Eventual election of multiple leaders for solving consensus in anonymous systems

Ernesto Jiménez[1,2] · Sergio Arévalo[1] · Carlos Herrera[3] · Jian Tang[1]

Abstract In classical distributed systems, each process has a unique identity. Today, new distributed systems have emerged where a unique identity is not always possible to be assigned to each process. For example, in many sensor networks a unique identity is not possible to be included in each device due to its small storage capacity, reduced computational power, or the huge number of devices to be identified. In these cases, we have to work with anonymous distributed systems where processes cannot be identified. Consensus cannot be solved in classical and anonymous asynchronous distributed systems where processes can crash. To bypass this impossibility result, failure detectors are added to these systems. It is known that Ω is the weakest failure detector class for solving consensus in classical asynchronous systems when a majority of processes never crashes. Although $A\Omega$ was introduced as an anonymous version of

[1] Universidad Politécnica de Madrid, Madrid, Spain

[2] Prometeo, Quito, Ecuador

[3] Escuela Poliécnica Nacional, Quito, Ecuador

Ω, to find the weakest failure detector in anonymous systems to solve consensus when a majority of processes never crashes is nowadays an open question. Furthermore, $A\Omega$ has the important drawback that it is not implementable. Very recently, $A\Omega'$ has been introduced as a counterpart of Ω for anonymous systems. In this paper, we show that the $A\Omega'$ failure detector class is strictly weaker than $A\Omega$ (i.e., $A\Omega'$ provides less information about process crashes than $A\Omega$). We also present in this paper the first implementation of $A\Omega'$ (hence, we also show that $A\Omega'$ is implementable), and, finally, we include the first implementation of consensus in anonymous asynchronous systems augmented with $A\Omega'$ and where a majority of processes does not crash.

Keywords Failure detectors · Consensus · Anonymity · Fault tolerance · Anonymous omega

1 Introduction

From a theoretical and practical point of view, we are accustomed to define and use distributed systems where each process has a unique identity (we can call it classical distributed systems). However, new distributed systems have emerged where a unique identity is not always possible to be assigned to each process. For example, in many sensor networks, a unique identity is not possible to be included in each device due to its small storage capacity, reduced computational power, or the huge number of devices to be identified [1,18]. In all these cases, we have to work with distributed systems where processes have no identity. Hence, we can use anonymous systems where processes are not identifiable because all of them are coded identically (i.e., processes have no identity, and there is no way to distinguish among them).

Another important context where anonymity is very important is when the users' privacy is involved [16]. In this case, identification is not possible without something that breaks the symmetry.

On the other hand, one of the most important coordination problems in distributed computing is consensus [13]. The consensus problem says that in a system where a set of values are proposed, only one of them can be decided. Consensus cannot be solved in anonymous (and classical) asynchronous systems when even one process may crash [20]. To bypass this impossibility result, failure detectors are added to these anonymous asynchronous systems [6,8].

A failure detector is a distributed device that provides information about process crashes [13]. It is well known that Ω is the class of failure detectors that provides the minimum information about process crashes (i.e., it is the weakest failure detector) for solving consensus in classical asynchronous systems when a majority of processes never crashes [12]. $A\Omega$ was introduced as an anonymous version of Ω [6].[1] Roughly speaking, $A\Omega$ states that eventually only a single process identifies itself as the leader of all non-crashed processes. Nevertheless, to find the weakest failure detector class to achieve consensus in anonymous systems when a majority of processes never crashes is still an open question [8]. Furthermore, $A\Omega$ has the important drawback that it is

[1] $A\Omega$ was first proposed by Bonnet and Raynal in the preliminary DISC 2010 conference paper [7].

not implementable in anonymous systems [6]. Hence, any algorithm that implements consensus with $A\Omega$ is not implementable.

Very recently, $A\Omega'$ has been introduced as a new counterpart of Ω for anonymous systems [10]. Roughly speaking, $A\Omega'$ states that eventually a set L of non-crashed processes will permanently identify themselves as leaders, and all these leader processes eventually will know the size of L (i.e., $|L|$).

Related work One of our main goals in this paper is to present the first implementation of the $A\Omega'$ failure detector [10]. This failure detector is important because it is weaker than other classes of failure detectors [5,6]. In [6], the anonymous classes AP, $A\Omega$ and $A\Sigma$ are introduced. They are the anonymous counterparts of the classes of perfect failure detector P [13], eventual leader failure detector Ω [12], and quorum failure detectors Σ [15], respectively. In the paper [11], another slightly different anonymous version of Σ denoted $A\Sigma'$ is introduced. In [5], the authors present the failure detector $\overline{AP}$ which is the anonymous counterpart of the perfect failure detector P when the membership of the system is unknown. With respect to the implementability, $A\Omega$ has the drawback that is not implementable even in anonymous synchronous systems [6]. If the membership is unknown, $\overline{AP}$ is not implementable either (applying similar techniques than in [22]).

In [19], a distributed model where the system is a collection of anonymous finite-state agents is presented. A protocol is self-stable if it does not require initialization to work, and it is always able to recover from temporary failures. In that paper [19], it is shown that self-stabilizing eventual leader election is impossible to achieve in such systems. To circumvent this result, they enrich the system with the failure detector $\Omega?$ When an agent invokes $\Omega?$ this failure detector returns the information of whether or not one or more processes are working as leaders. The information returned by $\Omega?$ may be incorrect by a finite period of time, but eventually $\Omega?$ will always provide accurate information. The authors show in [19] that in this system augmented with $\Omega?$ it is possible to achieve self-stabilizing eventual leader election in rings and complete graphs.

Failure detectors are important because they can help to solve important problems in distributed computing. One of the most important problems is consensus [13]. Consensus in anonymous systems is introduced for first time in [5]. In it, the authors solve consensus with a majority of processes that never crashes and using the failure detector $\overline{AP}$. They show that $2t + 1$ is the lower bound on the number of rounds to achieve consensus (t is the maximum number of crashed processes, and all processes must know this value of t). In [17], consensus in anonymous systems with different synchrony assumptions is also solved (that is, they assume that the system is not totally asynchronous but with partial synchrony). In the technical report [11], an algorithm is presented using the failure detector $\langle A\Omega', A\Sigma' \rangle$ to solve consensus in anonymous systems where all processes are interconnected using FIFO reliable links (hence, their anonymous system is stronger than the system we present in this paper). Nevertheless, their solution allows to solve consensus even if a majority of processes crashes.

