2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing

An Evaluation of Efficient Leader Election Algorithms for Crash-Recovery Systems

Carlos Gómez-Calzado, Mikel Larrea, Iratxe Soraluze, Alberto Lafuente, Roberto Cortiñas
Computer Architecture and Technology Department
University of the Basque Country UPV/EHU
20018 San Sebastián, Spain
{*carlos.gomez, mikel.larrea, iratxe.soraluze, alberto.lafuente, roberto.cortinas*}*@ehu.es*

***Abstract*—This paper presents an evaluation of three communication-efficient algorithms implementing the Omega class of failure detectors, which provides an eventual leader election functionality, in distributed systems where processes can crash and recover. Communication efficiency means that eventually only a correct process, i.e., the elected leader, keeps sending a message periodically to the rest of processes. The first algorithm relies on the use of stable storage to store the identity of the leader and an incarnation number. The second algorithm does not use stable storage, but requires a majority of correct processes. Also, it is *near*-communication-efficient, since besides the leader, unstable processes, i.e., those that crash and recover infinitely often, may send messages periodically before they receive a message from the leader. Finally, the third algorithm does neither use stable storage nor require a majority of correct processes, but assumes that each process has access to a nondecreasing and persistent local clock. Using the OMNeT++ network simulation framework, we evaluate the performance and the quality of service provided by these algorithms, in terms of the number of messages exchanged among processes and the capability of the failure detector to provide a single leader, respectively.**

***Keywords*-Omega failure detector; leader election; crash-recovery; communication efficiency;**

I. Introduction

It is well known that consensus [37], a fundamental problem in fault-tolerant distributed computing, cannot be solved deterministically in asynchronous systems prone to even a single process crash [17]. As a way to circumvent this impossibility result, Chandra and Toueg proposed in [7] the use of unreliable failure detectors, which provide (possibly incorrect) information about process failures. “Unreliable” means that, at a given instant, the information provided by most of the failure detectors proposed by Chandra and Toueg can be incorrect. However, eventually all of them provide enough information to solve consensus. Indeed, besides offering an elegant abstraction for solving consensus in a modular way (i.e., first implement a given failure detector and then implement a consensus protocol on top of it), the model of failure detectors allows to study the minimum properties (also known as “weakest failure detector”) that must be satisfied in order to solve consensus or other agreement problems like atomic commitment [10], [18] or, more recently, set-agreement [11], [12].

In this work, we focus on a failure detector class called Omega [6] in crash-recovery distributed systems. Informally, Omega provides an eventual leader election functionality, i.e., eventually all processes agree on a common and correct process. The importance of Omega comes from the fact that it has been proved to be the weakest failure detector for solving consensus [6]. In this regard, several consensus algorithms based on such a weak leader election mechanism have been proposed [19], [21], [23], [34].

A number of distributed algorithms implementing Omega in the more benign crash model, in which crashed processes do not recover, have been proposed [2], [3], [4], [5], [9], [14], [15], [16], [20], [22], [26], [31], [32], [33], [35], [36]. They basically differ in aspects like the assumptions on link reliability and synchrony, the communication pattern among processes, and the initial knowledge or not of the membership. In some of these algorithms, eventually only the elected leader keeps sending messages periodically to the rest of processes. Such an algorithm is said communication-efficient [3].

Failure detectors have also been studied in the crash-recovery failure model, i.e., in which crashed processes can recover (even infinitely often). Aguilera et al. define in [1] 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 [7], [13]. More recently, [29], [30] propose several algorithms implementing Omega in the crash-recovery failure model that rely on the use of stable storage to keep the identity of the current leader and the value of an incarnation number associated with each process. Alternatively, [27] proposes 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 periodically. Consequently, the cost of these algorithms in terms of the number of messages exchanged is high. It would be interesting to have efficient algorithms for Omega in which eventually only one process, i.e., the elected leader, sends a message periodically to the rest of processes. In this regard, three efficient Omega algorithms for crash-recovery systems have been proposed very recently [24], [25], [28]. In the line of the previously proposed algorithms

