Consensus in anonymous asynchronous systems with crash-recovery and omission failures

Ernesto Jiménez¹ · José Luis López-Presa¹ · Marta Patiño-Martínez¹

Received: 13 May 2021 / Accepted: 27 September 2021 / Published online: 8 October 2021

© The Author(s) 2021

Abstract

In anonymous distributed systems, processes are indistinguishable because they have no identity and execute the same algorithm. Currently, anonymous systems are receiving a lot of attention mainly because they preserve privacy, which is an important property when we want to avoid impersonation attacks. On the other hand, Consensus is a fundamental problem in distributed computing. It is well-known that Consensus cannot be deterministically solved in pure asynchronous anonymous systems if processes can crash (the so-called crash-stop failure model). This impossibility holds even if message losses never occur in transmission. Failure detectors are an elegant and powerful abstraction for achieving deterministic Consensus in asynchronous distributed systems. A failure detector is a distributed object that gives the processes information about crashed processes. Failure detectors have attracted so much attention in the crash-stop failure model because they provide a totally independent abstraction. Ω is the weakest failure detector to solve Consensus in classic asynchronous systems when a majority of processes never crash, and $A\Omega'$ is its implementable version for anonymous systems. As far as we know, there is a lack of works in the literature which tackle Consensus in anonymous asynchronous systems where crashed process can recover (the so-called crash-recovery failure model) and also assuming errors in transmission operations (the so-called omission failure model). Extending failure models in the system allows us to design more realistic systems and solve more practical security problems (i.e., fair exchange and the secure multiparty computation).

This work has been partially funded by the Ministry of Economy and Competitiveness (MINECO) Under Project QOSDATA (PID2020-119461GB-I00) and by the Regional Government of Madrid (CM) Under Project EDGEDATA (P2018/TCS-4499).

✉ Ernesto Jiménez
ernesto.jimenez.merino@upm.es

José Luis López-Presa
jose Luis.lopezp@upm.es

Marta Patiño-Martínez
mpatino@fi.upm.es

¹ Universidad Politécnica de Madrid, Madrid, Spain

We present, in this paper, an algorithm to solve Consensus using $A\Omega'$ in anonymous asynchronous systems under the crash-recovery and omission failure models. Another important contribution of this paper is a communication-efficient and latency-efficient implementation of $A\Omega'$ for these new failure models.

Keywords Anonymity · Consensus · Asynchrony · Failure detectors · Omission failures · Anonymous omega · Crash-recovery failures

Mathematics Subject Classification 68W10 · 68R01

1 Introduction

Among all agreement problems, Consensus [1] stands out as a fundamental paradigm in distributed computing. In this problem, each process proposes a value and, eventually, only one value must be decided upon. Solving Consensus gets more complicated as the power of the adversary increases: processes in the system that may fail, number of messages that may be lost in transmission, lack of synchrony among the processes in the distributed system, etc.

In the influential seminal paper [2], it was shown that, in a totally asynchronous system, even when messages are never lost, it is not possible to solve Consensus in a deterministic way if one process may fail by crashing forever. This scenario where processes can crash but cannot recover later is known as the *crash-stop* failure model. From this impossibility result, several lines of research have emerged to solve Consensus in asynchronous distributed systems, isolating these additional requirements. One of these lines proposes to enrich the asynchronous system with a *failure detector* [1]. A failure detector can be seen as a distributed object that gives the processes some information about the crashes of processes, isolating time requirements. Thus, an asynchronous (oblivious to time requirements) algorithm, augmented with a failure detector, can be used to solve deterministic Consensus. This concept of failure detectors in the crash-stop failure model has attracted so much attention because it provides an independent abstraction where time requirements are hidden. Then, when we want to design an implementation, we can have two totally independent algorithms, one for Consensus in a fully asynchronous system and another for failure detection in a not completely asynchronous system. The consensus algorithm will be simpler to design and easier to maintain because it can use the failure detector based on its properties, without any dependence on the failure detector's algorithm and timing. Similarly, the failure detector's algorithm will be designed independently of the consensus algorithm, focusing on the synchronization requirements.

Since the publication of the seminal paper [1], many classes of failure detectors [3], providing many different types of information about faults, have appeared not only to solve Consensus but many other agreement problems [4,5]. Then, an important point is to choose the most appropriate failure detector. It is known that Ω [6] provides the minimum information which is necessary and sufficient to solve Consensus in the crash-stop failure model when $f < n/2$, being f the maximum number of faulty

processes and n the total number of processes in the system. Ω states that, eventually, a non-crashed process will be the leader.

Trying to represent more realistic distributed systems, the *crash-recovery* failure model was introduced to solve Consensus with failure detectors [7,8]. In this new model, crashes are not always permanent. Hence, a process p_i crashed at a time t may recover at time $t' > t$ and keep working. If p_i recovers after t' , all messages sent in the time interval $[t, t']$ will not be received by p_i because its buffers will not be available in that interval. Besides, all values stored in p_i 's *regular variables* before t' will also be lost because volatile memory is used. The only way to avoid data loss when a process crashes is to keep essential information in stable memory. That way, the process will be able to reload it upon recovery. However, access to stable memory, like Hard Disks or even Solid State Drives, is much slower than access to variables in Main Memory. Hence, it is important to correctly identify which variables are critical and should be stored in stable memory, and which ones are not.

With the same objective of expanding the distributed systems to study, the *omission* failure model was introduced by Perry and Toueg [9]. A process p_i makes an omission failure if it deviates from its algorithm and avoids executing a communication (send or receive) operation. For example, a process prevents the spreading of a message m if it skips all send operations for m . Similarly, a process avoids being informed of a message m sent by another process if it omits the receive operation for m . From a practical point of view, the omission failure model describes errors in the communication buffers of the processes (e.g., overflow of a buffer, or bit errors in a packet placed in the transmission buffer).

It has been shown that solving Consensus in a system with omission and crash-stop failures allows solving practical security problems such as *fair exchange* and *secure multiparty computation* in a hybrid system model with security modules [10,11]. These arise, for example, in e-commerce transactions and distributed voting.

In classic distributed systems, each process has an identifier which must be unique. Therefore, the sender and the receiver of every message can be identified, so the sender knows who receives the message, and the receiver knows who sends it. This technique is widely used in the literature and simplifies the resulting algorithms to implement Consensus.

In an attempt to hide the identity of the processes and preserve privacy, nicknames or temporal identifiers may be used within classic algorithms [12–14]. However, the behavior of each process may be inferred as well as the relationship between processes, even if it is not possible to know their real identities.

Conversely, in anonymous systems, processes do not have identifiers and it is not possible to know which communication link was used to transmit a message [15], so classic algorithms cannot be used in this context. Without identifiers, attacks such as impersonation or forging are directly prevented and the behavior of each process is impossible to track.

Another important reason to use anonymous systems is to be able to work with networks where the processes have small storage memory, a huge membership, or computational restrictions. In these scenarios, it is not possible to manage identifiers [16–19].

It has also been shown that solving Consensus in a system with omission and crash-stop failures allows solving practical security problems such as fair exchange and secure multiparty computation in a hybrid system model with security modules [10,11]. These arise, for example, in e-commerce transactions and distributed voting. For example, in the case of e-commerce, if transactions have to be private, avoiding the need for identifiers to reach Consensus might allow for total privacy.

It is noteworthy that all the algorithms that work in anonymous systems under a certain failure model also work in their counterpart version of classic systems, but not the other way round.

1.1 Related work

As previously mentioned, Ω is an interesting failure detector to use in classic distributed systems because it has the minimum requirements to solve Consensus in asynchronous systems under the crash-stop failure model. Bonnet and Raynal introduced $A\Omega$ as its counterpart version for anonymous systems [20]. $A\Omega$ states that, eventually, one and only one non-crashed process will be the leader. The main drawback of $A\Omega$ is that it is not implementable because it is not always possible to break the symmetry regarding the ownership of the messages received by the processes in an anonymous system [20,21]. The $A\Omega'$ failure detector [22] was proposed as an implementable version of $A\Omega$ [21]. $A\Omega'$ differs from $A\Omega$ in that the number of leaders in the system is not restricted to only one, when the symmetry cannot be broken.

A lot of work has been published about Consensus and failure detectors in classic asynchronous systems under the crash-stop failure model [3–5,23,24]. Several works have also been published on their counterparts in anonymous systems under the crash-stop failure model [20–22,25,26] with reliable communication links, where errors or losses in the transmission of messages cannot occur.

In the case of classic asynchronous systems under the omission and crash-stop failure model, a few works are present in the literature to solve Consensus with failure detectors [27–29]. In all of them, the rotating coordinator paradigm is used to implement Consensus. Therefore, the existence of process identifiers known to all processes is essential in all these algorithms. Regarding failure detectors, the algorithms developed in these three works are also implemented based on the initial knowledge of all the process identifiers (i.e., the membership must be known). The need to know the identity of the processes in all these solutions makes it impossible to apply them to an anonymous environment. As far as we know, no previous works have been published which solve Consensus in anonymous systems under the omission failure model.

There are several works that study Consensus and failure detectors in classic distributed systems under the crash-recovery failure model [7,8,30,31]. In the crash-stop failure model, it is clear that a correct process is one that never crashes, and a process is incorrect if it eventually crashes. However, in the case of the crash-recovery failure model, these concepts have to be redefined [7]. A process that crashes any number of times, but ultimately remains permanently up, is considered correct because it can eventually collaborate to solve Consensus. Otherwise, it is considered incorrect. Regarding communication links, previous works that consider the crash-recovery fail-

ure model use stubborn links [32], which are a generalization of reliable links when processes can recover. These links guarantee that messages sent between two correct processes will eventually be received if the sender and the receiver remain up during transmission.