Not only in anonymous message-passing systems the consensus problem is solved, but also several solutions are presented in the literature to achieve consensus in anonymous shared memory systems [3,4,9,14]. In all of them, consensus is implemented in an anonymous shared memory system bounding the step complexity (i.e., the number of shared memory accesses) to $O(n)$ by each invocation on a read/write operation, being n the number of processes of the system. In [9] and [14], the failure detector $A\Omega$ [6] is used to solve consensus and there is no bounds in the number of crashed processes. In [9], the shared memory is formed by atomic multi-writer and multi-reader registers, and in [14], these shared memory is made up by the weak set object (this object is a set from which values are never removed). In [4], the authors implement consensus in an anonymous shared memory where no processes can crash and where the shared memory is implemented using atomic registers (namely, $\Omega(\log n)$ is the number of atomic registers needed to solve consensus). In [2] and [3], the anonymous shared memory is built by objects denoted *adopt-commit* [21]. In [2], consensus is solved for the probabilistic-write model. The algorithm presented is formed by an adopt-commit object to detect agreement, and by a conciliator object to guarantee the agreement not deterministically but with some probability. In [2], the step complexity is $O(\log m)$, being m the different values that processes can propose. In [3], the solution is improved to $O(n)$.

Our contribution In this paper, we show that $A\Omega'$ is strictly weaker than $A\Omega$ (i.e., $A\Omega'$ provides less information about process crashes than $A\Omega$). We also present the first implementation in the literature of $A\Omega'$ (hence, we also show that $A\Omega'$ is implementable). It is worth noting that this implementation is communication efficient (i.e., eventually only leader processes send messages). Finally, it is included in this paper the first implementation of consensus in anonymous asynchronous systems enriched with $A\Omega'$ and where a majority of processes does not crash. Therefore, we also show in this paper that consensus with this new and weaker version of Ω for anonymous systems $A\Omega'$ is also implementable.

This paper is organized as follows. The model of the anonymous distributed system is presented in Sect. 2. The failure detector $A\Omega'$ is presented in Sect. 3, and consensus using $A\Omega'$ can be found in Sect. 4. It is noteworthy that in Sect. 3.2, we prove that $A\Omega'$ is weaker than the traditional definition of anonymous omega failure detector $A\Omega$. Finally, we present the conclusions in Sect. 5.

2 Anonymous system *AS*

AS is a message-passing system formed by a finite set $\Pi = \{p_i\}_{i=1,\ldots,n}$ of n processes fully interconnected by links. Each process $p_i \in \Pi$ uses the primitive *broadcast* to send a message to every process $p_j \in \Pi$. This primitive, denoted by $broadcast(m)$, sends a copy of message m through each link.

Processes are executed by taking steps. A process crashes when it stops taking steps. We assume that crashes are permanent. We say that process p_i is *correct* in a run if it does not crash, and *faulty* if p_i crashes. We denote by *Correct* the set of correct processes, and by *Faulty* the set of faulty processes. We denote by f the maximum number of processes that may crash in a run. We consider that if some process p_i crashes while the primitive $broadcast(m)$ is invoked by p_i, a copy of the message m can be delivered to any unknown subset of processes (including the empty subset).

For analysis, we assume that time advances at discrete steps. We also assume a global clock whose values are the positive natural numbers, but processes cannot access it. We use the notation $\tau \in N$ to indicate an instant of time.

Processes are *anonymous* [6]. Then, processes have no identity, and there is no way to differentiate between any two processes of the system (i.e., processes have no identifier and execute the same code).

A failure detector *FD* is a distributed device with a local module FD_i for each process $p_i \in \Pi$. A failure detector *FD* returns information related with faulty processes each time that a process p_i invokes its module FD_i. The addition of a failure detector *FD* in a system S (denoted by $S[FD]$) allows to solve a certain problem P that it is impossible to overcome in S alone. According to the type and the quality of the information about crashed processes, several classes of failure detectors have been proposed [23,24].

3 $A\Omega'$ failure detector class

We introduce in this section the algorithm $\mathcal{A}_{A\Omega'}$ to implement the failure detector $A\Omega'$ in anonymous partially synchronous systems (see Fig. 1). This algorithm has a nice property: communication efficiency. That is, in every run, there is a time after which only leader processes broadcast messages.

3.1 Definition of $A\Omega'$

The $A\Omega'$ [10] failure detector provides each process $p_i \in \Pi$ with two output variables $leader_i$ and $quantity_i$. Let L (resp., NL) be the subset of correct processes such that eventually their variable $leader = true$ (resp., $leader = false$) permanently. We say that a correct process p_i is an *eventually leader process* (for shorten, a *leader*) if $p_i \in L$, and an *eventually non-leader process* (for shorten, a *non-leader*) if $p_i \in NL$. A failure detector of class $A\Omega'$ [10] satisfies that:

1. Every correct process is either an eventually leader process, or an eventually non-leader process.
2. There is at least one eventually leader process in the system.
3. There is a time after which every eventually leader process p_i has $quantity_i = |L|$ permanently, being L the set of eventually leader processes in the system.

More formally, the definition of $A\Omega'$ is the following. Let $leader_i^{\tau}$ and $quantity_i^{\tau}$ be the variables $leader_i$ and $quantity_i$ provided by $A\Omega'$ at time τ. Let $L = \{p_i \in Correct\colon \exists\tau\colon \forall\tau' \geq \tau,\ leader_i^{\tau'} = true\}$, and $NL = \{p_i \in Correct\colon \exists\tau : \forall\tau' \geq \tau,\ leader_i^{\tau'} = false\}$. In each run R of the system, any failure detector of class $A\Omega'$ must satisfy the following three properties:

1. $(L \cup NL = Correct) \wedge (L \cap NL = \emptyset)$.
2. $L \neq \emptyset$.
3. $\exists\tau : \forall\tau' \geq \tau,\ \forall p_i \in L,\ quantity_i^{\tau'} = |L|$.

Note that there is not a time after which a correct process $p_k \in NL$ must have in $quantity_k$ the number of leaders $|L|$ of the system.

3.2 $A\Omega'$ is strictly weaker than $A\Omega$

First, we define $A\Omega$ [6]. Let us consider that each process $p_i \in \Pi$ has a boolean variable l_i. Every failure detector of class $A\Omega$ satisfies that eventually: (1) there is a correct process p_l that has $l_l = true$ permanently, and (2) every correct process p_j other than p_l has $l_j = false$ permanently. More formally, $\exists\tau$, $\exists p_l \in Correct$: $\forall\tau' \geq \tau$, $\forall p_j \neq p_l \in Correct$, $l_l^{\tau'} = true$ and $l_j^{\tau'} = false$.

