and sets timers active to TRUE. On the other hand, $p$ includes $q$ in candidates$_p$ if required (incrementing Timeout$_p$[q]), resets timer$_p$(q) and calls the procedure updateLeader() in order to
update leader\textsubscript{p} to the process in candidates\textsubscript{p} with the minimum associated counter. In Task 4, when timer\textsubscript{p}(q) expires, p increments punish\textsubscript{p}[q], removes q from candidates\textsubscript{p}, and calls updateLeader(). Note that timer\textsubscript{p}(q) will not expire again unless it is previously reset in Line 23.

Observe that no timer expires in the algorithm until p has received ALIVE from a majority of processes. With this algorithm, eventually all the processes that are up will have in leader\textsubscript{p} either ⊥ (which indicates that they have not received ALIVE from a majority of processes yet) or the common correct leader l. Intuitively, after the reception of ALIVE from a majority of processes an unstable process u will have (1) punish\textsubscript{u} such that l is chosen as leader, and (2) Timeout\textsubscript{u}[l] such that timer\textsubscript{u}(l) will not expire any more.

Figures 2 to 4 present three scenarios of a system composed of five processes which satisfy the assumptions required by our algorithm. Observe that, since nothing can be said about the timeliness of fair lossy links, in the presented scenarios process p\textsubscript{2} will eventually become the leader (unless p\textsubscript{1} or p\textsubscript{5} communicate timely with p\textsubscript{2} through the fair lossy link/path).

**Figure 2. Scenario 1: three eventually up, one eventually down, one unstable.**

**Figure 3. Scenario 2: three eventually up, one eventually down, one unstable.**

**Figure 4. Scenario 3: three eventually up, two unstable.**

### 3.1. Correctness proof

We now show the correctness of the algorithm of Figure 1. Let R be the set of correct processes that eventually can reach by eventually timely links every alive process in \textit{S}. By definition, there is a constant \(\Delta\) and a time \(T\) after which every message sent by a process \(s, s \in R\), takes at most \(\Delta\) time to be received by every alive process. Let \(B\) be the set of correct processes \(p\) with bounded \(\text{punish}_{p}[p]\).

For the rest of the section we will assume that any time instant \(t\) is larger than a time \(t_{1}\), where \(t_{1}\) is a time instant that occurs after the stabilization time \(T\), and after every eventually down process has definitely crashed, and every eventually up process has definitely recovered.

We will denote \(\text{var}_{p}\), the value of the local variable var of \(p\) at time \(t\).

**Lemma 1** \(\forall q \in \text{correct}, \forall u \in \text{unstable}, \text{punish}_{q}[u]\) is unbounded.

**Proof:** Consider any unstable process \(u\). By definition, \(u\) will crash and recover an infinite number of times. Every time \(u\) recovers, it sends a \(\text{(RECOVERED, } u)\) message to all the processes, and hence \(u\) will send an infinite number of \(\text{(RECOVERED, } u)\) messages. An infinite subset of those messages will reach some correct process \(q\) which will increment \(\text{punish}_{q}[u]\) accordingly (Line 11). Since after time \(t_{1}\) correct processes will not crash, \(\text{punish}_{q}[u]\) is unbounded.

At any time \(t > t_{1}\), if process \(q \in R\) every message it sends will reach every correct processes \(p\) in at most \(\Delta\) time, setting \(\text{punish}_{q}[u] \geq \text{punish}_{q}[u]\). If process \(q \notin R\), by definition \(q\) will have at least one fair lossy asynchronous link/path to a correct process \(p \in R\), and eventually \(p\) will receive an \(\text{(ALIVE, } q, \text{punish}_{q})\) message, setting \(\text{punish}_{p}[u] \geq \text{punish}_{q}[u]\). After that the rest of alive processes will receive \(\text{punish}_{p} in
at most $\Delta$ time. Once a correct process $s$ receives a message from $p$, it will set $\text{punish}_s[u] \geq \text{punish}_p[u]$, and the lemma holds.

Henceforth we will assume that any time instant $t$ is larger than a time $t_2 > t_1$, where $t_2$ is a time instant that occurs after every correct process $s \in R$ has $\text{punish}_s[u]$ such that $\text{punish}_s[u] > \Delta$.

