Information Processing Letters 110 (2010) 83–87

Contents lists available at ScienceDirect

Information Processing Letters

www.elsevier.com/locate/ipl

A simple and communication-efficient Omega algorithm in the crash-recovery model ☆

Cristian Martín [1], Mikel Larrea *

The University of the Basque Country, 20018 San Sebastián, Spain

ARTICLE INFO

Article history:
Received 18 August 2009
Received in revised form 6 October 2009
Accepted 20 October 2009
Available online 24 October 2009
Communicated by A.A. Bertossi

Keywords:
Distributed computing
Fault tolerance
Consensus
Omega failure detector
Leader election
Crash-recovery
Communication-efficient algorithm

ABSTRACT

This paper presents a new algorithm implementing the Omega failure detector in the crash-recovery model. Contrary to previously proposed algorithms, this algorithm does not rely on the use of stable storage *and* is communication-efficient, i.e., eventually only one process (the elected leader) keeps sending messages. The algorithm relies on a nondecreasing local clock associated with each process. Since stable storage is not used to keep the identity of the leader in order to read it upon recovery, unstable processes, i.e., those that crash and recover infinitely often, output a special $\perp$ value upon recovery, and then agree with correct processes on the leader after receiving a first message from it.

1. Introduction

A fundamental problem in fault-tolerant distributed computing is the consensus problem [30]. Roughly speaking, consensus allows a set of processes to decide on a common value that has necessarily been proposed by one of them. The importance of consensus relies on the fact that other agreement problems like group membership and totally ordered broadcast can be reduced to some form of consensus, and hence solutions to these problems can be built on top of a consensus algorithm.

Since Fischer et al. showed the impossibility of solving consensus deterministically in asynchronous systems if at least one process can crash [13], several ways of circumventing this impossibility result have been studied. One of the most successful approaches, proposed by Chandra and Toueg in [7], consists in augmenting the asynchronous system with an unreliable failure detector, which provides (possibly incorrect) information about process failures. The completeness and accuracy properties satisfied by Chandra–Toueg's unreliable failure detectors give enough information to solve consensus. Moreover, with Hadzilacos they showed in [6] that a failure detector called Omega is the weakest failure detector for solving consensus. Informally, Omega provides an eventual leader election functionality, i.e., eventually all processes agree on a common correct process. Omega, or a similar weak leader election mechanism, is at the heart of several consensus algorithms that have been proposed [14,16,18,27].

A lot of algorithms implementing Omega in the crash model, i.e., in which a crashed process does not recover, have been proposed [2–5,8,10–12,15,17,20,24–26,28,29]. They differ in aspects like the communication reliability and synchrony assumptions (e.g., the number of eventually timely and of fair lossy links), the communication pattern among processes (all-to-all, logical ring, rotating

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

* Corresponding author.
E-mail addresses: martin.cristian@gmail.com, cmartin@ikerlan.es (C. Martín), mikel.larrea@ehu.es (M. Larrea).
[1] Current address: Ikerlan Research Center, 20500 Arrasate-Mondragón, Spain.

0020-0190/$ – see front matter
doi:10.1016/j.ipl.2009.10.011

star, etc.), and the initial knowledge or not of the membership. In some of these algorithms, eventually only one process (the elected leader) keeps sending messages periodically to the rest of processes. Such an algorithm is said communication-efficient [3], or more recently, quiescent [19].

Failure detection has also been studied in the crash-recovery model, i.e., in which a crashed process can recover (even infinitely often). Aguilera et al. defined in [1] an adaptation of the $\diamond\mathcal{S}$ failure detector to the crash-recovery model, proposing an algorithm implementing it in partially synchronous systems [7,9]. Regarding specific algorithms implementing Omega in the crash-recovery model, Martín et al. have proposed in [22,23] several Omega algorithms that rely on the use of stable storage to keep, among other informations, the identity of the leader and a local incarnation number associated with each process, while Martín and Larrea have proposed in [21] an Omega algorithm that does not use stable storage. These algorithms either rely on a message forwarding mechanism and/or have a permanent all-to-all communication pattern, and hence require a high number of messages to be exchanged. Recently, Larrea and Martín have proposed in [19] two more efficient Omega algorithms, one of which uses stable storage and is quiescent, i.e., eventually only one process keeps sending messages, while the other one does not use stable storage and is near-quiescent, i.e., eventually only one *correct* process keeps sending messages.[2]

In this work we present a simple and communication-efficient Omega algorithm in the crash-recovery model which does not rely on the use of stable storage but on a nondecreasing local clock associated with each process. With this algorithm, correct processes, i.e., those that eventually remain up forever, will eventually and permanently agree on the same correct process ℓ. Moreover, eventually ℓ will be the only process that keeps sending messages to the rest of processes. Regarding *unstable* processes, i.e., those that crash and recover infinitely often, since stable storage is not used they must "learn" from some other process(es) — actually, from ℓ — the identity of the leader upon recovery. In this regard, we make unstable processes not trust any process upon recovery, i.e., output a special value $\perp$, until either they trust the leader or crash.