A failure detector class X is *strictly weaker* than class Y in system S if (a) there is an algorithm that emulates the output of a failure detector D' of class X in the system S augmented with a failure detector D of class Y (denoted by $S[D]$), and (b) the opposite is not true (i.e., there is no algorithm that emulates the output of a failure detector D' of class Y in the system S augmented with a failure detector D of class X).

Then, we now prove that $A\Omega'$ is strictly weaker than $A\Omega$ with the following two cases.

Lemma 1 *Class $A\Omega'$ can be obtained from $AS[A\Omega]$.*

Proof Let D be any failure detector of class $A\Omega$. Let D' be an emulated failure detector with the following algorithm. Each process p_i sets $D'.quantity_i = 1$, and permanently updates $D'.leader_i$ with the value of $D.l_i$.

From definition of $A\Omega$, eventually a single correct process p_l has $D.leader_l = true$ permanently, and every correct process p_j other than p_l has $D.leader_j = false$ permanently. Hence, p_l belongs to L, and the rest of correct processes belong to NL (Condition 1 of $A\Omega'$). Then, $|L| = 1$, and, hence, $L \neq \emptyset$ (Condition 2 of $A\Omega'$). Finally, process p_l has $D'.quantity_l = |L| = 1$ permanently (Condition 3 of $A\Omega'$). Therefore, D' is a failure detector of class $A\Omega'$. □

Lemma 2 *Class $A\Omega$ cannot be obtained from $AS[A\Omega']$.*

Proof Let D be a failure detector of class $A\Omega'$ with a run R where the following six points are preserved: (1) the number of processes is greater than one, $|\Pi| > 1$, (2) all processes are correct, $Correct = \Pi$, and all of them are leaders, $L = Correct$, (3) from the beginning of the run, $D.leader_i = true$ and $D.quantity_i = |Correct|$ permanently in each process p_i [note that this is one of the possible outputs of $A\Omega'$ by previous points (1) and (2)], (4) all processes execute in R the same deterministic code at the same speed in lock step, broadcasting each message m at the same time, (5) the delay of m is the same in every link, and, hence, m will be received by every process in the same step of the execution, (6) if two messages m and m' are received in the same step, both messages will be delivered in the same order in every process.

Let us assume, by the way of contradiction, that $A\Omega$ can be deterministically obtained from $A\Omega'$ in all runs. Then, we construct a run R as described above. Then, because the six points of R and because processes have no identity, there is no way to distinguish among all correct processes in R deterministically, and it is impossible to break this symmetry. Thus, every process p_i either outputs $D'.l_i = true$ or $D'.l_i = false$ in R. Therefore, it is impossible to output $D'.l_l = true$ in a single correct process p_l, and $D'.l_j = false$ in every correct process p_j other than p_l in all executions (which contradicts the properties of $A\Omega$). Hence, a failure detector D' of class $A\Omega$ cannot be obtained from $AS[D]$. □

Theorem 1 *$A\Omega'$ is strictly weaker than $A\Omega$.*

Proof It derives directly from Lemmas 1 and 2. □

3.3 Anonymous partially synchronous system *APSS*

Let *APSS* be a system like *AS* but with the following particular features. Links are eventually timely. A link between processes p_i and p_j is *eventually timely* if there is an (unknown) *stabilization time* T_{st} after which if process p_i sends a message at time $t \geq T_{st}$, this message is delivered without errors to p_j in a bounded time $t' \leq t + \Delta$, being Δ an unknown but finite time. Messages sent by p_i at time $t'' < T_{st}$ (i.e., before the global stabilization time) can be lost or delivered to p_j after a finite time greater than $t'' + \Delta$.

We consider that the number of processes that may crash in the system *APSS* is at most $n - 1$ (i.e., $f \leq n - 1$).

Processes are *partially synchronous* in the sense that the time to execute a step by a process p_i is an unknown positive but bounded time.

3.4 The algorithm $\mathcal{A}_{A\Omega'}$ in *APSS*

We present in Fig. 1 an algorithm to implement the $A\Omega'$ failure detector in the system *APSS*. In every run, $\mathcal{A}_{A\Omega'}$ eventually elects a set of leaders among all correct processes of the system *APSS*. This algorithm has a nice property: communication efficiency. That is, in every run, there is a time after which only leader processes broadcast messages.

The description of the algorithm $\mathcal{A}_{A\Omega'}$ of Fig. 1 is the following. A correct process p_i is one of the leader processes if the condition of line 15 of Task 1 is ever satisfied, and hence, $leader_i$ contains $true$ forever. Note that this is so because after line 1 there is no line in Tasks 1 and 2 of Fig. 1 where $leader_i$ is set to $false$ again.

In Task 1, each leader process p_i broadcasts heartbeat messages (HB, seq_i) permanently, being seq_i its number of sequence (lines 5–8). A process p_i waits a time $timeout_i$ (line 9) after which it checks how many acknowledgments it has received (lines 10–16). If process p_i is a leader process, it stores in rec_i the set of messages (ACK_HB, s, s') received with $s \leq seq_i \leq s'$ (line 11). Note that rec_i, when this line 11 is executed, can return messages that had been received before line 7 is executed. Hence, $quantity_i$ has the number of these acknowledgments contained in rec_i (line 12). If process p_i is not a leader process, it stores in rec_i the set of new messages $(ACK\_HB, -, -)$ received since its latest execution of line 14. If it does not receive any acknowledgment message, then process p_i becomes a leader (line 15).