1066-6192/12 $26.00 © 2012 IEEE
DOI 10.1109/PDP.2013.33

that were not communication-efficient, the first algorithm relies on the use of stable storage, and the second algorithm requires a majority of correct processes. Concerning the third algorithm, it does neither use stable storage nor require a majority of correct processes, but alternatively assumes that each process is equipped with a nondecreasing local clock that keeps running despite the crash of the process. Regarding the degree of efficiency of these three algorithms, two of them are fully communication-efficient, i.e., eventually only one process (the elected leader) keeps sending a message periodically to the rest of processes. The other algorithm is *near*-communication-efficient, since besides the leader, unstable processes, i.e., those that crash and recover infinitely often, may send messages periodically before they receive a message from the leader. To the best of our knowledge, no other communication-efficient Omega algorithm for crash-recovery systems has been proposed.

Note that depending on the use or not of stable storage, the property that the algorithms satisfy regarding unstable processes 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 the leader itself the identity of the leader upon recovery. The algorithms make processes to be aware of being in this learning period by outputting a special "no-leader" value upon recovery.

In this paper, we present an evaluation of the mentioned three efficient Omega algorithms. Using the OMNeT++ network simulation framework (http://www.omnetpp.org/), we first evaluate the performance of the algorithms in terms of the number of messages exchanged among processes. Then, we evaluate the quality of service (QoS) provided by the algorithms [8], measured as the time percentage during which the failure detector provides a single leader to the upper layer, e.g., a leader-based consensus protocol. The evaluation shows that all the three algorithms perform reasonably well, although the third algorithm gives the best results due to its faster convergence on a single leader in the system. On the other side, the second algorithm has a higher message complexity than the other two.

The rest of the paper is organized as follows. In Section II, we describe the system model and the three specific crash-recovery systems S_1, S_2 and S_3 considered in this work. In Section III, we present the three efficient algorithms implementing Omega in systems S_1, S_2 and S_3 respectively. In Section IV, we evaluate by simulation the performance and QoS of the algorithms. Finally, Section V concludes the paper.

II. System Model

We consider a distributed 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 crash and may subsequently recover. In every execution of the system, Π is composed of the following three disjoint subsets:

- *Eventually up*, i.e., processes that eventually remain up forever.
- *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. By default, 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:

- *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).

Specific Crash-Recovery Systems S_1, S_2 and S_3

We consider the following three specific crash-recovery systems, denoted S_1, S_2 and S_3 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.

- 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.
  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.
- Finally, system S_3 does neither assume stable storage nor a majority of correct processes. Alternatively, it assumes that local clocks of processes are nondecreasing and continue running despite the crash of processes. Regarding communication reliability and synchrony, S_3 satisfies the same assumption as S_1:
  i) For every correct process p, there is an eventually timely link from p to every correct and every unstable process.

The Omega Failure Detector Class

Chandra et al. defined in [6] a failure detector class for the crash failure model called Omega, and proved that it is the weakest failure detector for solving consensus. 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. 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 [27], [29]:

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.

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.

Communication Efficiency Definitions

We present now the concepts of communication-efficient and near-communication-efficient implementations of the Omega failure detector class in crash-recovery systems [25]:

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.

III. The Algorithms

In this section, we present the three efficient Omega algorithms object of our study, namely a communication-efficient Omega (Property 1) algorithm for system S_1, a near-communication-efficient Omega (Property 2) algorithm for system S_2, and a communication-efficient Omega (Property 2) algorithm for system S_3 respectively. Detailed descriptions and the formal correctness proofs of the algorithms can be found in [25], [28].

A Communication-Efficient Omega for System S_1

Figure 1 presents the first Omega algorithm, which assumes that processes have access to stable storage. 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).

```
Every process p executes the following:

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 $\eta$ 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
    $Timeout_p[leader_p] \leftarrow Timeout_p[leader_p] + 1$
    $leader_p \leftarrow p$
```

Figure 1. Communication-efficient Omega algorithm for system S_1.