**Lemma 2** $\forall s \in R$, $\text{punish}_s[s]$ is bounded.

**Proof:** Consider any correct process $q \neq s$. Process $s$ sends a message $(ALIVE, s, \text{punish}_s)$ every $\eta$ time to every process. By definition, after time $T$ every message that $s$ sends is received by $q$ within $\Delta + \eta$ time from the time $q$ received the previous message from $s$. Since $q$ increases its timer $\text{Timeout}_q[s]$ every time it expires, eventually $\text{timer}_q(s)$ will cease expiring. Henceforth, $q$ will never punish $s$ (Line 26) anymore, and $s$ will not increase $\text{punish}_s[s]$ due to a message from any $q \in \text{correct}$.

On the other hand, every unstable process $u$ will not reset its timers until the reception of an $ALIVE$ message from a majority of processes. Since there is a majority of correct processes, we can assure that the process $u$ have received a message from at least one correct process, and $u$ will have $\text{punish}_u[u] \geq \text{punish}_s[u]$. Since after time $t_2$, at process $s$, $\text{punish}_s[u] > \Delta$, process $u$ will have $\text{Timeout}_u[s] \geq \text{punish}_s[u]$ (Line 15), and $\text{timer}_u(s)$ will never expire.

Henceforth, there is a time $t > t_2$ after which neither unstable nor correct processes will expire on $s$, $s$ will not be punished (Line 26), $\text{punish}_s[s]$ is bounded, and the lemma holds.

From the previous, note that $R \subseteq B$.

**Lemma 3** For every correct process $p \in B$ there exists a time after which every $q \in \text{correct}$ receives messages from $p$ infinitely often.

**Proof:** Consider a correct process $q \neq p$. We prove the contrapositive of the lemma. Suppose $q$ does not receive messages from $p$ infinitely often. Each time $q$ does not receive a message from $p$ and $\text{timer}_q(p)$ expires, process $p$ is punished by $q$ in $\text{punish}_q[p]$. Later, an infinite subset of the $ALIVE$ messages sent by $q$ could be received by $p$, increasing $\text{punish}_p[p]$, or at least by some process $s$, $s \in R$. The process will increase $\text{punish}_s[p]$, and the next time $p$ receives a message from $s$, it will increase $\text{punish}_p[p]$ accordingly. If this happens infinitely often, $\text{punish}_p[p]$ is not bounded, leading us to a contradiction.

For the rest of the section we will assume that any time instant $t$ is larger than time $t_3 > t_2$, where $t_3$ is a time instant that occurs after $\text{punish}_p[u] > \text{punish}_p[p], \forall u \in \text{unstable}, \forall p \in B$, and for every eventually down process $q$, $q \notin \text{candidates}_p$. This will eventually happen because $\text{punish}_p[p]$ is unbounded and timers on every eventually down process $q$ will expire. After that (Line 27) $q$ will be removed from $\text{candidates}_p$.

**Lemma 4** For every pair of correct processes $p$ and $q$, $p \in B$, there is a time after which for every time $t$, $\text{punish}_q[p] \geq \text{punish}_p[p]$.

**Proof:** For $q = p$, the lemma is trivial. Now assume $q \neq p$. Since $p \in B$, by Lemma 3 there exists a time after which every $q \in \text{correct}$ receives messages from $p$ infinitely often. Let $t > t_3$ be any time. There is a time $t' > t$ when $q$ receives $(ALIVE, p, \text{punish}_p)$, with $\text{punish}_p[p] = c$, originally sent by $p$ after time $t$, so $c \geq \text{punish}_p[p]$. Then at time $t'$, $q$ sets its $\text{punish}_q[p]$ to $c$, and so we have: $\text{punish}_q[p] \geq \text{punish}_p[p]$. The lemma now follows since $\text{punish}_q[p]$ is monotonically nondecreasing.

