Noname manuscript No.
(will be inserted by the editor)

Fault-tolerant Broadcast in Anonymous Systems

Ernesto Jiménez · Sergio Arévalo · Jian Tang

Received: date / Accepted: date

Abstract The broadcast service spreads a message m among all processes of a distributed system, such that each process eventually delivers m. A basic broadcast service does not impose any delivery guarantee in a system with failures. Fault-tolerant broadcast is a fundamental problem in distributed systems that adds certainty in the delivery of messages when crashes can happen in the system.

Traditionally, the fault-tolerant broadcast service has been studied in classical distributed systems when each process has a unique identity. However, very recently have appeared new distributed systems, such as sensor networks, where unique identity is not always possible to be included in each sensor node (due to small storage capacity, reduced computational power, a huge number of elements to be identified, etc.).

In this paper we study the definition and implementability of the fault-tolerant broadcast service in anonymous asynchronous systems, that is, in asynchronous systems where all processes have the same identity, and, hence, they are indistinguishable (they may have the same code).

Keywords Distributed computing · fault-tolerance · reliable, uniform and atomic broadcast services · failure detector · anonymous distributed system.

This work has been partially funded by PROMETEO Project-SENESCYT, Ecuador, the Spanish Research Council (MICCIN) under project BigDataPaaS (TIN2013-46883), and by the Regional Government of Madrid (CM) under project Cloud4BigData (S2013/ICE-2894) cofunded by FSE & FEDER.

Ernesto Jiménez.
Universidad Politécnica de Madrid, Spain, and Prometeo researcher, EPN, Ecuador.
E-mail: ernes@etsisi.upm.es.

Sergio Arévalo and Jian Tang.
Universidad Politécnica de Madrid, Spain.
E-mail: sergio.arevalo@etsisi.upm.es, tjapply@gmail.com.

1 Introduction

One of the most important communication abstractions for distributed systems is the *broadcast* service. This abstraction sends a message to all the processes of the system. However, it does not impose any fault-tolerant property. So, if a sender process crashes while it is broadcasting a message m, the outcome of the delivery of m is not known a priori. To avoid this indeterminism in the delivery when processes may crash, several types of fault-tolerant broadcast services have been introduced. The most popular are *reliable broadcast* (RB), *uniform reliable broadcast* (URB) and *atomic broadcast* (AB) services.

In the distributed systems that we study in this paper, a process can fail by crashing (that is, it stops working permanently). Thus, a process either may crash or not. In the first case we say that this process is a *crashed process* (or *faulty process*), and in the latter case, a *correct process*. Furthermore, in this paper we focus our study in asynchronous systems, that is, in distributed systems where the execution time of processes, and the delivery time of sent messages are unbounded.

The reliable broadcast (RB) service ([22], [27]) requires that (a) each message m sent by a correct process must be delivered by every correct process, and (b) if a correct process delivers a message m, then each correct process also delivers m.

The uniform reliable broadcast (URB) service ([22], [27]) is another type of fault-tolerant broadcast service that imposes a stronger delivery property. The URB service requires that (a) each message m sent by a correct process must be delivered by every correct process, and (b) if a process delivers a message m, then each correct process also delivers m. Note that the URB service is stronger than the RB service because the case (a) is the same in both services, and the delivery in the case (b) of the URB service includes all processes (correct or not) instead of only correct processes of the RB service.

The atomic broadcast (AB) service ([22], [27]) is also a fault-tolerant broadcast service which establishes a total order in the delivery. Thus, the AB service requires that if a process delivers the message m before m', then there is no other process that delivers m' before m.

Many papers in the literature present the broadcasting primitives analyzing how hardware and software can work in concert on scalable multi-processor and also distributed systems, and show how these primitives can be used as building blocks for more complex parallel operations ([3], [4], [5], [6], [7], [8], [12], [13], [14], [34]). In these papers can be found a number of illustrative examples and applications for broadcasting and also fault-tolerance.

To our knowledge, all works that study the fault-tolerant broadcast services rely on distributed systems where processes are distinguishable because each one of them has a unique identity ([10], [11], [15], [20], [22], [25], [27], [33]). In this paper we base our study in anonymous systems. In an anonymous system 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).

Nevertheless, we can find in the literature several works addressing the problem of counting the size of a network where processes are anonymous and the network topology constantly changes ([23], [29], [30]). In these works failures are limited to links rather than processes.

Anonymous processes are common in some practical distributed systems, such as sensor networks, where a unique identity is not always possible to be included in each device (due to, for example, small storage capacity, reduced computational power, or a huge number of elements to be identified) ([1], [2], [32]). Another practical issue where anonymous processes are used is related with privacy (for example, to hide the user identity in a system) [21].

Our work Up to now, the fault-tolerant broadcast service has been studied in classical distributed systems when each process has a different identity. However, this is the first paper, to our knowledge, that is devoted to the broadcast service with fault-tolerant guarantees in anonymous asynchronous systems.

In this paper we present several algorithms to prove the possibility to implement different types of fault-tolerant broadcast services in anonymous asynchronous systems. In particular, an implementation for the RB service, and another for the URB service when a minority of processes can crash.

We also include in this paper the impossibility to implement the URB service in anonymous asynchronous systems when a majority of processes can crash. To circumvent this impossibility result, we present in this paper an algorithm that implements the URB service independently of the number of crashed processes. To achieve it, we enrich the anonymous asynchronous system with a failure detector. A failure detector [19] is an oracle (i.e., a distributed component) that processes can invoke to obtain information about crashed processes.

Finally, it is very well known in the literature the impossibility to implement the AB service in the classical asynchronous system prone to failures ([19], [24]), and, hence, also in its anonymous version. However in this paper we present an algorithm that implements the AB service in anonymous asynchronous systems. We have circumvented the impossibility results of [19] and [24] augmenting the anonymous asynchronous system with the *Consensus* component. Consensus ([26], [28]) is one of the most fundamental building blocks in fault-tolerant distributed computing. Informally, Consensus states that all processes have to decide a same value v, and this value v has to be proposed by some process of the system. We prove that the AB service is implementable in anonymous asynchronous systems adapting the solution proposed in [19] (we use the anonymous version of RB and Consensus as building blocks).

The aim of this paper is to analyze the feasibility of the main types of fault-tolerant broadcast services in anonymous systems. Thus, we try to present our algorithms as simple as possible to solve each service. Other considerations such as performance or efficiency are out of our paper's scope (it is open for a future work).