Several Consensus algorithms have been presented with $f < n/2$ (i.e., with a majority of correct processes) and using stable variables [7,8,30,31]. These algorithms cannot be applied to anonymous systems because they use the rotating coordinator paradigm. Freiling et al. [30] adapt solutions that work under the crash-stop failure model to crash-recovery. In the case of the failure detectors, all papers use different classes of failure detectors but all of them must fulfill the *strong completeness* property: there is a time after which every incorrect process is permanently suspected by every correct process. It has been shown by Jiménez et al. [33] that these classes of failure detectors are not implementable when the membership is unknown. Hence, they are invalidated as possible candidates for use in anonymous systems. To our knowledge, there is no previous work that addresses how to solve Consensus in anonymous systems under the crash-recovery failure model. It is also worth mentioning the proposals which present several algorithms to implement Ω under the crash-recovery failure model [34–36]. Although some of these solutions do not need to know the membership of the system, they still need identifiers. Trying to adapt them to anonymous systems implies implementing its counterpart $A\Omega$. However, $A\Omega$ is not implementable in anonymous systems even under the crash-stop failure model [20].

In the case of classic asynchronous systems under omission and crash-recovery failure models, to our knowledge, only the paper of Fernández-Campusano et al. [37] tackles Consensus using failure detectors. The authors adapt the Consensus and failure detector algorithms of Cortiñas et al. [27] to crash-recovery. As previously mentioned, this solution is not applicable to anonymous systems where there are no identifiers. The same happens to the algorithms of Fernández-Campusano et al. [38,39], where the authors implement a version of Ω adapted to omission and crash-recovery.

1.2 Our contributions

The $A\Omega'$ failure detector has been previously defined and implemented in anonymous systems under the crash-stop failure model [21,22]. However, to our knowledge, this work is the first which extends the definition and includes an implementation of $A\Omega'$ in the more general and practical model of omission with crash-recovery failures. The $A\Omega'$ algorithm proposed in this paper is very effective in reducing the latency introduced by the use of stable variables: it just needs one integer stable variable that is accessed as little as possible: it is only read and written once when a process starts or recovers from a crash. This variable is a simple counter of the number of times the process has crashed. This algorithm is also *communication-efficient*, because only leaders eventually spread messages [40].

We also present an algorithm to solve Consensus in anonymous asynchronous systems with $f < n/2$ under the omission and crash-recovery failure models using an $A\Omega'$ failure detector. To our knowledge, this is the first such algorithm. Note that the combination of omissions and crash-recovery allows a much more realistic modeling

of systems than crash-stop or systems without communication omissions. Trying to reduce the latency of the Consensus algorithm, we follow the approach of Hurfin et al. [8] and Aguilera et al. [7] in the sense of keeping only critical values in stable storage, instead of the entire state of processes as in the paper of Oliveira et al. [31]. Nevertheless, our Consensus algorithm is not as efficient as these works at storing and accessing stable variables. The algorithm proposed by Aguilera et al. [7] writes to stable storage twice per round, and this number of write operations is reduced by Hurfin et al. [8] to just one per round. In fact, we need an indeterminate number (at least three) of this type of write operations per round. Regarding the content of these variables, they only store the critical values of the last message sent, while we also need to store data from previous messages. Note that, in our case, we have to face an important adversary, such as the lack of process identifiers, which adds uncertainty about what has already been sent/received to/from a certain process, in case of recovery. We believe it is a reasonable price to pay for anonymity in crash-recovery. However, reducing the amount of stable memory required is an open line for future research.

Another important parameter to measure latency in a Consensus algorithm is the number of *communication steps*¹ it requires to achieve consensus. As is common in the literature, to compare different consensus algorithms, we focus on executions where no failures occur. For this kind of runs, the solutions of Hurfin et al. [8] and Oliveira et al. [31] need one round formed by two communication steps, while that of Aguilera et al. [7] needs three communication steps, which is also the case of our algorithm. In this kind of runs, our algorithm needs to send $O(n^3)$ messages of size $O(1)$. These results are worse than those of most previous works for classic systems [8,30,31] which only need $O(n^2)$ messages, and also worse than that of Aguilera et al. [7], which needs $O(n)$ messages. This result is achieved because processes know the coordinator's identifier, and they can respond directly to it, rather than broadcast messages to all processes, which is the only possibility with anonymity.

1.3 Structure of the rest of the paper

This paper is organized as follows. We present, in Sect. 2, the model of the anonymous distributed system with omission and crash-recovery failures (denoted S). In Sect. 3 we introduce the definition of the A' failure detector, and Algorithm $\mathcal{A}_{A\Omega'}$ to show that A' is implementable in system S with partial synchrony (denoted PS). Then, we formally prove that $\mathcal{A}_{A\Omega'}$ fulfills the properties of A' in system PS . Sect. 4 is devoted to Consensus. We define the problem, propose Algorithm $\mathcal{A}_C$ to solve Consensus in system S with full asynchrony (denoted AS) and with the failure detector A' . As in the previous section, we also formally prove that $\mathcal{A}_C$ fulfills the properties of Consensus in system AS augmented with $A\Omega'$. We finish this section with a brief discussion about the performance of Algorithm $\mathcal{A}_C$ in system AS with $A\Omega'$. Finally, in Sect. 5, we present the conclusions and future work.

¹ A communication step is made up of the pair of broadcast/receive operations, such that corresponding reception operations are issued in response to these broadcast operations.

2 The anonymous system S with omissions and crash-recovery

System S is composed of a set of n processes $\Pi = \{p_1, p_2, \dots, p_n\}$ that communicate among them using message passing through a network of fully interconnected bidirectional links. This size n of Π is known to all processes. A process p_i invokes the primitive $broadcast_i(m)$ to send a message m through all its available links. The processes of Π are indistinguishable because they have no identity and they execute the same algorithm. We say, in this sense, that the system S is *anonymous* [20,41,42]. Nevertheless, for the sake of notation simplicity, we use the subscript i to identify the process p_i , but this process does not know its own index nor that of the other processes in Π . We assume that there is a discrete global clock in the system. However, this clock is not available to the processes in Π . Since the clock is discrete, we use natural numbers to represent instants of time. In System S , two types of failures are considered:

- *Omission* [9] A process makes a failure by omission if it does not execute a communication operation. From a practical point of view, this type of failures models errors in the communication with the operating system through I/O buffers. For example, a new message could be written to the output buffer before the transmission of the previous one was completed, preventing the transmission of a message. Another example may be the case that the buffer does not have enough capacity to store the received message, which generates an omission in reception.
- *Crash-Recovery* [7] A process p_i fails by crashing if it stops taking steps. Nevertheless, a crashed process can recover and restart its execution. A process is *down* while it is crashed, or *up* otherwise. Regarding these up/down states, a process $p_i \in \Pi$ belongs to one of these three classes:
 - *Eventually-up* if it crashes and recovers a finite number of times, but, at last, remains up forever. Note that the number of crashes might be zero.
 - *Eventually-down* if it crashes and recovers a finite number of times but, at last, remains crashed forever. Note that it might not recover after the first crash.
 - *Unstable* if it crashes and recovers an infinite number of times.

In S , when a process recovers, it loses all values stored in its *regular variables* before crashing, because they are kept in volatile memory. Similarly, after recovering, it also loses all previously received messages in its input buffer. To be able to restore the value of a variable after crashing, it must be kept in stable memory, the so-called *stable variables*. Thus, the content of these variables will be available and unaltered when the process recovers.

As proposed by Aguilera et al. [7], we consider that, when a process recovers, it knows it and starts executing from a predetermined line.

In system S , processes make finite omissions. Thus, there is a time after which no process makes omissions. Only eventually-up processes are considered *correct*. Other processes are considered *incorrect*. Hence, there is a time after which every correct process is permanently up and does not omit any more messages. We denote the set of correct processes by *Correct* and the set of incorrect ones by *Incorrect*. The maximum

number of incorrect processes f is a minority, i.e., $f < n/2$ being n the number of processes in the system.

It is assumed in S that every process is connected to every other process by a reliable link [1,20]. A link that connects process p_i to process p_j is reliable if every message sent by p_i is eventually received by p_j provided that both processes stay up during transmission, i.e., if p_i sends a message to p_j at time t , it will be received by p_j at some time $t', t' > t$, provided that p_i and p_j stay up during time interval $[t, t']$. Losses, modifications or duplication of messages are not possible in this case. Note that all messages sent to processes when they are down will be lost. When a process p_i crashes while executing $broadcast_i(m)$, nothing is guaranteed regarding the arrival of m to the processes connected to p_i 's reliable links. Thus, m could be received by an unpredictable subset of these processes. In the case that m is received by some process, it will be unaltered because links are reliable.

3 The failure detector $A\Omega'$

A failure detector F is a distributed tool where each process obtains information about incorrect or correct processes whenever it invokes its local module of F . Nevertheless, the information returned by F can contain (temporal and/or permanent) mistakes [1]. We define and implement the failure detector A' in an anonymous partially synchronous system where omission failures eventually cease. Algorithm $\mathcal{A}_{A\Omega'}$ (see Fig. 1) fulfills that only leader processes broadcast messages, so it is communication-efficient [40].

3.1 Definition of $A\Omega'$

It has been shown by Jiménez et al. [21] that $A\Omega'$ is an implementable version of the Ω failure detector for anonymous systems where processes may crash permanently [20]. Following, we are going to adapt this definition to crash-recovery systems. $A\Omega'$ provides two functions which allow a process to ask whether it is a leader and, in this case, how many leaders there are in the system. Eventually, the number of leaders must be bigger than zero. Let us define $A\Omega'$ more formally. Each process p_i has the boolean function $leader_i()$ and the integer function $quantity_i()$. Let $\mathcal{F}_i^t()$ be the value of function $\mathcal{F}_i()$ at time t . Let $L^t = \{p_i \in \text{Correct} : leader_i^t() = \text{true}\}$. We consider that function $leader_k^t()$ returns false if process p_k is down at time t .