**Lemma 5** For every correct process $p$: 1. If $\text{punish}_p[p]$ is bounded, then there exists a value $V_p$ and a time after which for every correct process $q$, $\text{punish}_q[p] = V_p$. 2. If $\text{punish}_p[p]$ is not bounded, then for every correct process $q$, $\text{punish}_q[p]$ is not bounded.

**Proof:** Let $p$ be a correct process.

(1) Suppose $\text{punish}_p[p]$ is bounded. Thus, by Lemma 4, for all correct processes $q$, there is a time $t > t_3$ after which $\text{punish}_q[p] \geq \text{punish}_p[p]$. Since $\text{punish}_p[p]$ is bounded and monotonically nondecreasing, there exists a value $V_p$ and a time after which $\text{punish}_p[p] = V_p$. Therefore, there exists a time after which, for all correct processes $q$, $\text{punish}_q[p] = V_p$.

(2) Suppose $\text{punish}_p[p]$ is not bounded. Lemma 4 implies that $\text{punish}_q[p]$ is also not bounded.

**Lemma 6** If process $k$ is not correct then for every correct process $q$ there is a time after which $k$ will not be leader$_q$.

**Proof:** If process $k$ is unstable, after time $t > t_3$, $\text{punish}_k[k] > \text{punish}_p[p]$, for every $p \in B$. As $q$ is correct every message broadcast by every process $p$ reaches timely every correct process $q$, $\text{punish}_q[k] \geq \text{punish}_p[k]$, and process $k$ will not be elected as leader anymore. If process $k$ is eventually
down, after time $t_3$, $k \notin \text{candidates}_p$. In both cases, $\text{leader}_q \neq k$ and the lemma holds. ■

**Lemma 7** There exists a correct process $l$ and a time after which, for every correct process $q$, $\text{leader}_q = l$.

**Proof:** Note that $B$ is not empty. By Lemma 5(1), for every process $p \in B$, there is a corresponding integer $V_p$ and a time after which for every correct process $q$, $\text{punish}_q[p] = V_p$ (forever). Let $l$ denote the process $p$ in $B$ with the smallest corresponding tuple $(V_p, p)$. We now show that eventually every correct process $q$ selects $l$ as its leader (forever). For any other process $p \neq l$: (*) there is a time after which $(\text{punish}_q[p], p) > (\text{punish}_q[l], l)$. This implies that eventually $q$ selects $l$ as its leader, forever. To show that (*) holds, consider the following cases. If $p$ is not correct then, by Lemma 6, eventually $p$ will never be elected as leader (forever). Now suppose that $p$ is correct. If $\text{punish}_p[p]$ is bounded, then $p$ is in $B$; so, by our selection of $l$ in $B$, eventually $(\text{punish}_q[p] = V_p, p) > (\text{punish}_q[l] = V_l, l)$ forever. Finally, if $\text{punish}_p[p]$ is not bounded, then, by Lemma 5(2), there is a time after which $\text{punish}_q[p] > \text{punish}_q[l] = V_l$ (because $\text{punish}_q[p]$ is unbounded and monotonically nondecreasing). In all cases (*) holds.

For the rest of the section we will assume that any time instant $t$ is larger than time $t_4 > t_3$, where $t_4$ is a time instant that occurs after Lemma 7 holds.

**Lemma 8** There is a time after which, for every unstable process $u$, when up, $\text{leader}_u = \bot$ or $\text{leader}_u = l$, being $l$ the same as in Lemma 7.

**Proof:** Every time an unstable process $u$ recovers from a crash, it will set $\text{leader}_u$ to $\bot$. Then, $u$ will wait until the reception of an ALIVE message from a majority of processes, in order to activate its timers and call $\text{updateLeader}()$. After the waiting period, $u$ has received a message from at least one correct process $q$. Once $u$ executes Line 14, $\forall p \in S$, $\text{punish}_u[p] \geq \text{punish}_q[p]$, and after Line 24 $\text{leader}_u = l$. Since $l \in B$, the timers of the unstable processes will not expire on $l$, and the lemma holds. ■

**Theorem 1** The algorithm of Figure 1 implements Omega (satisfies Property 2) in system $S$.

**Proof:** Follows directly from Lemma 7 and Lemma 8. ■

---

4. **On the Eventual Timeliness of Fair Lossy Links**

Figures 5 to 8 present several scenarios satisfying the assumptions required by the algorithm. In Figure 5, $p_1$ or $p_2$ will eventually become the leader (unless $p_5$ communicates timely with $p_1$). In Figures 6 and 7, any of the processes $p_1$, $p_2$ or $p_5$ will eventually become the leader. Figure 8 differs from Figure 5 in the direct fair lossy link from $p_5$ to $p_2$. In this scenario, besides $p_1$ and $p_2$, process $p_5$ could also become the leader, if it can communicate timely with either $p_1$ or $p_2$.