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: 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 $q \leq leader_p$. 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.

A Near-Communication-Efficient Omega for System S_2

Figure 2 presents the second Omega algorithm, which does not use stable storage but 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

```
Every process p executes the following:

Initialization:
  $leader_p \leftarrow \bot$
  for all $q \in \Pi$ except $p$:
    $Timeout_p[q] \leftarrow \eta$
    $Recovered_p[q] \leftarrow 0$
  $Recovered_p[p] \leftarrow 1$
  send (RECOVERED, $p$) to all processes except $p$
  start tasks 1, 2 and 3

Task 1:
  repeat forever every $\eta$ time units
    if $leader_p = p$ then
      send (LEADER, $p$, $Recovered_p$) to all processes except $p$
    else if $leader_p = \bot$ 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
      $Recovered_p[q] \leftarrow Recovered_p[q] + 1$
    else if message is of type ALIVE then
      if ($leader_p = \bot$) and ($p$ has received so far ALIVE from
                                   $\lfloor n/2 \rfloor$ different processes) then
        $leader_p \leftarrow p$
      end if
    else if message is of type LEADER then
      for all $r \in \Pi$:
        $Recovered_p[r] \leftarrow \max(Recovered_p[r], Recovered_q[r])$
      $Timeout_p[q] \leftarrow \max(Timeout_p[q], Recovered_p[p])$
      if (($leader_p = \bot$) and
          ($\langle Recovered_p[q], q\rangle < \langle Recovered_p[p], p\rangle$)) or
         (($leader_p \neq \bot$) and
          ($\langle Recovered_p[q], q\rangle \leq \langle Recovered_p[leader_p], leader_p\rangle$))
      then
        $leader_p \leftarrow q$
        reset $timer_p$ to $Timeout_p[q]$
      end if
      if ($leader_p = \bot$) or
         ($\langle Recovered_p[p], p\rangle < \langle Recovered_p[leader_p], leader_p\rangle$)
      then
        $leader_p \leftarrow p$
        stop $timer_p$
      end if
    end if

Task 3:
  upon expiration of $timer_p$ do
    $Timeout_p[leader_p] \leftarrow Timeout_p[leader_p] + 1$
    $leader_p \leftarrow \bot$
    empty the set of ALIVE messages received so far
```

Figure 2. Near-communication-efficient Omega algorithm for system S_2.

being $\bot$ 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 $\bot$ and empties the set of *ALIVE* messages received so far. Observe that resetting $leader_p$ to $\bot$ leads p to start sending again *ALIVE* messages periodically by Task 1.

A Communication-Efficient Omega for System S_3

Figure 3 presents the third Omega algorithm, which does neither use stable storage nor require a majority of correct processes, but assumes that processes are equipped with nondecreasing and persistent local clocks. As in the second algorithm, $leader_p$ is initialized to the special value $\bot$, 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.

```
Every process p executes the following:

Initialization:
  $leader_p \leftarrow \bot$
  $Timeout_p \leftarrow clock()$
  $ts_p \leftarrow clock()$
  $ts_{min} \leftarrow ts_p$
  start tasks 1, 2 and 3

Task 1:
  wait ($Timeout_p$) time units
  if $leader_p = \bot$ then
    $leader_p \leftarrow p$
  else
    reset $timer_p$ to $Timeout_p$
  end if
  repeat forever every $\eta$ 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 ($leader_p = \bot$) and ($q < p$)]
       or [($ts_q = ts_{min}$) and ($leader_p \neq \bot$) and ($q \leq leader_p$)] then
      $leader_p \leftarrow q$
      $ts_{min} \leftarrow ts_q$
      reset $timer_p$ to $Timeout_p$
    end if

Task 3:
  upon expiration of $timer_p$ do
    $Timeout_p \leftarrow Timeout_p + 1$
    $leader_p \leftarrow p$
    $ts_{min} \leftarrow ts_p$
```

Figure 3. Communication-efficient Omega algorithm for system S_3.

