Journal of Computer and System Sciences 75 (2009) 178–189

Contents lists available at ScienceDirect

Journal of Computer and System Sciences

www.elsevier.com/locate/jcss

Implementing the Omega failure detector in the crash-recovery failure model ☆

Cristian Martín [a], Mikel Larrea [a,*], Ernesto Jiménez [b]

[a] Departamento de Arquitectura y Tecnología de Computadores, Universidad del País Vasco, Paseo Manuel de Lardizabal 1, 20018 San Sebastián, Spain
[b] Departamento de Arquitectura y Tecnología de Computadores, Universidad Politécnica de Madrid, Ctra Valencia km. 7, 28031 Madrid, Spain

ARTICLE INFO

Article history:
Received 23 July 2007
Received in revised form 9 October 2008
Available online 1 November 2008

Keywords:
Distributed algorithms
Consensus
Omega failure detector
Eventual leader election

ABSTRACT

Unreliable failure detectors are mechanisms providing information about process failures, that allow to solve several problems in asynchronous systems, e.g., Consensus. A particular failure detector, Omega, provides an eventual leader election functionality. This paper addresses the implementation of Omega in the crash-recovery failure model. We first propose an algorithm assuming that processes are reachable from the correct process that crashes and recovers a minimum number of times. Then, we propose two algorithms which assume only that processes are reachable from some correct process. Besides this, one of the algorithms requires the membership to be known a priori, while the other two do not.

1. Introduction

Unreliable failure detectors were proposed by Chandra and Toueg [1] as a mechanism that provides (possibly incorrect) information about process failures. This mechanism has been used to solve several problems in crash-prone asynchronous distributed systems, in particular the Consensus problem [2]. In this paper, we focus on a failure detector called Omega, proposed by Chandra, Hadzilacos and Toueg in [3], that provides eventual agreement on a common leader among all non-faulty processes in a system. The Omega failure detector has been shown to be the weakest failure detector for solving Consensus [3]. In this regard, algorithms solving Consensus using Omega have been proposed, e.g., [4].

Several algorithms implementing Omega have been proposed in the literature. Among them, we can include all the algorithms that implement the Eventually Perfect class of failure detectors $\diamond\mathcal{P}$, e.g., [1,5,6], since Omega can be trivially obtained from $\diamond\mathcal{P}$, e.g., by choosing as leader the nonsuspected process with the lowest identifier. The algorithms in [1,5,6] assume that every pair of processes (p, q) is connected by two unidirectional communication links $p \rightarrow q$ and $q \rightarrow p$, being all the links *eventually timely*, i.e., eventually all messages are delivered within an unknown time bound.

Specific algorithms for Omega have also been proposed. Larrea et al. [7] propose an algorithm implementing Omega and $\diamond\mathcal{S}$ which also requires all links to be eventually timely. Aguilera et al. [8] propose an Omega algorithm for systems where some unknown correct process must have all its (incoming and outgoing) links eventually timely, while all other links can be *lossy asynchronous*, i.e., messages can be lost or arbitrarily delayed. In [9], Aguilera et al. propose another Omega algorithm for a weaker system in which only the outgoing links from some unknown correct process to the rest of processes must be eventually timely. In [10], Aguilera et al. propose an algorithm for a system in which at most f processes can crash, links are fair lossy, and some correct process has f eventually timely outgoing links. In [11], Jiménez et al. propose an algorithm

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

* Corresponding author. Fax: +34 943 015590.
E-mail address: mikel.larrea@ehu.es (M. Larrea).

0022-0000/$ – see front matter
doi:10.1016/j.jcss.2008.10.002

implementing Omega with unknown membership which requires that eventually all correct processes are reachable timely from the correct process with smallest identifier. Finally, in [12], Jiménez et al. propose another algorithm implementing Omega with unknown membership which requires that eventually all correct processes are reachable timely from some correct process.

All the above mentioned algorithms consider a crash failure model, in which once a process crashes it does not recover. Failure detection and Consensus in the crash-recovery failure model has been studied in [13] (with crash-recovery as a form of omission failure) and [14–16]. In [13,15,16], adaptations of unreliable failure detector classes $\diamond\mathcal{W}$ and/or $\diamond\mathcal{S}$ to the crash-recovery failure model are defined and Consensus protocols based on the new classes are proposed. However, no algorithm implementing those failure detectors is provided. In [14], Aguilera et al. also define an adaptation of $\diamond\mathcal{S}$ to the crash-recovery failure model and propose an algorithm implementing it in partially synchronous systems [1,17]. The algorithm assumes a fully connected system and requires the membership to be known a priori by processes.

In this paper, we address the implementation of Omega in the crash-recovery failure model for systems not necessarily fully connected, proposing three algorithms:

- A first algorithm assuming that eventually all processes are reachable timely from the correct process that crashes and recovers a minimum number of times. This algorithm does not require the membership to be known a priori.
- A second algorithm assuming that eventually all processes are reachable timely from some correct process. This algorithm requires the membership to be known a priori.
- A third algorithm assuming that eventually all processes are reachable timely from some correct process (as in the second algorithm), which does not require the membership to be known a priori (as in the first algorithm).

The reachability condition assumed for the second and third algorithms, which will be further refined to adapt to our new definition of Omega in the crash-recovery failure model, has been shown to be minimal for implementing Omega in the crash failure model [18]. As we will see, the second and third algorithms will choose as leader the correct process that is "less suspected" among those that reach timely all processes.

The rest of the paper is organized as follows. In Section 2, we describe the system model considered in this work, and redefine the property of Omega in the crash-recovery failure model. In Section 3, we present the first algorithm implementing Omega. In Section 4, we weaken the synchrony assumption in order to implement Omega, and present the second and third algorithms. In Section 5, we discuss about system models regarding their connectivity and synchrony. Finally, Section 6 concludes the paper.

2. System model

We consider a system S composed of a finite and totally ordered set Π of $n > 1$ processes that communicate only by sending and receiving messages. Processes are connected by unidirectional communication links. In general, not all pairs of processes can communicate *directly*, i.e., there not need to be a communication link between every pair of processes. To send messages, processes have a broadcasting primitive allowing a process to send the same message m through each of its outgoing links.

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*. This is the subset of processes that, after crashing and recovering a finite number of times, remain up forever, i.e., they do not crash any more. Processes that never crash are included in this subset.
(2) *Eventually down*. This is the subset of processes that, after crashing and recovering a finite number of times, remain down forever, i.e., they do not recover any more. Processes that never start their execution are included in this subset.
(3) *Unstable*. This is the subset of processes that crash and recover an infinite number of times, i.e., there is not a time after which either they remain up forever, or they remain down forever.

By definition, processes in (1) are *correct*, and processes in (2) and (3) are *incorrect*. We assume that the number of correct processes in the system in any run is at least one. We also assume that every process has access to stable storage to keep the value of some private variables, in particular an incarnation number, initialized to 0, which is incremented during initialization and every time a process recovers from a crash.

Processes execute by taking atomic steps. We assume the existence of a lower bound σ on the number of steps per unit of time taken by any process. For simplicity, we assume that each line of our algorithms represents one step. We also assume that each task of our algorithms is allowed to run. Processes have clocks that accurately measure intervals of time but are not necessarily synchronized.

We consider two types of links in S: eventually timely links and lossy asynchronous links. In eventually timely links, 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 through any of these links at a time $t \geqslant T$, then this message is received by time $t + \delta$ (if the receiver

process is up).[1] In lossy asynchronous links, messages can be lost or arbitrarily delayed. We consider that no link in S modifies its messages nor generates spontaneous messages. However, it may duplicate messages or deliver them out of order. For simplicity, we assume that messages are unique, in the sense that we can determine whether a message received is a duplicate of a previously received message. This can be achieved by including the sender process identifier and a sequence number into each message.

2.1. Redefinition of Omega in the crash-recovery failure model

Chandra, Hadzilacos and Toueg defined in [3] 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 *(Omega-crash). There is a time after which all the* correct *processes always trust the same correct process.*

Note that the output of the failure detector module of Omega at a process p may change through the time, i.e., p may trust different processes at different times. Furthermore, at any given time t, two processes p and q may trust different processes.

In practice, Omega can be queried by application processes at any time, e.g., to solve Consensus. Note that in the crash-recovery failure model it is not possible for a process to determine if it is a correct process, or on the contrary it is an eventually down (but still up) process, or even an unstable (but up) process. Moreover, usually termination of Consensus cannot be ensured if correct processes select a leader different from the one selected by unstable processes. It could be interesting in such a scenario that eventually all the *active* processes, i.e., processes that are up and have completed the initialization phase, agree on a common (correct) leader process. Hence, we redefine the property that Omega must satisfy, adapted to the crash-recovery failure model.

Property 2 *(Omega-crash-recovery). There is a time after which all the* active *processes always trust the same correct process.*

As we will see, the three algorithms implementing Omega proposed in this work satisfy Property 2.

3. A first algorithm

In this section we present a first algorithm, adapted from [11], that implements Omega in system S. This algorithm assumes that eventually all processes are reachable timely from the correct process that crashes and recovers a minimum number of times. It does not require the membership of the system — the process identifiers — to be known a priori by processes,

Let us denote by c_{min} the correct process in S with the smallest identifier among those that have the minimum incarnation number $incarnation_{min}$. Let us denote by $G(S)$ the directed graph, obtained from S, with the sets of correct and unstable processes as vertex set and the set of eventually timely outgoing links of correct processes as edge set. We assume that all vertexes in $G(S)$ are reachable (either directly or indirectly) from c_{min}.

Property 3. *Eventually every process $q \in (correct \cup unstable)$ can be reached from c_{min} in $G(S)$.*

Clearly, unstable processes will only be reached if, each time they recover, they remain up a sufficiently long time. If this does not happen, we do not need to care about them.

Fig. 1 presents the first algorithm in detail. The process chosen as leader by a process p, i.e., trusted by p, is held in a variable $leader_p$. We will show that with this algorithm there is a time after which every active process permanently has $leader_p = c_{min}$, and hence satisfies Property 2.

The basic idea of the algorithm is that eventually only process c_{min} broadcasts new alive messages (*ALIVE*, c_{min}, $incarnation_{min}$) every η time units, and that (copies of) these messages reach the rest of active processes, either directly, or indirectly by re-broadcasting. In the worst case $O(n^2)$ links carry messages forever. In the algorithm, we assume that for any process to send (*ALIVE*, –, –) messages (Lines a11 or a16), it has necessarily incremented its incarnation number by 1 in stable storage during initialization (Line a1). In order to satisfy Property 2, besides $incarnation_p$ every process p keeps in stable storage the value of $leader_p$ (initialized to p), which is read during initialization. We also assume that every unstable process will be able to write in stable storage infinitely often (Line a8).

In order to satisfy Property 2, we include a wait instruction at the beginning of Task 1 (Line a7). After this wait, p writes the value of $leader_p$ in stable storage (Line a8). By the assumption that every unstable process is able to execute Line a8 infinitely often, eventually every unstable process will always write the right leader in stable storage. From this point, whenever an unstable process recovers, it will initialize its leader to the right value (Line a3), satisfying Property 2.

[1] Actually, the bound σ on relative process speeds does not need to hold from the beginning, but from T.

Every process p executes the following:

Initialization:
(a1) increment $INCARNATION_p$ by 1 in stable storage
(a2) $incarnation_p \leftarrow$ read $INCARNATION_p$ from stable storage
(a3) $leader_p \leftarrow$ read $LEADER_p$ from stable storage
(a4) $incarnation_{leader} \leftarrow incarnation_p$
(a5) $Timeout_p \leftarrow \eta + incarnation_p$
(a6) **start tasks** 1, 2 and 3

Task 1:
(a7) wait $(\eta + incarnation_p)$
(a8) write $leader_p$ in stable storage
(a9) **loop forever**
(a10) **if** $[leader_p = p]$ **then**
(a11) broadcast $(ALIVE, p, incarnation_p)$
(a12) **end if**
(a13) wait (η)

Task 2:
(a14) **upon reception of** message $(ALIVE, q, incarnation_q)$ with $q \neq p$ for the first time **do**
(a15) **if** $[incarnation_q < incarnation_{leader}]$ **or** $[(incarnation_q = incarnation_{leader})$ **and** $(q \leqslant leader_p)]$ **then**
(a16) broadcast $(ALIVE, q, incarnation_q)$
(a17) $leader_p \leftarrow q$
(a18) $incarnation_{leader} \leftarrow incarnation_q$
(a19) reset $timer_p$ to $Timeout_p$
(a20) **end if**

Task 3:
(a21) **upon expiration of** $timer_p$ **do**
(a22) $Timeout_p \leftarrow Timeout_p + 1$
(a23) $leader_p \leftarrow p$
(a24) $incarnation_{leader} \leftarrow incarnation_p$

Fig. 1. First algorithm implementing Omega in system S.