The $A\Omega'$ failure detector must preserve that there is a time $\mathcal{T}$ such that, for all process $p_i \in \Pi$, for all time $t > \mathcal{T}$ $leader_i^t() = leader_i^{\mathcal{T}}()$, and for all process $p_j \in L^{\mathcal{T}}$ $quantity_j^t() = |L^{\mathcal{T}}| \neq 0$.

3.2 The anonymous partially synchronous system PS

Let PS be a system like S but augmented with the following features regarding links' reliability, synchrony and maximum number of failures.

Links in PS are eventually timely. A link between processes p_i and p_j is *eventually timely* if there is an unknown time T after which every message sent through this link

is delivered within Δ units of time for some unknown but finite value Δ , provided that processes p_i and p_j remain up in the time interval $[t, t + \Delta]$. Δ is the maximum delivery delay of the messages after time T . Note that the reception of m is not guaranteed before T or when the sender or the receiver crash during transit time.

In *PS*, processes are partially synchronous because we consider that there is a maximum and finite time to execute a step but it is unknown to all processes.

To solve Consensus with a failure detector as independent modules, we have to reduce the number of omission failures that processes can experience. Thus in *PS* we consider that the number of omissions of every process is finite but unknown. Conversely, while the maximum number of incorrect processes in system S was limited to $f < n/2$, in *PS*, we have been able to relax that limitation to $f < n$.

3.3 The algorithm $\mathcal{A}_{A\Omega'}$

An algorithm that implements the $A\Omega'$ failure detector is depicted in Fig. 1. This algorithm is structured in four parts: *initialization* (Lines 1–4) which is performed once when the process is started, *recovery* (Lines 34–39) which is executed when the process is restarted after a crash, *access* (Lines 30–33) which implements the functions provided by the failure detector, and the *main task* (Lines 5–29) which provides the access functions with the data they need.

In order to withstand crashes and recoveries of processes, Algorithm $\mathcal{A}_{A\Omega'}$ relies on an integer global variable named $STAGE_i$ kept in stable storage. We use capital letters to denote stable storage variables. Note that it is the only variable of this type needed in the algorithm. This variable $STAGE_i$ counts the number of crashes the process p_i has suffered (see Line 35). In volatile memory it is copied into $stage_i$. Additional variables are used to keep the leadership state of the process (ld_i) and the estimated number of leaders (rec_i). When a process starts, it becomes a leader (Line 2). However, when it recovers from a crash, it restarts as non-leader (Line 38).

The main task of the algorithm (Task 1) works in rounds. While leaders keep count of the rounds, non-leaders do not care about them. Each time it is restarted, it starts from round 0 (Line 5). Only leaders broadcast heartbeat messages at Line 9, so this algorithm is communication efficient.

At each round, leaders broadcast one heartbeat message and wait to receive the heartbeats from the leaders. However, since a process does not know how many leaders there are, it waits for a limited period of time (variable $timeout_i$) before analyzing the messages received. If it has not received any message, or all the messages it has received from processes in its own stage correspond to previous rounds (Line 13), then it is not waiting for a sufficiently large period of time, so it increments its timeout (Line 14). If it has received a message from a process that has crashed less times than itself or, having crashed the same number of times, is at a later round, i.e. the other is faster (see Line 16), then this process stops being a leader (Line 17). This way, only the fastest processes among these which have crashed the minimum number of times will remain as leaders.

At each round, the number of leaders is estimated as the number of heartbeat messages received (Line 12). Note that the messages considered at Line 11 are those

```

Init:
(1)  $STAGE_i \leftarrow 0$ ;
(2)  $stage_i \leftarrow STAGE_i$ ;  $ld_i \leftarrow \text{true}$ ;  $rec_i \leftarrow 0$ ;
(3)  $timeout_i \leftarrow 1$ ;
(4) start Task 1.

Task 1:
(5)  $r_i \leftarrow 0$ ;
(6) repeat
(7)   if  $ld_i$  then
(8)      $r_i \leftarrow r_i + 1$ ;
(9)      $\text{broadcast}(\text{HB}, stage_i, r_i)$ ;
(10)    wait until  $timeout_i$ ;
(11)    let  $Received$  be the set of  $(\text{HB}, -, -)$  messages received during wait;
(12)     $rec_i \leftarrow |Received|$ ;
(13)    if  $((rec_i = 0) \vee (\forall (\text{HB}, stage, r) \in Received : stage = stage_i, r < r_i))$  then
(14)       $timeout_i \leftarrow timeout_i + 1$ ;
(15)    end if;
(16)    if  $(\exists (\text{HB}, stage, r) \in Received : stage < stage_i \vee stage = stage_i \wedge r > r_i)$  then
(17)       $ld_i \leftarrow \text{false}$ ;
(18)    end if;
(19)  else
(20)    wait until  $timeout_i$ ;
(21)    let  $Received$  be the set of  $(\text{HB}, -, -)$  messages received during wait;
(22)    if  $(|Received| = 0)$  then
(23)       $ld_i \leftarrow \text{true}$ ;
(24)       $timeout_i \leftarrow timeout_i + 1$ ;
(25)    else if  $(\forall (\text{HB}, stage, -) \in Received, stage > stage_i)$  then
(26)       $ld_i \leftarrow \text{true}$ ;
(27)    end if;
(28)  end if;
(29) forever.

(30) function  $leader_i()$ :
(31)   return  $(ld_i)$ .

(32) function  $quantity_i()$ :
(33)   return  $(rec_i)$ .

(34) upon recovering of process  $p_i$  do:
(35)    $STAGE_i \leftarrow STAGE_i + 1$ ;
(36)    $stage_i \leftarrow STAGE_i$ ;
(37)    $timeout_i \leftarrow stage_i$ ;
(38)    $ld_i \leftarrow \text{false}$ ;
(39)   start Task 1.

```

Fig. 1 The algorithm $\mathcal{A}_{A\Omega'}$ in System PS where $f < n$ (code for process p_i)

which have been received since the last time this line was executed, or from the moment this process was (re)started, if this line was not previously executed since the process was (re)started.

Non-leaders wait (at each round) for a period of time determined by variable $timeout_i$ (Line 20). As in the case of the leaders, the messages considered at Line 21 are those received from the last time this line was executed or from the time the process (re)started. If no message is received in this lapse of time (Line 22), then the process assumes there is no other leader in the system and becomes a leader (Line 23). Addi-

tionally, it increments its timeout since it might not be waiting long enough (Line 24). In case all the received messages come from processes that have crashed more times than itself (Line 25), then it becomes a leader (Line 26), since the goal is for the leaders to be the lowest-stage processes.

To eventually prevent unstable processes from becoming leaders, the initial timeout of a process recovering from a crash is set to its stage (Line 37). This way, it is guaranteed that its timeout will grow indefinitely for unstable processes so, eventually, the condition of Line 22 will not hold anymore.

3.4 Correctness of $\mathcal{A}_{A\Omega'}$ in PS

Since System PS has finite omissions, there is a time t_1 after which there are no omissions in the system. Besides, there is a time t_2 after which correct processes do not crash anymore. There is also a time t_3 after which no more processes crash permanently. Finally, since links are eventually timely, there is a time t_4 such that, after that time, every message sent between correct processes is delivered within t_d (unknown but finite) units of time. Let $t_s = \max(t_1, t_2, t_3, t_4)$. Two direct properties of Algorithm $\mathcal{A}_{A\Omega'}$ can be stated.

Property 1 *After time t_s , variables $STAGE_i$ and, consequently, $stage_i$ do not grow for any correct process p_i .*

Property 2 *After time t_s , variables $STAGE_i$ and, consequently, $stage_i$ grow indefinitely for every unstable process p_i .*

Lemma 1 *Variable $timeout_i$ of a correct leader p_i does not grow indefinitely.*

Proof Since links are eventually timely, after time t_s , every message sent between correct processes is delivered within t_d units of time. A correct leader only increases $timeout_i$ when every message considered in Line 11 has $stage = stage_i$ and $r < r_i$. However, after time t_s , every message sent from p_i takes at most t_d units of time. Hence, once $timeout_i > t_d$, for each subsequent round, the message sent (by p_i to itself) at Line 9 will be considered in Line 11. Hence, the condition of Line 13 will not hold anymore for process p_i . Hence, $timeout_i$ will not grow indefinitely for a correct leader. $\square$

Lemma 2 *Variable $timeout_i$ of an unstable process p_i grows indefinitely.*

Proof Note that $STAGE_i$ grows indefinitely from Property 2. Since $timeout_i$ is set to $STAGE_i$ (Lines 36–37) each time a process recovers from a crash and, in Algorithm $\mathcal{A}_{A\Omega'}$, $timeout_i$ is never decreased, then $timeout_i$ will grow indefinitely for all unstable process p_i . $\square$

Lemma 3 *Eventually, at least one correct process becomes a leader.*

Proof By the way of contradiction, let us assume that there is a time after which no correct process is a leader anymore. Then, every correct process will be executing Lines 19–28 in every iteration of the loop in Task 1. However, from Properties 1 and

2, eventually, for all correct process p_i , $stage_i < stage_k$ for all unstable process p_k . Hence, if there is an unstable leader, eventually, the condition of Line 25 will hold for process p_i , and it will become a leader. If there is no leader sending messages, the condition of Line 22 will hold for process p_i , and it will also become a leader, what contradicts the initial assumption and completes the proof. $\square$

Lemma 4 *Eventually, no unstable process becomes a leader anymore.*

Proof Note that, when recovering from a crash, a process is not a leader (see Line 38). Thus, the only way for it to become a leader is that the condition of Line 22 or the condition of Line 25 holds. From Properties 1 and 2, eventually, for all unstable process p_u and correct process p_c , $stage_u > stage_c$. Hence, eventually, the condition of Line 25 will not hold anymore. Let us consider now the condition of Line 22. From Lemma 1, the timeout of a correct leader does not grow indefinitely. However, from Lemma 2, the timeout of an unstable process grows indefinitely. Besides, from Lemma 3, eventually, at least one correct process becomes a leader. Then, the time spent at Line 20 by an unstable process will be sufficient to receive some message from a correct leader. Thus, eventually, the condition of Line 22 will not hold anymore for an unstable process. Hence, there is a time after which no unstable process becomes a leader anymore. $\square$

Corollary 1 *Eventually, every leader process is correct.*

Lemma 5 *There is a time after which all the leaders are at the minimum stage and maximum round.*

Proof Since, from Corollary 1, there is a time after which all leaders are correct, let us assume, by the way of contradiction, that there are two (correct) leaders at different stages after that time and they remain up as leaders and not omitting forever after. Then, they both must be executing Lines 7–19 at each iteration of the loop. Recall that, after time t_s , all messages sent between correct processes are delivered within t_d units of time. Hence, both correct leaders will eventually receive the other's messages. Therefore, the condition of Line 16 will hold for the one with the highest stage, and it will stop being a leader (Line 17), what contradicts the initial assumption. Hence, eventually, all leaders must be at the same (minimum) stage. Let us focus now on the round. Analogously to the case of the stages, if a process detects another leader in a higher round (within the same stage) at Line 16, it will stop being a leader. Hence, all (correct) leaders must eventually be at the minimum stage and maximum round. $\square$

Since, eventually, all leaders must be at the same stage and round, if their relative speeds are not the same, there will be a time where they will not be at the same round and the one at the smaller round will stop being a leader. Thus, eventually, there will be only one leader, or all the leaders will be permanently synchronized.

Lemma 6 *There is a time after which, for all $p_i \in \Pi$, every subsequent call to $leader_i()$ returns the same value.*

Proof Note first that crashed processes never call $leader_i()$. Besides, from Lemma 4, eventually, every unstable process is not a leader anymore, so there is a time after

which every call to $leader_i()$ for an unstable process returns false. Let us focus now on the case of correct processes. From Lemma 5, there is a time after which all the leaders are at the same (minimum) stage and maximum round, i.e., only the fastest (and synchronized) leaders eventually remain as leaders. Once they are the unique leaders in the system, they remain like that forever after since the condition of Line 16 will not hold anymore. Thus, eventually, every subsequent call to $leader_i()$ will return true for all those processes. Conversely, the processes which stopped being leaders because there were faster leaders will not become leaders anymore once they have a sufficiently large timeout since neither the condition of Line 22 nor that of Line 25 will be satisfied anymore. Thus every subsequent call to $leader_i()$ will return false for all these processes. $\square$

Lemma 7 *There is a time after which, for any two processes $p_i, p_j \in \Pi$, if $leader_i() = leader_j() = \text{true}$, then $quantity_i() = quantity_j()$ for all subsequent calls.*

Proof From Lemma 5, there is a time after which all the leaders are at the same stage and round. As commented, there can be only one leader or various synchronized leaders. In either case, the leaders will be able to receive all the messages from their round since their timeout will be bigger than t_d . Hence, for any pair of leaders p_i, p_j , there is a time after which every call to $quantity_i()$ and $quantity_j()$ return the same value. $\square$

Theorem 1 *Algorithm $\mathcal{A}_{A\Omega'}$ implements the $A\Omega'$ failure detector in System PS where $f < n$.*

Proof Direct from Lemmas 6 and 7. $\square$

4 Consensus in the anonymous asynchronous system

We introduce in this section the algorithm $\mathcal{A}_C$, which implements Consensus in anonymous asynchronous systems with omissions and crash-recover failures augmented with the failure detector A' , and where $f < n/2$ (see Figs. 2, 3, 4 and 5). We prove in a formal way that $\mathcal{A}_C$ in this system preserves the properties of Consensus, and finish this section analyzing its performance.

4.1 Definition of consensus

Each process p_i proposes its own value v_i , which is initially unknown to any other process. Consensus is achieved if the following three properties hold [1]:

1. *Validity* The decided value is one of the proposed values.
2. *Termination* Every correct process eventually decides.
3. *Agreement* If two processes decide, they decide the same value.

4.2 The anonymous asynchronous system AS

As previously mentioned, it is impossible to solve Consensus in a fully asynchronous system. Thus, a partially synchronous Consensus algorithm could be developed. However, it is much simpler to move all the time assumptions to an independent module, called a failure detector, which hides all the synchrony from the Consensus algorithm.

To solve Consensus with a failure detector as independent modules, we have to limit the number of omissions for eventually-up processes. Thus, in AS (as in PS) the number of omissions of eventually-up processes is finite but unknown.

We say that AS is a system such as S but asynchronous. Therefore, the time to execute a step of a process and the delay time to receive a message through a reliable link are unknown and unbounded.

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

Consequently, to circumvent the impossibility to solve consensus in totally anonymous asynchronous systems [2], we consider that AS is an asynchronous system augmented with the failure detector A' .

4.3 The algorithm $\mathcal{A}_C$ in AS with $A\Omega'$

Algorithm $\mathcal{A}_C$ is structured as a set of tasks that run in parallel and are coordinated to avoid concurrency problems. The read and write operations on variables stored in stable memory are assumed to be atomic. When *consensus()* is invoked, it executes the *init* part in Fig. 2, which sets the initial values of the local variables and starts tasks 1 (Fig. 3), 2 (Fig. 4) and 3 (Fig. 5). These tasks are used to achieve consensus. When a process recovers from a crash, it executes Lines 5–14 in Fig. 2. Task 4 (Fig. 5) is used to permanently advert the decided value (once a decision has been made).

Communication among processes is performed by means of a broadcasting primitive. Messages are not altered during transmission and they are delivered provided that the sender and receiver are not crashed during transmission as specified in System AS . It is also assumed, in AS , that a process receives all the messages it broadcasts.

An $A\Omega'$ failure detector is used to determine which processes are considered leaders at each moment. Only proposals from leader processes are considered to reach consensus. The main consensus procedure works in rounds divided in phases. In order to allow crashes and recoveries from processes, the algorithm periodically broadcasts old messages. Thus, when a process recovers, it can receive the messages it missed from previous rounds to catch up with the other processes. Besides, it keeps its *STATUS* in stable memory to know at which round and phase it was before the crash, its current estimation of the value to be agreed and whether that value was supported by a sufficient amount of processes or not (in case it had passed the second phase).

Since the system is anonymous, there is no way to distinguish two identical messages sent by the same process from two identical messages sent by different processes. Hence, it is necessary to devise a way to identify the messages that form part of a certain round and make sure that only one message per process, round and phase is considered. In order to identify sets of messages in a round that come from different processes, each message is provided with a tag. These tags are locally generated. Processes keep

```

function Consensus( $v_i$ ):
  init:
    (1)  $r_i \leftarrow 1$ ;  $est_i[1][ph1] \leftarrow v_i$ ;  $passed_i \leftarrow \emptyset$ ;  $accepted_i[1] \leftarrow \text{false}$ ;
    (2)  $STATUS_i \leftarrow \langle r_i, est_i, passed_i, accepted_i \rangle$ ; % stable storage variable
    (3)  $TAGS_i \leftarrow \emptyset$ ; % stable storage variable
    (4) start Tasks 1, 2 and 3.

  upon recovering of process  $p_i$  do
    (5)  $r_i \leftarrow STATUS_i.r$ ;
    (6)  $est_i \leftarrow STATUS_i.est$ ;
    (7)  $passed_i \leftarrow STATUS_i.passed$ ;
    (8)  $accepted_i \leftarrow STATUS_i.accepted$ ;
    (9) if ( $ph3 \in passed_i$ ) then
      (10) start Task 4;
      (11) return ( $est_i[r_i][ph3]$ );
    (12) else
      (13) start Tasks 1, 2 and 3;
    (14) end if.

```

Fig. 2 Startup and recovery of the Algorithm $\mathcal{A}_C$ (code for process p_i)

track of the tags they use in variable $TAGS$ stored in stable memory, and make sure to send only one message per round, phase and tag. New tags are generated by adding one to the highest tag stored in $TAGS$. This way, it will be possible to enforce that the set of messages for some round, phase and tag does not have two messages from the same process.

Before sending a message, the process tests if it has already recorded a triplet with the same round, phase and tag of the message to be sent in $TAGS$. Only if it has not previously sent such a message, the new message is sent (see, for example, Lines 5–8 Fig. 3). Prior to sending the message, the triplet is recorded to ensure that no other such message will be sent in the future. Note that, if the process crashes before sending the message, that message will never be sent again. However, that is not a problem for the algorithm. If the process recovers, then it will periodically send new messages for that round and phase with newly generated tags.

When a process is waiting for a condition to hold, it is assumed that the waiting is active and the condition is permanently evaluated until it reaches true. For example, at Line 4 Fig. 4, the leadership condition determined by the failure detector, as well as the amount of leaders in the system are constantly evaluated during the wait (the second condition is only evaluated if the first is false, i.e., the process is a leader).

When a process recovers from a crash, it reloads the content of variable $STATUS$ from stable storage into volatile storage variables (Lines 5–8 in Fig. 2). If a decision was already made, i.e. $ph3$ was reached, then Task 4 is started and the decided value is returned (Lines 10–12 in Fig. 2). Otherwise, tasks 1, 2 and 3 are started (Line 13 in Fig. 2). It is assumed that, if a return operation was previously executed, this new return operation has no effect (provided that the value returned is the same, i.e., the algorithm is correct).

Task 1:

```

(1) repeat each  $\eta$  time
(2)   let last be the highest tag in  $TAGS_i$ ;
(3)    $tag_i \leftarrow last + 1$ ;
(4)   for  $k = 1$  to  $r_i$  do
(5)     if  $(D.leader_i()) \wedge \langle \text{NOTIFY}, k, tag_i \rangle \notin TAGS_i$  then
(6)        $TAGS_i \leftarrow TAGS_i \cup \{\langle \text{NOTIFY}, k, tag_i \rangle\}$ ;
(7)        $broadcast(\text{NOTIFY}, k, est_i[k][ph1], tag_i)$ ;
(8)     end if;
(9)     if  $((k < r_i \vee ph1 \in passed_i) \wedge \langle \text{VERIFY}, k, tag_i \rangle \notin TAGS_i)$  then
(10)       $TAGS_i \leftarrow TAGS_i \cup \{\langle \text{VERIFY}, k, tag_i \rangle\}$ ;
(11)       $broadcast(\text{VERIFY}, k, est_i[k][ph2], tag_i)$ ;
(12)    end if;
(13)    if  $((k < r_i \vee ph2 \in passed_i) \wedge \langle \text{COMMIT}, k, tag_i \rangle \notin TAGS_i)$  then
(14)       $TAGS_i \leftarrow TAGS_i \cup \{\langle \text{COMMIT}, k, tag_i \rangle\}$ ;
(15)       $broadcast(\text{COMMIT}, k, est_i[k][ph3], accepted_i[k], tag_i)$ ;
(16)    end if;
(17)  end for;
(18) forever.

```

Fig. 3 Task 1 of the Algorithm $\mathcal{A}_C$ (code for process p_i)

Task 1 (Fig. 3) is responsible for periodically sending the necessary messages. For each past round, it sends one message per phase with the information available at that round (unless a message for the same round, phase and tag was previously sent). This way, the actions taken at that round by slow or crashing processes will be based on the same information the faster processes had at that round. For the current round, messages are sent in case the previous phases have been already passed by the process.

To ensure that each process receives a large enough set of messages from different processes in each phase and round, Task 3 (Fig. 5) ensures to send the corresponding message for each round and phase with all the tags generated by all processes for that round, unless the corresponding triplet is already stored in $TAGS$. It is assumed that the *upon reception* clauses in Fig. 5 are executed in mutual exclusion. Furthermore, it is also assumed that they are executed in mutual exclusion with Lines 5–16 Fig. 3. Thus, it can be guaranteed that a process never sends two messages for the same round and phase with the same tag and every triplet is stored in $TAGS$ only once.

Task 2 (Fig. 4) performs the basic consensus procedure. Rounds are performed until consensus is reached. Each round consists of three phases:

- In the first phase (Lines 2–21 Fig. 4), the leaders make proposals and wait to receive the proposals from all the leaders. Recall that, all the messages with the same tag come from different processes. Hence, if five messages are received, then they are proposals from five different leaders. Among the proposals received, they make a deterministic choice (to guarantee that every process makes the same election based on the same set of messages). In this case, the minimum value of those proposed is chosen, but any other deterministic choice would be valid. Since non-leaders do not know how many leaders there are, they wait until a leader gets to the next phase and assumes the same value that was assumed by that leader. In

Task 2:

```

(1) repeat
(2)   if ( $ph1 \notin passed_i$ ) then
(3)     if ( $D.leader_i()$ ) then
(4)       wait until ( $\neg D.leader_i() \vee \exists tag : D.quantity_i() = |\{(NOTIFY, r_i, -, tag) \text{ received}\}|$ );
(5)       let Received be the set of (NOTIFY,  $r_i, -, tag$ ) messages considered in previous line;
(6)       if ( $|Received| > 0$ ) then
(7)          $est_i[r_i][ph2] \leftarrow \min est : (NOTIFY, r_i, est, tag) \in Received$ ;
(8)       else
(9)          $est_i[r_i][ph2] \leftarrow est_i[r_i][ph1]$ ;
(10)      end if;
(11)    else
(12)      wait until ( $D.leader_i() \vee (VERIFY, r_i, -, -) \text{ received}$ );
(13)      if ( $(VERIFY, r_i, -, -) \text{ received}$ ) then
(14)         $est_i[r_i][ph2] \leftarrow$  the estimation of the message received in Line 12;
(15)      else
(16)         $est_i[r_i][ph2] \leftarrow est_i[r_i][ph1]$ ;
(17)      end if;
(18)    end if;
(19)     $passed_i \leftarrow \{ph1\}$ ;
(20)     $STATUS_i \leftarrow \langle r_i, est_i, passed_i, accepted_i \rangle$ ;
(21)  end if;
(22)  if ( $ph2 \notin passed_i$ ) then
(23)    wait until  $\exists tag : n - f = |\{(VERIFY, r_i, -, tag) \text{ received}\}|$ ;
(24)    let Received be the set of (VERIFY,  $r_i, -, tag$ ) messages considered in previous line;
(25)     $est_i[r_i][ph3] \leftarrow \min est : (VERIFY, r_i, est, tag) \in Received$ ;
(26)     $accepted_i[r_i] \leftarrow (\forall est : (VERIFY, r_i, est, tag) \in Received, est = est_i[r_i][ph3])$ ;
(27)     $passed_i \leftarrow passed_i \cup \{ph2\}$ ;
(28)     $STATUS_i \leftarrow \langle r_i, est_i, passed_i, accepted_i \rangle$ ;
(29)  end if;
(30)  wait until  $\exists tag : n - f = |\{(COMMIT, r_i, -, -, tag) \text{ received}\}|$ ;
(31)  let Received be the set of (COMMIT,  $r_i, -, -, tag$ ) messages considered in previous line;
(32)  if ( $\forall (COMMIT, r_i, est, accepted, tag) \in Received, est = est_i[r_i][ph3] \wedge accepted = \text{true}$ ) then
(33)     $passed_i \leftarrow \{ph3\}$ ;
(34)  else
(35)    if ( $\neg accepted_i[r_i] \wedge \exists est : (COMMIT, r_i, est, \text{true}, tag) \in Received$ ) then
(36)       $est_i[r_i + 1][ph1] \leftarrow est$ ;
(37)    else
(38)       $est_i[r_i + 1][ph1] \leftarrow est_i[r_i][ph3]$ ;
(39)    end if;
(40)     $r_i \leftarrow r_i + 1$ ;
(41)     $passed_i \leftarrow \emptyset$ ;
(42)  end if;
(43)   $STATUS_i \leftarrow \langle r_i, est_i, passed_i, accepted_i \rangle$ ;
(44) until ( $passed_i = \{ph3\}$ );
(45) start Task 4;
(46) return ( $est_i[r_i][ph3]$ ).

```

Fig. 4 Task 2 of the Algorithm $\mathcal{A}_C$ (code for process p_i)

case a process stops being a leader while it was waiting for the leaders' proposals, if it has already received any messages during the wait, it assumes the minimum of the proposals it has received, otherwise it keeps its own proposal. In case a process becomes a leader while it was waiting for the verified choice of a leader, it keeps its own estimation for the next phase. In any case, the process updates its *STATUS* (in stable memory) before getting to the next phase. In case the leadership state of the process changes during the wait and the *wait until* operation ends without receiving any message, then the process keeps its own estimation for the next phase.

- In the second phase (Lines 22–29 Fig. 4), each process waits for $(n - f)$ VERIFY messages (which are sent by Tasks 1 and 3) with the same tag. This condition will be satisfied when there is a sufficient number of non-crashed processes that respond with the same tag. Although there are, potentially, many different subsets of $(n - f)$ messages in a set of n messages, as it will be proved, eventually, the minimum of the proposals of the leaders will be the value sent by all non-crashed processes in the VERIFY messages. Thus, $accepted_i[r_i]$ will eventually be set to true. If the failure detector is not accurate yet, then $accepted_i[r_i]$ may be set to false and new rounds may be necessary to achieve consensus. The estimation for the next phase is chosen as the minimum of the proposals received and, as in the previous phase, the *STATUS* is recorded before proceeding to the next phase.
- In the third phase (Lines 30–43 Fig. 4), processes wait until they receive a sufficient number of COMMIT messages from different processes. If $(n - f)$ messages have the *accepted* field set to true, then that value is decided. Hence, it is returned and Task 4 is started to permanently notify the decision made (Lines 45–46 Fig. 4). If there is, at least, one message with *accepted* set to true, then that value is set as the estimation for the first phase of the next round; otherwise, this process keeps its own estimation for the next round. If a decision has not been made, then the process updates its *STATUS* and proceeds to the next round.

Note that the estimation for each round and phase is kept in *STATUS* to guarantee that the values exchanged by processes at each round and phase are the same no matter how many times they are broadcast and how many times a process crashes.

Task 3 (Fig. 5) responds to the reception of messages from other processes in order to provide a sufficient number of messages from different processes for each round and phase to allow the processes to satisfy their waiting conditions. Note that, when a process sends a message in Task 1, it has previously stored the corresponding triplet in *TAGS*, so Task 3 will only send new messages for tags generated by other processes. Additionally, it responds to the reception of a DECISION message by updating the current estimation, updating the *STATUS* of the process, returning the decided value and starting Task 4. It is assumed that, when the code of the *upon reception* clause corresponding to the reception of the DECISION message starts being executed, Task 2 is interrupted. Thus, if Line 19 Fig. 5 is executed, i.e., the process does not crash before executing that line, then Task 2 will not be restarted anymore. However, if the process crashes before executing Line 19, then, if the process recovers, it will restart Task 2 upon recovering.

4.4 Correctness of $\mathcal{A}_C$ in AS with $A\Omega'$

In this section, we prove that Algorithm $\mathcal{A}_C$ correctly solves Consensus in System AS when provided with an $A\Omega'$ failure detector.

Let us start with some basic properties of the algorithm which are essential in the correctness of the algorithm.

Lemma 8 *For each type, round and tag, each process sends at most one message.*

Task 3:

```

(1) upon reception of (NOTIFY,  $r$ ,  $-$ ,  $tag$ ) :  $r \leq r_i \wedge D.leader_i()$  do
(2)   if ( $\langle \text{NOTIFY}, r, tag_i \rangle \notin TAGS_i$ ) then
(3)      $TAGS_i \leftarrow TAGS_i \cup \{\langle \text{NOTIFY}, r, tag_i \rangle\}$ ;
(4)      $broadcast(\text{NOTIFY}, r, est_i[r][ph1], tag)$ ;
(5)   end if.
(6) upon reception of (VERIFY,  $r$ ,  $-$ ,  $tag$ ) :  $r < r_i \vee r = r_i \wedge ph1 \in passed_i$  do
(7)   if ( $\langle \text{VERIFY}, r, tag_i \rangle \notin TAGS_i$ ) then
(8)      $TAGS_i \leftarrow TAGS_i \cup \{\langle \text{VERIFY}, r, tag_i \rangle\}$ ;
(9)      $broadcast(\text{VERIFY}, r, est_i[r][ph2], tag)$ ;
(10)  end if.
(11) upon reception of (COMMIT,  $r$ ,  $-$ ,  $-$ ,  $tag$ ) :  $r < r_i \vee r = r_i \wedge ph2 \in passed_i$  do
(12)  if ( $\langle \text{COMMIT}, r, tag_i \rangle \notin TAGS_i$ ) then
(13)     $TAGS_i \leftarrow TAGS_i \cup \{\langle \text{VERIFY}, r, tag_i \rangle\}$ ;
(14)     $broadcast(\text{COMMIT}, r, est_i[r][ph3], accept_i[r], tag)$ ;
(15)  end if.
(16) upon reception of (DECISION,  $dec$ ) do
(17)   $est_i[r_i][ph3] \leftarrow dec$ ;
(18)   $passed_i \leftarrow \{ph3\}$ ;
(19)   $STATUS_i \leftarrow \langle r_i, est_i, passed_i, accepted_i \rangle$ ;
(20)  start Task 4;
(21) return ( $est_i[r_i][ph3]$ ).

```

Task 4:

```

(22) stop Tasks 1, 2 and 3;
(23) repeat  $broadcast(\text{DECISION}, est_i[r_i][ph3])$  forever.

```

Fig. 5 Tasks 3 and 4 of the algorithm $\mathcal{A}_C$ (code for process p_i)

Proof Note that, before a message is sent, the corresponding triplet $\langle type, round, tag \rangle$ is previously stored in stable memory (variable $STATUS$) and a message is never sent if the corresponding triplet is already present in that variable, as it can be seen in Task 1 (Fig. 3) and Task 3 (Fig. 5). $\square$

Lemma 9 For all round r of Task 2 (Fig. 4), for all process p_i , $est_i[r][ph3] = est_j[r][ph1]$ for some process p_j .

Proof The value of $est_i[r][ph3]$ is set at Line 25 Fig. 4 with the value received in a VERIFY message sent by some process p_j at Line 11 Fig. 3 or at Line 9 Fig. 5. In both cases, the value sent comes from $est_j[r][ph2]$ (recall that messages are not altered in transmission). This value must have been set at Lines 7, 9, 14 or 16 Fig. 4. In the case of Lines 9 and 16 Fig. 4, it is set with the value of $est_j[r][ph1]$. In the case of Line 7 Fig. 4, the value comes in a NOTIFY message sent by some process p_j at Line 7 Fig. 3 or at Line 4 Fig. 5 and corresponds to $est_k[r][ph1]$ for some process k . In the case of Line 14 Fig. 4, the process which sent that message must have executed Line 7, 9, 14 or 16 Fig. 4 before Line 19 Fig. 4, i.e., before proceeding to the second phase. Hence, for all process p_i , $est_i[r][ph3] = est_j[r][ph1]$ for some process p_j . $\square$

Lemma 10 For all round $r > 1$ of Task 2 (Fig. 4), for all process p_i , $est_i[r][ph1] = est_j[r-1][ph3]$ for some process p_j .

Proof Note that, $est_i[r+1][ph1]$ is set at Line 38 or Line 36. In the first case, it is set to $est_i[r][ph3]$. In the second case, it is set to the value that comes in a COMMIT

sent at Line 15 Fig. 3 or Line 14 Fig. 5. In both cases, the value sent is $est_j[r][ph3]$ for some process p_j . $\square$

Lemma 11 *For all round r of Task 2 (Fig. 4), for all process p_i , $est_i[r][ph3] = v_j$ for some process p_j .*

Proof By simple induction on the round number: It is easy to see that, in the first round, $est_i[1][ph1] = v_i$ for all process p_i (see Line 1 Fig. 2). Hence, from Lemma 9, $est_i[1][ph3] = v_k$ for some process p_k . Assume that the initial assumption holds for round r , i.e., for all process p_i , $est_i[r][ph3] = v_j$ for some process p_j . Then, from Lemma 10, $est_i[r+1][ph1] = est_j[r][ph3]$ for some process p_j . Again from Lemma 9, $est_i[r+1][ph3] = est_j[r+1][ph1]$ for some process p_j . Hence, $est_i[r+1][ph3] = v_j$ for some process p_j , what completes the proof. $\square$

Lemma 12 *Validity: The decided value is one of the proposed values.*

Proof It is assumed that a process decides v when it executes `return v`. Hence, a decision can be made at Line 46 Fig. 4, Line 11 Fig. 2, or Line 21 Fig. 5. In all cases, the value decided is that stored in $est_i[r_i][ph3]$. If the decision is made in Task 2 (Fig. 4), then, from Lemma 11, the value decided is a value proposed by some process. If the decision is made in Task 3 (Fig. 5), then the decided value arrives in a DECISION message sent by some process at Line 23 of Task 4 (Fig. 5). There, the value sent is the estimation for the third phase which, from Lemma 11, is a value proposed by some process. In case the decision is made during recovery (Fig. 2), the estimation has been restored from stable memory, which was saved in Task 2 at Line 43 Fig. 4, but the process crashed before executing Line 46. In this case, again, the decided value is one of the proposed values, from Lemma 11. $\square$

Lemma 13 *If a correct process decides, then every correct process eventually decides.*

Proof If a correct process decides, it executes Task 4 (see Lines 10–11 Fig. 2, Lines 45–46 Fig. 4 and Lines 20–21 Fig. 5). Thus, eventually, every correct process will receive a DECIDE message (just in case it does not decide in Task 2) and will decide in Task 3 (see Lines 16–21 Fig. 5). Recall that there is a time after which correct processes do not crash anymore and messages are not omitted. $\square$

Note that there is a time after which the failure detector stabilizes, i.e., each process p_i permanently gets the same result from every subsequent invocation to $D.leader_i()$ and every process that gets true in such invocation gets the same result when invoking $D.quantity_i()$. Let t_s be the time after which correct processes do not crash anymore, do not make omissions and the failure detector is stable.

Lemma 14 *If no process has decided yet, every wait until sentence executed by Task 2 (Fig. 4) ends within a finite time.*

Proof Recall that the sequence of tags generated by a process is monotonically increasing. After time t_s , this sequence is strictly monotonically increasing since the process does not crash. Let t be the highest tag among the correct processes which start a new

iteration of the loop of Task 1 after time t_s . This tag has not been previously used by any of the correct processes and all the messages sent with this tag will be received by all correct processes. Hence, every correct process will respond to them in Task 3 (Fig. 5) unless they obtained this tag in Task 1 (Fig. 3). In any case, it is guaranteed that every correct process will send one message with that tag, either in Task 1 or Task 3, for all rounds and phases up to their current one (with the exception of the first phase of each round where only leaders send NOTIFY messages). Since not all correct processes need to be at the same round, consider the lowest such round. All the processes which are at that round will receive the enough messages to continue when they wait at Lines 4,12,23 and 30 Fig. 4. Thus, they will proceed to the following round. Since Task 1 periodically generates new tags and sends new messages, this argument can be applied to every round at which there are correct processes, while no process decides. $\square$

Consider the first new round r_s started by a process after time t_s . We will prove now that, if a decision is not made before, it will be made at this round.

Lemma 15 *If no process has previously decided while executing Task 2 (Fig. 4), at round r_s , every leader process p_i sets the same value to $est_i[r_s][ph2]$ at Line 7 Fig. 4.*

Proof From Lemma 14, the time spent by a process at every *wait until* sentence is finite. Assuming that no process has decided before, at round r_s , $D.leader_i()$ returns the same value permanently for each process p_i and the value returned by $D.quantity_i()$ is the same for every process to which $D.leader_i()$ returns true. Hence, all leader processes will get the same set of messages in $Received_i$. Recall that all the leaders are correct, do not crash anymore and do not make omissions after time t_s . Thus, when they reach Line 7, all of them will choose the same value to store in $est_i[r_s][ph2]$. $\square$

Lemma 16 *If no process has previously decided while executing Task 2 (Fig. 4), at round r_s , every non-leader process p_i sets the same value to $est_i[r_s][ph2]$ at Line 14 Fig. 4.*

Proof Recall that the value returned by $D.leader_i()$ does not change after time t_s . Hence, every non-leader process will execute Line 14. The VERIFY message that allows a non-leader process to stop waiting at Line 12 Fig. 4 must be sent by some leader p_k . Since these messages, sent in Task 1 (Fig. 3) and Task 3 (Fig. 5), carry the value in $est_k[r_s][ph2]$, which has been set at Line 7 Fig. 4, from Lemma 15, all of them have the same value. Hence, all non-leader processes will set $est_i[r_s][ph2]$ to the same value. $\square$

Lemma 17 *Termination: Every correct process eventually decides.*

Proof Assume, by the way of contradiction, that no process is able to decide. Then, the condition of Line 32 Fig. 4 does not hold in any round for any process. Hence, every process receives at least one COMMIT message at Line 30 Fig. 4 with *accepted* set to false in every round. Therefore, at least one process must see at least two different estimations in the received at Line 23 Fig. 4. However, from Lemmas 15 and 16, every non-crashed process that reaches round r_s gets to the second phase with the same

estimation. Then, all the VERIFY messages sent in round r_s have the same estimation and we reach a contradiction. Hence, at least one process decides and, from Lemma 13, all of them decide. $\square$

Lemma 18 *When executing Task 2 (Fig. 4), if a process decides v at round r , then no process may decide v' , $v \neq v'$ at round r .*

Proof When executing Task 2, if a process decides v (Line 46) at round r , then it must have received $(n - f)$ COMMIT messages with the same round r , the same tag (i.e., from different processes from Lemma 8), *accepted* set to true and the same proposed value v (see Line 32). If some other process decides v' at round r , then it must have received $(n - f)$ COMMIT messages with the same round r , the same tag (i.e., from different processes from Lemma 8), *accepted* set to true and the same proposed value v' (see Line 32). However, the estimation sent in the COMMIT messages is the one set at the second phase and it is never changed, once it is set at Line 25. Hence, those $2(n - f)$ messages must come from different processes. However, from the definition of the system, $(n - f) > n/2$ and we reach a contradiction. Hence, it is not possible to decide two different values at the same round of Task 2. $\square$

Lemma 19 *In Task 2 (Fig. 4), if a process sends a COMMIT message with value v and *accepted* set to true at round r , then no process may send a COMMIT message with value v' , $v' \neq v$ and *accepted* set to true at round r .*

Proof Recall that the estimation for one phase of a round is once set in the previous phase of that round or (in the case of the first phase) in the last phase of the previous round (or during initialization), and *accepted* _{i} [r_i] is only set at Line 26. If a process sends a COMMIT message with value v and *accepted* set to true, it must have received $(n - f)$ VERIFY messages from different (from Lemma 8) processes (see Lines 23–28 Fig. 4). Since it is a majority, there can not be any other majority with the same set of possible values. Hence, no process may have received $(n - f)$ VERIFY messages with value v' , $v' \neq v$ at round r and, then, no process may send a COMMIT message with value v' , $v' \neq v$ and *accepted* set to true. $\square$

Lemma 20 *If a process decides v at round r in Line 46 Fig. 4, all processes that proceed to the following round assume v as their estimation.*

Proof If a process decides v at round r , it must have received $(n - f)$ COMMIT messages from different (from Lemma 8) processes with the same proposal v and *accepted* set to true (see Lines 30–33 and Lines 44–46 Fig. 4). Since it is a majority, every process must receive, at this round and phase, at least one such message. From Lemma 19, if a process sent a COMMIT message with value v and *accepted* set to true, then no other process might send such a message with value v' , $v' \neq v$. Hence, every process that might not have v as its estimation, sets its estimation for the next round to v (see Lines 35–39 Fig. 4). $\square$

Lemma 21 *If at round r in Task 2 (Fig. 4), $est_i[r][phI] = v$ for all process p_i , then the only possible value decided in Task 2 at round r or later is v .*

Proof Recall that the value decided in Task 2 at round r is always a value stored in $est_i[r][ph3]$ for some process p_i (see Line 11 Fig. 2, Line 46 Fig. 4 and Line 21 Fig. 5). Let us apply induction on the round number. Since $est_i[r][ph1] = v$ for all process p_i , from Lemma 9, $est_i[r][ph3] = est_j[r][ph1]$ for some process p_j . Hence, only v can be decided at round r . From Lemma 10, for all process p_i , $est_i[r][ph1] = est_j[r-1][ph3]$ for some process p_j . Hence, at round $r+1$, $est_i[r+1][ph1] = v$ for all process p_i and, from Lemma 9, $est_i[r+1][ph3] = est_j[r+1][ph1]$ for some process p_j . Hence, only v can be decided at round r or later. $\square$

Lemma 22 *Agreement: If two processes decide, they decide the same value.*

Proof Note first that, prior to sending a DECIDE message, in Task 4 (Fig. 5), it is necessary that, at least, one process exits the main loop of Task 2 (Fig. 4). Without loss of generality, assume that the first process to exit that loop, exits at round r with value v in its estimation. From Lemma 18, if any other process decides at round r , it will decide v . If some process does not decide at round r , then, from Lemma 20, it must set its estimation to v for round $r+1$. Then, from Lemma 21, the only possible value decided in Task 2 at round r or later is v . If the decision is made in Task 3 (Lines 16–21 Fig. 5), then a DECISION message must have been sent by a process which has made the decision previously. Consider the first process which made the decision. This process must have decided in Task 2 and, in that Task 2, the only value that can be decided is v . If a process decides at recovering from a crash, it must have executed Line 43 Fig. 4 or Line 19 Fig. 5 with $passed_i = \{ph3\}$ previously. Therefore, the decision was already made on value v . Hence, if two processes decide, they decide the same value. $\square$

Theorem 2 *Algorithm $\mathcal{A}_C$ solves Consensus in AS with $A\Omega'$.*

Proof Direct from Lemmas 12, 17 and 22. $\square$

4.5 Complexity analysis

As is common in the literature, to be able to compare different Consensus algorithms, we focus on executions where no failures occur, i.e., no process crashes and the failure detector makes no mistakes. For this type of runs, Hurfin et al. [8] and Oliveira et al. [31] present solutions which reach consensus in one round formed by two communication steps, while that of Aguilera et al. [7] needs one more communication step. For the same scenario, our algorithm also reaches consensus in one round, which consists of three communication steps. As previously mentioned, all previous works use the rotating coordinator paradigm based on the existence and knowledge of process' identifiers.

Roughly speaking, the procedure used by Hurfin et al. [8] and Oliveira et al. [31] to reach consensus in one round with two communication steps works as follows. In phase 1, the coordinator p_c sends one message with its proposal to all processes, and each process responds by resending that proposal to all the processes. Then, phase 2 starts and each process sends a message with the decision made to all the processes. Hence, each phase needs n^2 messages. The algorithm of Aguilera et al. [7] needs three communications steps for the same scenario: in phase 1, the coordinator p_c sends

one message asking for proposals to all processes, and each process responds to p_c with its own proposal. After processing the responses, phase 2 starts and p_c sends the coordinator proposal and waits until it receives an acknowledgment from all processes. Finally, in phase 3, p_c sends a message with the decision to all processes. Therefore, the total number of messages exchanged (considering the three phases) is $2n + 2n + n$. Note that, in our case, with anonymous processes we cannot use a coordinator because we cannot send a message to a single process.

Our Consensus algorithm, in a run where no process crashes nor omits any message, and the failure detector makes no mistakes, reaches consensus in one round as follows. We consider η to be large enough that the loop of Task 1 is executed only once per round. If it is executed k times per round, then the estimation of the number of messages would have to be multiplied by the constant factor k . In phase 1, each leader sends a message with its proposal to all processes, and every leader responds to these messages by sending their respective proposals. Since all the leaders receive the proposals from all leaders, all of them choose the same value v to be agreed. Hence, in phase 1 a total of $ln + l^2n$ messages are sent, being n the number of processes in the system and l the number of leaders. In phase 2, each leader broadcasts this value v to all processes. This allows non-leaders to proceed to phase 2. Thus, all non-leaders also broadcast this value v in phase 2. Then, each process responds to all these messages, sending their estimation v to all processes. Hence, the number of messages sent in phase 2 is $n + n^2$. Since, all processes propose the same value, all of them send the *COMMIT* message with *accepted* set to true in phase 3. Then, all of them respond to these messages with new broadcasts. Hence, the total number of messages of phase 3 is $n + n^2$. Thus, considering that $l = O(n)$, the total number of messages until consensus is achieved is $O(n^3)$. Recall that η induces only a constant factor.

Considering the size of the messages exchanged, all of them are of fixed size, so we can say that their size is $O(1)$.

5 Conclusions and future work

Anonymous systems are currently receiving a lot of attention mainly because the lack of identifiers in the processes allows to preserve privacy, which is a major property when we want to avoid impersonation attacks. This paper is intended to be a step forward in solving Consensus in anonymous systems where new adversaries, not previously considered in the literature, have been added, e.g., crash-recovery and omission failures. To our knowledge, all previously proposed consensus algorithms in anonymous distributed systems assume that a crashed process can never recover (the so-called crash-stop failure model) and, what is more, they do not consider omission failures, i.e., they assume that errors in input/output communication buffers never happen. Note that this latter model is also important because it allows us to solve practical security problems (e.g., the fair exchange and the secure multiparty computation). This extension of failure models allows us to design more realistic systems.

One of the main contributions of this paper is an agreement algorithm (called $\mathcal{A}_C$) to solve Consensus in anonymous asynchronous systems under the crash-recovery and omission failure models such that up to $f < n/2$ processes can be crashing

and recovering permanently. To overcome the impossibility result of Fischer, Lynch and Paterson [2], $\mathcal{A}_C$ needs the asynchronous system to be augmented with a failure detector. We have proved that $O(n^3)$ messages of size $O(1)$ are interchanged to reach Consensus in a run where there are no failures. This moderate complexity is due to the fact that processes do not have identifiers, which has two major consequences: (a) there cannot be a round coordinator, and (b) messages must be broadcast instead of being sent to specific processes.

The other main contributions of this paper are to adapt the definition of the $A\Omega'$ failure detector to the new failure models and an implementation (Algorithm $\mathcal{A}_{A\Omega'}$) of $A\Omega'$ for the case where $f < n$. This algorithm $\mathcal{A}_{A\Omega'}$ exhibits two major properties: communication-efficiency and latency-efficiency in the use of stable storage. $\mathcal{A}_{A\Omega'}$ is communication-efficient because there is a time after which only leaders broadcast messages. $\mathcal{A}_{A\Omega'}$ is also latency-efficient in its use of stable variables because its unique variable of this kind is an integer value which is used (i.e., read and written) only once each time a process recovers from a crash. It is used as a counter of the number of times that a process recovers from a crash.

For future work we are considering adding more powerful adversaries to the system. For example, it would be interesting to consider processes which may have transient or permanent malicious behavior. Another interesting line of research for future work is to study how anonymous Consensus can be used to solve security problems in a hybrid model with security modules [10,11].

Funding Open Access funding provided thanks to the CRUE-CSIC agreement with Springer Nature.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

References

1. Chandra TD, Toueg S (1996) Unreliable failure detectors for reliable distributed systems. *J ACM* 43(2):225–267
2. Fischer MJ, Lynch NA, Paterson MS (1985) Impossibility of distributed consensus with one faulty process. *J ACM* 32(2):374–382
3. Freiling FC, Guerraoui R, Kuznetsov P (2011) The failure detector abstraction. *ACM Comput Surv* 43(2):1–40
4. Raynal M (2009) Failure detectors for asynchronous distributed systems: an introduction. *Wiley Encyclop Comput Sci Eng* 2:1181–1191
5. Raynal M (2010) Communication and agreement abstractions for fault-tolerant asynchronous distributed systems. In: *Synthesis Lectures on Distributed Computing Theory*. Morgan & Claypool Publishers,
6. Chandra TD, Hadzilacos V (1996) The weakest failure detector for solving consensus. *J ACM* 43(4):685–722
7. Aguilera MK, Chen W, Toueg S (2000) Failure detection and consensus in the crash-recovery model. *Distrib Comput* 13(2):99–125

8. Hurfin M, Mostéfaoui A, Raynal M (1998) Consensus in asynchronous systems where processes can crash and recover. In: The seventeenth symposium on reliable distributed systems, SRDS 1998, West Lafayette, Indiana, USA, October 20–22, 1998, proceedings. pp 280–286. IEEE Computer Society
9. Perry kJ, Toueg S (1986) Distributed agreement in the presence of processor and communication faults. *IEEE Trans Software Eng* 12(3):477–482
10. Avoine G, Gärtner FC, Guerraoui R, Vukolic M (2005) Gracefully degrading fair exchange with security modules. In: Dal Cin M, Kaâniche M, Pataricza A (eds) Dependable computing - EDCC-5, 5th European dependable computing conference, Budapest, Hungary, April 20–22, 2005, Proceedings, vol 3463. Springer, Lecture notes in computer science, pp 55–71
11. Fort M, Freiling F, Penso LD, Benenson Z, Kesdogan D (2006) Trustedpals: Secure multiparty computation implemented with smart cards. In: Dieter G, Jan M, Andrei S (eds.), Computer Security: ESORICS 2006, 11th European symposium on research in computer security, Hamburg, Germany, September 18–20, 2006, Proceedings, vol 4189. Springer, Lecture notes in computer science, pp 34–48
12. Chang C-C, Lin C-Y, Lin K-C (2007) Simple efficient mutual anonymity protocols for peer-to-peer network based on primitive roots. *J Netw Comput Appl* 30(2):662–676
13. Chaum D (1981) Untraceable electronic mail, return addresses, and digital pseudonyms. *Commun ACM* 24(2):84–88
14. Chothia T, Chatzikokolakis K (2005) A survey of anonymous peer-to-peer file-sharing. In: Enokido T, Yan L, Xiao B, Kim D, Dai Y-S, Yang LT (eds.), Embedded and ubiquitous computing: EUC 2005 Workshops, EUC 2005 Workshops: UISW, NCUS, SecUbiq, USN, and TAUES, Nagasaki, Japan, December 6–9, 2005, Proceedings, vol 3823. Lecture notes in computer science. Springer, pp 744–755
15. Angluin D, Aspnes J, Eisenstat D, Ruppert E (2005) On the power of anonymous one-way communication. In: J.H. Anderson, G. Prencipe, R. Wattenhofer (Eds.), Principles of distributed systems, 9th international conference. OPODIS 2005, Pisa, Italy, December 12–14, 2005, Revised Selected Papers, 3974. Lecture notes in computer science. Springer, pp 396–411
16. Akyildiz IF, Su W, Sankarasubramaniam Y, Cayirci E (2002) Wireless sensor networks: a survey. *Comput Netw* 38(4):393–422
17. 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
18. Durresi A, Paruchuri V, Durresi M, Barolli L (2005) A hierarchical anonymous communication protocol for sensor networks. In: Yang LT, Amamiya M, Liu Z, Guo M, Rammig FJ (Eds.), Embedded and ubiquitous computing: EUC 2005, international conference EUC 2005, Nagasaki, Japan, December 6–9, 2005, proceedings. volume 3824 of Lecture notes in computer science. pp 1123–1132. Springer
19. Ould-Ahmed-Vall EM, Blough Douglas M, Heck-Ferri Bonnie S, Riley George F (2009) Distributed global ID assignment for wireless sensor networks. *Ad Hoc Netw* 7(6):1194–1216
20. Bonnet F, Raynal M (2013) Anonymous asynchronous systems: the case of failure detectors. *Distrib Comput* 26(3):141–158
21. Jiménez E, Arévalo S, Herrera C, Tang J (2015) Eventual election of multiple leaders for solving consensus in anonymous systems. *J Supercomput* 71(10):3726–3743
22. Bouzid Z, Travers C (2012) Brief announcement: anonymity, failures, detectors and consensus. In: Marcos KA (ed.), Distributed computing: 26th international symposium, DISC 2012, Salvador, Brazil, October 16–18, 2012. Proceedings
23. Attiya H, Welch JL (2004) Distributed computing: fundamentals, simulations, and advanced topics, 2nd edn. Wiley, Hoboken
24. Nancy A (1996) Lynch. Morgan Kaufmann, Distributed Algorithms
25. Bonnet F, Raynal M (2011) The price of anonymity: optimal consensus despite asynchrony, crash, and anonymity. *TAAS* 6(4):1–28
26. Bouzid Z, Travers C (2012). Anonymity, failures, detectors and consensus. Working paper or preprint, August
27. Cortiñas R, Freiling Felix C, Ghajar-Azadanlou M, Lafuente A, Larrea M, Lucia DP, Iratxe SA (2012) Secure failure detection and consensus in trustedpals. *IEEE Trans Dependable Secur Comput* 9(4):610–625
28. Delporte-Gallet C, Fauconnier H, FC Freiling (2005) Revisiting failure detection and consensus in omission failure environments. In: D Van Hung, Wirsing M (Eds.), Theoretical aspects of computing: ICTAC 2005, second international colloquium, Hanoi, Vietnam. October 17–21, 2005, Proceedings

29. Delporte-Gallet C, Fauconnier H, A Tielmann, Freiling FC, Kilic M (2009) Message-efficient omission-tolerant consensus with limited synchrony. In 23rd IEEE international symposium on parallel and distributed processing, IPDPS 2009, Rome, Italy, May 23–29, 2009. pp 1–8. IEEE
30. Freiling FC, Lambertz C, Majster-Cederbaum M (2009). Modular consensus algorithms for the crash-recovery model. In: 2009 International conference on parallel and distributed computing, applications and technologies. pp 287–292
31. Oliveira RC, Guerraoui R, Schiper A (1997). Consensus in the crash-recovery model. Technical report, 97-239, Ecole Polytechnique Fédérale, Département d'Informatique, Lausanne, Switzerland, August
32. Guerraoui R, Oliveira RC, Schiper A (1996) Stubborn communication channels. Département d'Informatique, Lausanne, Switzerland, Technical report, Ecole Polytechnique Fédérale
33. Jiménez E, Arévalo S, Fernández A (2006) Implementing unreliable failure detectors with unknown membership. *Inf Process Lett* 100(2):60–63
34. Larrea M, Martín C, Iratxe SA (2011) Communication-efficient leader election in crash-recovery systems. *J Syst Softw* 84(12):2186–2195
35. Martín C, Larrea M (2010) A simple and communication-efficient omega algorithm in the crash-recovery model. *Inf Process Lett* 110(3):83–87
36. Martín C, Larrea M, Jiménez E (2009) Implementing the omega failure detector in the crash-recovery failure model. *J Comput Syst Sci* 75(3):178–189
37. Fernández-Campusano C, Cortiñas R, Larrea M (2014) A performance study of consensus algorithms in omission and crash-recovery scenarios. In: 2014 22nd Euromicro international conference on parallel, distributed, and network-based processing. pp 240–243
38. Fernández-Campusano C, Larrea M, Cortiñas R, Raynal M (2016) A communication-efficient leader election algorithm in partially synchronous systems prone to crash-recovery and omission failures. In: Proceedings of the 17th international conference on distributed computing and networking, Singapore. January 4–7, 2016, pp 8:1–8:4. ACM
39. Fernández-Campusano C, Larrea M, Cortiñas R, Raynal M (2017) A distributed leader election algorithm in crash-recovery and omissive systems. *Inf Process Lett* 118:100–104
40. Aguilera MK, Delporte-Gallet C, Fauconnier H, Toueg S (2008) On implementing omega in systems with weak reliability and synchrony assumptions. *Distrib Comput* 21(4):285–314
41. Angluin D (1980) Local and global properties in networks of processors (extended abstract). In: E.M. Raymond, G. Seymour, A.B. Walter, J.L. Richard (Eds.), Proceedings of the 12th annual ACM symposium on theory of computing, April 28–30, 1980, Los Angeles, California, USA. pp 82–93. ACM
42. Attiya H, Snir M, Warmuth MK (1988) Computing on an anonymous ring. *J ACM* 35(4):845–875

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.