This paper is organized as follows. The anonymous system model is presented in Section 2. Definitions of fault-tolerant broadcast services in anonymous systems are included in Section 3. In Section 4 we include an implementation of the RB service in the anonymous asynchronous system. In Section 5 we prove that the URB service is impossible to be implemented in the anonymous asynchronous system when a majority of processes can crash. In Section 6 we study the implementability of the URB service in anonymous asynchronous systems. In particular, Section 6.1 includes an implementation of the URB service in the anonymous asynchronous system when a minority of processes can crash, and Section 6.2 presents an algorithm that circumvents the impossibility result of Section 5 by using a failure detector. Section 7 presents an implementation of the AB service in the anonymous asynchronous system augmented with Consensus, the RB and URB services. Finally, we finish our paper with the conclusion in Section 8.

2 The Anonymous System

The anonymous asynchronous system (denoted $AAS[\emptyset]$) is formed by a set of processes $\Pi = \{p_i\}_{i=1,\ldots,n}$ such that its size $|\Pi|$ is n, and i is the index of each process p_i, $1 \le i \le n$.

Processes are anonymous [18]. Hence, they have no identity, and there is no a way to differentiate between any two processes of the system (i.e., processes have no identifier, and execute the same code). So, anonymity implies that process indexes are fictitious in the sense that each process $p_i \in \Pi$ does not know its index i. We only use process indexes from an external observer point of view, and with the purpose of simplifying the notation.

A run R is formed by the set of steps taken by each process $p_i \in \Pi$. We assume that time advances at discrete steps in each run R, and there is a global clock T whose values are the positive natural numbers. Note that T is an auxiliary concept that we only use for notation, but that processes can not check or modify. Processes are *asynchronous*, that is, the time to execute a step by a process in a run R is unbounded.

When a process crashes it stops taking steps. We assume that a crashed process never recovers. A process $p_i \in \Pi$ is *correct* if it does not crash, and *faulty* if it crashes. Let $Correct$ be the set of correct processes, and let $Faulty$ be the set of faulty processes. We denote by f the maximum number of processes that may crash. Unless otherwise is stated, we consider that this maximum number is $n-1$ (i.e., $f \le n-1$).

In $AAS[\emptyset]$ processes communicate among them sending and receiving messages through links. Each pair of processes is connected by a link. We assume that links neither duplicate nor create spurious messages. We consider that links are *reliable*. A link l is reliable if it is guaranteed that every message sent using l is eventually received as long as sender and receiver are correct processes. Note that messages can be lost in a reliable link if either sender or receiver is a faulty process. Unless otherwise is stated, links do not enforce any restriction with respect to the order in which messages are sent or received (that is, FIFO order is not necessarily preserved).

The system $AAS[\emptyset]$ has two primitives to send and receive messages: $bcast(m)$ and $del(m)$. We say that a process p_i broadcasts a message m when it invokes $bcast_i(m)$. Similarly, a process p_i delivers a message m when it invokes $del_i(m)$. The delivery of a message m by a process p_i can be seen as the fact of passing the message m to the upper layer where this process p_i is (the user p_i in the case of the top layer). We omit the index i in these primitives when the process p_i that invokes these primitives is not important.

With $bcast_i(m)$ process p_i asynchronously sends a message m to each process $p_k \in \Pi$, and $del_i(m)$ reports to the invoking process p_i that m is the received message which is delivered. To preserve the anonymity of the system, we also consider that delivering processes can not identify the link through which a broadcast message is received.

In the literature is always considered that broadcast and delivered messages are unique. It is traditionally assumed that every broadcast message m includes the different sender's process identity as part of the content of m to distinguish it from other messages ([10], [22], [27], [33]). Since in $AAS[\emptyset]$ processes are anonymous, we have to consider that messages are not unique. Hence, in $AAS[\emptyset]$ several instances of a same message m can be broadcast or delivered. Thus, it is more accurate to say that in $AAS[\emptyset]$ process p_i sends an instance of message m to each process $p_k \in \Pi$ when it invokes $bcast_i(m)$, and process p_i is reported of the delivering of an instance of a message m when it invokes $del_i(m)$. To simplify, we abuse of the notation and we only distinguish between an instance of a message and the message itself when it is absolutely necessary.

Let $\mathcal{B}_i$ be the multiset of all instances of messages broadcast by process p_i, and let $\mathcal{D}_i$ be the multiset of all instances of messages delivered by process p_i. Let $\mathcal{B}$ be the multiset of all instances of messages broadcast in the system, i.e., $\mathcal{B} = \bigcup_{p_i \in \Pi} \mathcal{B}_i$. Similarly, $\mathcal{D} = \bigcup_{p_i \in \Pi} \mathcal{D}_i$ is the multiset formed by all instances of messages delivered in the system. Hence, for example, if we have the following five primitives with the same message m: $bcast_i(m)$, $bcast_j(m)$, $del_i(m)$, $del_j(m)$, and $del_k(m)$, then the multiset $\mathcal{B}$ has two instances of m, and $\mathcal{D}$ have three instances (i.e., $\mathcal{B} = \{m, m\}$, and $\mathcal{D} = \{m, m, m\}$).

We assume that broadcast and deliver primitives of $AAS[\emptyset]$ do not give any fault-tolerant guarantees if a process crashes. Specifically, if a process crashes while it is executing $bcast(m)$, m can be received by any subset of processes, and, hence, $del(m)$ can be invoked only by this subset of processes. Therefore, the system $AAS[\emptyset]$, with these two communication primitives, offers an *unreliable broadcast* service.

Before finishing this section, we explain the nomenclature of the system that we are going to use in this paper. As we have said, $AAS[\emptyset]$ is the notation of the anonymous asynchronous system previously defined. As we will see later in this paper, several results are only possible to achieve if we constrain or enhance the system $AAS[\emptyset]$. We use the brackets in the notation to indicate it. For example, if we limit the number of faulty processes to a minority, we use $AAS[f < n/2]$. In other cases we have to enrich the system with another component, for example, with a failure detector. Thus, if we enhance the system $AAS[\emptyset]$ with the failure detector ψ, we use $AAS[\psi]$. Finally, we use AAS omitting the brackets when it is not important to determine whether the anonymous system is enhanced or constrained by some component or condition.

3 Definitions

Now, we define broadcast services that include fault-tolerance.

Three properties have to be satisfied by the broadcast and deliver primitives to provide a *reliable broadcast* (RB) service in the anonymous system AAS:

- *Integrity*: Each instance i_m of each message m delivered by a process has to be the result of broadcasting i_m.
- *Validity*: Each instance of each message m broadcast by a correct process has to be delivered by every correct process.