The algorithm works as follows. In Task 1, p first waits $Timeout_p$ time units, after which if p still has no leader, i.e., $leader_p = \bot$, then p sets $leader_p$ to p. Otherwise, p resets $timer_p$ to $Timeout_p$ 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. 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 $(leader_p = \perp)$ and $(q < p)$, i.e., p has no leader yet and q is a good candidate, or (3) $(ts_q = ts_{min})$ and $(leader_p \neq \perp)$ and $(q \leq leader_p)$, 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.

IV. Evaluation

In this section, we present an evaluation by simulation of the three efficient Omega algorithms presented in the previous section. For that, we have used the OMNeT++ network simulation framework (http://www.omnetpp.org/). We first evaluate the performance of the algorithms in terms of the number of messages exchanged among processes. Then, we evaluate the quality of service provided by the algorithms, measured as the time percentage during which the failure detector provides a single leader to the upper layer, e.g., a leader-based consensus protocol.

We have evaluated the algorithms in the following three scenarios:

- Small: composed of 5 processes, out of which 3 are eventually up, 1 is eventually down, and 1 is unstable. The eventually up processes crash and recover 3, 0 and 1 times respectively, while the eventually down process does not recover after its 4th crash.
- Medium: composed of 10 processes, out of which 6 are eventually up, 1 is eventually down, and 3 are unstable. The eventually up processes crash and recover 3, 3, 0, 1, 4 and 1 times respectively, while the eventually down process does not recover after its 5th crash.
- Large: composed of 20 processes, out of which 11 are eventually up, 2 are eventually down, and 7 are unstable. The eventually up processes crash and recover 4, 4, 1, 4, 5, 0, 4, 5, 3, 5 and 3 times respectively, while the 2 eventually down processes do not recover after their 5th crash.

For each scenario, five executions have been simulated. In order to make a fair comparison of the results, in all the executions processes crash and recover at the same time instants, which are randomly calculated in advance (with the last crash and/or recovery over half of the execution duration). Nevertheless, the use of uniformly distributed message delays gives us different executions in terms of timeout expirations and leader changes. The duration of the executions has been of 4000 seconds in all cases (for evaluating the quality of service, additional executions of 8000 and 12000 seconds have been simulated too). The value chosen for η, i.e., the periodicity for sending heartbeat messages by the elected leader, has been 20 seconds. These values have been empirically proved to be sufficient for comparative purposes.

Performance

Figure 4 presents the average number of messages sent per link by each algorithm (note that the number of links for each scenario is $n(n-1)$, i.e., 20, 90 and 380 for small, medium and large, respectively). It can be observed that in all cases the number decreases for larger scenarios, which is clearly due to the fact that the algorithms are communication-efficient, i.e., eventually only the elected leader keeps sending messages periodically, and hence many links are not used any more by the algorithms. Note also that the second algorithm uses more messages than the rest, due to the fact that it has three types of messages (*RECOVERED*, *ALIVE* and *LEADER*), and that the third algorithm uses less messages than the first, which is due to the fact that it has longer waiting times upon recovery (since it uses the local clock instead of a counter stored in stable storage).

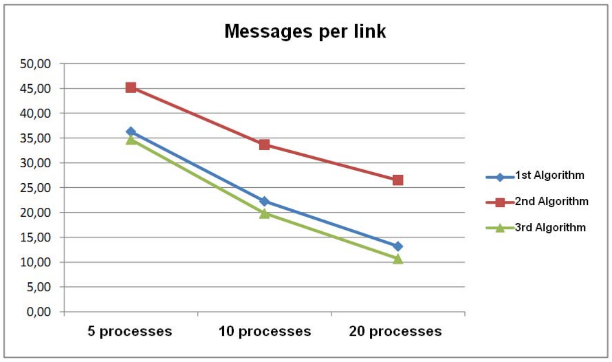

Figure 4. Average number of messages sent per link.