2. System model

We consider a system S composed of a finite and totally ordered set $\Pi = \{p_1, p_2, \ldots, p_n\}$ of $n > 1$ processes that communicate only by sending and receiving messages. 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 [22]: (1) *eventually up*, i.e., processes that eventually remain up forever, (2) *eventually down*, i.e., processes that eventually remain crashed forever, and (3) *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.

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.

Each process has a nondecreasing local clock that can measure intervals of time with a bounded drift (the bound is unknown). The clocks of the processes are not synchronized. We assume that clocks continue running despite the crash of processes.

Communication links cannot create or alter messages, and are not assumed to be FIFO. Concerning timeliness or loss properties, we consider the following types of links [3]: (1) *eventually timely links*, where there is an unknown bound δ on message delays and an unknown global stabilization time T, such that if a message is sent at a time $t \geqslant T$, then this message is received by time $t+\delta$, and (2) *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. More precisely, we assume that 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, i.e., the links from/to eventually down processes and the links from unstable processes, can be lossy asynchronous.

Finally, the Omega failure detector, adapted to system S, satisfies the following property [21]: *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.*

3. The algorithm

In this section, we present a communication-efficient algorithm implementing Omega in system S without using stable storage. Fig. 1 presents the algorithm in detail. The process chosen as leader by a process p, i.e., trusted by p, is held in a variable $leader_p$, which is initialized to the special value $\perp$, indicating that no process is trusted by p yet. Every process p also has a $Timeout_p$ variable used to set a timer with respect to its current leader, initialized to the value returned by the local clock $clock()$, as well as two timestamps ts_p and $ts_{\min}$, initialized to $clock()$ and to ts_p, respectively. Note that the initialization part of the algorithm is executed by p at each recovery.

The algorithm, which is composed of three concurrent tasks that are started at the end of the initialization, works as follows. In Task 1, p first waits $Timeout_p$ time units, after which if p still has no leader, i.e., $leader_p = \perp$, then p sets $leader_p$ to p. Otherwise, p resets $timer_p$ to $Timeout_p$

[2] The small difference between a quiescent Omega algorithm and a near-quiescent Omega algorithm is that in the latter, besides the leader, unstable processes can send messages forever.

Every process p executes the following:

```
Initialization:
  leader_p ← ⊥
  Timeout_p ← clock()
  ts_p ← clock()
  ts_min ← ts_p
  start tasks 1, 2 and 3

Task 1:
  wait (Timeout_p) time units
  if leader_p = ⊥ then
    leader_p ← p
  else
    reset timer_p to Timeout_p
  end if
  repeat forever every η time units
    if leader_p = p then
      send (LEADER, p, ts_p) to all processes except p
    end if

Task 2:
  upon reception of (LEADER, q, ts_q) do
    if (ts_q < ts_min)
      or [(ts_q = ts_min) and (leader_p = ⊥) and (q < p)]
      or [(ts_q = ts_min) and (leader_p ≠ ⊥) and (q ⩽ leader_p)] then
        leader_p ← q
        ts_min ← ts_q
        reset timer_p to Timeout_p
    end if

Task 3:
  upon expiration of timer_p do
    Timeout_p ← Timeout_p + 1
    leader_p ← p
    ts_min ← ts_p
```

Fig. 1. Communication-efficient Omega algorithm in the crash-recovery model.

in order to monitor its current leader. Then, p enters a permanent loop in which every η time units it checks if it is the leader, i.e., $leader_p = p$, in which case p sends a $(LEADER, p, ts_p)$ message to the rest of processes.

Task 2 is activated whenever p receives a $(LEADER, q, ts_q)$ message from another process q. Observe that this task is active during p's waiting instruction of Task 1. The received message is taken into account if either (1) $ts_q < ts_{\min}$, i.e., q has recovered earlier than p's current leader, (2) $(ts_q = ts_{\min})$ and $(leader_p = \bot)$ and $(q < p)$, i.e., p has no leader yet and q is a good candidate, or (3) $(ts_q = ts_{\min})$ and $(leader_p \neq \bot)$ and $(q \leqslant leader_p)$, i.e., q is a better candidate than $leader_p$ (or $q = leader_p$). In all these cases p adopts q as its current leader, setting $leader_p$ to q and $ts_{\min}$ to ts_q, and resets $timer_p$ to $Timeout_p$.

In Task 3, which is activated whenever $timer_p$ expires, p "suspects" its current leader: it increments $Timeout_p$ in order to avoid premature erroneous suspicions in the future, and considers itself as the new leader, setting $leader_p$ to p and $ts_{\min}$ to ts_p.

