The Journal of Systems and Software 84 (2011) 2186–2195

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Communication-efficient leader election in crash–recovery systems

Mikel Larrea*, Cristian Martín[1], Iratxe Soraluze

University of the Basque Country, UPV/EHU, 20018 San Sebastián, Spain

ARTICLE INFO

Article history:
Received 11 November 2010
Received in revised form 6 June 2011
Accepted 6 June 2011
Available online 17 June 2011

Keywords:
Fault-tolerant distributed computing
Consensus
Omega failure detector
Leader election
Crash–recovery
Communication-efficient algorithm

ABSTRACT

This work addresses the leader election problem in partially synchronous distributed systems where processes can crash and recover. More precisely, it focuses on implementing the Omega failure detector class, which provides a leader election functionality, in the crash–recovery failure model. The concepts of communication efficiency and near-efficiency for an algorithm implementing Omega are defined. Depending on the use or not of stable storage, the property satisfied by unstable processes, i.e., those that crash and recover infinitely often, varies. Two algorithms implementing Omega are presented. In the first algorithm, which is communication-efficient and uses stable storage, eventually and permanently unstable processes agree on the leader with correct processes. In the second algorithm, which is near-communication-efficient and does not use stable storage, processes start their execution with no leader in order to avoid the disagreement among unstable processes, that will agree on the leader with correct processes after receiving a first message from the leader.

1. Introduction

Unreliable failure detectors, proposed by Chandra and Toueg (1996), provide (possibly incorrect) information about process failures, allowing to solve fault-tolerant agreement problems in asynchronous distributed systems, e.g., the consensus problem (Pease et al., 1980) (a fundamental result in fault-tolerant distributed computing is that consensus cannot be solved deterministically in asynchronous systems prone to even a single process crash (Fischer et al., 1985)). In this work, we address the implementation of a failure detector class called Omega (Chandra et al., 1996) in the crash–recovery failure model. Informally, Omega provides an eventual leader election functionality, i.e., eventually all processes agree on a common and correct process. Several consensus algorithms based on such a weak leader election mechanism have been proposed (Guerraoui and Raynal, 2004; Lamport, 1998; Larrea et al., 2005; Mostéfaoui and Raynal, 2001).

Many algorithms implementing Omega in the crash failure model, i.e., in which crashed processes do not recover, have been proposed (Aguilera et al., 2004; Chu, 1998; Fernández et al., 2006a,b; Fernández and Raynal, 2007; Malkhi et al., 2005; Mostéfaoui et al., 2003, 2004, 2006a,b, 2007). Larrea et al. (2000) proposed an algorithm requiring all links to be eventually timely (i.e., there is an unknown bound δ and an unknown time T, such that if a message is sent at a time $t \geq T$, then this message is received by time $t+\delta$). Aguilera et al. (2001) proposed an algorithm requiring all links of some unknown correct process to be eventually timely. Aguilera et al. (2003, 2008) also proposed several algorithms requiring only the outgoing links from some unknown correct process to be eventually timely. Jiménez et al. (2006) have proposed an algorithm with unknown membership which requires that eventually all correct processes are reachable timely from some correct process.

Failure detection has also been studied in the crash–recovery failure model, i.e., in which crashed processes can recover (even infinitely often). However, few specific algorithms implementing Omega in this failure model have been proposed. Aguilera et al. (2000) define an adaptation of the $\Diamond\mathcal{S}$ failure detector class to the crash–recovery failure model, proposing an algorithm implementing it in partially synchronous systems (Chandra and Toueg, 1996; Dwork et al., 1988). Martín et al. (2007, 2009) proposed several algorithms implementing Omega in the crash–recovery failure model that rely on the use of stable storage to keep the value of an incarnation number associated with each process. Recently, Martín and Larrea (2008) have proposed an algorithm for Omega which does not use stable storage but requires a majority of correct processes. In all these algorithms, every alive process sends messages to the rest of processes. Consequently, the cost of these algorithms in terms of the number of messages exchanged is high. It would be interesting to have algorithms for Omega in which eventually only

* Corresponding author. Tel.: +34 943015084; fax: +34 943015590.
E-mail addresses: mikel.larrea@ehu.es (M. Larrea), martin.cristian@gmail.com, cmartin@ikerlan.es (C. Martín), iratxe.soraluze@ehu.es (I. Soraluze).
[1] Current address: Ikerlan Research Center, 20500 Arrasate-Mondragón, Spain.

0164-1212/$ – see front matter © 2011 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2011.06.019

one process, i.e., the elected leader, sends a message periodically to the rest of processes. In this regard, Martín and Larrea (2010) have proposed a simple Omega algorithm that relies on a nondecreasing and persistent local clock associated with each process, in which eventually only the elected leader keeps sending messages to the rest of processes.

Apart from the seminal work of Aguilera et al. (2000), a few works dealing with failure detection and consensus in the crash–recovery model have been published (e.g., Hurfin et al., 1998; Oliveira et al., 1997, and more recently Freiling et al., 2009). The consensus algorithms in Hurfin et al. (1998) and Oliveira et al. (1997) use $\Diamond\mathcal{S}$-like failure detectors that require unstable processes to be eventually suspected forever, which is unrealistic since it involves predicting the future. Also, all the algorithms use stable storage. The work of Freiling et al. (2009) focuses on failure detector classes $\mathcal{P}$ and $\Diamond\mathcal{P}$, which are redefined for the crash–recovery model. Their approach to solve consensus consists in re-using existing algorithms for the crash model in a modular way. To do so, they emulate a crash system on top of a crash–recovery system to be able to run a crash consensus algorithm.

In this work, we first define the concepts of communication efficiency and near-efficiency when implementing the Omega failure detector class in crash–recovery systems. They are related to the fact that eventually either only one process, or only one among correct processes, sends messages forever, respectively. Then, we propose a communication-efficient Omega algorithm which uses stable storage, and a near-communication-efficient Omega algorithm which does not use stable storage but requires a majority of correct processes. In this regard, and following the line of Martín and Larrea (2008), we replace the use of stable storage by the need of 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 correct majority has been discussed by Wiesmann and Défago (2006) on the implementation of end-to-end communication primitives. Depending on the use or not of stable storage, the property that the algorithms satisfy regarding unstable processes, i.e., those that crash and recover infinitely often, varies. When stable storage is used, unstable processes can agree with correct processes by reading the identity of the leader from stable storage upon recovery. On the other hand, when stable storage is not used unstable processes must learn from some other process(es) the identity of the leader upon recovery. As in Martín and Larrea (2008), we make processes to be aware of being in this learning period by outputting a special "no-leader" value upon recovery. Furthermore, the near-communication-efficient Omega algorithm not using stable storage proposed in this paper does not rely on any persistent clock, which makes possible to implement Omega in a communication-efficient way as shown in Martín and Larrea (2010).

The rest of the paper is organized as follows. In Section 2, we describe the system model and the two specific systems S_1 and S_2 considered in this work, and give the definitions of communication efficiency and near-efficiency for the Omega failure detector class in crash–recovery systems. Sections 3 and 4 present a communication-efficient Omega algorithm for system S_1 using stable storage and a near-communication-efficient Omega algorithm for system S_2 not using stable storage, respectively. In Section 5, we discuss about the relaxation of the communication reliability and synchrony assumptions. Finally, Section 6 concludes the paper.