– *Agreement*: All correct processes deliver the same number of instances of each message m.

Let us define the RB service more formally.

Definition 1 (RB properties) The RB service has to preserve the following three properties in AAS:

1. *Integrity*: $\forall p_i \in \Pi$, $\mathcal{D}_i \subseteq \mathcal{B}$.
2. *Validity*: $\forall p_i \in Correct$, $\bigcup_{p_j \in Correct} \mathcal{B}_j \subseteq \mathcal{D}_i$.
3. *Agreement*: $\forall p_i, p_j \in Correct$, $\mathcal{D}_i = \mathcal{D}_j$.

The uniformity in the delivery has to be added to provide *uniform reliable broadcast* (URB) service. Roughly speaking, the basic idea of uniformity in classical systems, where each broadcast message m is unique, is that if a faulty process delivers a message m, then each correct process also has to deliver m once ([27], [33]). Then, we can define it for an anonymous system AAS as follows:

– *Uniformity*: If a faulty process delivers x instances of a message m, then each correct process delivers at least x instances of m.

Hence, more formally.

Definition 2 (URB properties) The URB service has to preserve the following four properties in AAS:

1. *Integrity*: $\forall p_i \in \Pi$, $\mathcal{D}_i \subseteq \mathcal{B}$.
2. *Validity*: $\forall p_i \in Correct$, $\bigcup_{p_j \in Correct} \mathcal{B}_j \subseteq \mathcal{D}_i$.
3. *Agreement*: $\forall p_i, p_j \in Correct$, $\mathcal{D}_i = \mathcal{D}_j$.
4. *Uniformity*: $\forall p_i \in Faulty$, and $\forall p_j \in Correct$, $\mathcal{D}_i \subseteq \mathcal{D}_j$.

The URB service can not be solved in anonymous asynchronous systems where any number of processes can crash (see Theorem 1 of Section 5). To circumvent this impossibility result, there is a distributed device that processes can use to get information about process failures, namely, a failure detector [19].

All original classes of failure detectors presented by Chandra and Toueg in [19] return information about the identifiers of crashed processes. In this work we use the failure detector class ψ [31] because it does not handle identifiers. Roughly speaking, the failure detector class ψ returns at time τ a number c, such that c is an upper bound of the number of correct processes at time τ (transient period), but eventually c converges towards exactly the number of correct processes (permanent period). Let us define ψ more formally.

Definition 3 (ψ failure detector) Let us consider that each process p_i has a local variable $output_i$ that always returns an integer and positive value. We denote by $output_i^\tau$ this variable at time τ. Let $|Correct|^\tau$ be the number of processes that are correct up to time τ. For any process $p_i \in \Pi$ and run R, the variable $output_i$ must satisfy the following two properties:

1. $\forall \tau$, $output_i^\tau \geq |Correct|^\tau$ (transient period).
2. $\exists \tau : \forall \tau' \geq \tau$, $output_i^{\tau'} = |Correct|^{\tau'}$ (permanent period).

We enrich the anonymous asynchronous system $AAS[\emptyset]$ with the failure detector ψ to solve the URB service even when a majority of processes can crash (that is, $AAS[\psi]$).

The total order property in the delivery of messages has to be added to the URB service to provide the *atomic broadcast* (AB) service[1]. In classical systems, where each broadcast message m is unique, the total order is defined as the delivery of each pair of messages m and m' in a same order in all processes ([10], [11], [15], [22], [25], [27], [33]). Then, we define the total order in the anonymous asynchronous system AAS as follows.

– *Total Order*: For any two broadcast instances i_m and $i_{m'}$, if a process delivers i_m before delivering $i_{m'}$, then no process can deliver $i_{m'}$ before delivering i_m.

Then, we define the AB service in AAS more formally.

Definition 4 (AB properties) The AB service has to preserve the following five properties in AAS:

1. *Integrity*: $\forall p_i \in \Pi$, $\mathcal{D}_i \subseteq \mathcal{B}$.
2. *Validity*: $\forall p_i \in Correct$, $\bigcup_{p_j \in Correct} \mathcal{B}_j \subseteq \mathcal{D}_i$.
3. *Agreement*: $\forall p_i, p_j \in Correct$, $\mathcal{D}_i = \mathcal{D}_j$.
4. *Uniformity*: $\forall p_i \in Faulty$, and $\forall p_j \in Correct$, $\mathcal{D}_i \subseteq \mathcal{D}_j$.
5. *Total Order*: For all processes $p_i, p_j \in \Pi$, and for all instances $i_m \in \mathcal{B}$ and $i_{m'} \in \mathcal{B}$, if process p_i delivers i_m before it delivers $i_{m'}$, then process p_j cannot deliver $i_{m'}$ before it delivers i_m.

4 Implementing the RB service in $AAS[\emptyset]$

We show in this section that the algorithm of Figure 1 implements the RB service in an anonymous asynchronous system independently of the number of faulty processes (i.e., $AAS[\emptyset]$).

Description of the algorithm of Figure 1. Recall that we say that a process p_i broadcasts an instance of message m in $AAS[\emptyset]$ if it executes $bcast_i(m)$, and p_i delivers an instance of m if it executes $del_i(m)$. To avoid ambiguity, we say that process p_i RB-broadcasts an instance of message m if it invokes RB_bcast$_i(m)$ (line 3). Similarly, we say that process p_i RB-delivers an instance of message m if it invokes RB_del$_i(m)$ (line 15).

When process p_i invokes RB_bcast$_i(m)$, it sends a message $(m, seq_i[m])$ to every process of the system $AAS[\emptyset]$, such that m is the instance of the message to spread, and $seq_i[m]$ is the p_i's number of sequence of m (line 5). The variable $seq_i[m]$ allows each process p_j to distinguish among several instances of m RB-broadcast by process p_i (initially, $seq_i[m]$ is 0, line 2).

When process p_i delivers (m, s), that is, message m with number of sequence s (line 6), it uses $count\_msg_i[m, s]$ to increase the number of messages m with

[1] A weaker version of the AB service is also possible using the RB properties instead of the URB properties. Sometimes this stronger version is called the *uniform atomic broadcast* (UAB) service [27].

the same number of sequence s delivered by process p_i (line 7). Then, it sends $(ACK, m, s, count\_msg_i[m, s])$ to every process of the system $AAS[\emptyset]$ (line 8).

When process p_i delivers (ACK, m, s, c) for the first time, that is, an instance of m with number of sequence s and a counter c (line 9), it relays this message (ACK, m, s, c) (line 10) to spread this message even if the sender process that broadcast the message (ACK, m, s, c) crashes. To avoid relaying a same message indefinitely, lines 9-11 are executed only the first time that a same message is delivered (line 9).

To RB-deliver an instance of message m as many times as needed, process p_i uses $exec_i[m, s]$ and the function $apply\_msg(m, s, c)$. The variable $exec_i[m, s]$ remembers the number of times that process p_i executed RB_del$_i(m)$ due to the reception of $(ACK, m, s, -)$ (initially $exec_i[m, s]$ is 0, line 2). The function $apply\_msg(m, s, c)$ allows process p_i to execute RB_del$_i(m)$ from the last time, indicated by $exec_i[m, s]+1$, until the value of the counter of messages (m, s), indicated by c (line 14). To avoid to RB-deliver instances of message m due to outdated delivery of messages (ACK, m, s, c), c has to be greater than $exec_i[m, s]$ (line 13).

```
(1) init
(2)     arrays seq_i, exec_i and count_msg_i have 0 in all positions.

(3) when RB_bcast_i(m) is executed:
(4)     seq_i[m] ← seq_i[m] + 1;
(5)     bcast_i(m, seq_i[m]).

(6) when del_i(m, s) is executed:
(7)     count_msg_i[m, s] ← count_msg_i[m, s] + 1;
(8)     bcast_i(ACK, m, s, count_msg_i[m, s]).

(9) when del_i(ACK, m, s, c) is executed for first time:
(10)    bcast_i(ACK, m, s, c);
(11)    apply_msg(m, s, c).

(12) function apply_msg(m, s, c):
(13)    if (exec_i[m, s] < c) then
(14)        for (j = exec_i[m, s] + 1 to c) do
(15)            RB_del_i(m)
(16)        end for;
(17)        exec_i[m, s] ← c
(18)    end if.
```

Fig. 1 RB service in $AAS[\emptyset]$ (code for process p_i).

Correctness of the algorithm.

Lemma 1 *Integrity:* $\forall p_i \in \Pi$, $\mathcal{D}_i \subseteq \mathcal{B}$.

Proof Let us consider, by the way of contradiction, that the claim is not true. Then, there is a process p_i such that $\mathcal{D}_i \supset \mathcal{B}$. That is, following the contradiction, we have that RB_bcast(m) is executed x times, and RB_del$_i(m)$ is executed y times, being $y > x$. Note that in one extreme case x processes can execute RB_bcast(m) once, and, in the other, a same process can execute RB_bcast(m) x times.

A process p_k increments its local sequence number of instance s of m by one (line 4) previously to execute $bcast(m, s)$ (line 5). Then, for each process p_k, the values of s for m that are broadcast are $1, 2, 3, \ldots$. So, in this case, these values of s for m that are broadcast by any process will be in the range from $1, 2, 3, \ldots$ up to (at most) x. On the other hand, each time that a process p_k delivers a number of instance s of m executing $del_k(m, s)$ (line 6), it counts this number of instances incrementing $count\_msg_k[m, s]$ by one (line 7). Hence, because links are reliable and neither duplicate nor create spurious messages, if $r \leq x$ processes executes $bcast(m, s)$, then every process p_k broadcast the sequence of messages $bcast_k(ACK, m, s, 1)$, $bcast_k(ACK, m, s, 2)$, $\ldots$ $bcast_k(ACK, m, s, c)$, such that $c \leq r$. Note that c could be less than r because some of these r processes can crash before its broadcasting. Thus, because links are reliable and neither duplicate nor create spurious messages, process p_i, while it is alive, eventually receives the messages of these broadcast primitives, and executes their corresponding $del_i(ACK, m, s, -)$. Note that, because links do not force any delivery order, these executions may not be in the same order than their respective broadcast primitives were issued.

We can observe that process p_i stores in $exec_i[m, s]$ the number of invocations of RB_del$_i(m)$ for each instance s of m when $del_i(ACK, m, s, -)$ is executed (lines 14-17). We can also observe that process p_i only RB-delivers the instance s of m (line 15) when $del_i(ACK, m, s, c)$ is also executed, but if it has not been applied yet, i.e., if $c > exec_i[m, s]$ (line 13). Then, RB_bcast(m) is executed x times, and, RB_del$_i(m)$ is executed c times, being $c \leq x$. So, we reach a contradiction, and, hence, $\forall p_i \in \Pi$, $\mathcal{D}_i \subseteq \mathcal{B}$.

Lemma 2 *Validity:* $\forall p_i \in Correct, \bigcup_{p_j \in Correct} \mathcal{B}_j \subseteq \mathcal{D}_i$.

Proof A correct process p_j increments its local sequence number of instance s of m by one (line 4) previously to execute $bcast(m, s)$ (line 5). So, its values of s for m that are broadcast are $1, 2, 3, \ldots$.

On the other hand, each time that a correct process p_j delivers a number of instance s of m executing $del_j(m, s)$ (line 6), it counts this number of instances incrementing $count\_msg_j[m, s]$ by one (line 7). Hence, because links are reliable and neither duplicate nor create spurious messages, if c correct processes execute $bcast(m, s)$, then every correct process p_j eventually broadcasts the sequence of messages $bcast_j(ACK, m, s, 1)$, $bcast_j(ACK, m, s, 2)$, $\ldots$ $bcast_j(ACK, m, s, q)$, being $q \geq c$. Thus, because links are reliable and neither duplicate nor create spurious messages, every correct process p_i eventually receives the messages of these broadcast primitives, and executes their corresponding $del_i(ACK, m, s, -)$.

We can observe that each correct process p_i stores in $exec_i[m, s]$ the number of invocations of RB_del$_i(m)$ for each instance s of m when $del_i(ACK, m, s, -)$ is executed (lines 14-17). Note that process p_i only RB-delivers the instance s of m (line 15) when $del_i(ACK, m, s, c)$ is also executed, but if it has not been applied yet, that is, if $c > exec_i[m, s]$ (line 13). Then, if RB_bcast(m) is executed x times, c of these x times are due to correct processes, and, hence, RB_del$_i(m)$ is executed at least c times. Therefore, $\forall p_i \in Correct, \bigcup_{p_j \in Correct} \mathcal{B}_j \subseteq \mathcal{D}_i$.

Lemma 3 *Agreement:* $\forall p_i, p_j \in Correct$, $\mathcal{D}_i = \mathcal{D}_j$.

Proof Let us consider, by the way of contradiction, that the claim is not true. Following the contradiction, let us consider, w.l.o.g., that a correct process p_i RB-delivers x instances of a message m, and a correct process p_j RB-delivers $x' < x$ instances of this message m.

If correct process p_i RB-delivers x instances of m, it always executes x times $del_i(ACK, m, -, c)$ such that $c \leq x$ (lines 12-18), and, hence, it also executes their corresponding $bcast_i(ACK, m, -, c)$ (lines 10-11). Note that $c \leq x$ because each process increments $count\_msg_j[m, s]$ by one (line 7), and links are reliable and not duplicate or create spurious messages. After all this happens, correct process p_j will also eventually execute x times the primitive $del_j(ACK, m, -, c)$, being $c \leq x$ (line 9), and process p_j will eventually have to deliver from x' to x instances of m (lines 14-17). Therefore, we reach a contradiction, and $\forall p_i, p_j \in Correct$, $\mathcal{D}_i = \mathcal{D}_j$.

5 Impossibility of the URB service in $AAS[f > n/2]$

We show in this section that the URB service is impossible to solve in the anonymous asynchronous system if a majority of processes can crash.

Theorem 1 *There is no algorithm $\mathcal{A}$ that implements the URB service in every run of an anonymous asynchronous system when a majority of processes can crash (i.e., $AAS[f > n/2]$), and when processes do not know the maximum number of faulty processes (i.e., f is unknown).*

Proof By contradiction, let us assume that there is an algorithm $\mathcal{A}$ that implements the URB service in every run of $AAS[f > n/2]$ when processes do not know f. Let us consider the following two valid runs R_1 and R_2 of $\mathcal{A}$.

In R_1 a correct process p_b executes the URB broadcast primitive with the message m, and, to preserve the Validity Property of the URB service, a correct process p_d executes at time τ the URB deliver primitive with the message m. We consider that p_d delivers m after receiving x messages acknowledging that x processes have also delivered this message m. Note that a process only knows that in a run the rest of processes can crash, but it does not know how many processes will crash in R_1 or who they will be. So, x is not related to the number of correct processes. Finally, we consider that in R_1 the transmission of any other message not previously specified is delayed in this asynchronous system until time τ', $\tau' > \tau$.

R_2 is the same execution of R_1 until time τ. So, R_1 and R_2 are indistinguishable until time τ. Then, let us consider in R_2 that p_b crashes at time τ'', $\tau < \tau'' < \tau'$. We also consider in R_2 that p_d crashes at time τ'', hence, after delivering m. Similarly, let us consider that these x processes, that informed p_d about their delivery of m, also crash at a time τ''. Note that, as process p_d does not know a priori anything about correct processes, it can happen that the intersection between the set of these x processes and the set of correct processes can be empty. We also assume in R_2 that all transmitted messages in R_1 sent by faulty processes that were delayed until time τ' are lost in R_2. Note that this can happen because reliable channels only guarantee the delivery of messages if sender and receiver processes are correct. Then after τ', there is no correct process in R_2 that has received any message related to m. Hence, we reach a contradiction, and there is a process p_d

that delivers m at time τ, but there is no correct process that can deliver m in R_2 (which violate the Uniformity Property of the URB service).

Therefore, there is no algorithm $\mathcal{A}$ that implements the URB service in every run of $AAS[f > n/2]$ when processes do not know f.

As we can see, the unique identity of the processes has no influence in the proof of the previous theorem. Hence, note that the following corollary is also preserved.

Corollary 1 *There is no algorithm $\mathcal{A}$ that implements the URB service in every run of a classical asynchronous system when a majority of processes can crash, and when processes do not know the maximum number of faulty processes.*

6 Implementing the URB service in *AAS*

In this section we present an algorithm (see Figure 2) that implements the URB service in the anonymous asynchronous system when a majority of processes are correct (i.e., $AAS[f < n/2]$).

Another algorithm (see Figure 3) is presented in this section that implements the URB service in $AAS[\emptyset]$ with the failure detector ψ, see Definition 3, (i.e., $AAS[\psi]$). Note that, due to the impossibility result of Theorem 1, we need to use a failure detector to enhance the system and circumvent this impossibility. Thus, the implementation of the URB service in AAS is possible independently of the number of correct processes.

6.1 Implementing the URB service in $AAS[f < n/2]$

We show that the algorithm of Figure 2 implements the URB service in $AAS[f < n/2]$.

Description of the algorithm of Figure 2. Similarly to Figure 1, we say in Figure 2 that a process p_i URB-broadcasts an instance of message m if it invokes URB_bcast$_i(m)$ (line 3), and that a process p_i URB-delivers an instance of message m if it invokes URB_del$_i(m)$ (line 20).

The algorithm of Figure 2 is basically the same of Figure 1 except when process p_i executes $del_i(ACK, m, s, c)$ (lines 9-16). In this case of Figure 2, process p_i also relays this message (ACK, m, s, c) when it is executed by p_i for the first time (lines 10-12). Process p_i uses $count\_ack_i[m, s, c]$ to count the number of messages (ACK, m, s, c) received it (line 13). If process p_i has received a message (ACK, m, s, c) from a majority of processes, then process p_i applies this message (lines 14-16), executing $URB\_del_i(m)$, from the last time, indicated in $exec_i[m, s] + 1$, until the value of the counter of messages (m, s), indicated by c (lines 19-22). Similarly to Figure 1, to avoid to URB-deliver messages due to outdated delivery of messages (ACK, m, s, c), the value c has to be greater than $exec_i[m, s]$ (line 18).

```
(1) init
(2)     arrays seq_i, exec_i, count_msg_i and count_ack_i
        have 0 in all positions.

(3) when URB_bcast_i(m) is executed:
(4)     seq_i[m] ← seq_i[m] + 1;
(5)     bcast_i(m, seq_i[m]).

(6) when del_i(m, s) is executed:
(7)     count_msg_i[m, s] ← count_msg_i[m, s] + 1;
(8)     bcast_i(ACK, m, s, count_msg_i[m, s]).

(9) when del_i(ACK, m, s, c) is executed:
(10)    if (del_i(ACK, m, s, c) is executed for first time) then
(11)        bcast_i(ACK, m, s, c)
(12)    end if;
(13)    count_ack_i[m, s, c] ← count_ack_i[m, s, c] + 1;
(14)    if (count_ack_i[m, s, c] > n/2) then
(15)        apply_msg(m, s, c)
(16)    end if.

(17) function apply_msg(m, s, c):
(18)    if (exec_i[m, s] < c) then
(19)        for (j = exec_i[m, s] + 1 to c) do
(20)            URB_del_i(m)
(21)        end for;
(22)        exec_i[m, s] ← c
(23)    end if.
```

Fig. 2 URB service in $AAS[f < n/2]$ (code for process p_i).

Correctness of URB in $AAS[f < n/2]$

Lemma 4 *Integrity:* $\forall p_i \in \Pi$, $\mathcal{D}_i \subseteq \mathcal{B}$.

Proof It is similar to the proof of Lemma 1.

Lemma 5 *Validity:* $\forall p_i \in Correct$, $\bigcup_{p_j \in Correct} \mathcal{B}_j \subseteq \mathcal{D}_i$.

Proof A correct process p_j increments its local sequence number of instance s of m by one (line 4) previously to execute $bcast(m, s)$ (line 5). So, its values of s for m that are broadcast are $1, 2, 3, \ldots$

On the other hand, each time that a correct process p_j delivers a number of instance s of m executing $del_j(m, s)$ (line 6), it counts this number of instance incrementing $count\_msg_j[m, s]$ by one (line 7). Hence, because links are reliable and neither duplicate nor create spurious messages, if c correct processes execute $bcast(m, s)$, then every correct process p_j broadcasts the sequence of messages $bcast_j(ACK, m, s, 1)$, $bcast_j(ACK, m, s, 2)$, $\ldots$ $bcast_j(ACK, m, s, c)$. Thus, because links are reliable, there are no duplicated or spurious messages, and a majority of processes are correct (due to $AAS[f < n/2]$), every correct process p_i eventually receives the messages of these broadcast primitives from a majority of processes, and it executes its corresponding line 15. As p_i stores in $exec_i[m, s]$ the number of invocations of URB_del$_i(m)$ for each instance s of m when apply_msg(m,s,c) is executed, and process p_i only URB-delivers the instance s of m

if it has not been applied yet (line 18), hence, p_i URB-delivers m at least c times because URB_bcast(m) is executed at least c times. Therefore, $\forall p_i \in Correct$, $\bigcup_{p_j \in Correct} \mathcal{B}_j \subseteq \mathcal{D}_i$.

Lemma 6 *Agreement:* $\forall p_i, p_j \in Correct$, $\mathcal{D}_i = \mathcal{D}_j$.

Proof Let us consider, w.l.o.g., that a correct process p_i URB-delivers x instances of a message m, and a correct process p_j URB-delivers $x' < x$ instances of this message m.

If correct process p_i URB-delivers x instances of m, it eventually executes line 20 of function *apply_msg*() with parameters $(m, -, c_1)$, $\ldots$ $(m, -, c_s)$ such that $c_1 + \ldots + c_s = x$. To do so, process p_i has to receive each corresponding message $(ACK, m, -, c_1)$, $\ldots$ $(ACK, m, -, c_s)$ from at least a majority of processes (lines 14-16). Then, a majority of processes executes $bcast_i(ACK, m, -, c_1)$, $\ldots$ $bcast_i(ACK, m, -, c_s)$, and each one of them rebroadcasts these messages the first time they receive them (lines 10-12). Thus, because links are reliable, there are no duplicated or spurious messages, and a majority of processes are correct (i.e., $f < n/2$), all these x messages $(ACK, m, -, c_1)$ $\ldots$ $(ACK, m, -, c_s)$ will be received by correct process p_j, and it eventually also has to URB-deliver from x' to x instances of m (line 20). Therefore, $\forall p_i, p_j \in Correct$, $\mathcal{D}_i = \mathcal{D}_j$.

Lemma 7 *Uniformity:* $\forall p_i \in Faulty$, *and* $p_j \in Correct$, $\mathcal{D}_i \subseteq \mathcal{D}_j$.

Proof Each time that a faulty process p_i URB-delivers m, it executes line 20 into the function *apply_msg*() with parameters, w.l.o.g, (m, s', c'). Note that this happens because process p_i has received the message (ACK, m, s', c') from a majority of processes (lines 14-16). Then, a majority of processes executes $bcast_i(ACK, m, s', c')$, and each one of them rebroadcasts this message the first time they receive it (lines 10-12). Thus, because links are reliable, there are no duplicated or spurious messages, and a majority of processes are correct (i.e., $f < n/2$), this message (ACK, m, s', c') will be received by correct process p_j, and it eventually also has to URB-deliver at least c' instances of m (lines 19-21). Therefore, $\forall p_i \in Faulty$, and $p_j \in Correct$, $\mathcal{D}_i \subseteq \mathcal{D}_j$.

6.2 Implementing the URB service in $AAS[\psi]$

In this section we show that the algorithm of Figure 3 implements the URB service in $AAS[\psi]$ independently of the number of correct processes.

Description of the algorithm of Figure 3. As in the Figure 2, we say that a process p_i URB-broadcasts an instance of message m if it invokes URB_bcast$_i(m)$ (line 5), and URB-delivers an instance of message m if p_i invokes URB_del$_i(m)$ (line 25).

This algorithm of Figure 3 is similar to the algorithm of Figure 2. The main difference now is that a process p_i, based on the value returned by the failure detector ψ in *FD.output$_i$*, has to wait until it delivers a number of messages (ACK, m', s', c'), indicated by $count\_ack_i[m', s', c']$, broadcast by all correct processes. As the number of correct processes may change over time, process p_i needs a task (task T2 of Figure 3) where it can know this variation. In this task T2 process p_i checks permanently the variable $FD.output_i$ of the failure detector ψ. Hence, if process p_i delivered a number of messages (ACK, m', s', c') at least equal to the current number of correct processes (line 18), it applies this message m' in the same way that the algorithm of Figure 2 does (line 19, and lines 22-28).

```
(1) init
(2)     arrays seq_i, exec_i, count_msg_i and count_ack_i
        have 0 in all positions;
(3)     start tasks T1 and T2.

(4) task T1:
(5) when URB_bcast_i(m) is executed:
(6)     seq_i[m] ← seq_i[m] + 1;
(7)     bcast_i(m, seq_i[m]).

(8) when del_i(m, s) is executed:
(9)     count_msg_i[m, s] ← count_msg_i[m, s] + 1;
(10)    bcast_i(ACK, m, s, count_msg_i[m, s]).

(11) when del_i(ACK, m, s, c) is executed:
(12)    if (del_i(ACK, m, s, c) is executed for first time) then
(13)        bcast_i(ACK, m, s, c);
(14)    end if;
(15)    count_ack_i[m, s, c] ← count_ack_i[m, s, c] + 1.

(16) task T2:
(17)    repeat forever
(18)        for_each (count_ack_i[m', s', c'] ≥ FD.output_i) do
(19)            apply_msg(m', s', c')
(20)        end_for
(21)    end repeat.

(22) function apply_msg(m, s, c):
(23)    if (exec_i[m, s] < c) then
(24)        for (j = exec_i[m, s] + 1 to c) do
(25)            URB_del_i(m)
(26)        end for;
(27)        exec_i[m, s] ← c
(28)    end if.
```

Fig. 3 URB service in $AAS[\psi]$ (code for process p_i).

Correctness of URB in $AAS[\psi]$

Lemma 8 *Integrity:* $\forall p_i \in \Pi$, $\mathcal{D}_i \subseteq \mathcal{B}$.

Proof It is similar to the proof of Lemma 1.

Lemma 9 *Validity:* $\forall p_i \in Correct$, $\bigcup_{p_j \in Correct} \mathcal{B}_j \subseteq \mathcal{D}_i$.

Proof A correct process p_j increments its local number of instance s of m by one (line 4) previously to execute $bcast(m, s)$ (line 5). So, its values of s for m that are broadcast are $1, 2, 3, \ldots$.

On the other hand, each time that a correct process p_j delivers a number of instance s of m executing $del_j(m, s)$ (line 8), it counts this number of instances incrementing $count\_msg_j[m, s]$ by one (line 9). Hence, because links are reliable and neither duplicate nor create spurious messages, if c correct processes execute $bcast(m, s)$, then every correct process p_j broadcasts the sequence of messages $bcast_j(ACK, m, s, 1)$, $bcast_j(ACK, m, s, 2)$, $\ldots$ $bcast_j(ACK, m, s, c)$. Thus, because in the system $AAS[\psi]$ links are reliable, there are no duplicated or spurious messages, and every correct process p_i eventually has in its variable $FD.output_i$ of the failure detector ψ the number of correct processes (from Property 2 of Definition 3), it eventually receives at least $FD.output_i$ messages (line 18), and, hence, executing its corresponding line 19. Note that correct process p_i stores in $exec_i[m, s]$ the number of invocations of URB_del$_i(m)$, for each instance s of m when apply_msg(m,s,c) is executed. Similarly, note that process p_i only URB-delivers the instance s of m if it has not been applied yet (line 23). Hence, as a consequence, process p_i URB-delivers m at least c times, because URB_bcast(m) is executed at least c times. Therefore, $\forall p_i \in Correct$, $\bigcup\limits_{p_j \in Correct} \mathcal{B}_j \subseteq \mathcal{D}_i$.

Lemma 10 *Agreement:* $\forall p_i, p_j \in Correct$, $\mathcal{D}_i = \mathcal{D}_j$.

Proof Let us consider, w.l.o.g., that a correct process p_i URB-delivers x instances of a message m, and a correct process p_j URB-delivers $x' < x$ instances of this message m.

If correct process p_i URB-delivers x instances of m, it eventually executes line 25 of function $apply\_msg()$ with parameters $(m, -, c_1)$, $\ldots$ $(m, -, c_s)$ such that $c_1 + \ldots + c_s = x$. To do so, process p_i has to receive each corresponding messages $(ACK, m, -, c_1)$, $\ldots$ $(ACK, m, -, c_s)$ from at least $FD.output_i$ processes (lines 18-20). Then, eventually $FD.output_i$ is equal to the number of correct processes (from Property 2 of Definition 3), and it executes $bcast_i(ACK, m, -, c_1)$, $\ldots$ $bcast_i(ACK, m, -, c_s)$, and each correct process rebroadcasts these messages the first time they receive them (lines 12-14). Thus, because links are reliable and the variable $FD.output_i$ of the failure detector ψ of all correct processes eventually converges towards the number of correct processes (from Property 2 of Definition 3), all these x messages $(ACK, m, -, c_1)$ $\ldots$ $(ACK, m, -, c_s)$ will be received by correct process p_j, and it eventually also has to URB-deliver from x' to x instances of m (lines 24-26).

Lemma 11 *Uniformity:* $\forall p_i \in Faulty$, *and* $p_j \in Correct$, $\mathcal{D}_i \subseteq \mathcal{D}_j$.

Proof Each time that process p_i URB-delivers m, it executes line 25 into the function $apply\_msg()$ with parameters, w.l.o.g, (m, s', c'). Thus, process p_i receives the message (ACK, m, s', c') from $FD.output_i$ processes (lines 18-20). Then, eventually $FD.output_i$ is equal to the number of correct processes (from Property 2 of Definition 3), and it executes $bcast_i(ACK, m, s', c')$, and each one of them rebroadcasts this message the first time they receive it (lines 12-14). Thus, because links are reliable and the variable $FD.output$ of all correct processes contains this exact number of correct processes (from Property 2 of Definition 3), this message (ACK, m, s', c') will be received by correct process p_j, and it eventually also has to deliver at least c' instances of m (line 25). Therefore, $\forall p_i \in Faulty$, and $p_j \in Correct$, $\mathcal{D}_i \subseteq \mathcal{D}_j$.

7 Implementing the AB service in *AAS*

It is known that the AB service is not possible to be solved in classical asynchronous systems prone to process crashes ([19], [24]), and, hence, in anonymous asynchronous systems. To circumvent this impossibility, in this section we show that the algorithm of Figure 4 implements the AB service in the anonymous asynchronous system $AAS[\emptyset]$ if we enrich it with the following two components: Consensus and the RB service. Note that the algorithm of Figure 4 is a simple adaptation to anonymous synchronous systems the solution of [19] presented for classical asynchronous systems. We include it in this paper with the aim of proving that the AB service is also possible with anonymity.

Roughly speaking, *Consensus* ([26], [28]) specifies that all processes decide a same value v, and this value v is proposed by some process. Let us define consensus more formally.

Definition 5 Let us consider that a process p_i of the anonymous system AAS proposes a value v_i invoking the primitive $consensus_i(v_i)$. This primitive returns to process p_i the decided value v preserving the following three properties.

1. *CON-Termination*: Every correct process eventually decides.
2. *CON-Validity*: The value v decided by any process is one of the proposed values.
3. *CON-Agreement*: All decided values are the same value v.

Let us see the requirements of the two components that Figure 4 needs to implement the AB service. As we have shown in Section 4, the RB service can be implemented in $AAS[\emptyset]$. It is also known that Consensus cannot be solved in $AAS[\emptyset]$ [16]. Hence, the algorithm of Figure 4 is implementable in $AAS[\emptyset]$ enhanced with the requirements that Consensus enforce. Then, we denote $AAS[Consensus]$ the anonymous system $AAS[\emptyset]$ enriched with the requirements of the Consensus component.

As examples of the implementability of Consensus, and hence of the AB service in AAS, there are in the literature several works that solve Consensus augmenting the anonymous asynchronous system with an implementable failure detector ([9], [17])[2]. [9] implements Consensus in the anonymous asynchronous system $AAS[\emptyset]$ enhanced with a failure detector[3]. [17] implements Consensus in $AAS[\psi]$, that is, in an anonymous asynchronous system enriched with the failure detector ψ. Thus, the AB service can be solved at least in an anonymous system $AAS[\psi]$.

Description of the algorithm of Figure 4 First of all, note that it is a simple adaptation of [19] with multisets of messages and with anonymous Consensus. The necessity of multisets is due to the fact that messages in anonymous systems are not unique (as we have already explained in Section 2). Thus, several instances of a same message can be maintained in the multiset.

Similarly to the rest of algorithms in this paper, we say that a process p_i AB-broadcasts an instance of message m if it invokes $\text{AB\_bcast}_i(m)$ (line 6), and a process p_i AB-delivers an instance of message m if it invokes $\text{AB\_del}_i(m)$ (line 17).

[2] We can find other papers in the literature that solve Consensus in anonymous systems with a failure detector that can not be implementable ([18], [16]). That is, they can solve it theoretically, but not practically.

[3] Actually, [9] solves Consensus in a more general system such that a particular case is the anonymous asynchronous system.

Each process p_i has in $consensus_i$ an array of instances of the Consensus component (line 3). Each instance $consensus_i[k]$ is totally independent of the rest of instances $consensus_i[s]$, being $k \neq s$. We consider that instances of Consensus can be executed concurrently if it is necessary.

Process p_i RB-broadcasts a message m each time it AB-broadcasts an instance of m (lines 6-7). An instance of message m is stored by process p_i in the multiset $received_i$ each time it RB-delivers m (lines 8-9). Process p_i stores an instance of message m in the multiset $delivered_i$ each time it AB-delivers an instance of m (lines 17-18). All messages received by processes p_i but not yet delivered are stored in the multiset $pending_i$ (line 12). Note that $pending_i$, $received_i$ and $delivered_i$ have to be a multiset variable because messages are not unique, and, hence, there can be multiples instances of a same message m. Initially, $received_i$ and $delivered_i$ are empty (line 2).

If process p_i has some message in $pending_i$, process p_i proposes a value to be consensuated (lines 15-16). This proposed value is the first message considering the FIFO (first-in and first-out) sequence of messages in order of arrival to the process and not yet AB-delivered. Then, process p_i proposes a message to get consensus in the instance of Consensus indicated by $next\_order_i$ when it invokes the primitive $consensus_i[next\_order_i](proposal_i)$ (line 16). This decided message is in the variable $decision_i[next\_order_i]$ when the primitive $consensus_i$ finishes. Finally, process p_i AB-delivers the decided message in $decision_i[next\_order_i]$ (line 17), and it also includes this decided message in the multiset $delivered_i[next\_order_i]$ (line 18). Then, the instance of the message in $decision_i[next\_order_i]$ can be removed from the multiset $pending_i$ in the next iteration of process p_i (line 12).

Correctness of AB in AAS[Consensus] The proofs of the properties CON-Termination, CON-Validity and CON-Agreement of Consensus are similar to [19] but using multisets of messages and the anonymous Consensus component.

8 Conclusion

Fault-tolerant broadcast is a fundamental problem in distributed systems that includes several guarantees in the delivery of messages when crashes can happen in the system. Traditionally, the fault-tolerant broadcast service has been studied in classical distributed systems where each process has a unique identity.

In this paper we have studied for first time the fault-tolerant broadcast service in anonymous systems. First, we include an implementation of the reliable broadcast (RB) service for anonymous systems. On the possibility to implement the uniform reliable broadcast (URB) service, in this paper we prove the impossibility to implement the uniform reliable broadcast (URB) service when a majority of processes can crash and the amount of crashed processes is unknown by the correct processes, and the possibility of implement it when only a minority can crash. To extend the implementability of the URB service circumventing this impossibility result, we present an algorithm that implements the URB service in anonymous asynchronous systems independently of the number of crashed processes. We do it enriching the system with a failure detector (we use ψ because it is a failure detector that works without knowing the identities of the processes). We also prove in this paper that the atomic broadcast (AB) service is implementable if we augment

init
multisets $received_i$ and $delivered_i$ are empty;
array $consensus_i$ shared by all processes;
$next\_order_i \leftarrow 1$;
start task T.

when AB_$bcast_i(m)$ **is executed**:
RB_$bcast_i(m)$.

when RB_$del_i(m)$ **is executed**:
$received_i \leftarrow received_i \cup \{m\}$.

task T:
repeat forever
  $pending_i \leftarrow received_i \setminus delivered_i$;
  if $(pending_i \neq \emptyset)$ **then**
    $next\_order_i \leftarrow next\_order_i + 1$
    $proposal_i \leftarrow$ first message of $pending_i$ in FIFO order;
    $decision_i[next\_order_i] \leftarrow$
        $consensus_i[next\_order_i](proposal_i)$;
    AB_$del_i(decision_i[next_order_i])$;
    $delivered_i \leftarrow delivered_i \cup \{decision_i[next\_order_i]\}$
  end if
end repeat.

Fig. 4 AB service in *AAS* (code for process p_i).

the anonymous asynchronous system with the requirements needed by Consensus. Hence, as there are in the literature anonymous Consensus components that are implementable in the anonymous systems, hence, we prove in this paper that AB service is implementable in anonymous systems.

As future work, we have to study other fault-tolerant broadcast services with different properties of delivery (such as FIFO or Causal order). Another future line is to search the weakest failure detector that allows to implement each type of fault-tolerant broadcast service in asynchronous anonymous systems. Finally, the solutions included in this paper have been focused to prove the possibility results with algorithms as simple as possible. Hence, we aim to the researchers to study new algorithms that solve the fault-tolerant broadcast services regarding the performance or efficiency of the anonymous systems.

References

1. Ian F. Akyildiz, