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, Π 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 δ 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 $\perp$ (i.e., it does not trust any process) or l. More precisely, upon recovery it trusts first $\perp$, 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:

procedure updateLeader()
(p1)    $leader_p \leftarrow l$ such that $punish_p[l] = min\{punish_p[q]\}, \forall q \in candidates_p$
end procedure

Initialization:
( 1)    $leader_p \leftarrow \perp$
( 2)    $candidates_p \leftarrow \Pi$
( 3)    $\forall q \neq p : Timeout_p[q] \leftarrow$ default time-out interval
( 4)    $\forall q : punish_p[q] \leftarrow 0$
( 5)    send $(RECOVERED, p)$ to all processes
( 6)    $timers\_active \leftarrow FALSE$
( 7)    start tasks 1, 2, 3 and 4

Task 1:
( 8)    repeat forever every $\eta$ time units
( 9)        send $(ALIVE, p, punish_p)$ to all processes

Task 2:
(10)    upon reception of message $(RECOVERED, q)$ do
(11)        $punish_p[q] \leftarrow punish_p[q] + 1$

Task 3:
(12)    upon reception of message $(ALIVE, q, punish_q)$ with $q \neq p$ for the first time do
(13)        send $(ALIVE, q, punish_q)$ to all processes
(14)        $\forall r : punish_p[r] \leftarrow \max\{punish_p[r], punish_q[r]\}$
(15)        $\forall r : Timeout_p[r] \leftarrow \max\{Timeout_p[r], punish_p[p]\}$
(16)        if $p$ has received so far $ALIVE$ from a majority of processes then
(17)            if $timers\_active = FALSE$ then
(18)                $\forall q \neq p$ : reset $timer_p(q)$ to $Timeout_p[q]$
(19)                $timers\_active \leftarrow TRUE$
(20)            if $q \notin candidates_p$ then
(21)                $candidates_p \leftarrow candidates_p \cup \{q\}$
(22)                $Timeout_p[q] \leftarrow Timeout_p[q] + 1$
(23)            reset $timer_p(q)$ to $Timeout_p[q]$
(24)            $updateLeader()$

Task 4:
(25)    upon expiration of $timer_p(q)$ do
(26)        $punish_p[q] \leftarrow punish_p[q] + 1$
(27)        $candidates_p \leftarrow candidates_p - \{q\}$
(28)        $updateLeader()$
```

Figure 1. Algorithm implementing Omega in system S.

Every process p has a $leader_p$ variable containing its trusted process (initialized to $\perp$), and a $candidates_p$ set containing the processes among which p will choose $leader_p$ (initialized to Π). 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 $punish_p[q]$.

In Task 3, when p receives an $ALIVE$ message from $q \neq p$ 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 $leader_p$ to the process in $candidates_p$ with the minimum associated counter. In Task 4, when $timer_p(q)$ expires, p increments $punish_p[q]$, removes q from $candidates_p$ and calls $updateLeader()$. Note that $timer_p(q)$ 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 $leader_p$ either $\perp$ (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) $punish_u$ such that l is chosen as leader, and (2) $Timeout_u[l]$ such that $timer_u(l)$ 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_2 will eventually become the leader (unless p_1 or p_5 communicate timely with p_2 through the fair lossy link/path).

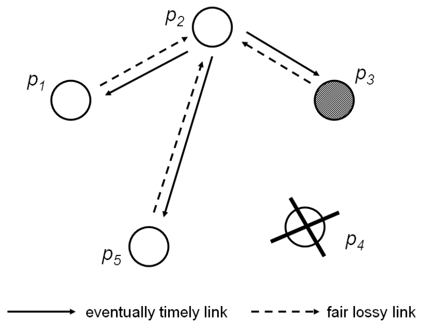

Figure 2. Scenario 1: three eventually up, one eventually down, one unstable.

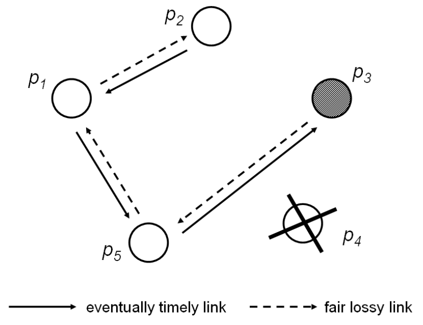

Figure 3. Scenario 2: three eventually up, one eventually down, one unstable.

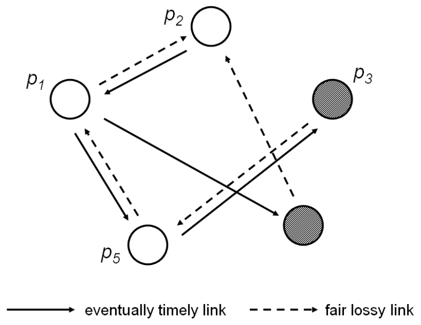

Figure 4. Scenario 3: three eventually up, two unstable.

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 S. By definition, there is a constant Δ and a time T after which every message sent by a process s, $s \in R$, takes at most Δ time to be received by every alive process. Let B be the set of correct processes p with bounded $punish_p[p]$.

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 var_{p_t} the value of the local variable var of p at time t.

Lemma 1 *$\forall q \in correct, \forall u \in unstable, punish_q[u]$ 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 ($RECOVERED$, u) message to all the processes, and hence u will send an infinite number of ($RECOVERED$, u) messages. An infinite subset of those messages will reach some correct process q which will increment $punish_q[u]$ accordingly (Line 11). Since after time t_1 correct processes will not crash, $punish_q[u]$ 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 Δ time, setting $punish_p[u] \geq punish_{q_t}[u]$. 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 ($ALIVE$, q, $punish_q$) message, setting $punish_p[u] \geq punish_{q_t}[u]$. After that the rest of alive processes will receive $punish_p$ in at most Δ time. Once a correct process s receives a message from p, it will set $punish_s[u] \geq punish_{q_t}[u]$, 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 $punish_s[u]$ such that $punish_s[u] > \Delta$.

Lemma 2 *$\forall s \in R$, $punish_s[s]$ is bounded.*

Proof: Consider any correct process $q \neq s$. Process s sends a message ($ALIVE$, s, $punish_s$) every η 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 $Timeout_q[s]$ every time it expires, eventually $timer_q(s)$ will cease expiring. Thenceforth, q will never punish s (Line 26) anymore, and s will not increase $punish_s[s]$ due to a message from any $q \in correct$.

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 $punish_u[u] \geq punish_s[u]$. Since after time t_2, at process s, $punish_s[u] > \Delta$, process u will have $Timeout_u[s] \geq punish_u[u]$ (Line 15), and $timer_u(s)$ will never expire.

Thenceforth, 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), $punish_s[s]$ 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 $q \in correct$ 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 $timer_q(p)$ expires, process p is punished by q in $punish_q[p]$. Later, an infinite subset of the $ALIVE$ messages sent by q could be received by p, increasing $punish_p[p]$, or at least by some process s, $s \in R$. The process will increase $punish_s[p]$, and the next time p receives a message from s, it will increase $punish_p[p]$ accordingly. If this happens infinitely often, $punish_p[p]$ 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 $punish_p[u] > punish_p[p]$, $\forall u \in unstable$, $\forall p \in B$, and for every eventually down process q, $q \notin candidates_p$. This will eventually happen because $punish_p[u]$ is unbounded and timers on every eventually down process q will expire. After that (Line 27) q will be removed from $candidates_p$.

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

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 $q \in correct$ receives messages from p infinitely often. Let $t > t_3$ be any time. There is a time $t' > t$ when q receives ($ALIVE$, p, $punish_p$), with $punish_p[p] = c$, originally sent by p after time t, so $c \geq punish_{p_t}[p]$. Then at time t', q sets its $punish_q[p]$ to c, and so we have: $punish_q[p] \geq punish_{p_t}[p]$. The lemma now follows since $punish_q[p]$ is monotonically nondecreasing. ■

Lemma 5 *For every correct process p: 1. If $punish_p[p]$ is bounded, then there exists a value V_p and a time after which for every correct process q, $punish_q[p] = V_p$. 2. If $punish_p[p]$ is not bounded, then for every correct process q, $punish_q[p]$ is not bounded.*

Proof: Let p be a correct process.

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

(2) Suppose $punish_p[p]$ is not bounded. Lemma 4 implies that $punish_q[p]$ 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 $t > t_3$, $punish_p[k] > punish_p[p]$, for every $p \in B$. As q is correct every message broadcast by every process p reaches timely every correct process q, $punish_q[k] \geq punish_p[k]$, and process k will not be elected as leader anymore. If process k is eventually down, after time t_3, $k \notin candidates_p$. In both cases, $leader_q \neq k$ and the lemma holds. ■

Lemma 7 *There exists a correct process l and a time after which, for every correct process q, $leader_q = l$.*

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, $punish_q[p] = V_p$ (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 $(punish_q[p], p) > (punish_q[l], l)$. This implies that eventually q selects l as its leader, forever. To show that (*) holds, consider the following 3 possible 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 $punish_p[p]$ is bounded, then p is in B; so, by our selection of l in B, eventually $(punish_q[p] = V_p, p) > (punish_q[l] = V_l, l)$ forever. Finally, if $punish_p[p]$ is not bounded, then, by Lemma 5(2), there is a time after which $punish_q[p] > punish_q[l] = V_l$ (because $punish_q[p]$ 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, $leader_u = \perp$ or $leader_u = l$, being l the same as in Lemma 7.*

Proof: Every time an unstable process u recovers from a crash, it will set $leader_u$ to $\perp$. Then, u will wait until the reception of an $ALIVE$ message from a majority of processes, in order to activate its timers and call $updateLeader()$. After the waiting period, u has received a message from at least one correct process q. Once u executes Line 14, $\forall p \in S$, $punish_u[p] \geq punish_q[p]$, and after Line 24 $leader_u = l$. 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.

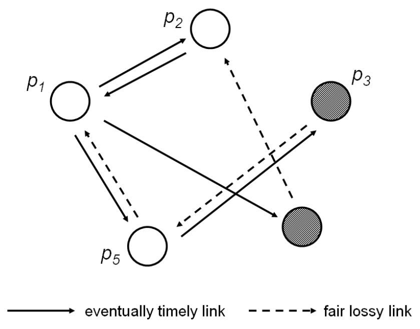

Figure 5. Scenario 4: three eventually up, two unstable.

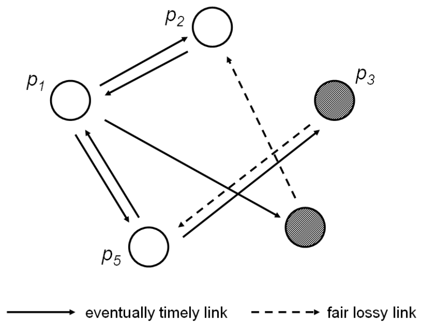

Figure 6. Scenario 5: three eventually up, two unstable.

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-

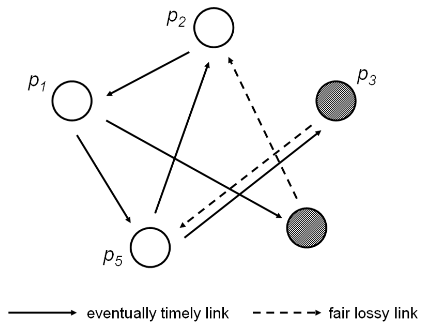

Figure 7. Scenario 6: three eventually up, two unstable.

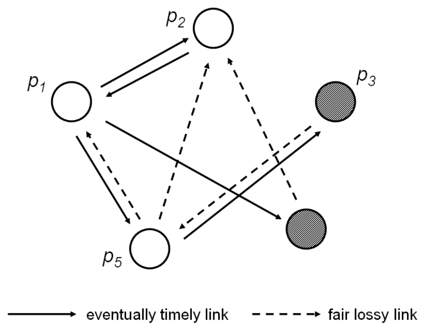

Figure 8. Scenario 7: three eventually up, two unstable.

tence of a correct process having an eventually timely link/path with the rest of correct and unstable processes, since it ensures that the algorithm stabilizes on a common and correct leader, independently of the behavior of fair lossy links.

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 $candidates_p$ infinitely often. Even other correct processes (except the leader) can be included and removed from $candidates_p$ 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 $candidates_p$.

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 $correct\_set$, and (2) every unstable process, when up, always trusts either $\emptyset$ (i.e., it does not trust any process) or $correct\_set$. More precisely, upon recovery it trusts first $\emptyset$, and —if it remains up for sufficiently long— then $correct\_set$ 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 $candidates_p$ with lowest associated punish value. This can be done inside a procedure $updateCorrectSet()$, similar to the procedure $updateLeader()$ of the algorithm of Figure 1.

```
Every process p executes the following:

procedure $updateCorrectSet()$
(p1)    $correct\_set_p \leftarrow k$ processes $\in candidates_p$
                      with lowest associated value in $punish_p$
end procedure

Initialization:
( 1)    $correct\_set_p \leftarrow \emptyset$
...
```

Figure 9. Agreeing on a common set of correct processes.

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

[1] M. Aguilera, W. Chen, and S. Toueg. Failure detection and consensus in the crash-recovery model. *Distributed Computing*, 13(2):99–125, 2000.

[2] M. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. Stable leader election. In *Proceedings of the 15th International Symposium on Distributed Computing (DISC'2001)*, pages 108–122, Lisbon, Portugal, October 2001. LNCS 2180, Springer-Verlag.

[3] M. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. On implementing Ω with weak reliability and synchrony assumptions. In *Proceedings of the 22nd ACM Symposium on Principles of Distributed Computing (PODC'2003)*, pages 306–314, Boston, Massachusetts, July 2003.

[4] T. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722, July 1996.

[5] T. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.

[6] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, April 1988.

[7] A. Fernández, E. Jiménez, and M. Raynal. Eventual leader election with weak assumptions on initial knowledge, communication reliability, and synchrony. In *Proceedings of the IEEE International Conference on Dependable Systems and Networks (DSN'2006)*, pages 166–178, Philadelphia, Pennsylvania, June 2006.

[8] R. Guerraoui and M. Raynal. The information structure of indulgent consensus. *IEEE Transactions on Computers*, 53(4):453–466, April 2004.

[9] E. Jiménez, S. Arévalo, and A. Fernández. Implementing Unreliable Failure Detectors with Unknown Membership. *Information Processing Letters*, 100(2):60–63, 2006.

[10] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.

[11] M. Larrea, A. Fernández, and S. Arévalo. Optimal implementation of the weakest failure detector for solving consensus. In *Proceedings of the 19th IEEE Symposium on Reliable Distributed Systems (SRDS'2000)*, pages 52–59, Nurenberg, Germany, October 2000.

[12] M. Larrea, A. Fernández, and S. Arévalo. Eventually consistent failure detectors. *Journal of Parallel and Distributed Computing*, 65(3):361–373, March 2005.

[13] C. Martín, M. Larrea, and E. Jiménez. On the implementation of the Omega failure detector in the crash-recovery failure model. In *Proceedings of the ARES 2007 Workshop on Foundations of Fault-tolerant Distributed Computing (FOFDC'2007)*, pages 975–982, Vienna, Austria, April 2007.

[14] A. Mostéfaoui and M. Raynal. Leader-based consensus. *Parallel Processing Letters*, 11(1):95–107, 2001.

[15] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, April 1980.

[16] M. Wiesmann and X. Défago. End-to-end consensus using end-to-end channels. In *Proceedings of the 12th Pacific Rim International Symposium on Dependable Computing (PRDC'2006)*, pages 341–350, Riverside, CA, USA, December 2006.