2. System model and communication efficiency definitions

We consider a system model 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. We also use p, q, r, etc. to denote processes. Every 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 execution of the system, Π is composed of the following three disjoint subsets (Martín et al., 2007):

- *Eventually up*, i.e., processes that eventually remain up forever. We naturally include in this subset processes that never crash, also called *always up*.
- *Eventually down*, i.e., processes that eventually remain crashed forever.
- *Unstable*, i.e., processes that crash and recover an infinite number of times.

By definition, eventually up processes are correct, while eventually down and unstable processes are incorrect. We assume that the number of correct processes in the system in any execution is at least one.

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

Communication links cannot create or alter messages, and are not assumed to be FIFO. Concerning timeliness or loss properties, we consider the following three types of links (Aguilera et al., 2003):

- *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 $t + \delta$. Note that if the message is sent before T, then it is eventually lost or received at its destination.
- *Lossy asynchronous links*, where there is no bound on message delay, and the link can lose an arbitrary number of messages (possibly all). Note however that every message that is not lost is eventually received at its destination.
- *(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).

2.1. Specific crash–recovery systems S_1 and S_2

We consider two specific systems, denoted S_1 and S_2, respectively. System S_1 assumes that processes have access to stable storage. Regarding communication reliability and synchrony, S_1 satisfies the following assumption:

(i) For every correct process p, there is an eventually timely link from p to every correct and every unstable process.

The rest of links of S_1, i.e., the links from/to eventually down processes and the links from unstable processes, can be lossy asynchronous. Fig. 1 presents a scenario of a system composed of five processes which satisfies the assumptions of S_1.

System S_2 assumes that processes do not have access to any form of stable storage. Alternatively, it is assumed that a majority of processes are correct. Regarding communication reliability and synchrony, S_2 satisfies the following assumptions:

(i) For every correct process p, there is an eventually timely link from p to every correct and every unstable process.

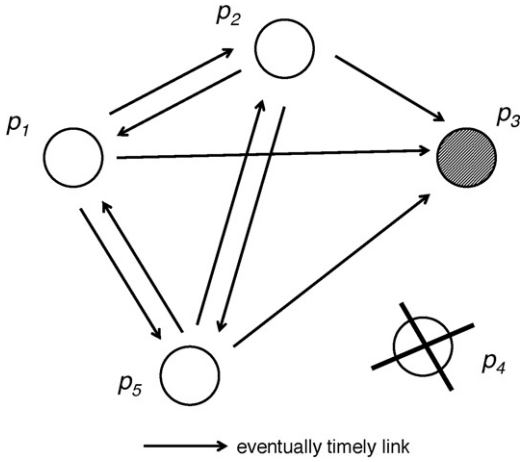

Fig. 1. Scenario of system S_1: three processes eventually up, one eventually down, one unstable.

(ii) For every unstable process u, there is a fair lossy link from u to every correct process.

The rest of links of S_2, i.e., the links from/to eventually down processes and the links between unstable processes, can be lossy asynchronous. Fig. 2 presents a scenario of a system satisfying the assumptions of S_2.

2.2. The Omega failure detector class

Chandra et al. (1996) defined a failure detector class 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 (it is said that p *trusts* q). The Omega failure detector class satisfies the following property: *there is a time after which every correct process always trusts the same correct process.* Since this definition was made for the crash failure model, it 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 at any time with correct processes, which can be a drawback, e.g., making an attempt to solve consensus fail due to the existence of several leaders (Lamport, 1998; Mostéfaoui and Raynal, 2001). 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. In this regard, the quality of the agreement of unstable processes with correct ones will depend on the use or not of stable storage. Intuitively, the use of stable storage allows unstable processes to agree from the beginning of their execution (by reading the identity of the leader from stable storage), while the absence of stable storage forces unstable processes to communicate with some correct process(es) in order to learn the identity of the leader. Hence, we consider the following two definitions for Omega in the crash–recovery failure model, for systems with and without stable storage, respectively (Martín and Larrea, 2008; Martín et al., 2007):

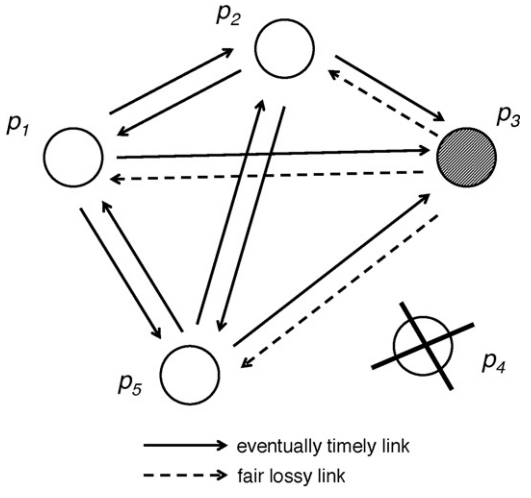

Fig. 2. Scenario of system S_2: three processes eventually up, one eventually down, one unstable.

Property 1 (Omega-crash–recovery, stable storage). *There is a time after which every process that is up, either correct or unstable, always trusts the same correct process.*

Regarding the behavior of unstable processes in order to satisfy this property, in our first Omega algorithm every process will read the identity of its leader from stable storage at the beginning of the execution. In order to eventually agree permanently with correct processes, an unstable process u must have written definitely the identity of the final leader in stable storage. Note that it will suffice to write it once, provided it is not changed later. We will assume that unstable processes remain alive long enough to write definitely the identity of the final leader in stable storage, and we have included in our algorithm a mechanism to ensure it with high probability. Said this, there could be some unstable processes that do not write definitely the identity of the final leader in stable storage. The Omega property defined above does not apply to those unstable processes.

Property 2 (Omega-crash–recovery, no stable storage). *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.*

As we will see, in our second Omega algorithm processes start setting their leader to $\perp$, and no assumption about how long an unstable process u should be alive when it recovers is made. Said this, u will only agree on the final leader ℓ with correct processes if, when up, it receives a message from ℓ. Observe that this definition of Omega for crash–recovery systems without stable storage is very useful for leader-based protocols, e.g., consensus, since it allows unstable processes to delay their participation in the protocol until they really trust a process, thus ensuring eventual agreement and making consensus solvable.

2.3. Communication efficiency definitions

We define now the concepts of communication-efficient and near-communication-efficient implementations of the Omega failure detector class in crash–recovery systems.

Definition 1. An algorithm implementing the Omega failure detector class in the crash–recovery failure model is communication-efficient if there is a time after which only one process sends messages forever.

Definition 2. An algorithm implementing the Omega failure detector class in the crash–recovery failure model is near-communication-efficient if there is a time after which, among correct processes, only one sends messages forever.

Intuitively, since the (correct) leader process in an Omega algorithm must send messages forever in order to keep being trusted by the rest of processes, we can derive that a communication-efficient Omega algorithm is also near-communication-efficient.

The small difference between both definitions is that in a near-communication-efficient Omega algorithm, besides the leader, unstable processes can send messages forever.

In the following two sections, we propose a communication-efficient Omega (Property 1) algorithm which uses stable storage for system S_1, and a near-communication-efficient Omega (Property 2) algorithm which does not use stable storage for system S_2, respectively.

3. A communication-efficient Omega algorithm for system S_1

In this section, we present a communication-efficient algorithm implementing Omega (Property 1) in system S_1 using stable storage. Fig. 3 presents the algorithm in detail. The process chosen as leader by a process p, i.e., trusted by p, is held in a variable $leader_p$. Every process p uses stable storage to keep the value of two local variables: $leader_p$, initially set to p, and an incarnation number $incarnation_p$, initially set to 0, which is incremented during initialization and every time p recovers from a crash. Both $incarnation_p$ and $leader_p$ are read from stable storage (from the $INCARNATION_p$ and $LEADER_p$ stable storage variables, respectively) by p during initialization. Also, p has a time-out $Timeout_p[q]$ with respect to every other process q (initialized to $\eta + incarnation_p$, being η a constant value), and a $Recovered_p$ vector to count the number of times that each process has recovered (initialized to 0 for every other process, and to $incarnation_p$ for p itself).

The algorithm works as follows. After the initializations, if process p does not trust itself, then it resets a timer with respect to $leader_p$. After that, p starts the three tasks of the algorithm. In Task 1, p first waits $\eta + incarnation_p$ time units, after which it writes $leader_p$ in stable storage. Then, every η time units p checks if it trusts itself, in which case p sends a *LEADER* message containing $Recovered_p$ to the rest of processes. Task 2 is activated whenever p receives a *LEADER* message from another process q (note that this task is active during p's waiting of Task 1): p updates $Recovered_p$ with $Recovered_q$, taking the highest value for each component of the vector. After that, p checks if q is a better candidate than $leader_p$ to become p's leader, which is the case if either (1) $Recovered_p[q] < Recovered_p[leader_p]$, or (2) $Recovered_p[q] = Recovered_p[leader_p]$ and $q \leq leader_p$.[1] In that case, p sets q as its leader and resets $timer_p$ to $Timeout_p[q]$ in order to monitor q (i.e., $leader_p$) again. Finally, p also checks if it deserves to be leader comparing $Recovered_p[p]$ with $Recovered_p[leader_p]$. If it is the case $leader_p$ is set to p and $timer_p$ is stopped. This way, $leader_p$ will be the process with the smallest associated recovery value in $Recovered_p$ among $leader_p$, q and p. In Task 3, which is activated whenever $timer_p$ expires, p increments $Timeout_p[leader_p]$ in order to avoid new premature suspicions on $leader_p$, and resets $leader_p$ to p.

As we will show, with this algorithm eventually every correct process always trusts the same correct process ℓ. Consequently, by Task 1 eventually only one correct process sends messages forever, i.e., the algorithm is at least near-communication-efficient. Concerning the behavior of unstable processes, the wait instruction followed by the write of $leader_p$ in stable storage at the beginning of Task 1 ensure with high probability that eventually p writes definitely ℓ in stable storage.[2] Actually, in practice it is sufficient that every unstable process p writes at least once $leader_p = \ell$ in stable storage, provided that all subsequent writes (if any) correspond to ℓ too. Hence, the required number of writes in stable storage, although unknown, is finite.

From this point, whenever p recovers, it will initialize $leader_p$ to ℓ from stable storage. Moreover, the initializations of $Timeout_p[\ell]$ to $\eta + incarnation_p$, and of $Recovered_p[p]$ to $incarnation_p$ prevent unstable processes from disturbing the leader election, because they ensure that eventually (1) every unstable process p does never suspect the leader ℓ (since p's time-out with respect to ℓ keeps increasing forever, and hence eventually $timer_p$ never expires), and (2) every unstable process p will never be elected as the leader in Task 2 (since $incarnation_p$, and hence $Recovered_p[p]$, keeps increasing forever). Hence, the algorithm is communication-efficient.

3.1. Correctness proof

We show now that the algorithm of Fig. 3 implements Omega (Property 1) in system S_1, and that it is communication-efficient.

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

Proof. A message m cannot remain forever in a link, since it remains at most δ time in an eventually timely link if sent after T (otherwise, i.e., if m is sent before T, then it is eventually lost or received), and is eventually lost or received in a lossy asynchronous link. Also, m cannot remain forever in the destination process, since processes are assumed to be synchronous. Hence, m eventually disappears from the system. □

For the rest of the proof we will assume that any time instant t is larger than $t_1 > t_0$, 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 correct (i.e., eventually up) process has definitely recovered, and every unstable process has an incarnation value bigger than any correct process, i.e., $\forall u \in unstable$, $\forall p \in correct$: $incarnation_u > incarnation_p$,
(2) and t_1 is a time instant such that all messages sent before t_0 have disappeared from the system (this eventually happens by Lemma 1). In particular, this includes (a) all messages sent by eventually down processes, (b) all messages sent by correct processes before recovering definitely, and (c) all messages sent by every unstable process u with $Recovered_u[u] = incarnation_u \leq incarnation_p$, for every correct process p. This eventually happens, since by definition unstable processes crash and recover an infinite number of times, while correct processes crash and recover a finite number of times.

Let be ℓ the correct process with the smallest value for its $incarnation_\ell$ variable, i.e., the correct process that crashes and recovers fewer times. If two or more correct processes have the same final value for their *incarnation* variables, then let ℓ be the process with smallest identifier among them. We will show that eventually and permanently, for every correct and every unstable process p, $leader_p = \ell$.

Lemma 2. *Eventually and permanently*, $leader_\ell = \ell$.

Proof. After time t_1, the only way for process ℓ to maintain as leader another process q is by receiving a message (*LEADER*, q, $Recovered_q$) such that $Recovered_q[q] < Recovered_\ell[\ell]$. However, it is simple to see that such a scenario cannot happen, since any (*LEADER*, q, $Recovered_q$) message that ℓ can receive necessarily has either (1) $Recovered_q[q] = incarnation_q > incarnation_\ell = Recovered_\ell[\ell]$, or (2) $Recovered_q[q] = incarnation_q = incarnation_\ell = Recovered_\ell[\ell]$ and $q > \ell$. Hence, if at a given time $leader_\ell = q \neq \ell$,

[1] We use $\langle Recovered_p[q], q\rangle \leq \langle Recovered_p[leader_p], leader_p\rangle$ to denote it. Observe that the case where $q = leader_p$ satisfies the relation.

[2] A way to cope with processes not satisfying this assumption could consist in returning, together with the identity of the current leader, a Boolean flag indicating if the process has already completed the waiting instruction of Task 1. Clearly, this flag will be eventually and permanently true at correct processes, while it will be eventually and permanently false at unstable processes.