Table I presents the total number of messages sent.

	1st algorithm (Figure 1)	2nd algorithm (Figure 2)	3rd algorithm (Figure 3)
5 proc.	725 messages	904 messages	694 messages
10 proc.	2002 messages	3030 messages	1784 messages
20 proc.	5008 messages	10078 messages	4065 messages

Table I
Total number of messages sent.

Figure 5 presents the average number of messages of each type (*RECOVERED*, *ALIVE* and *LEADER*) sent per link by the second algorithm. Note that the number of *RECOVERED* and *ALIVE* messages per link grows very smoothly for larger scenarios, while the number of *LEADER* messages per link logically decreases due to the communication efficiency of the algorithm.

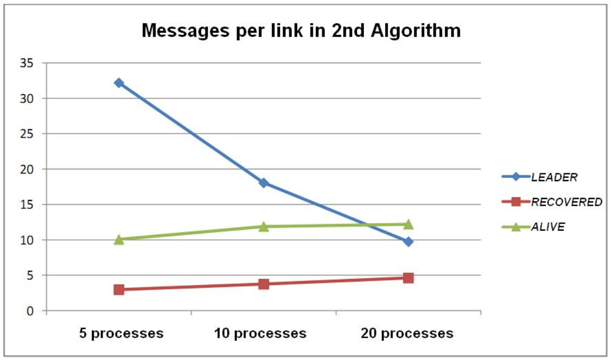

Figure 5. Average number of messages of each type sent per link by the second algorithm.

	RECOVERED	*ALIVE*	*LEADER*
5 proc.	59 messages	201 messages	644 messages
10 proc.	338 messages	1069 messages	1624 messages
20 proc.	1744 messages	4639 messages	3696 messages

Table II
TOTAL NUMBER OF MESSAGES OF EACH TYPE SENT BY THE SECOND ALGORITHM.

Table II presents the total number of messages of each type sent by the second algorithm.

Finally, Figure 6 presents the minimum, maximum and average number of *ALIVE* messages sent in the second algorithm by a process while it does not have a leader yet, i.e., its leader is set to ⊥. Observe that, for a large scenario, the value ranges from a minimum value of 6 *ALIVE* messages to a maximum value of 52 messages, with an average value of 15 messages. Also, the maximum values for small and medium scenarios are 9 and 20 messages, respectively.

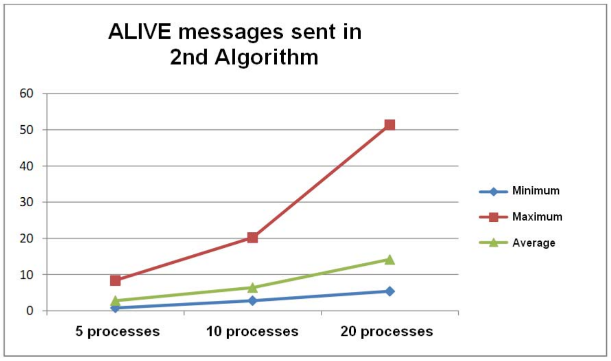

Figure 6. Number of *ALIVE* messages sent in the second algorithm.

Quality of Service

Figure 7 presents the time percentage during which the failure detector has provided a single leader, for executions lasting 4000 seconds. Additionally, Table III presents the percentages for executions lasting 8000 and 12000 seconds respectively. In general, the three algorithms exhibit good quality of service, although it can be observed that the 3rd algorithm gives the best result, due to the fact that it converges faster than the other two towards a single leader.

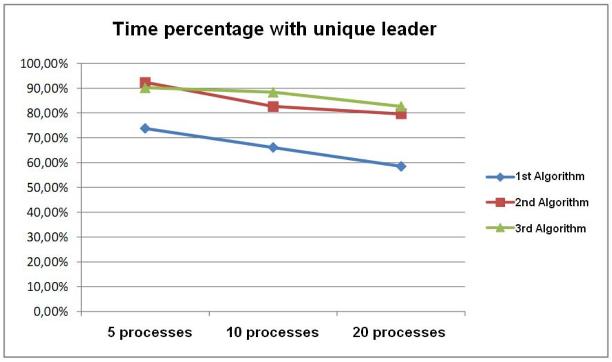

Figure 7. Time percentage during which the failure detector provides a single leader.