With this algorithm, the elected leader ℓ will be the "oldest" correct process, i.e., the process that first recovers definitely (using the process identifiers to break ties). Hence, eventually every correct process will permanently trust ℓ. Consequently, by Task 1 eventually only one correct process will keep sending messages. Concerning the behavior of unstable processes, the waiting instruction at the beginning of Task 1 guarantees that, eventually and permanently, unstable processes always receive a first $(LEADER, \ell, ts_\ell)$ message from ℓ before the end of the waiting, changing their leader from $\bot$ to ℓ in Task 2. Moreover, the initialization of $Timeout_p$ to $clock()$ prevents unstable processes from disturbing the leader election, because it ensures that eventually every unstable process u will never suspect the leader ℓ (since u's timeout with respect to ℓ keeps increasing forever, and hence eventually $timer_u$ will never expire). By the previous, it is simple to see that the algorithm is communication-efficient, i.e., eventually only one process (the elected leader ℓ) keeps sending messages.

Correctness proof

We show now that the algorithm of Fig. 1 implements Omega in system S, and that it is communication-efficient.

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

Proof. A message m cannot remain forever in a link, since it remains at most $T + \delta$ time in an eventually timely link, and is lost or eventually received in a lossy asynchronous link. Also, m cannot remain forever in the destination process, since processes are assumed to be synchronous. Then, the destination process will eventually by Task 2 either take m into account or drop it. Hence, m will eventually disappear 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 a clock value bigger than ts_p for every correct process p, i.e., $\forall u \in unstable$, $\forall p \in correct$: $ts_u > ts_p$,

(2) and t_1 is a time instant such that all messages sent before t_0 have disappeared from the system (this eventually happens from 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 $ts_u \leqslant ts_p$, for every correct process p.

Let be ℓ the correct process with the smallest value for its ts variable, i.e., the correct process that first recovers definitely. If two or more correct processes have the same final value for their ts variables, then let ℓ be the process with smallest identifier among them. We will show that eventually and permanently (1) for every correct process p, $leader_p = \ell$, and (2) for every unstable process u, either $leader_u = \bot$ or $leader_u = \ell$.

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

Proof. By the algorithm, the only way for process ℓ to have as leader another process q is by receiving an "acceptable" message from it in Task 2. However, it is simple to see that such a scenario cannot happen, since any $(LEADER, q, ts_q)$ message that ℓ can receive necessarily has either (1) $ts_q > ts_{\min} = ts_\ell$ at ℓ, or (2) $ts_q = ts_{\min}$ at ℓ and $q > \ell$, and hence is discarded in Task 2. As a result, eventually and permanently process ℓ considers itself the leader, i.e., $leader_\ell = \ell$. □

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

Proof. Follows directly from Lemma 2 and 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 2 and Task 1 of the algorithm, ℓ will periodically send a $(LEADER, \ell, ts_\ell)$ 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, and take it into account, setting $leader_p$ to ℓ and $ts_{\min}$ to ts_ℓ, and resetting $timer_p$ to $Timeout_p$. Observe that $timer_p$ can expire a finite number of times, since by Task 3 every time it expires p increments $Timeout_p$. Hence, eventually by Task 2 p will receive a $(LEADER, \ell, ts_\ell)$ 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. □

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

Proof. Follows directly from Lemma 4 and the algorithm.

Lemma 6. *Eventually, every unstable process u will not send any more messages, and $leader_u$ will be either $\perp$ or ℓ forever.*

Proof. By Lemma 2 and Task 1 of the algorithm, ℓ will periodically send a $(LEADER, \ell, ts_\ell)$ 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 $clock()$ time units at the beginning of Task 1, eventually by Task 2 u will always receive a first $(LEADER, \ell, ts_\ell)$ message from ℓ before the end of the waiting instruction of Task 1. Upon reception of that message, and since necessarily $ts_\ell < ts_{\min}$ at process u at that instant, u adopts ℓ as its leader, changing the value of $leader_u$ from $\perp$ to ℓ. Moreover, by the fact that u initializes $Timeout_u$ to $clock()$, eventually $timer_u$ will not expire any more. After this happens, u will not send any more messages. Also, the value of $leader_u$ will be either $\perp$ or ℓ forever. □

Theorem 1. *The algorithm of Fig.* 1 *implements Omega in system S.*

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

Theorem 2. *The algorithm of Fig.* 1 *is communication-efficient.*

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

References

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

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

[3] 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, Massachusetts, July 2003, pp. 306–314.

[4] 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, July 2004, pp. 328–337.

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

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

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

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

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

[10] 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, December 2006, pp. 63–72.

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

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

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

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

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

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

[17] 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, October 2000, pp. 52–59.

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

[19] M. Larrea, C. Martín, Quiescent leader election in crash-recovery systems, in: Proceedings of the 15th Pacific Rim International Symposium on Dependable Computing (PRDC'2009), Shanghai, China, November 2009, pp. 325–330.

[20] D. Malkhi, F. Oprea, L. Zhou, Omega meets paxos: Leader election and stability without eventual timely links, in: Proceedings of the 19th International Symposium on Distributed Computing

(DISC'2005), Krakow, Poland, in: LNCS, vol. 3724, Springer-Verlag, September 2005, pp. 199–213.

[21] C. Martín, M. Larrea, 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.

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

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

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

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

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

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

[28] A. Mostéfaoui, M. Raynal, C. Travers, 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.

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

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