Note that every process writes only once $leader_p$ in stable storage every time it starts executing the algorithm. Hence, from the point of view of the number of write operations in stable storage the algorithm is extremely efficient. Another approach consists in writing this value in stable storage more frequently, e.g., (1) periodically, or even better (2) every time it changes. This could help in speeding up stabilization, at the price of a higher number of write operations in stable storage.

Removing the re-broadcast of *ALIVE* messages (Line a16) we get a simplified version of the algorithm that works in a *fully* (*eventually*) *timely connected* system S_F, i.e., a system in which every process has a direct communication link with every other process, all the links being eventually timely. This ensures that eventually every new alive message that process c_{min} broadcasts will be received timely by the rest of active processes directly from c_{min}. Since eventually only process c_{min} broadcasts alive messages, eventually $O(n)$ links would carry messages forever. Note that if S_F is weakened by either (1) removing some links or (2) considering some links as lossy asynchronous, then messages must be re-broadcast in order to guarantee their reception by all the active processes.

Correctness proof

Lemma 1. *Any message* $(ALIVE, p, incarnation_p)$, $p \in \Pi$, *eventually disappears from the system.*

Proof. Note first that a message cannot remain forever in a link, since it remains at most $T + \delta$ time in an eventually timely link, and is lost or eventually delivered in a lossy asynchronous link. Note as well that a message cannot remain forever in a process, since by assumption processes take at least one step (execute at least one line of the algorithm) per unit of time. Then, a process will eventually crash, drop the message (Lines a14 and a15), or (re-)broadcast it (Lines a11 or a16). Finally, note that a process never re-broadcasts twice the same message and never re-broadcasts its own messages (Line a14). Hence a message can be (re-)broadcast at most n times, and will eventually disappear from the system. □

For the rest of the proof we will assume that any time instant t is larger than a time $t'_{base} > t_{base}$, where:

(1) t_{base} is a time instant that occurs after the stabilization time T (i.e., $t_{base} > T$), and after every eventually down process has definitely crashed, every eventually up process has definitely recovered, and every unstable process has an incarnation number bigger than $incarnation_{min}$,

(2) and t'_{base} is a time instant such that all messages broadcast for the first time before t_{base} have disappeared from the system (this eventually happens from Lemma 1). In particular, this includes (a) all messages broadcast by eventually down processes, (b) all messages broadcast by eventually up processes before recovering definitely, and (c) all messages broadcast by unstable processes with incarnation number less or equal to $incarnation_{min}$.

Lemma 2. *There is a time after which process c_{min} permanently has $leader_{c_{min}} = c_{min}$ and broadcasts a new (ALIVE, c_{min}, $incarnation_{min}$) message every η time.*

Proof. Note that after time t'_{base} process c_{min} will never receive an (*ALIVE*, q, $incarnation_q$) message with $incarnation_q < incarnation_{min}$, or with $incarnation_q = incarnation_{min}$ from a process q such that $q < c_{min}$. Therefore, after time t'_{base} process c_{min} will never execute Lines a16–a19 of the algorithm. Hence once $leader_{c_{min}} = c_{min}$ it will remain so forever. To show that this eventually happens, note that if $leader_{c_{min}} \neq c_{min}$ at time $t > t'_{base}$, then $timer_{c_{min}}$ must be active at that time (actually, $timer_{c_{min}}$ was reset the last time Line a19 was executed). Since after time t'_{base} Line a19 will never be executed, $timer_{c_{min}}$ will not be reset any more. Then $timer_{c_{min}}$ will eventually expire (Line a21), and c_{min} will set $leader_{c_{min}} = c_{min}$ and $incarnation_{leader} = incarnation_{min}$ (Lines a23–a24). Finally, from Task 1, once $leader_{c_{min}} = c_{min}$, process c_{min} will permanently broadcast a new (*ALIVE*, c_{min}, $incarnation_{min}$) message every η time. □

Lemma 3. *There is a time after which every process $p \in correct$, $p \neq c_{min}$, permanently has either (1) $incarnation_{leader} > incarnation_{min}$, or (2) $incarnation_{leader} = incarnation_{min}$ and $leader_p \geqslant c_{min}$. Hence, p re-broadcasts each new (ALIVE, c_{min}, $incarnation_{min}$) message it receives (Line* a16*), since Line* a15 *of the algorithm will be satisfied.*

Proof. Note that, after t'_{base}, once [$incarnation_{leader} > incarnation_{min}$] or [($incarnation_{leader} = incarnation_{min}$) and ($leader_p \geqslant c_{min}$)] is satisfied, it will remain so forever, since no (*ALIVE*, q, $incarnation_q$) message with $incarnation_q < incarnation_{min}$, or with $incarnation_q = incarnation_{min}$ from a process q such that $q < c_{min}$ will be received. Then, if $incarnation_{leader} < incarnation_{min}$, or $incarnation_{leader} = incarnation_{min}$ and $leader_p < c_{min}$ at time $t > t'_{base}$ with either (1) $incarnation_p > incarnation_{min}$, or (2) $incarnation_p = incarnation_{min}$ and $p > c_{min}$, then $timer_p$ must be active at that time. Then $timer_p$ will eventually expire (Line a21), setting either (1) $incarnation_{leader} = incarnation_p > incarnation_{min}$, or (2) $incarnation_{leader} = incarnation_p = incarnation_{min}$ and $leader_p = p > c_{min}$. □

Lemma 4. *There is a time after which every process $p \in correct$, $p \neq c_{min}$, permanently receives new (ALIVE, c_{min}, $incarnation_{min}$) messages with intervals of at most $(\eta + d(p)(\delta + 3\sigma))$ time between consecutive messages, where $d(p)$ is the distance in $G(S)$ from c_{min} to p.*

Proof. The proof uses induction on $d(p)$. If $d(p) = 1$ then p receives new (*ALIVE*, c_{min}, $incarnation_{min}$) messages directly from c_{min}. From Lemma 2, there is a time after which c_{min} sends new messages every η time. The messages take at most δ time to cross the eventually timely link from c_{min} to p. Then, the base case is clearly satisfied. Then, let us assume $d(p) = i > 1$. There must be some process $q \in correct$ with $d(q) = i - 1$ and whose link from q to p is eventually timely. Then, by induction hypothesis q eventually receives new (*ALIVE*, c_{min}, $incarnation_{min}$) messages forever within intervals of $\eta + (i-1)(\delta + 3\sigma)$. Then, from Lemma 3 q will eventually re-broadcast all these messages (Task 2) in at most 3σ time units. Since the messages take at most δ time to cross the link from q to p, the lemma holds. □

Theorem 1. *There is a time after which every active process p permanently has $leader_p = c_{min}$, i.e., p trusts c_{min}. Hence, the algorithm of Fig.* 1 *satisfies Property* 2*, and implements Omega in system S.*

Proof. Lemma 2 shows the claim for $p = c_{min}$. For $p \in correct$, such that $p \neq c_{min}$, from Lemma 3 there is a time after which p permanently has either (1) $incarnation_{leader} > incarnation_{min}$, or (2) $incarnation_{leader} = incarnation_{min}$ and $leader_p \geqslant c_{min}$. From Lemma 4, whenever $leader_p \neq c_{min}$ after this time, $leader_p$ changes back to c_{min} in at most $(\eta + d(p)(\delta + 3\sigma))$ time. Furthermore, once $leader_p = c_{min}$, it only changes (to p) by executing Lines a21–a24, since the conditions in Lines a14 and a15 prevent $leader_p$ from changing in Line a17. Finally, $leader_p$ changes from c_{min} to p a finite number of times, since each time this happens $Timeout_p$ is incremented by 1. By contradiction, assuming this happens an infinite number of times, $Timeout_p$ eventually grows to the point in which $timer_p$ never expires, because new (*ALIVE*, c_{min}, $incarnation_{min}$) messages are received timely and $timer_p$ is reset before expiration. Hence, eventually $leader_p = c_{min}$ permanently. Finally, every unstable process p will eventually receive an (*ALIVE*, c_{min}, $incarnation_{min}$) message during the waiting instruction of Line a7, setting $leader_p = c_{min}$ (Line a17). Then, p will write c_{min} in stable storage (Line a8). After that, p will have $leader_p = c_{min}$ permanently, even upon initialization (Line a3). Hence, the algorithm of Fig. 1 satisfies Property 2, and implements Omega in system S. □

4. Weakening the synchrony assumption

In this section, we weaken the synchrony assumption of the previous section in order to implement Omega in the crash-recovery failure model, proposing two algorithms which assume that eventually all processes are reachable timely from

Every process p executes the following:

procedure *updateLeader*()
(b1) $leader_p \leftarrow l$ such that $counter_p[l] = min\{counter_p\}$
end procedure

Initialization:
(b2) increment $INCARNATION_p$ by 1 in stable storage
(b3) $incarnation_p \leftarrow$ read $INCARNATION_p$ from stable storage
(b4) $leader_p \leftarrow$ read $LEADER_p$ from stable storage
(b5) $\forall q \neq p : Timeout_p[q] \leftarrow \eta + incarnation_p$
(b6) $\forall q \neq p$: reset $timer_p(q)$ to $Timeout_p[q]$
(b7) $\forall q \neq p : counter_p[q] \leftarrow 0$
(b8) $counter_p[p] \leftarrow incarnation_p$
(b9) **start tasks** 1, 2 and 3

Task 1:
(b10) wait $(\eta + incarnation_p)$
(b11) write $leader_p$ in stable storage
(b12) **loop forever**
(b13) broadcast (*ALIVE*, p, $counter_p$)
(b14) wait (η)

Task 2:
(b15) **upon reception of** message (*ALIVE*, q, $counter_q$) with $q \neq p$ for the first time **do**
(b16) broadcast (*ALIVE*, q, $counter_q$)
(b17) $\forall r : counter_p[r] \leftarrow \max\{(counter_p[r], counter_q[r])\}$
(b18) reset $timer_p(q)$ to $Timeout_p[q]$
(b19) *updateLeader*()

Task 3:
(b20) **upon expiration of** $timer_p(q)$ **do**
(b21) $counter_p[q] \leftarrow counter_p[q] + 1$
(b22) $Timeout_p[q] \leftarrow Timeout_p[q] + 1$
(b23) reset $timer_p(q)$ to $Timeout_p[q]$
(b24) *updateLeader*()

Fig. 2. Second algorithm implementing Omega in system S.

some correct process, independently of its identifier and incarnation number. As we will see, the strategy followed by our algorithms is to choose as leader the correct process that is "less suspected" among those that reach timely all processes. Besides this, one of the algorithms requires the membership of the system to be known a priori by processes, while the other one relaxes this assumption too.

Let us denote by $G(S)$ the directed graph, obtained from S, with the sets of correct and unstable processes as vertex set and the set of eventually timely outgoing links of correct processes as edge set. We assume that all vertexes in $G(S)$ are reachable (either directly or indirectly) from some process $p \in correct$.

Property 4. *There is some process $p \in correct$ such that eventually every process $q \in (correct \cup unstable)$ can be reached from p in $G(S)$.*

4.1. A second algorithm

In this section we present a second algorithm, adapted from [9], that implements Omega in system S. This algorithm requires the membership of the system — the process identifiers — to be known a priori by processes. Fig. 2 presents the algorithm in detail. With this algorithm there is a time after which every active process permanently has $leader_p = l$, being l the less suspected process among those that eventually communicate timely with the rest of processes. As in our first algorithm, we assume that every unstable process will be able to write in stable storage infinitely often.

The algorithm works as follows. Every process p has a $counter_p[q]$ for each process q, which is p's estimation of the number of times q has been suspected. Process p selects as its leader the process l with the smallest $counter_p[l]$ value. In order to keep the $counter_p$ variable up to date, every process p broadcasts every η time units an (*ALIVE*, p, $counter_p$) message.[2] If a process p receives a message (*ALIVE*, q, $counter_q$) with $q \neq p$ for the first time, p re-broadcasts the message, updates its $counter_p$ vector accordingly, resets $timer_p(q)$ for when it expects to receive the next (*ALIVE*, q, $counter_q$) message, and calls the procedure *updateLeader*().

[2] The value η should be bigger than σ multiplied by the number of processes in the system, or messages would possibly be queued at each process after arriving timely, and only be processed after their respective timeouts had expired.

If $timer_p(q)$ expires before receiving a new (*ALIVE*, q, $counter_q$) message, then p increments the suspicion counter $counter_p[q]$, increments the value $Timeout_p[q]$, resets $timer_p(q)$, and calls *updateLeader*().

The algorithm includes a mechanism to eventually avoid unstable processes from disturbing the leader election. This mechanism is based on the incarnation number of processes. Observe that, during initialization, every process p initializes its timeouts with respect to the rest of processes to $\eta + incarnation_p$ (Line b5). Also, p initializes $counter_p[p]$ to $incarnation_p$ (Line b8). These initializations ensure that eventually (1) every unstable process p will never suspect a correct process q that reaches timely every other process (since p's timeout with respect q keeps increasing forever, and hence eventually $timer_p(q)$ will never expire), and consequently p will not increment $counter_p[q]$ anymore, and (2) every unstable process p will never be elected as the leader in the *updateLeader*() procedure (since $incarnation_p$, and hence $counter_p[p]$, keeps increasing forever).

Also, the algorithm includes a waiting instruction followed by the write of the leader in stable storage in order to force unstable processes to agree permanently with correct processes on the leader (Lines b10–b11).

The number of messages sent periodically (every η time) in this algorithm is bounded by $n * ul$, being n and ul the number of processes and unidirectional links in the system respectively. This derives from the fact that periodically every alive process sends a new *ALIVE* message (Line b13), and that every message is (re-)sent exactly once through every link of the system (Line b16).

In the algorithm for the crash model of [9], processes (re-)broadcast explicit *ACCUSATION* messages to notify suspicions. By including the whole vector of suspicion counters into *ALIVE* messages, the algorithm of Fig. 2 avoids the broadcast of *ACCUSATION* messages. Observe that the system model allows scenarios in which many pairs of processes cannot communicate timely (either directly or indirectly). In the algorithm in [9], these processes would suspect each other and hence broadcast *ACCUSATION* messages permanently. Thus, avoiding those messages reduces notably the message complexity of the algorithm.

Correctness proof

Let R be the set of correct processes that eventually reach timely all the correct and unstable processes in S. Let B be the set of correct processes p with bounded $counter_p[p]$. By definition, there is a constant Δ and a time after which every message sent by s, $s \in R$, takes at most $\Delta = (n-1)(\delta + 2\sigma)$ time to be received by every correct and unstable (if active) process.

Lemma 5. *Any message (ALIVE, p, $counter_p$), $p \in \Pi$, eventually disappears from the system.*

Proof. Note first that a message cannot remain forever in a link, since it remains at most $T + \delta$ time in an eventually timely link, and is lost or eventually delivered in a lossy asynchronous link. Note as well that a message cannot remain forever in a process, since by assumption processes take at least one step (execute at least one line of the algorithm) per unit of time. Then, a process will eventually crash, drop the message (Line b15), or (re-)broadcast it (Lines b13 or b16). Finally, note that a process never re-broadcasts twice the same message and never re-broadcasts its own messages (Line b15). Hence a message can be (re-)broadcast at most n times, and will eventually disappear from the system. □

For the rest of the proof we will assume that any time instant t is larger than a time $t_1 > t_0$, where:

(1) t_0 is a time instant that occurs after the stabilization time T (i.e., $t_0 > T$), and after every eventually down process has definitely crashed, every eventually up process has definitely recovered, and every unstable process u has an incarnation number such that $incarnation_u > \Delta + 4\sigma$. Note that by definition u will crash and recover an infinite number of times, and hence eventually $incarnation_u > \Delta + 4\sigma$,
(2) and t_1 is a time instant such that all messages broadcast for the first time before t_0 have disappeared from the system (this eventually happens from Lemma 5).

Lemma 6. *$\forall s \in R$, $counter_s[s]$ is bounded.*

Proof. Consider any correct process $q \neq s$. Process s sends a message (*ALIVE*, s, $counter_s$) every η time. Eventually, every (*ALIVE*, s, $counter_s$) message that s sends is received directly or indirectly by q within $\Delta + \eta$ time from the time q received the previous message from s. Since q increases $Timeout_q[s]$ every time $timer_q(s)$ expires, eventually $timer_q(s)$ will not expire any more. After this, q will not punish s (Line b21) again, and s will not increase $counter_s[s]$ due to a message from any $q \in correct$.

On the other hand, every unstable process u will eventually and permanently set $timer_u(s) > \Delta + \eta + 4\sigma$ during initialization. Every time u resets $timer_u(s)$, we know that $timer_u(s)$ will expire after time $\Delta + \eta + 4\sigma$ time. As messages from s are sent every η time, in the worst case process s will send a message at time $t + \eta$, and the message will be received at process u at time $t + \Delta + \eta$, and $timer_u(s)$ will be reset at $t + \Delta + \eta + 4\sigma$. Hence, $timer_u(s)$ will never expire on any $s \in R$. After this, u will not punish s (Line b21) again, and s will not increase $counter_s[s]$ due to a message from any $u \in unstable$. □

The following observation derives from Lemma 6:

Observation 1. $R \subseteq B$.

Lemma 7. *For every process $p \in B$, every process $s \in R$ receives messages from p infinitely often.*

Proof. The proof is by contradiction. Assume that s does not receive messages from p infinitely often. Each time $timer_s[p]$ expires, process p is punished by s (Line b21). Eventually, a new *ALIVE* message sent by s will be received by p and p will increase $counter_p[p]$ (Line b17). Since this happens infinitely often, $counter_p[p]$ is not bounded, which is a contradiction with the fact that $p \in B$. □

The following observation derives from Lemma 7:

Observation 2. *There is a constant Δ' and a time $t_2 > t_1$ after which every message sent by $p \in B$ takes at most Δ' time to be received by every correct and unstable (if active) process.*

For the rest of the proof we will assume that any time instant t is larger than time $t_2 > t_1$, where t_2 is a time instant that occurs after $counter_p[q] > counter_p[p]$, $\forall q \notin correct$ and $\forall p \in B$, and $incarnation_u > counter_p[p]$, $\forall u \in unstable$. This will eventually happen because $counter_p[q]$ and $incarnation_u$ grow infinitely, and by definition $counter_p[p]$ is bounded. Note that during initialization (Line b8) $counter_u[u]$ is set to $incarnation_u$, so $counter_u[u] > counter_p[p]$.

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

Lemma 8. *For every pair of correct processes p and q, $p \in B$, there is a time after which for every time t, $counter_q[p] \geqslant counter_{p_t}[p]$.*

Proof. For $p = q$, the lemma is trivial. Now assume $p \neq q$. As $p \in B$, by Lemma 7 every process $s \in R$ receives messages from p infinitely often, and hence by rebroadcast q will receive messages of type (*ALIVE*, p, $counter_p$) infinitely often. Let $t > t_2$ be any time. There is a time $t' > t$ when q receives (*ALIVE*, p, $counter_p$) with $counter_p[p] = c$, originally sent by p after time t, so $c \geqslant counter_{p_t}[p]$. Then at time t', q sets its $counter_q[p]$ to c, and so we have: $counter_q[p] \geqslant counter_{p_t}[p]$. The lemma now follows since $counter_q[p]$ is monotonically nondecreasing. □

Lemma 9. *For every correct process p:*

1. *If $counter_p[p]$ is bounded, then there exists a value V_p and a time after which for every correct process q, $counter_q[p] = V_p$.*
2. *If $counter_p[p]$ is not bounded, then for every correct process q, $counter_q[p]$ is not bounded.*

Proof. Let p be a correct process.

(1) Suppose $counter_p[p]$ is bounded. Thus, by Lemma 8, for every correct process q, there is a time $t > t_2$ after which $counter_q[p] \geqslant counter_{p_t}[p]$. Since $counter_p[p]$ is bounded and monotonically nondecreasing, there exists a value V_p and a time after which $counter_p[p] = V_p$. Therefore, there exists a time after which, for every correct process q, $counter_q[p] = V_p$.
(2) Suppose $counter_p[p]$ is not bounded. Lemma 8 implies that $counter_q[p]$ is also not bounded. □

Lemma 10. *For every correct process p:*

1. *If $counter_p[p]$ is bounded, then there is a time after which for every unstable process u, $counter_u[p] = V_p$ in at most $\Delta' + \eta + 3\sigma$ time after its initialization.*
2. *If $counter_p[p]$ is not bounded, then for every unstable process u, $counter_u[p]$ is not bounded.*

Proof. Let p be a correct process.

(1) Suppose $counter_p[p]$ is bounded. Thus, by Lemma 9 there is a time after which $counter_p[p] = V_p$. From Observation 2, every unstable process u will receive (if active) an alive message from every process $p \in B$ in at most $\Delta' + \eta$ time. Hence, at most $\Delta' + \eta + 3\sigma$ time after initialization, $counter_u[p] = V_p$.
(2) Suppose $counter_p[p]$ is not bounded. By definition every unstable process u will receive (if active) an alive message infinitely often from every process $q \in B$, and will update $counter_u[p]$ (Line b17). By Lemma 9, if $counter_p[p]$ is not bounded, then $counter_q[p]$ is not bounded. Hence, $counter_u[p]$ is also unbounded. □

The following observation derives from Lemmas 9 and 10:

Observation 3. *There is a time $t' > t_2$ after which every message sent by every process q will contain $counter_q[p] = V_p$, $\forall p \in B$.*

For the rest of the proof we will assume that any time instant t is larger than t' of Observation 3.

Lemma 11. *If process k is not correct then for every process q there is a time after which k will not be $leader_q$.*

Proof. As process k is not correct, there is a time $t > t_2$ after which $counter_k[k] > counter_p[p]$, and $counter_p[k] > counter_p[p]$, for every $p \in B$. If q is correct, since eventually every message broadcast by every process p reaches timely every correct process q, $counter_q[k] \geqslant counter_p[k]$, and process k will not be elected as leader anymore. If q is unstable, by definition q will execute Line b11 infinitely often. By Lemma 10 there is a time after which $counter_q[p] = V_p$ and $counter_q[k] > counter_q[p]$ in at most $\Delta' + \eta + 3\sigma$ time after initialization. Hence, eventually, $leader_q \neq k$ will be permanently saved in stable storage, and process k will not be elected as leader anymore. □

Lemma 12. *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 9(1), for every process $p \in B$, there is a corresponding integer V_p and a time after which for every correct process q, $counter_q[p] = V_p$ (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 $(counter_q[p], p) > (counter_q[l], l)$. This implies that eventually q selects l as its leader, forever. To show (*) holds, consider the following 3 possible cases. If p is not correct then, by Lemma 11, eventually p will never be elected as leader (forever). Now suppose that p is correct. If $counter_p[p]$ is bounded, then $p \in B$; so, by our selection of l in B, eventually $(counter_q[p] = V_p, p) > (counter_q[l] = V_l, l)$ forever. Finally, if $counter_p[p]$ is not bounded, then, by Lemma 9(2), there is a time after which $counter_q[p] > counter_q[l] = V_l$ (because $counter_q[p]$ is unbounded and monotonically nondecreasing). In all cases (*) holds. □

Lemma 13. *There exists a correct process l and a time after which, for every unstable process u, $leader_u = l$.*

Proof. By Lemma 10(1), for every process $p \in B$, there is a corresponding integer V_p and a time after which for every unstable process u, $counter_u[p] = V_p$ $\Delta' + \eta + 3\sigma$ time after initialization. By definition, u executes Line b11 infinitely often, saving $leader_u$ in stable storage. Let l denote the process $p \in B$ with the smallest corresponding tuple (V_p, p). We now show that eventually every unstable process u selects l as its leader (forever). For any other process $p \neq l$: (*) there is a time after which $(counter_u[p], p) > (counter_u[l], l)$. This implies that eventually u selects l as its leader, writes $leader_u = l$ in stable storage (forever), and reads $leader_u = l$ from stable storage during initialization. To show (*) holds, consider the following 3 possible cases. If p is not correct then, by Lemma 11, eventually p will never be elected as leader (forever). Now suppose that p is correct. If $counter_p[p]$ is bounded, then $p \in B$; so, eventually every (*ALIVE*, z, $counter_z$) message that u receives after initialization will contain always $(counter_z[p] = V_p, p) > (counter_z[l] = V_l, l)$ forever. Since during initialization every counter is set to 0 except $counter_u[u]$ that is unbounded, $counter_u[p]$ will be set to $counter_z[p]$ and $counter_u[l]$ to $counter_z[l]$ respectively (Line b17). By our selection of l in B, l will be chosen as leader and written in stable store at Line b11. Finally, if $counter_p[p]$ is not bounded, then, by Lemma 10(2), there is a time after which $counter_u[p] > counter_u[l] = V_l$ (because $counter_u[p]$ is unbounded and monotonically nondecreasing). In all cases (*) holds. □

Theorem 2. *There is a time after which every active process p permanently has $leader_p = l$, i.e., p trusts l. Hence, the algorithm of Fig. 2 satisfies Property 2, and implements Omega in system S.*

Proof. It follows directly from Lemmas 12 and 13, and the common definition of process l made in both lemmas. □

4.2. A third algorithm

In this section we present a third algorithm, adapted from [12], that implements Omega in system S. Contrary to the algorithm of Fig. 2, this algorithm does not require the membership of the system to be known a priori by processes. The process identifiers are totally ordered, but need not be consecutive. Furthermore, processes have no knowledge about the total number of processes n. Fig. 3 presents the algorithm in detail. As in the previous two algorithms, we assume that every unstable process will be able to write in stable storage infinitely often.

The algorithm works as follows. Processes send messages periodically to show they are alive. These messages are re-broadcast to attempt reaching all processes. Each process p maintains a set $membership_p$ of pairs (q, v) (initially $(p, incarnation_p)$), where q is a process that p knows, and $v \geqslant 0$ is roughly the number of times that q has been "punished." Every message sent by p contains this set $membership_p$.

When a process p receives a message from $q \neq p$ for the first time, after re-broadcasting it, for every pair $(r, -) \in membership_q$, p checks if $(r, -) \notin membership_p$, in which case p includes (r, v) in $membership_p$ (being v the value associated

Every process p executes the following:

```
procedure updateLeader()
( c1) leader_p ← process in min{membership_p}
end procedure

Initialization:
( c2) increment INCARNATION_p by 1 in stable storage
( c3) incarnation_p ← read INCARNATION_p from stable storage
( c4) leader_p ← read LEADER_p from stable storage
( c5) membership_p ← {(p, incarnation_p)}
( c6) start tasks 1, 2 and 3

Task 1:
( c7) wait (η + incarnation_p)
( c8) write leader_p in stable storage
( c9) loop forever
(c10)   broadcast (ALIVE, p, membership_p)
(c11)   wait (η)

Task 2:
(c12) upon reception of message (ALIVE, q, membership_q) with q ≠ p for the first time do
(c13)   broadcast (ALIVE, q, membership_q)
(c14)   ∀(r, −) ∈ membership_q:
(c15)     if (r, −) ∉ membership_p then
(c16)       membership_p ← membership_p ∪ {(r, v)} : (r, v) ∈ membership_q
(c17)       create timer_p(r) and Timeout_p[r]
(c18)       Timeout_p[r] ← η + incarnation_p
(c19)       reset timer_p(r) to Timeout_p[r]
(c20)     else
(c21)       replace in membership_p (r, v) by (r, max{v, v'}): (r, v') ∈ membership_q
(c22)     end if
(c23)   reset timer_p(q) to Timeout_p[q]
(c24)   if (p, −) ∉ membership_q then
(c25)     replace in membership_p (p, v) by (p, v + 1)
(c26)   end if
(c27)   updateLeader()

Task 3:
(c28) upon expiration of timer_p(q) do
(c29)   replace in membership_p (q, v) by (q, v + 1)
(c30)   Timeout_p[q] ← Timeout_p[q] + 1
(c31)   reset timer_p(q) to Timeout_p[q]
(c32)   updateLeader()
```

Fig. 3. Third algorithm implementing Omega in system S.

to r in $membership_q$), creates $timer_p(r)$ and $Timeout_p[r]$, initializes $Timeout_p[r]$ to $\eta + incarnation_p$, and resets $timer_p(r)$. Otherwise, if $(r, -) \in membership_p$, then p updates the value associated to r in $membership_p$. After that, p resets $timer_p(q)$ to $Timeout_p[q]$. Then, if $(p, -) \notin membership_q$, then p punishes itself by incrementing its associated counter in $membership_p$. Finally, the *updateLeader*() procedure is called to change $leader_p$ if required. A process p will hold in $leader_p$ its current leader, which is the process q whose pair (q, v) in $membership_p$ has the smallest value v, using the process identifier to break ties.

If $timer_p(q)$ expires before receiving a new (*ALIVE*, q, $membership_q$) message, then p increments the value associated to q in $membership_p$, increments the value $Timeout_p[q]$, resets $timer_p(q)$ to $Timeout_p[q]$, and calls *updateLeader*().

To avoid unstable processes from disturbing the leader election, during initialization every process p initializes $membership_p$ with the pair $(p, incarnation_p)$ (Line c5). Also, in Task 1 p waits $\eta + incarnation_p$ units of time (Line c7) before sending messages (that include $membership_p$) periodically. This waiting ensures that eventually every unstable process p will only send messages with $membership_p$ containing a pair (l, v) such that l is a correct process and v is smaller than the value associated to any other (correct or unstable) process in the system.

As in the algorithm of Fig. 2, the number of messages sent periodically (every η time) in this algorithm is also bounded by $n * ul$, being n and ul the number of processes and unidirectional links in the system respectively.

In the algorithm for the crash model of [12], an additional set $candidates_p$, containing the processes considered alive, is maintained by every process p, and *ALIVE* messages include the set $candidates_p$. Upon a suspicion on a process q, p removes q from $candidates_p$ and broadcasts an explicit *ALIVE* message to notify the suspicion. Again, our algorithm for the crash-recovery model avoids the explicit broadcast of messages to notify suspicions, reducing the message complexity of the algorithm.

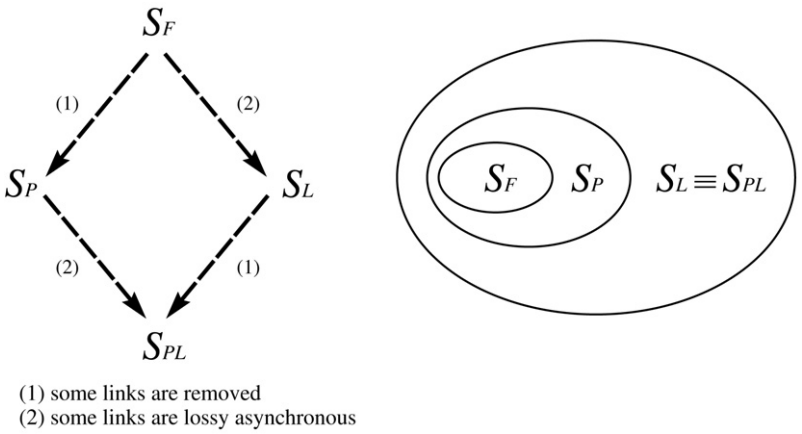

Fig. 4. Relation between systems S_F, S_P, S_L and S_{PL}.

Regarding the correctness proof of this algorithm, it is close to that of algorithm in Fig. 2. The main difference is the unknown membership, which is addressed with a nondecreasing membership ($membership_p$), dynamically created timers, and a mechanism by which a process punishes itself (Lines c24–c26). This mechanism is needed because a process may never be known by the rest of processes.

Theorem 3. *There is a time after which every active process p permanently trust the same correct process. Hence, the algorithm of Fig.* 3 *satisfies Property* 2*, and implements Omega in system S.*

5. A note about connectivity and synchrony

From the point of view of the communication, in order to get to the system S considered in this work, we can start from a system *fully* (*eventually*) *timely connected* S_F, i.e., a system in which every process has a direct communication link with every other process, all the links being eventually timely. The system S_F can be weakened by (1) removing some links, leading to systems *partially* (*eventually*) *timely connected* S_P, or (2) considering some links as lossy asynchronous, leading to systems *fully connected with* (*some*) *lossy links* S_L. Clearly, $S_F \subset S_P$ and $S_F \subset S_L$. Also, $S_P \subset S_L$, since links that have been removed in S_P can be seen as lossy asynchronous links that systematically drop all the messages in S_L. If we apply both (1) and (2) to S_F, we get systems *partially connected with* (*some*) *lossy links* S_{PL}. Note that $S_{PL} \equiv S_L$, since links that have been removed in S_{PL} can be seen as lossy asynchronous links that systematically drop all the messages in S_L, while S_L can be seen as a subset of S_{PL} in which no links are removed. In other words, any Omega algorithm that works correctly in S_{PL} must also work correctly in S_L, and vice versa. Also, $S_P \subset S_{PL}$, since a subset of the links that have been removed in S_P can be seen as lossy asynchronous links that systematically drop all the messages in S_{PL}. To summarize, we have $S_F \subset S_P \subset S_L \equiv S_{PL}$ (see Fig. 4).

The system S considered in this work assumes partial connectivity as in S_P and some lossy asynchronous links as in S_L, so $S \subset S_{PL}$. Nevertheless, S requires timely connectivity from some correct process to the rest of processes. More precisely, S requires that there is a path formed exclusively by eventually timely links between some correct process (which must be c_{min} in the case of the first algorithm) and the rest of correct and unstable processes.

6. Conclusion

In this paper, we have presented three algorithms that implement the Omega failure detector in the crash-recovery failure model. The algorithms work in systems in which not every pair of processes is connected by a direct communication link and also some links can be lossy asynchronous. The first algorithm assumes that eventually all processes are reachable timely from the correct process that crashes and recovers a minimum number of times. This algorithm does not require the membership to be known a priori. The second algorithm assumes that eventually all processes are reachable timely from some correct process. This algorithm requires the membership to be known a priori. Finally, the third algorithm also assumes that eventually all processes are reachable timely from some correct process, but does not require the membership to be known a priori.

Acknowledgments

We are grateful to Antonio Fernández for his helpful comments.

References

[1] T. Chandra, S. Toueg, Unreliable failure detectors for reliable distributed systems, J. ACM 43 (2) (1996) 225–267.
[2] M. Pease, R. Shostak, L. Lamport, Reaching agreement in the presence of faults, J. ACM 27 (2) (1980) 228–234.
[3] T. Chandra, V. Hadzilacos, S. Toueg, The weakest failure detector for solving consensus, J. ACM 43 (4) (1996) 685–722.
[4] A. Mostéfaoui, M. Raynal, Leader-based consensus, Parallel Process. Lett. 11 (1) (2001) 95–107.
[5] M. Larrea, A. Fernández, S. Arévalo, On the implementation of unreliable failure detectors in partially synchronous systems, IEEE Trans. Comput. 53 (7) (2004) 815–828.
[6] M. Larrea, A. Lafuente, Brief Announcement: Communication-efficient implementation of failure detector classes $\diamond\mathcal{Q}$ and $\diamond\mathcal{P}$, in: Proceedings of the 19th International Symposium on Distributed Computing, DISC 2005, in: Lecture Notes in Comput. Sci., vol. 3724, Springer-Verlag, Krakow, Poland, 2005, pp. 495–496.
[7] M. Larrea, A. Fernández, 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, Nurenberg, Germany, 2000, pp. 52–59.
[8] M. Aguilera, C. Delporte-Gallet, H. Fauconnier, S. Toueg, Stable leader election, in: Proceedings of the 15th International Symposium on Distributed Computing, DISC 2001, in: Lecture Notes in Comput. Sci., vol. 2180, Springer-Verlag, Lisbon, Portugal, 2001, pp. 108–122.
[9] M. Aguilera, C. Delporte-Gallet, H. Fauconnier, 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, Boston, MA, 2003, pp. 306–314.
[10] M. Aguilera, C. Delporte-Gallet, H. Fauconnier, S. Toueg, Communication-efficient leader election and consensus with limited link synchrony, in: Proceedings of the 23rd ACM Symposium on Principles of Distributed Computing, PODC 2004, St. John's, Newfoundland, Canada, 2004, pp. 328–337.
[11] E. Jiménez, S. Arévalo, A. Fernández, Implementing the Ω failure detector with unknown membership and weak synchrony, Technical Report RoSaC-2005-2, Universidad Rey Juan Carlos, Spain, January 2005.
[12] E. Jiménez, S. Arévalo, A. Fernández, Implementing unreliable failure detectors with unknown membership, Inform. Process. Lett. 100 (2) (2006) 60–63.
[13] D. Dolev, R. Friedman, I. Keidar, D. Malkhi, Failure detectors in omission failure environments, in: Proceedings of the 16th ACM Symposium on Principles of Distributed Computing, PODC 97, Santa Barbara, CA, USA, 1997, p. 286.
[14] M. Aguilera, W. Chen, S. Toueg, Failure detection and consensus in the crash-recovery model, Distrib. Comput. 13 (2) (2000) 99–125.
[15] M. Hurfin, A. Mostéfaoui, M. Raynal, Consensus in asynchronous systems where processes can crash and recover, in: Symposium on Reliable Distributed Systems, SRDS 98, West Lafayette, IN, USA, 1998, pp. 280–286.
[16] R. Oliveira, R. Guerraoui, A. Schiper, Consensus in the crash-recover model, Technical Report TR-97/239, Swiss Federal Institute of Technology, Lausanne, 1997.
[17] C. Dwork, N. Lynch, L. Stockmeyer, Consensus in the presence of partial synchrony, J. ACM 35 (2) (1988) 288–323.
[18] A. Fernández, E. Jiménez, S. Arévalo, Minimal system conditions to implement unreliable failure detectors, in: Proceedings of the 12th IEEE Pacific Rim International Symposium on Dependable Computing, PRDC 2006, University of California, Riverside, USA, 2006, pp. 63–72.