Initialization:
  if $INCARNATION_p$ and $LEADER_p$ do not exist in stable storage **then**
    $incarnation_p \leftarrow 0$
    $leader_p \leftarrow p$
    write $incarnation_p$ into $INCARNATION_p$ in stable storage
    write $leader_p$ into $LEADER_p$ in stable storage
  end if
  $incarnation_p \leftarrow$ read $INCARNATION_p$ from stable storage
  $incarnation_p \leftarrow incarnation_p + 1$
  write $incarnation_p$ into $INCARNATION_p$ in stable storage
  $leader_p \leftarrow$ read $LEADER_p$ from stable storage
  for all $q \in \Pi$ except p:
    $Timeout_p[q] \leftarrow \eta + incarnation_p$
    $Recovered_p[q] \leftarrow 0$
  $Recovered_p[p] \leftarrow incarnation_p$
  if $leader_p \neq p$ **then**
    reset $timer_p$ to $Timeout_p[leader_p]$
  end if
  start tasks 1, 2 and 3

Task 1:
  wait $(\eta + incarnation_p)$ time units
  write $leader_p$ into $LEADER_p$ in stable storage
  repeat forever every η **time units**
    if $leader_p = p$ **then**
      send $(LEADER, p, Recovered_p)$ to all processes except p
    end if

Task 2:
  upon reception of message $(LEADER, q, Recovered_q)$ **do**
    for all $r \in \Pi$:
      $Recovered_p[r] \leftarrow \max(Recovered_p[r], Recovered_q[r])$
    if $\langle Recovered_p[q], q\rangle \leq \langle Recovered_p[leader_p], leader_p\rangle$ **then**
      $leader_p \leftarrow q$
      reset $timer_p$ to $Timeout_p[q]$
    end if
    if $\langle Recovered_p[p], p\rangle < \langle Recovered_p[leader_p], leader_p\rangle$ **then**
      $leader_p \leftarrow p$
      stop $timer_p$
    end if

Task 3:
  upon expiration of $timer_p$ **do**
    $Timeout_p[leader_p] \leftarrow Timeout_p[leader_p] + 1$
    $leader_p \leftarrow p$

Fig. 3. Communication-efficient Omega algorithm in system S_1.

then either ℓ will receive a (*LEADER*, *q*, $Recovered_q$) message from *q* or $timer_\ell$ will eventually expire. If ℓ receives a (*LEADER*, *q*, $Recovered_q$) message from *q*, then by Task 2 of the algorithm ℓ becomes the leader and stops $timer_\ell$. Otherwise, if $timer_\ell$ expires, then by Task 3 of the algorithm ℓ becomes the leader too. After that, ℓ will not change its leader any more. As a result, eventually and permanently process ℓ considers itself the leader, i.e., $leader_\ell = \ell$. □

Lemma 3. *Eventually and permanently, process ℓ periodically sends a (LEADER, ℓ, $Recovered_\ell$) message to the rest of processes.*

Proof. Follows directly from Lemma 2 and Task 1 of the algorithm. □

Lemma 4. *Eventually and permanently, for every correct process p, $leader_p = \ell$.*

Proof. Follows from Lemma 2 for process ℓ. Let be any other correct process *p*. By Lemma 3, ℓ will periodically send a (*LEADER*, ℓ, $Recovered_\ell$) message to the rest of processes, including *p*. By the fact that the communication link between ℓ and *p* is eventually timely, by Task 2 *p* will receive the message in at most δ time units. Since by definition, for every correct process *p*, $Recovered_\ell[\ell] \leq Recovered_p[p]$, ℓ is a better candidate to be *p*'s leader than both *p* itself and $leader_p$ (in case $leader_p \neq \ell$). Hence, *p* will set $leader_p$ to ℓ, and will reset $timer_p$ to $Timeout_p[\ell]$. Observe that $timer_p$ can expire a finite number of times on ℓ, since by Task 3 every time it expires *p* increments $Timeout_p[\ell]$. Hence, eventually by Task 2 *p* receives a (*LEADER*, ℓ, $Recovered_\ell$) message from ℓ periodically and timely, i.e., before $timer_p$ expires. After this happens, *p* will not change $leader_p$ to a value different from ℓ any more. Observe also that the leader of process *p* can be an eventually down process *q* whose incarnation number is smaller than the incarnation number of every correct process (including ℓ). However, eventually *q* will definitely crash, $timer_p$ will expire, and by Task 3 $leader_p$ will be set to *p* so that when *p* receives a (*LEADER*, ℓ, $Recovered_\ell$) message from ℓ it will adopt ℓ as its leader. □

Lemma 5. *Eventually and permanently, every correct process $p \neq \ell$ does not send any more messages.*

Proof. Follows directly from Lemma 4 and the algorithm. □

Lemma 6. *Eventually, every unstable process u does not send any more messages, and $leader_u$ is ℓ forever.*

Proof. By Lemma 3, ℓ will periodically send a (*LEADER*, ℓ, $Recovered_\ell$) message to the rest of processes, including u. By the facts that (1) the communication link between ℓ and u is eventually timely, and (2) u waits $\eta + incarnation_u$ time units at the beginning of Task 1, eventually by Task 2 u always receives a first (*LEADER*, ℓ, $Recovered_\ell$) message from ℓ before the end of the waiting instruction of Task 1. Upon reception of that message, and since necessarily $Recovered_\ell[\ell] < Recovered_u[u]$ at process u at that instant, u adopts ℓ as its leader in Task 2. Moreover, by the fact that u initializes $Timeout_u[\ell]$ to $\eta + incarnation_u$, eventually $timer_u$ does not expire on ℓ any more. Also, at the end of the wait of Task 1, u will write $leader_u = \ell$ in stable storage. After this happens, u will not send any more messages, and the value of $leader_u$ will be ℓ forever, since upon recovery u will read ℓ as its leader from stable storage. □

Theorem 1. *The algorithm of* Fig. 3 *implements Omega* (Property 1) *in system S_1: there is a time after which every process that is up, either correct or unstable, always trusts the same correct process ℓ.*

Proof. Follows directly from Lemmas 2, 4 and 6. □

Theorem 2. *The algorithm of* Fig. 3 *is communication-efficient: there is a time after which only process ℓ sends messages forever.*

Proof. Follows directly from Lemmas 3, 5 and 6. □

4. A near-communication-efficient Omega algorithm for system S_2

In this section, we present a near-communication-efficient algorithm implementing Omega (Property 2) in system S_2, which assumes that processes do not have access to any form of stable storage. In particular, when a process crashes all its variables loose their values. Fig. 4 presents the algorithm in detail, which requires a majority of the processes in the system to be correct. Contrary to the previous algorithm, where the variable $leader_p$ was initialized from stable storage, $leader_p$ is now initialized to the "no-leader" $\perp$ value. Also, since processes do not have an incarnation counter in stable storage, $Timeout_p[q]$ is initialized to η for every other process q, and $Recovered_p[p]$ is initialized to 1.

The algorithm works as follows. During initialization (and upon recovery), p sends a *RECOVERED* message to the rest of processes, in order to inform them that it has recovered. After that, p starts the three tasks of the algorithm. In Task 1, which is periodically activated every η time units, if p trusts itself, then it sends a *LEADER* message containing $Recovered_p$ to the rest of processes. Otherwise, if $leader_p = \perp$ then p sends an *ALIVE* message to the rest of processes in order to help choosing an initial leader. Task 2 is activated whenever p receives either a *RECOVERED*, an *ALIVE* or a *LEADER* message from another process q. If p receives a *RECOVERED* message from q, p increments $Recovered_p[q]$. Otherwise, if p receives an *ALIVE* message from q, then if $leader_p = \perp$ and p has received so far *ALIVE* from $\lfloor n/2 \rfloor$ different processes, p considers itself the leader, setting $leader_p$ to p. Finally, if p receives a *LEADER* message from q, then p updates $Recovered_p$ with $Recovered_q$ as in the previous algorithm (i.e., taking the highest value for each component of the vector), as well as its time-out with respect to q, $Timeout_p[q]$, taking the highest value between its current value and $Recovered_p[p]$. After that, p checks if q deserves to become p's leader, which is the case if either (1) $leader_p = \perp$ and $Recovered_p[q] \leq Recovered_p[p]$, or (2) $leader_p \neq \perp$ and $Recovered_p[q] \leq Recovered_p[leader_p]$ (using process identifiers to break ties). In that case, p sets q as its leader and resets $timer_p$ to $Timeout_p[q]$ in order to monitor q (i.e., $leader_p$) again. Finally, p also checks if it deserves to become the leader, which is the case if $leader_p$ continues being $\perp$ or $Recovered_p[p] \leq Recovered_p[leader_p]$ (using process identifiers to break ties). If it is the case, then p sets $leader_p$ to p and stops $timer_p$.

In Task 3, whenever $timer_p$ expires, as in the previous algorithm p increments $Timeout_p[leader_p]$ in order to avoid new premature suspicions on $leader_p$. But differently, now p resets $leader_p$ to $\perp$ and empties the set of *ALIVE* messages received so far. This is done in order to avoid several unstable processes to alternate forever being one of them the leader of the rest, which could occur if they are continuously crashing and recovering and their respective timers always expire before receiving a *LEADER* message from the correct leader ℓ. Observe that resetting $leader_p$ to $\perp$ leads p to start sending again *ALIVE* messages periodically by Task 1.

In this algorithm, all the processes set $leader_p$ to $\perp$ during initialization. Since a majority of the processes are correct, at least a process p will receive a majority of *ALIVE* messages, setting $leader_p$ to p in Task 2 and starting to send *LEADER* messages by Task 1. Since unstable processes crash and recover an infinite number of times, $\forall p \in correct, \forall u \in unstable$: $Recovered_p[u]$ is unbounded. However, eventually, when all the correct processes recover definitely, the recovery counters for correct processes will not increase any more, i.e., $\forall p \in \Pi, \forall q \in correct$: $Recovered_p[q]$ is bounded. This recovery counter values will be propagated among correct processes in the *LEADER* messages sent. The correct process, ℓ, with the smallest propagated recovery value will set $leader_\ell$ to ℓ in Task 2, and after that $leader_\ell$ will be ℓ permanently. Every other correct process p will receive *LEADER* messages from ℓ periodically and will adjust $timer_p$ so that $leader_p = \ell$ permanently. Any unstable process u will set $leader_u$ to $\perp$ every time it recovers and to ℓ when it receives a *LEADER* message from ℓ. Thanks to the line $Timeout_u[\ell] \leftarrow \max(Timeout_u[\ell], Recovered_u[u])$ of Task 2, eventually the timer u sets on ℓ will not expire any more, since $Recovered_\ell[u]$ is unbounded and u will update $Recovered_u[u]$ when it receives a *LEADER* message from ℓ. Therefore, for every unstable process u, initially $leader_u = \perp$ and then $leader_u = \ell$ when u receives a *LEADER* message from ℓ.

4.1. Correctness proof

We show now that the algorithm of Fig. 4 implements Omega (Property 2) in system S_2, and that it is near-communication-efficient.

Lemma 7. *Any message* (*RECOVERED*, p), (*LEADER*, p, $Recovered_p$) *or* (*ALIVE*, p), $p \in \Pi$, *eventually disappears from the system.*

Proof. A message m cannot remain forever in a link, since it remains at most δ time in an eventually timely link if sent after T (otherwise, i.e., if m is sent before T, then it is eventually lost or received), and is eventually lost or received in a lossy asynchronous link or a fair lossy link. Also, m cannot remain forever in the destination process, since processes are assumed to be synchronous. Hence, m eventually disappears from the system. □

Observation 1. *Since correct processes crash and recover a finite number of times, and RECOVERED messages are only sent during initialization, $\forall p \in \Pi, \forall q \in correct$: $Recovered_p[q]$ is bounded.*

We naturally assume that every unstable process u sends infinite *RECOVERED* messages, i.e., infinitely often, whenever u recovers from a crash, it executes the instruction which sends a *RECOVERED* message to the rest of processes. Note that if eventually u does no longer execute that instruction, then u is indistinguishable from an eventually down process (and $leader_u = \perp$ forever when u is up).

Observation 2. *Since unstable processes crash and recover an infinite number of times, $\forall p \in correct$, $\forall u \in unstable$: $Recovered_p[u]$ is unbounded.*

For the rest of the proof, we will consider a process as unstable only if it completes the initialization of the algorithm an infinite

```
Initialization:
    leader_p ← ⊥
    for all q ∈ Π except p:
        Timeout_p[q] ← η
        Recovered_p[q] ← 0
    Recovered_p[p] ← 1
    send (RECOVERED, p) to all processes except p
    start tasks 1, 2 and 3

Task 1:
    repeat forever every η time units
        if leader_p = p then
            send (LEADER, p, Recovered_p) to all processes except p
        else if leader_p = ⊥ then
            send (ALIVE, p) to all processes except p
        end if

Task 2:
    upon reception of message (RECOVERED, q) or (ALIVE, q) or (LEADER, q, Recovered_q) do
        if message is of type RECOVERED then
            Recovered_p[q] ← Recovered_p[q] + 1
        else if message is of type ALIVE then
            if (leader_p = ⊥) and (p has received so far ALIVE from ⌊n/2⌋ different processes) then
                leader_p ← p
            end if
        else if message is of type LEADER then
            for all r ∈ Π:
                Recovered_p[r] ← max(Recovered_p[r], Recovered_q[r])
            Timeout_p[q] ← max(Timeout_p[q], Recovered_p[p])
            if ((leader_p = ⊥) and (⟨Recovered_p[q], q⟩ < ⟨Recovered_p[p], p⟩)) or
              ((leader_p ≠ ⊥) and (⟨Recovered_p[q], q⟩ ≤ ⟨Recovered_p[leader_p], leader_p⟩)) then
                leader_p ← q
                reset timer_p to Timeout_p[q]
            end if
            if (leader_p = ⊥) or (⟨Recovered_p[p], p⟩ < ⟨Recovered_p[leader_p], leader_p⟩) then
                leader_p ← p
                stop timer_p
            end if
        end if

Task 3:
    upon expiration of timer_p do
        Timeout_p[leader_p] ← Timeout_p[leader_p] + 1
        leader_p ← ⊥
        empty the set of ALIVE messages received so far
```

Fig. 4. Near-communication-efficient Omega algorithm in system S_2.

number of times, including the sending of the *RECOVERED* message to the rest of processes (otherwise, although formally unstable, the process is considered eventually down). We will also assume that any time instant t is larger than $t_1 > t_0$, 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 correct (i.e., eventually up) process has definitely recovered, all *RECOVERED* messages sent by correct processes have disappeared from the system (this eventually happens by Lemma 7 and the fact that *RECOVERED* messages are only sent during initialization), and the counter of the number of times that every unstable process has recovered at every correct process p is bigger than the counter of the number of times that every correct process has recovered at p, i.e., $\forall p, q \in correct, \forall u \in unstable$: $Recovered_p[u] > Recovered_p[q]$ (this eventually happens by Observations 1 and 2),
(2) and t_1 is a time instant such that all *LEADER* messages sent before t_0 have disappeared from the system (this eventually happens by Lemma 7).

Lemma 8. *Eventually for every correct process p and every unstable process u, $leader_p \neq u$ permanently.*

Proof. Let p and u be any correct process and any unstable process, respectively. By Observations 1 and 2 eventually $Recovered_p[p] < Recovered_p[u]$ permanently. When this holds, if $leader_p$ is still an unstable process u, $leader_p$ will be set to p at the end of Task 2 when p receives a *LEADER* message from u. If p does not receive timely a *LEADER* message from u, then $timer_p$ will expire and $leader_p$ will be set to $\perp$ in Task 3. After that, p will not set $leader_p$ to u any more. Otherwise, if $leader_p$ is different from an unstable process u, $leader_p$ will not be set to u in Task 2 any more, because p itself is a better candidate to be the leader. □

Lemma 9. *Eventually for every correct process p and every eventually down process q, $leader_p \neq q$ permanently.*

Proof. Let p and q be any correct process and any eventually down process, respectively. By definition, eventually q will crash and will not recover. If when this occurs for some correct process p $leader_p = q$, then $timer_p$ will expire and $leader_p$ will be set to $\perp$ in Task 3. After that, $leader_p \neq q$ permanently. □

Observation 3. *By the algorithm, no process p can have another process q as its leader without having received a LEADER message from q.*

Lemma 10. *There exists a time t such that for every $t' > t$, for some correct process p, $leader_p = p$.*

Proof. By Lemma 8, eventually no correct process p has an unstable process as its leader. By Lemma 9, eventually no correct process p has an eventually down process as its leader. Therefore, eventually $leader_p = \perp$, $leader_p = q$ being q another correct process, or $leader_p = p$ for every correct process p. Observe that if $leader_p = q$ for two correct processes p and q, then by Observation 3 q has already set $leader_q = q$. Observe also that all correct processes will not have permanently their leader set to $\perp$, because in that case at least a correct process p would receive a majority of *ALIVE* messages and would set $leader_p = p$ in Task 2. Therefore, eventually some correct process p will have $leader_p = p$. Once this occurs, considering Observations 1 and 2, p only will change $leader_p$ when it receives a *LEADER* message from another correct process q with $Recovered_p[q] \leq Recovered_p[p]$ (using process identifiers to break ties), setting $leader_p = q$. By Observation 3, this means that there is another correct process q with $leader_q = q$. □

From the previous, we have that eventually there will be always some correct process p such that $leader_p = p$ that sends *LEADER* messages to the rest of processes by Task 1 of the algorithm. Apart from correct processes, unstable processes can send *LEADER* messages as well. Observe that after time t_1 the recovery counters for correct processes will not increase any more. Let K be the set of correct processes which send *LEADER* messages to the rest of processes after time t_1. By the algorithm, eventually the recovery counter for every correct process q at every correct process $p \in K$, $Recovered_p[q]$, is set forever to the highest recovery counter for q propagated in *LEADER* messages. Observe that some correct process $r \notin K$ could have a higher recovery counter for q, $Recovered_r[q]$, that is not propagated. Let be $\ell \in K$ the correct process such that $Recovered_p[\ell] \leq Recovered_p[q]$ (using process identifiers to break ties) for every correct processes $p, q \in K$. We will show that eventually and permanently, (1) for every correct process p, $leader_p = \ell$, and (2) for every unstable process u, initially $leader_u = \perp$ and then $leader_u = \ell$. Observe that for every correct process $r \notin K$, $Recovered_r[\ell] \leq Recovered_r[r]$ (with $l < r$ in case of tie), otherwise r would set $leader_r$ to r at the reception of a *LEADER* message from ℓ in Task 2 and therefore r would be in K.

Lemma 11. *Eventually and permanently $leader_\ell = \ell$.*

Proof. By definition of ℓ, eventually and permanently $Recovered_\ell[\ell] \leq Recovered_\ell[q]$ for every correct process $q \in K$ (using process identifiers to break ties). Therefore, if ℓ sets $leader_\ell$ to ℓ in Task 2 ℓ will not change it neither in Task 2 nor in Task 3 any more, and hence $leader_\ell = \ell$ permanently. So we have to prove that eventually ℓ sets $leader_\ell$ to ℓ. If $leader_\ell$ is $\perp$ and ℓ receives *ALIVE* from $\lfloor n/2 \rfloor$ different processes, by Task 2 ℓ will set $leader_\ell$ to ℓ. If ℓ receives a *LEADER* message from an unstable process, ℓ will set $leader_\ell$ to ℓ at the end of Task 2. If neither of this previous conditions are given before, by Lemma 10 ℓ will receive a *LEADER* message from another correct process $q \in K$, and again ℓ will set $leader_\ell$ to ℓ in Task 2. □

Lemma 12. *Eventually and permanently $leader_p = \ell$ for every correct process $p \in K$.*

Proof. The lemma directly follows from Lemma 11 for $p = \ell$. Let be now $p \neq \ell$, with $p \in K$. By Lemma 11 and Task 1 of the algorithm, eventually ℓ sends *LEADER* messages permanently. By Lemmas 8 and 9, eventually p does not choose an unstable or an eventually down process as its leader. Whenever p receives a *LEADER* message from ℓ, p sets its leader to ℓ in Task 2, since by definition of ℓ $Recovered_p[\ell] \leq Recovered_p[q]$ for every correct process $q \in K$ (using process identifiers to break ties). After that, every time $timer_p$ expires, p will increment $Timeout_p[\ell]$ and will set $leader_p = \perp$. Eventually p will receive another *LEADER* message from ℓ and will set $leader_p$ to ℓ again. Observe that $timer_p$ can expire a finite number of times on ℓ, since by Task 3 every time it expires p increments $Timeout_p[\ell]$, and the link from ℓ to p is eventually timely. Hence, eventually by Task 2 p receives a *LEADER* message from ℓ periodically and timely, i.e., before $timer_p$ expires. After this happens, p will not change $leader_p$ to a value different from ℓ any more. □

Lemma 13. *Eventually and permanently $leader_p = \ell$ for every correct process p.*

Proof. The lemma directly follows from Lemma 12 for $p \in K$. Let be now p a correct process such that $p \notin K$. Eventually, when Lemma 12 holds, among correct processes only ℓ sends *LEADER* messages and by Lemmas 8 and 9, eventually p does not choose an unstable or an eventually down process as its leader. Therefore, when p receives a *LEADER* message from ℓ it sets $leader_p$ to ℓ in Task 2. Observe that p will not set $leader_p$ to p, since otherwise p would send a *LEADER* message and p would be in K. After that, every time $timer_p$ expires, p will increment $Timeout_p[\ell]$ and will set $leader_p = \perp$. Eventually p will receive another *LEADER* message from ℓ and will set $leader_p$ to ℓ again. Observe that $timer_p$ can expire a finite number of times on ℓ, since by Task 3 every time it expires p increments $Timeout_p[\ell]$, and the link from ℓ to p is eventually timely. Hence, eventually by Task 2 p receives a *LEADER* message from ℓ periodically and timely, i.e., before $timer_p$ expires. After this happens, p will not change $leader_p$ to a value different from ℓ any more. □

Lemma 14. *Eventually, every unstable process will stop sending LEADER messages forever.*

Proof. Eventually an unstable process u will not set $leader_u$ to u and will not send *LEADER* messages any more. There are two cases to consider:

(a) An unstable process u could set $leader_u$ to u when, having $leader_u = \perp$, u receives *ALIVE* from $\lfloor n/2 \rfloor$ different processes. Observe that eventually, when Lemma 13 holds, every correct process p will have $leader_p \neq \perp$ permanently, and therefore, correct processes, a majority of the processes in the system, stop sending *ALIVE* messages definitely, and hence an unstable process u will never receive *ALIVE* from $\lfloor n/2 \rfloor$ different processes.

(b) An unstable process u could also change $leader_u$ from $\perp$ to u in Task 2 after receiving a *LEADER* message from a process q, if $Recovered_u[u] < Recovered_u[q]$ or $Recovered_u[u] = Recovered_u[q]$ and $u < q$. Observe that eventually, when Lemma 12 holds, by Observations 1 and 2, this only might occur if u receives a *LEADER* message from another unstable process v before receiving a *LEADER* message from ℓ. In this case, if $Recovered_u[u] \leq Recovered_u[v]$ (using process identifiers to break ties), then u will set $leader_u$ to u and will start to send *LEADER* messages. However, several unstable processes will not alternate forever being one of them the leader of the rest. Observe that if unstable processes which are up remain up sufficiently long, they will receive a *LEADER* message from ℓ and they will take ℓ as their leader and no unstable process u will set $leader_u$ to u any more. Besides, if all unstable processes put their leader to $\perp$ or to ℓ at the same time no unstable process u will set $leader_u$ to u any more. Therefore, the leadership alternance among unstable processes only may occur if there is always at least an unstable process u up with $leader_u$ set to u that sends *LEADER* messages that make another unstable process v set $leader_v$ to v after which u crashes. But with this purpose, the

unstable process u such that $leader_u$ is u will change from an unstable process with a higher recovery counter to another with a lower recovery counter (using process identifiers to break ties). At the end, when w, the unstable process with the smallest recovery counter, sets $leader_w$ to w and sends *LEADER* messages, no other unstable process will become leader. Therefore, when w crashes, the timers that unstable processes with w as their leader have set on w will expire (if they do not crash before) and they will set their leader to $\perp$ in Task 3. After that, no other unstable process will send *LEADER* messages again.

□

Lemma 15. *Eventually, every unstable process upon recovery will have* $leader_u = \perp$ *first and — if it remains up for sufficiently long — then* $leader_u = \ell$ *until it crashes.*

Proof. Eventually, when Lemmas 12 and 14 hold, process ℓ will be the unique process sending *LEADER* messages in the system. Hence, whenever an unstable process u recovers, if it remains up for sufficiently long, it will receive a *LEADER* message from ℓ, and it will update $Recovered_u$ from $Recovered_\ell$, as well as the value of $Timeout_u[\ell]$. On the one hand, by Observations 1 and 2, $Recovered_u[u] > Recovered_u[\ell]$ when u updates $Recovered_u$ from $Recovered_\ell$. On the other hand, $Timeout_u[\ell]$ will be updated with the maximum value between $Timeout_u[\ell]$ and $Recovered_u[u]$. This way, considering that $Recovered_u[u]$ increases forever for every unstable process u, $Timeout_u[\ell]$ will be such that $timer_u$ will not expire any more (since the link from ℓ to u is eventually timely). Thus, when u recovers, if it receives a message from ℓ, u will change its leader from $\perp$ to ℓ and ℓ will remain as u's leader until u crashes. If u crashes before receiving the *LEADER* message from ℓ, $leader_u$ will continue being $\perp$. □

Theorem 3. *The algorithm of Fig. 4 implements Omega (Property 2) in system* S_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.*

Proof. Follows directly from Lemmas 13 and 15. □

Theorem 4. *The algorithm of Fig. 4 is near-communication-efficient: there is a time after which, among correct processes, only* ℓ *sends messages forever.*

Proof. Follows directly from Lemma 13 and Task 1 of the algorithm. □

Finally, observe that the algorithm of Fig. 4 is not communication-efficient, since besides the leader ℓ, unstable processes also send messages forever. Interestingly, eventually every time an unstable process u recovers, if it remains up for sufficiently long, as soon as it sets $leader_u = \ell$ it stops sending messages until it crashes.

5. Relaxing the communication reliability and synchrony assumptions

In the algorithms presented in this work, it is possible to relax the assumptions on communication reliability and synchrony, by means of the use of message relaying, i.e., the first time a process p receives a message m, before delivering it p resends m to the rest of processes (a small optimization consists in not sending m neither to its original sender nor to the process from which m has been received, if different from the original sender). This approach requires messages to be uniquely identified, in order to detect duplicates. A usual way to do it is to add a pair (*sender_id*, *sequence_number*) to every message. In the crash–recovery failure model, uniqueness of the sequence number requires to store it in stable storage. An alternative consists in adding a timestamp given by the sender's clock, assuming that clocks continue running despite the crash of processes.

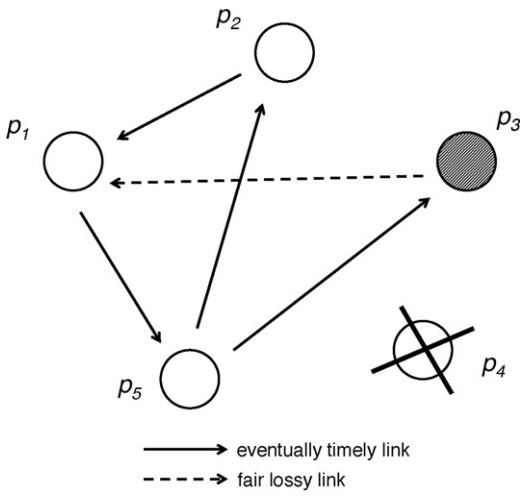

Fig. 5. Relaxed scenario of system S_2: three processes eventually up, one eventually down, one unstable.

According to the above, the algorithm of Fig. 3 works under the following weaker assumption:

(i') For every correct process p, there is an eventually timely *path* from p to every correct and every unstable process.

Similarly, the algorithm of Fig. 4 works under the following weaker assumptions:

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

Fig. 5 presents a scenario which satisfies the weaker assumptions required by the algorithm of Fig. 4. A consequence of the use of message relaying is that the algorithms will no longer be (near-)communication-efficient *sensu stricto*, i.e., they remain (near-)communication-efficient only regarding the number of (correct) processes that send "new" messages forever.

6. Conclusion

In this paper, we have studied the leader election problem in distributed systems where processes can crash and recover. The concepts of communication efficiency and near-efficiency for an algorithm implementing the Omega failure detector class have been defined. Depending on the use or not of stable storage, the property satisfied by unstable processes varies. Then, two Omega algorithms have been presented, one of which is communication-efficient and relies on the use of stable storage, while the other is near-communication-efficient and does not rely on stable storage but on a majority of correct processes.

We believe that Omega, as defined in this paper, can be useful to solve consensus in the crash–recovery model, in particular because its definition avoids the disagreement among unstable processes and correct processes. However, designing efficient consensus protocols based on Omega in the crash–recovery model remains an open research field. In this regard, we think that existing

leader-based consensus protocols for the crash model can be adapted to the crash–recovery model with the help of the Omega algorithms proposed in this paper.

Compared to the state of the art in implementing Omega in the crash failure model, the algorithms presented in this paper rely on stronger synchrony assumptions, e.g., they require every pair of correct processes to be connected by an eventually timely link. Our aim has been to keep algorithms relatively simple, but at the same time to achieve communication efficiency as defined in this paper. Said this, we believe that it could be possible to weaken the synchrony assumptions while preserving communication efficiency. However, determining the weakest synchrony assumptions to implement Omega efficiently in crash–recovery is an open research line.

Acknowledgements

The authors would like to thank the anonymous referees for their helpful comments. Research partially supported by the Spanish Research Council (MCeI), under grant TIN2010-17170, the Basque Government, under grants IT395-10 and S-PE10UN55, and the Comunidad de Madrid, under grant S2009/TIC-1692.

References

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

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

Aguilera, M., Delporte-Gallet, C., Fauconnier, H., Toueg, S., 2003. 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, July 2004, pp. 306–314.

Aguilera, M., Delporte-Gallet, C., Fauconnier, H., Toueg, S., 2004. 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, July 2004, pp. 328–337.

Aguilera, M., Delporte-Gallet, C., Fauconnier, H., Toueg, S., 2008. On implementing omega in systems with weak reliability and synchrony assumptions. Distributed Computing 21 (4), 285–314.

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

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

Chu, F., 1998. Reducing Ω to $\Diamond\mathcal{W}$. Information Processing Letters 67 (September (6)), 289–293.

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

Fernández, A., Jiménez, E., Arévalo, S., 2006a. 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, December 2006, pp. 63–72.

Fernández, A., Jiménez, E., Raynal, M., 2006b. 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), Philadelphia, PA, June 2006, pp. 166–178.

Fernández, A., Raynal, M., 2007. From an intermittent rotating star to a leader. In: Proceedings of the 11th International Conference on Principles of Distributed Systems (OPODIS'2007), Springer-Verlag, Guadeloupe, French West Indies, December 2007, LNCS 4878, pp. 189–203.

Fischer, M., Lynch, N., Paterson, M., 1985. Impossibility of distributed consensus with one faulty process. Journal of the ACM 32 (April (2)), 374–382.

Freiling, F., Lambertz, C., Majster-Cederbaum, M., 2009. Modular consensus algorithms for the crash–recovery model. In: Proceedings of the 10th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT'2009), Higashi Hiroshima, Japan, December 2009, pp. 287–292.

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

Hurfin, M., Mostéfaoui, A., Raynal, M., 1998. Consensus in asynchronous systems where processes can crash and recover. In: Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems (SRDS'1998), West Lafayette, IN, USA, October 1998, pp. 280–286.

Jiménez, E., Arévalo, S., Fernández, A., 2006. Implementing unreliable failure detectors with unknown membership. Information Processing Letters 100 (2), 60–63.

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

Larrea, M., Fernández, A., Arévalo, S., 2000. 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, October 2000, pp. 52–59.

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

Malkhi, D., Oprea, F., Zhou, L., 2005. *Omega* meets paxos: leader election and stability without eventual timely links. In: Proceedings of the 19th International Symposium on Distributed Computing (DISC'2005), Springer-Verlag, Krakow, Poland, September 2005, LNCS 3724, pp. 199–213.

Martín, C., Larrea, M., 2008. Eventual leader election in the crash–recovery failure model. In: Proceedings of the 14th Pacific Rim International Symposium on Dependable Computing (PRDC'2008), Taipei, Taiwan, December 2008, pp. 208–215.

Martín, C., Larrea, M., 2010. A simple and communication-efficient Omega algorithm in the crash–recovery model. Information Processing Letters 110 (3), 83–87.

Martín, C., Larrea, M., Jiménez, E., 2007. 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), Vienna, Austria, April 2007, pp. 975–982.

Martín, C., Larrea, M., Jiménez, E., 2009. Implementing the Omega failure detector in the crash–recovery failure model. Journal of Computer and System Sciences 75 (May (3)), 178–189.

Mostéfaoui, A., Mourgaya, E., Raynal, M., 2003. Asynchronous implementation of failure detectors. In: Proceedings of the IEEE International Conference on Dependable Systems and Networks (DSN'2003), San Francisco, CA, June 2003, pp. 351–360.

Mostéfaoui, A., Mourgaya, E., Raynal, M., Travers, C., 2006a. A time-free assumption to implement eventual leadership. Parallel Processing Letters 16 (June (2)), 189–208.

Mostéfaoui, A., Rajsbaum, S., Raynal, M., Travers, C., 2007. From omega to Omega: A simple bounded quiescent reliable broadcast-based transformation. Journal of Parallel and Distributed Computing 67 (January (1)), 125–129.

Mostéfaoui, A., Raynal, M., 2001. Leader-based consensus. Parallel Processing Letters 11 (March (1)), 95–107.

Mostéfaoui, A., Raynal, M., Travers, C., 2004. Crash–resilient time-free eventual leadership. In: Proceedings of the 23rd IEEE Symposium on Reliable Distributed Systems (SRDS'2004), Florianópolis, Brazil, October 2004, pp. 208–217.

Mostéfaoui, A., Raynal, M., Travers, C., 2006b. Time-free and timer-based assumptions can be combined to obtain eventual leadership. IEEE Transactions on Parallel and Distributed Systems 17 (July (7)), 656–666.

Oliveira, R., Guerraoui, R., Schiper, A., 1997. Consensus in the crash–recover model. Technical Report TR-97/239. Swiss Federal Institute of Technology, Lausanne, July 1997.

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

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

Mikel Larrea received his MS degree in Computer Science from the Swiss Federal Institute of Technology in 1995, and his PhD degree in Computer Science from the University of the Basque Country in 2000. He is currently an associate professor of Computer Science at the University of the Basque Country. His research interests include distributed algorithms and systems, fault tolerance and ubiquitous computing.

Cristian Martín received his MS and PhD degrees in Computer Science from the University of the Basque Country in 2002 and 2011, respectively. He is currently a researcher at the Ikerlan Research Center. His research interests include distributed and ubiquitous systems, and fault tolerance.

Iratxe Soraluze received her MS and PhD degrees in Computer Science from the University of the Basque Country in 1999 and 2004, respectively. She is currently an assistant professor of Computer Science at the University of the Basque Country. Her research interests include distributed algorithms and systems, fault tolerance and ubiquitous computing.