In Task 2, each leader process p_i uses the variable $next\_ack_i$ to know the next number of sequence s of the acknowledgment message $(ACK\_HB, s, -)$ that process p_i has to broadcast. Initially, $next\_ack_i \leftarrow 1$ (line 2). When a leader process p_i

```
Init:
 (1) $timeout_i \leftarrow 1$; $leader_i \leftarrow false$; $seq_i \leftarrow 0$;
 (2) $next\_ack_i \leftarrow 1$; $quantity_i \leftarrow 0$;
 (3) start Tasks 1 and 2.

Task 1:
 (4) while true do
 (5)     if ($leader_i$) then
 (6)         $seq_i \leftarrow seq_i + 1$;
 (7)         $broadcast(HB, seq_i)$
 (8)     end if;
 (9)     wait until $timeout_i$ units;
 (10)    if ($leader_i$) then
 (11)        let $rec_i$ be the set of $(ACK\_HB, s, s')$
             received such that $s \leq seq_i \leq s'$;
 (12)        $quantity_i \leftarrow |rec_i|$
 (13)    else
 (14)        let $rec_i$ be the set of new $(ACK\_HB, -, -)$
             received;
 (15)        if ($rec_i = \emptyset$) then $leader_i \leftarrow true$ end if
 (16)    end if
 (17) end while.

Task 2:
 (18) upon reception of message $(HB, s_k)$
      such that ($s_k \geq next\_ack_i$) do:
 (19)        if ($leader_i$) then
 (20)            $broadcast(ACK\_HB, next\_ack_i, s_k)$;
 (21)            $next\_ack_i \leftarrow s_k + 1$
 (22)        end if.

 (23) upon reception of message $(ACK\_HB, s_k, s'_k)$
      such that ($s_k < seq_i$) do:
 (24)    if ($leader_i$) then $timeout_i \leftarrow timeout_i + 1$ end if.
```

Fig. 1 The algorithm $\mathcal{A}_{A\Omega'}$ in the system *APSS* (code for process p_i)

receives a message (HB, s_k) not previously acknowledged (i.e., $s_k \geq next\_ack_i$) (line 18), it broadcasts a message $(ACK\_HB, next\_ack_i, s_k)$ which acknowledges (in only one message) all heartbeat messages with the number of sequence in the range $[next\_ack_i, s_k]$ (line 20).

A leader process p_i may broadcast heartbeat messages (HB, seq_i) faster than the time that another leader process p_k broadcasts messages (ACK_HB, s_k, s'_k) with $s_k \leq seq_i$. In this case, process p_i will receive outdated acknowledgment messages, and $timeout_i$ will be incremented in one unit (lines 23–24). Then, leader process p_i will slow down its heartbeat broadcasting speed because it increases the time that it is waiting at line 9.

3.5 Correctness of $\mathcal{A}_{A\Omega'}$ in *APSS*

We now present the formal proofs to show that $\mathcal{A}_{A\Omega'}$ implements $A\Omega'$ in *APSS*.

The following lemma shows that there is a time after which every correct process p_i has $leader_i = x$ permanently. This value x is either $true$ or $false$.

Lemma 3 *For each run,* $(L \cup NL = Correct) \wedge (L \cap NL = \emptyset)$.

Proof Let us consider, by contradiction, that there is a run with a correct process p_i such that $p_i \notin L$ and $p_i \notin NL$. Then, by this hypothesis of contradiction, there is some correct process p_i such that $leader_i$ is changing its boolean value infinitely often. However, process p_i initially has $leader_i = false$ (line 1), and it only may change to $true$ once (when the condition of line 15 is satisfied). Note that there is no line in Tasks 1 and 2 of Fig. 1 where $leader_i$ is set to $false$ again. Hence, we reach a contradiction. Therefore, every correct process p_i either $p_i \in L$ or $p_i \in NL$, and hence, $(L \cup NL = Correct) \wedge (L \cap NL = \emptyset)$. □

Let T_F be the time when every faulty process p_f has crashed, and all messages $(HB, -)$ and $(ACK\_HB, -)$ broadcast by p_f have already been delivered or lost.

We prove in the following lemma that at least one correct process p_c eventually has $leader_c = true$ permanently.

Lemma 4 *For each run,* $L \neq \emptyset$

Proof By contradiction, let us consider that there is a run such that $L = \emptyset$. Note that in Fig. 1 if process p_i changes from $leader_i = false$ (line 1) to $leader_i = true$ (line 15), $leader_i$ will never change to $false$ again. So, if the hypothesis of contradiction holds, there is no process that broadcasts messages $(HB, -)$ and $(ACK\_HB, -)$ after T_F, because $leader = false$ in all correct processes (lines 5–8 and lines 19–22). Note that the maximum number of faulty processes in the system is $n - 1$ (i.e., $f \leq n - 1$). Then, after T_F, at least one correct process p_c will execute $leader_c \leftarrow true$ because it has not received any message since its latest execution of line 14, and rec_c is empty (lines 14–15). Therefore, we reach a contradiction because at least a correct process p_c has $leader_c = true$ permanently, and hence, for each run, $L \neq \emptyset$. □

Let it_i^s be the sth iteration of process p_i. This iteration is formed by all operations from line 4 to line 17 of Task 1 of Fig. 1 executed by process p_i for the sth time.

We show in the following lemma that eventually each leader process p_i has in rec_i, when it executes line 11, one (and only one) message (ACK_HB, s, s') with $s \leq seq_i \leq s'$ from every leader process p_j.

Lemma 5 *In each run, given processes* $p_i \in L$ *and* $p_j \in L$*, there is an iteration* $it_i^{s_a}$ *such that* $\forall s_b \geq s_a$ *process* p_i *has in* rec_i *exactly one message* (ACK_HB, s, s') *with* $s \leq s_b \leq s'$ *of process* p_j *when process* p_i *executes line* 11 *at iteration* $it_i^{s_b}$.

Proof Note that, after executing $leader_i \leftarrow true$ of line 15, correct process $p_i \in L$ broadcasts messages (HB, s_i) permanently, increasing in one unit the value of the sequence number s_i at each iteration $it_i^{s_i}$.

Let us define a time T_l such that $T_l \geq T_{st}$, and process p_i and process p_j are already leaders. Then, leader process p_i will be broadcasting messages (HB, s_i) permanently at each iteration $it_i^{s_i}$ with an increasing number of sequence s_i, such that after time T_l

we know that all these heartbeat messages will be received by leader process $p_j \in L$. So, we also know that process p_j after time $T_l + \Delta$ will broadcast acknowledgment messages (ACK_HB, s_j, s'_j) permanently with increasing values of s_j and s'_j, being $s_j \leq s'_j$. Note that process p_j broadcasts one (and only one) message (ACK_HB, s', s'') in response to all messages (HB, s_i) received from all leaders, $s' \leq s_i \leq s''$, (lines 18–21).

Let us consider the following sequence of iteration numbers $s_1 < s_2 \cdots < s_a$. Let $(ACK\_HB, s_1, -)$ be the first acknowledgment message broadcast by p_j after time T_l. Then, for the iteration $it_i^{s_2}$, there is a message (ACK_HB, s, s') with $s \leq s_2 \leq s'$ broadcast by process p_j and delivered at process p_i at most Δ units of time after being broadcast. Note that (ACK_HB, s, s') with $s \leq s_i \leq s'$ can be the same message for several consecutive iterations.