As a complementary result, Figure 8 presents the average number of simultaneous leaders provided by the algorithms (whenever they do not provide a single leader), which gives an idea of their degree of divergence. Note that only non-⊥ values are considered in the second and third algorithms. Logically, for larger scenarios the average number of simultaneous leaders increases. The bigger value for the third algorithm is due to the fact that it is very sensitive to the growth of the local clock, which grows faster than the counter used in the first algorithm. As a consequence, the third algorithm sends less messages and has less leader changes, which involves a bigger number of simultaneous leaders.

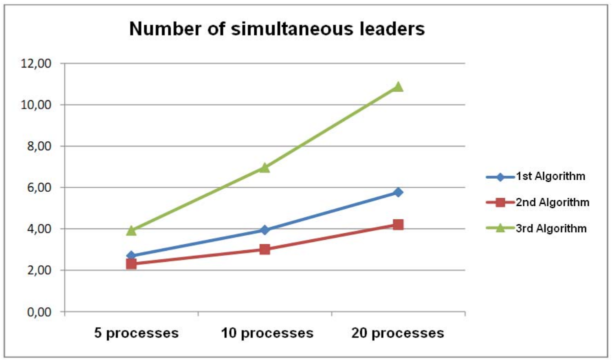

Figure 8. Average number of simultaneous leaders.

Finally, Figure 9 presents an example of execution history for the small scenario composed of 5 processes. It can be observed how the third algorithm converges faster than the

	8000 seconds			12000 seconds		
	5 processes	10 processes	20 processes	5 processes	10 processes	20 processes
1st algorithm (Figure 1)	86,44%	81,79%	79,06%	90,04%	89,10%	85,70%
2nd algorithm (Figure 2)	94,13%	92,63%	91,19%	95,05%	94,22%	90,62%
3rd algorithm (Figure 3)	94,86%	94,33%	91,33%	96,58%	96,22%	94,21%

Table III
TIME PERCENTAGE DURING WHICH THE FAILURE DETECTOR PROVIDES A SINGLE LEADER.

other two, among which the second converges faster than the first. Note also that the third algorithm has a bigger average number of simultaneous leaders when there is no single leader.

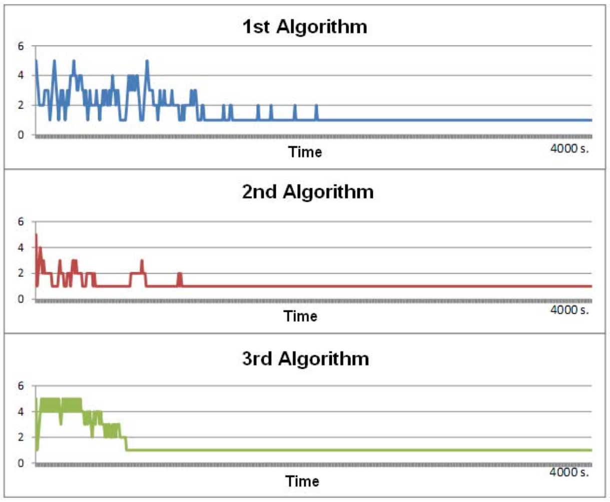

Figure 9. Example of execution history for the small scenario. The vertical axis stands for the number of concurrent leaders.

V. CONCLUSION

In this paper, we have evaluated by simulation both the performance and the quality of service of three efficient algorithms implementing the Omega failure detector class in crash-recovery systems. All the algorithms avoid the disagreement among unstable processes and correct processes, although depending on the use or not of stable storage, the property satisfied by unstable processes varies. The first algorithm, which is communication-efficient, uses stable storage. The second algorithm, which is near-communication-efficient, assumes a majority of correct processes. The third algorithm, which is communication-efficient, assumes that processes are equipped with a monotonically increasing local clock that continues running upon process crashes. In the second and third Omega algorithms, 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 ⊥ value upon recovery, and then agree with correct processes on the leader after receiving a first message from it.

The evaluation shows that all the three algorithms perform reasonably well, although the third algorithm gives the best results due to its faster convergence towards a single leader in the system. On the other side, the second algorithm uses more messages than the rest. All in all, one has to consider carefully the requirement of each algorithm before choosing one of them as a candidate for a real deployment, namely (1) the use of stable storage, (2) a majority of correct processes, and (3) the use of a nondecreasing and persistent local clock.

The experiments carried out so far, even being rather limited and specific (i.e., number of processes of each type, number and instants of crashes and recoveries of processes. . .), have allowed us to compare the behavior of the three algorithms under the same conditions. Nevertheless, in order to better understand the absolute values obtained, we plan to carry out more exhaustive experiments. Additionally, we envisage the realization of experiments in a real scenario such as PlanetLab (http://www.planet-lab.org/), which will allow to compare the results obtained by simulation with those of a real deployment.

ACKNOWLEDGMENTS

Research partially supported by the Spanish Research Council, under grant TIN2010-17170, the Basque Government, grants IT395-10, S-PE11UN099 and S-PE12UN109, and the University of the Basque Country UPV/EHU, under grant UFI11/45. Also, Carlos Gómez-Calzado is recipient of a doctoral fellowship from the Basque Government.

REFERENCES

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

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

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

[4] M. Aguilera, C. Delporte-Gallet, H. Fauconnier, and 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)*, pages 328–337, St. John's, Newfoundland, Canada, July 2004.

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

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

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

[8] W. Chen, S. Toueg, and M. K. Aguilera. On the quality of service of failure detectors. *IEEE Transactions on Computers*, 51(5):561–580, 2002.

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

[10] C. Delporte-Gallet, H. Fauconnier, R. Guerraoui, V. Hadzilacos, P. Kouznetsov, and S. Toueg. The weakest failure detectors to solve certain fundamental problems in distributed computing. In *Proceedings of the 23rd ACM Symposium on Principles of Distributed Computing (PODC'2004)*, pages 338–346, St. John's, Newfoundland, Canada, July 2004.

[11] C. Delporte-Gallet, H. Fauconnier, R. Guerraoui, and A. Tielmann. The Weakest Failure Detector for Message Passing Set-Agreement. In *Proceedings of the 22nd International Symposium on Distributed Computing (DISC'2008)*, pages 109–120, Arcachon, France, September 2008. LNCS 5218, Springer-Verlag.

[12] C. Delporte-Gallet, H. Fauconnier, R. Guerraoui, and A. Tielmann. The disagreement power of an adversary. *Distributed Computing*, 24(3-4):137–147, 2011.

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

[14] A. Fernández, E. Jiménez, and 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)*, pages 63–72, University of California, Riverside, USA, December 2006.

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

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

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

[18] R. Guerraoui, M. Larrea, and A. Schiper. Non-Blocking Atomic Commitment with an Unreliable Failure Detector. In *Proceedings of the 14th IEEE Symposium on Reliable Distributed Systems (SRDS'1995)*, pages 41–50, Bad Neuenahr, Germany, September 1995.

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

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

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

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

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

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

[25] M. Larrea, C. Martín, and I. Soraluze. Communication-efficient leader election in crash-recovery systems. *Journal of Systems and Software*, 84(12):2186–2195, December 2011.

[26] D. Malkhi, F. Oprea, and 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)*, pages 199–213, Krakow, Poland, September 2005. LNCS 3724, Springer-Verlag.

[27] C. Martín and 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)*, pages 208–215, Taipei, Taiwan, December 2008.

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

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

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

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

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

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

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

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

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

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