In summary, a correct process could become the leader even if it does not have an eventually timely link/path with the rest of correct and unstable processes, provided it can communicate timely with those processes (through fair lossy links/paths). If it is the case, the links/paths from such process to the rest of correct and unstable processes can be defined as *lossy but eventually timely*. Clearly, this is a behavioral definition, since a priori nothing can be said about the timeliness of fair lossy links. That’s why we require the exis-
5. Agreeing on a Common Set of Correct Processes

In the algorithm of Figure 1, for every correct process \( p \) (included the leader), unstable processes can be included and removed from \( \text{candidates}_p \) infinitely often. Even other correct processes (except the leader) can be included and removed from \( \text{candidates}_p \) infinitely often, since there is not an eventually timely path between every pair of correct processes. The only thing that can be ensured is that eventually down processes will be removed definitely from \( \text{candidates}_p \).

However, it is possible to adapt the algorithm of Figure 1 in order to agree on a common set of \( k \) correct processes, being \( k \) a value provided by the application/protocol using the failure detector, e.g., a Consensus protocol.\(^2\) Obviously, \( k \) should be less or equal the minimum number of correct processes. As a particular case, when \( k = 1 \) we get an Omega failure detector. The property satisfied by the adapted algorithm is the following:

**Property 3** There is a time after which (1) every correct process always trusts the same set of \( k \) correct processes \( \text{correct}_p \), and (2) every unstable process, when up, always trusts either \( \emptyset \) (i.e., it does not trust any process) or \( \text{correct}_p \). More precisely, upon recovery it trusts first \( \emptyset \), and —if it remains up for sufficiently long— then \( \text{correct}_p \) until it crashes.

The adapted algorithm relies on the following assumptions on communication reliability and synchrony, which are stronger than for Omega:

- For every correct process \( p \), there is an eventually timely path from \( p \) to every correct and every unstable process.
- For every unstable process \( u \), there is a fair lossy link from \( u \) to some correct process.

The adaptation, presented in Figure 9 is straightforward. It consists in selecting the \( k \) processes in \( \text{candidates}_p \) with lowest associated punish value. This can be done inside a procedure \( \text{updateCorrectSet}() \), similar to the procedure \( \text{updateLeader}() \) of the algorithm of Figure 1.

Every process \( p \) executes the following:

\[
\begin{align*}
&\text{procedure } \text{updateCorrectSet}() \\
&(p1) \quad \text{correct}_p \leftarrow k \text{ processes} \in \text{candidates}_p \quad \text{with lowest associated value in } \text{punish}_p \\
&\text{end procedure}
\end{align*}
\]

**Initialization:**

( 1 ) \( \text{correct}_p \leftarrow \emptyset \)

![Figure 9. Agreeing on a common set of correct processes.](image)

6. Conclusion

In this work, we have addressed the implementation of the Omega failure detector in the crash-recovery failure model without using any form of stable storage. To do that, we have first redefined the property of Omega in

\(^2\)We assume in this work that a majority of processes are correct.
that model. Then, we have proposed an algorithm implementing Omega under some weak assumptions on reliability and synchrony. Finally, we have discussed about the eventual timeliness of fair lossy links, and presented an adaptation of our algorithm in order to agree on a common set of correct processes.

Acknowledgments

Research partially supported by the Spanish Research Council, under grants TIN2007-67353-C02-02 and TIN2006-15617-C03-01, the Basque Government, under grant S-PE07IK03, and the Comunidad de Madrid, under grant S-0505/TIC/0285.

References