Note that if in an iteration $it_i^{s_x}$, with $s_x > s_1$, when leader process p_i executes line 11, it has not received the message (ACK_HB, s, s') with $s \leq s_x \leq s'$ from process p_j, then, each time this happens, $timeout_i$ will be incremented when this message (ACK_HB, s, s') with $s \leq s_x \leq s'$ is finally received (lines 23–24). This is so because seq_i will be greater than s_x.

Let s_a be the iteration number where for the first time the value of $timeout_i$ will be greater than time $T_{reply_j} = 2\Delta + \phi_j$, being Δ the maximum time to deliver a message from p_j to p_i, and where ϕ_j is the maximum time that process p_j takes to execute lines 18–22.

Now, let us assume, by contradiction, that there is an iteration $it_i^{s_b}$, with $s_b > s_a$, such that when leader process p_i executes line 11 at this iteration $it_i^{s_b}$, it has not received the message (ACK_HB, s, s') with $s \leq s_b \leq s'$ from process p_j. Note that in this iteration process p_i broadcasts the message (HB, s_b), and waits until $timeout_i > T_{reply_j}$ because this time is never decreased in the algorithm. Then, when process p_i executes line 11 at this iteration $it_i^{s_b}$, either (a) will receive one message (ACK_HB, s, s') with $s \leq s_b \leq s'$ from process p_j, or (b) has already received one message (ACK_HB, s, s') with $s \leq s_b \leq s'$ from process p_j in response to a faster leader.

Thus, for every iteration $it_i^{s_b}$ with $s_b \geq s_a$, exactly one message (ACK_HB, s, s') with $s \leq s_b \leq s'$ from process p_j will be received by process p_i when it executes line 11 at $it_i^{s_b}$. Hence, we reach a contradiction and the claim of the lemma follows. □

This theorem proves that $\mathcal{A}_{A\Omega'}$ is communication efficient. Note that in the worst case all correct processes are in L.

Theorem 2 *In the algorithm of Fig.* 1, *there is a time after which only processes in L broadcast messages permanently.*

Proof From Lemma 3 and definition of T_F, we can observe in the algorithm of Fig. 1 that eventually after T_F only correct processes are alive and all broadcast and delivered messages belong to these correct processes. Then, if a correct process p_i broadcasts a message $(HB, -)$ or $(ACK\_HB, -)$, it must have $leader_i = true$ (lines 5–8 and lines 19–22, respectively). So, if this case happens, it has already executed $leader_i \leftarrow true$ of line 15. Finally, note that if process p_i changes from $leader_i = false$ (line 1) to $leader_i = true$ (line 15), this variable $leader_i$ will never change to $false$ again, and hence, p_i is in L. Therefore, there is a time after which only processes in L broadcast messages permanently. □

Theorem 3 *The algorithm of Fig.* 1 *implements the failure detector $A\Omega'$ in APSS.*

Proof From Lemmas 3 and 4, Conditions 1 and 2 of $A\Omega'$ are preserved in each run. From Theorem 2 and Lemma 5, in each run, every process $p_i \in L$ eventually has $rec_i = L$ permanently when it executes line 11, and hence, $quantity_i = |L|$ (line 12). Thus, Condition 3 of $A\Omega'$ is also preserved in each run. Therefore, the algorithm of Fig. 1 implements the failure detector $A\Omega'$ in a system *APSS*. □

4 Consensus with $A\Omega'$

We introduce in this section the algorithm $\mathcal{A}_{cons}$ to implement consensus in anonymous asynchronous systems augmented with the failure detector $A\Omega'$, and with a majority of correct processes (see Fig. 2).

The *consensus* problem [13] specifies that all processes that take a decision have to decide the same value v, and this value v has to be proposed by some process. More formally, the definition of consensus for anonymous systems is the following.

4.1 Definition of consensus

In each run, every process of the system proposes a value, and has to decide a value satisfying the following three properties:

1. *Validity* Every decided value has to be proposed by some process of the system.
2. *Termination* Every correct process of the system eventually has to decide a value.
3. *Agreement* Every decided value has to be the same value.

4.2 Anonymous asynchronous system *AAS*

Let *AAS* be a system such as *AS* but with the following particular features. Links are reliable. A link between processes p_i and p_j is *reliable* if every message sent by p_i is delivered once to p_j without errors in an unknown, positive and unbounded time.

We consider that a majority of processes are correct in this system (i.e., $f < n/2$).

Each process p_i initially has no information about any other different process p_j of Π $(i \neq j)$ except that the size of the system is n and $f < n/2$. In other words, in every run, process p_i only knows that of n processes at least the majority of them are correct, but it does not know who they are or the exact number of them.

As we have mentioned in the introduction, it is impossible to solve consensus in anonymous asynchronous systems. To avoid this result, failure detectors are included. We denote by $AAS[A\Omega']$ the anonymous asynchronous system defined in this section augmented with the failure detector $A\Omega'$.

4.3 The algorithm $\mathcal{A}_{cons}$ in $AAS[A\Omega']$

We present in Fig. 2 an algorithm to solve consensus in $AAS[A\Omega']$. This algorithm is an adaptation of the leader-based consensus algorithm of [8] to the case in which multiple leaders coexist in the anonymous system. The changes between both algorithms are mainly focused in the phase *PH0* where the failure detector is used. Every process p_i uses the while sentence of Task 1 to execute asynchronous rounds permanently (lines 4–24). Each round is formed by three phases: *PH0*, *PH1* and *PH2*. Process p_i uses the variable r_i to know the number of the round that it is executing. The variable est_i contains the value proposed by p_i in round r_i to be decided. Note that initially est_i

```
function propose($v_i$):
Init:
 (1)   $r_i \leftarrow 0$; $est_i \leftarrow v_i$;
 (2)   start Tasks 1 and 2.
Task 1:
 (3)   while $true$ do
 (4)     $r_i \leftarrow r_i + 1$;
         % phase $PH0$
 (5)     $l_i \leftarrow D.leader_i$;
 (6)     if ($l_i$) then $broadcast(PH0, true, r_i, est_i)$ end if;
 (7)     wait until
(7-a)      $((l_i \neq D.leader_i)$
                $\vee$
(7-b)       $((l_i) \wedge (D.quantity_i\ (PH0, true, r_i, -)$ received)
                $\vee$
(7-c)       $((PH0, false, r_i, -)$ received));
 (8)     if $((PH0, -, r_i, -)$ received) then
 (9)       $est_i \leftarrow \min\{est_k : (PH0, -, r_i, est_k)$ received$\}$
 (10)    end if;
 (11)    $broadcast(PH0, false, r_i, est_i)$;
         % phase $PH1$
 (12)    $broadcast(PH1, r_i, est_i)$;
 (13)    wait until $(PH1, r_i, -)$ received
                                from $> n/2$ processes;
 (14)    if (all $(PH1, r_i, est)$ received : $est_i = est$) then
 (15)      $agree_i \leftarrow true$ else $agree_i \leftarrow false$
 (16)    end if;
         % phase $PH2$
 (17)    $broadcast(PH2, r_i, est_i, agree_i)$;
 (18)    wait until $(PH2, r_i, -, -)$ received
                                from $> n/2$ processes;
 (19)    if $((PH2, r_i, est, true)$ received) then
 (20)      $est_i \leftarrow est$
 (21)    end if
 (22)    if (all $(PH2, r_i, est, true)$ received) then
 (23)      $broadcast(DECIDE, est_i)$; return($est_i$)
 (24)    end if
 (25)  end while.

Task 2:
 (26)  upon reception of $(DECIDE, v)$ do:
 (27)    $broadcast(DECIDE, v)$; return($v$).
```

Fig. 2 The algorithm $\mathcal{A}_{cons}$ for solving consensus in the system $AAS[A\Omega']$ where is known that a majority of processes are correct (process p_i's code)

contains the value v_i, that is, the proposal of process p_i (line 1). The boolean variable $agree_i$ allows process p_i to indicate whether it knows that a majority of processes has the same value in est_i in round r_i. The boolean variable l_i is used to know whether process p_i is a leader process in round r_i.

In phase *PH0*, the goal is to reach a round r after which every leader process p_j has the same value v in est_j in each next round $r' \geq r$. To know if a process p_i is a leader process in a round x, it stores in l_i the boolean value of $leader_i$ returned by the failure detector D of class $A\Omega'$ (line 5). If it is a leader, process p_i broadcasts a message $(PH0, true, x, est_i)$ (line 6), and waits to receive a number $D.quantity_i$ of messages $(PH0, true, x, -)$ (line 7-b). Note that this value $quantity_i$, returned by the failure detector D of class $A\Omega'$, is eventually the number of all leader processes if process p_i is a leader process (Case 3 of $A\Omega'$). After that, a process p_i sets in est_i the minimal value of all received messages of phase *PH0* (lines 8–10), and broadcasts a message $(PH0, false, x, est_i)$ (line 11). This latter message allows non-leader processes to finish waiting in line 7-c in round x. Note that there is an unknown time t' after which the value $leader_i$ returned by D stabilizes (by definition of $A\Omega'$). Before this time t', a process p_i can believe that it is a leader in round x because $l_i = true$, but $D.leader$ may change to $false$ in the same round x. To avoid being blocked forever in the wait sentence of line 7, the failure detector is checked permanently to know if the value of l_i changes with respect to $D.leader_i$ while it is waiting in phase *PH0* (line 7-a). Similarly, the value $D.quantity_i$ is also checked permanently to eventually know the exact number of leader processes in this round x.

In *PH1*, the goal is that processes can check whether in a round r each process p_j of a majority of processes has the same value v as the proposed value (i.e., $est_j = v$). To do so, each process p_i broadcasts in round x a message $(PH1, x, est_i)$ (line 12), checking whether it receives the same value est in all messages $(PH1, x, est)$ from a majority of processes, and $est_i = est$ (line 14). If this happens, $agree_i \leftarrow true$, otherwise, $agree_i \leftarrow false$ (lines 14–16).

In *PH2*, the goal is that a process p_i can decide a value v in a round r. If this happens, v has to be always the unique possible value to be decided by any other process p_j in any of the next rounds $r' \geq r$. To do so, each process p_i broadcasts, in a round x, a message $(PH2, x, est_i, agree_i)$ and waits to receive a message $(PH2, x, -, -)$ from a majority of processes (lines 17–18). Note that, due to phase *PH1*, given any two received messages $(PH2, x, est_j, true)$ and $(PH2, x, est_k, true)$, the proposed value in this round x has to be the same (i..e, $est_j = est_k$). Hence, if the fourth parameter of some received message is $true$, process p_i establishes est as the value to be decided in x or in a next round (lines 19–21). On the other hand, if in all these messages the fourth parameter is $true$, process p_i decides in this round x the value est of all received messages $(PH2, x, est, true)$, and broadcasts a message $(DECIDE, est_i)$ with its decision (lines 22–24).

4.4 Correctness of $\mathcal{A}_{cons}$ in $AAS[A\Omega']$

We define a *round* r as the set of sentences that every process p_i executes while it has $r_i = r$.

Lemma 6 *Validity: for each run, every decided value has to be proposed by some process of the system.*

Proof Let us use induction on the number of rounds r to show that when a process p_i finishes a round in a run, it holds in its variable est_i a value proposed by some process. For simplicity, we assume round $r = 0$ as base of our induction. Clearly, the claim holds since the variable est_i of process p_i is initialized with its own proposed value v_i (line 1). *Induction hypothesis*: Let us assume that the claim is true for round $r = k$. Then, for every process p_i that finished round k, the value held in variable est_i, when $r_i = k$, was proposed by some process. *Step* $r = k + 1$: We now show that the claim also holds in round $r = k + 1$. For every process p_i that finished round k, the variable est_i can be changed in round $k + 1$ with the proposed value *est* broadcast in phase *PH0* by some process (lines 8–10). After that, the variable est_i can only be changed with proposed values *est* broadcast by processes in phase *PH1* (line 14) and *PH2* (line 20). Then, the claim is also held in this step $r = k + 1$. Therefore, it is shown by induction that when a process p_i finishes round r it holds in its variable est_i a value proposed by some process.

Thus, if a process p_i finishes a round and decides $est_i = v$ when it executes line 23, this value v was proposed by some process. On the other hand, if p_i decides v executing Task 2, this value v was also proposed by some process because it is broadcast when line 23 is executed. So, for each run, every decided value has to be proposed by some process of the system. □

Observation 1 *If some correct process does not wait forever at line 7 of a round r, then no other correct process will wait forever at line 7 of this round r.*

Proof If a correct process p_i reaches line 11, it broadcasts a message (*PH0*, *false*, r, est_i) that makes the condition of line 7-c true for every other correct process (recall that links are reliable). □

We say that a process p_i changes its *leading state* if the value of l_i is changed.

Observation 2 *If a correct process changes its leading state after executing line 5 of round r, then no correct process will wait forever at line 7 of this round r.*

Proof If a correct process changes its leading state after executing line 5 of round r, then the condition of line 7-a becomes true and it unblocks. Then, from Observation 1, every other correct process stops waiting at line 7 of this round r. □

Observation 3 *If any correct process waits forever at line 7 of a round r, then the condition of line 7-b evaluates to false forever for every correct process.*

Proof Otherwise, some correct process would stop waiting and, from Observation 1, every other correct process would stop waiting at line 7 of round r. □

Lemma 7 *No correct process waits forever at line* 7.

Proof Let us assume, by the way of contradiction, that there is a correct process that waits forever at line 7 of a round r. Then, every other correct process waits forever too. Otherwise, Observation 1 would not hold. Besides, every correct process keeps its initial leading state. Otherwise Observation 2 would not hold. Finally, since every other correct process waits forever, then from Observation 3, the condition of line 7-b evaluates to false forever for every correct process. Since every correct process keeps its initial leading state, the failure detector of every correct process eventually computes $D.quantity$ correctly. Since all the correct leader processes execute line 6, and they are leaders forever from Observation 2, then, eventually, at least one correct process will receive at least $D.quantity$ $(PH0, true, r, -)$ messages (recall that the links are reliable), what contradicts the initial assumption, completing the proof. □

Lemma 8 *Let p_l be a leader process in run R. There is a round r of run R after which every process p_i that finishes phase PH0 has $est_i = est_l$ at the end of phase PH0 of each round $r' \geq r$.*

Proof Let t be the time in a run R when (a) all faulty processes have crashed and their broadcast messages have already been delivered, (b) $D.leader$ does not change in any correct process anymore, and (c) $D.quantity$ does not change in any leader process anymore. Let r be the largest round in run R reached by any correct process at time t. Let us consider that this process is p_i. From Lemma 7, no process blocks in phase *PH0* of this round r. From the assumption that a majority of processes are correct, no process blocks in phases *PH1* and *PH2* of round r. Then, all leader processes eventually reach this round r and broadcast $(PH0, true, r, est)$. Hence, we have two cases:
Case 1 Process p_i is a leader process. Its variable $D.quantity_i$ has the total number of leader processes and it receives this number of messages $(PH0, true, r, est)$ (line 7-b). Then, it sets est_i with the minimum value est of all processes. Each leader process $p_l \neq p_i$ will also receive the same messages and will also set in its variable est_l the same minimum value est. After that, process p_i broadcasts $(PH0, false, r, est_i)$ (line 11). Therefore, variable est of all leader processes have the same value.
Case 2 Process p_i is a non-leader process. Each leader process, when finishes phase *PH0*, broadcasts $(PH0, false, r, est)$ with the minimum value est of all leader processes (line 11). Hence, all messages $(PH0, false, r, est)$ received by process p_i have the value est of a same leader process.

Therefore, by the two previous cases, every process p_i that finishes phase *PH0* has est_i with the same value of a leader process at the end of phase *PH0* of this round r. Note that after phase *PH0*, the value in the variable est does not change in the following two phases of the same round. Then, process p_i keeps the same common value in est_i in phases *PH1* and *PH2*. Thus, in every round $r' \geq r$ of a run, every process p_i that finishes phase *PH0* of r' has $est_i = est_l$ at the end of this phase *PH0*, being p_l a leader process. □

Lemma 9 *Agreement: for each run, every decided value has to be the same value.*

Proof Let us suppose that a process p_i decides a value v in the round r of a run, and a process p_j decides a value v' in round $r' \geq r$ of the same run. Then, this lemma is true if we show that $v = v'$.

Let us use induction on the number of rounds r' to show this result. Let us assume that the base case of our induction is $r' = r$ (both processes p_i and p_j decide in the same round). If a process p_i decides a value v in this round r, it is because a majority of processes broadcasts $(PH1, r, v)$ and $(PH2, r, v, true)$. Similarly, if a process p_j also decides in this same round r, another (or the same) majority of processes broadcasts $(PH1, r, v')$ and $(PH2, r, v', true)$. Then, because two majorities has at least one process in common, $v = v'$. Hence, the base case is satisfied. *Induction hypothesis:* Let us assume that the claim is true until round $r' = r + k$, $k \geq 1$. Then, if a process p_i decides a value v in round r, every process p_j that decides until round $r + k$ holds v in variable est_j. *Step* $k+1$: We now show that the claim also holds in round $r' = r+k+1$. Note that if a process decides a value v' in round $r + k$, this process receives messages $(PH1, r + k, v')$ and $(PH2, r + k, v', true)$ of a majority of processes. Also note that if a process that does not decide in round $r + k$ wants to finish this round, it also has to receive messages $(PH2, r + k, -, -)$ of a majority of processes. Then, at least one message of this majority has to be $(PH2, r + k, v', true)$. By induction hypothesis, $v' = v$. Hence, every process p_j that reaches round $r + k + 1$ holds in its variable est_j the value v in round $r + k$. Clearly, if any process p_j decides in this round $r + k + 1$, the value only can be v, hence, the claim is also satisfied for $r' = r + k + 1$. Therefore, the lemma is shown by induction. □

Lemma 10 *Termination: for each run, every correct process of the system eventually has to decide a value.*

Proof From Lemma 8, there is a round r in every run after which every process p_i that finishes phase *PH0* has $est_i = est_l$, being p_l a leader process. Then, from Lemma 7 and because a majority of processes never crashes, all received messages in phase *PH1* of round r are $(PH1, r, est_l)$ and its number is greater than $n/2$. Hence, process p_i does not block in phase *PH1* and all received messages in phase *PH2* of round r are $(PH2, r, est_l, true)$, and its number is also greater than $n/2$. So, every process p_i that finishes phase *PH2* in round r can decide. Therefore, every correct process of the system decides. □

Theorem 1 *The algorithm described in Fig.* 2 *solves the consensus problem in* $AAS[A\Omega']$.

Proof From Lemmas 6, 9 and 10, validity, agreement and termination properties are satisfied in every run. □

4.5 Analysis of rounds

A way to consider the costs of an algorithm for consensus in message-passing systems is to evaluate the number of rounds needed to decide a value. In [5] is shown that $2t + 1$ is the lower bound on the number of rounds to achieve consensus (t is the maximum number of crashed processes). They can determine it exactly because they use the perfect failure detector for anonymous systems $\overline{AP}$. Differently of [5] because we do not use a perfect failure detector, the maximum number of rounds in $\mathcal{A}_{cons}$ for each run can not be bounded a priori. Therefore, we are going to analyze the extreme cases. Our algorithm $\mathcal{A}_{cons}$ works in asynchronous consecutive rounds, such that each round is formed by three phases (*PH0*, *PH1* and *PH2*).

The best case is when each process knows if it is a leader or a non-leader since the beginning of the run. That is, $(\forall p_i \in L, \forall p_k \in NL, \forall \tau) \implies (l_i^\tau = true$ and $l_k^\tau = false)$. Note that this happens if the failure detector D of class $A\Omega'$ stabilizes since time $\tau = 0$ and returns *true* or *false* accurately. Hence, in this best case, a process decides in the phase *PH2* of the first round (line 23). The worst case for a correct process p_i is if when the failure detector stabilizes, it is in the highest round r' of all processes. In this case, p_i has to wait until the rest of processes that form the majority reach its round r' (line 7). Then, this process p_i will decide in the phase *PH2* of this round r' (line 23).

5 Conclusion

Anonymous systems are necessary when an identity is not possible in the processes of the system. Such cases are common, for example, in sensor networks where devices have constraints in computational power, a small storage capacity, or when there are a very big number of devices in the system.

The $A\Omega'$ failure detector [10] has been proposed very recently as a new counterpart of the omega failure detector for anonymous systems. We prove in this paper that $A\Omega'$ is strictly weaker than $A\Omega$ (which is the previously proposed version of the anonymous omega failure detector proposed in the literature).

It has been shown in [6] that $A\Omega$ is not implementable, and we present in this paper the first implementation of $A\Omega'$. Therefore, we prove in this paper that $A\Omega'$ is implementable.

Finally, we prove in this paper that consensus can be solved in anonymous asynchronous systems using $A\Omega'$ when a majority of processes does not crash. Hence, we also show here that consensus is implementable in anonymous systems.

References

1. Angluin D, Aspnes J, Diamadi Z, Fischer MJ, Peralta R (2006) Computation in networks of passively mobile finite-state sensors. Distrib. Comput 18(4):235–253
2. Aspnes J (2010) A modular approach to shared-memory consensus, with applications to the probabilistic-write model. In: Proceedings of 29th symposium on principles of distributed computing (PODC), pp 460–467
3. Aspnes J, Ellen F (2011) Tight bounds for anonymous adopt-commit objects. In: Proceedings of 23rd symposium on parallelism in algorithms and architectures (SPAA), pp 317–324
4. Attiya H, Gorbach A, Moran S (2002) Computing in totally anonymous asynchronous shared memory systems. Inf Comput 173(2):162–183
5. Bonnet F, Raynal M (2011) The price of anonymity: optimal consensus despite asynchrony, crash, and anonymity. ACM Trans Auton Adapt Syst TAAS 6(4):23
6. Bonnet F, Raynal M (2013) Anonymous asynchronous systems: the case of failure detectors. Distrib Comput 26(3):141–158
7. Bonnet F, Raynal M (2010) Anonymous asynchronous systems: the case of failure detectors. In: Proceedings of 24th international symposium on distributed computing (DISC), pp 206–220
8. Bonnet F, Raynal M (2010) Consensus in anonymous distributed systems: is there a weakest failure detector? In: AINA. IEEE Computer Society, Australia, pp 206–213
9. Bouzid Z, Sutra P, Travers C (2011) Anonymous agreement: the janus algorithm. In: Proceedings of 15th international conference on principles of distributed systems (OPODIS). France, pp 175–190

10. Bouzid Z, Travers C (2012) Anonymity, failures, detectors and consensus. In: Proceedings of 26th international symposium on distributed computing (DISC), pp 427–428
11. Bouzid Z, Travers C (2012) Anonymity, failures, detectors and consensus. Technical report. http://hal.inria.fr/hal-00723309. INRIA, August 2012
12. Chandra T, Hadzilacos V, Toueg S (1996) The weakest failure detector for solving consensus. J ACM 43(4):685–722
13. Chandra T, Toueg S (1996) Unreliable failure detectors for reliable distributed systems. J ACM 43(2):225–267
14. Delporte-Gallet C, Fauconnier H (2009) Two consensus algorithms with atomic registers and failure detector Ω. In: Proceedings of 10th international conference on distributed computing and networking (ICDCN'09), vol 5408. LNCS, pp 251–262
15. Delporte-Gallet C, Fauconnier H, Guerraoui R (2010) Tight failure detection bounds on atomic object implementations. J ACM 57(4)
16. Delporte-Gallet C, Fauconnier H, Guerraoui R, Kermarrec AM, Ruppert E, Tran-The H (2013) Byzantine agreement with homonyms. Distrib Comput 26(5–6):321–340
17. Delporte-Gallet C, Fauconnier H, Tielmann A (2009) Fault-tolerant consensus in unknown and anonymous networks. In: Proceedings of 29th IEEE international conference on distributed computing systems (ICDCS'09). Canada, pp 368–375
18. Durresi A, Paruchuri V, Durresi M, Barolli L (2005) A hierarchical anonymous communication protocol for sensor networks. In: Proceedings of international conference on embedded and ubiquitous systems (EUS'05). LNCS, vol 3824. Springer, Berlin, pp 1123–1132
19. Fischer M, Jiang H (2006) Self-stabilizing leader election in networks of finite-state anonymous agents. In: Proceedings of 10th international conference on principles of distributed systems (OPODIS). France, pp 395–409
20. Fischer MJ, Lynch N, Paterson MS (1985) Impossibility of distributed consensus with one faulty process. J ACM 32(2):374–382
21. Gafni E (1998) Round-by-round fault detectors: unifying synchrony and asynchrony. In: Proceedings of 17th symposium on principles of distributed computing (PODC), pp 143–152
22. Jiménez E, Arévalo S, Fernández A (2006) Implementing unreliable failure detectors with unknown membership. Inf Process Lett 100(2):60–63
23. Raynal M (2009) Failure detectors for asynchronous distributed systems: an introduction. In: Wiley encyclopedia of computer science and engineering, vol 2, pp 1181–1191
24. Raynal M (2010) Communication and agreement abstractions for fault-tolerant asynchronous distributed systems. In: Synthesis lectures on distributed computing theory. Morgan & Claypool Publishers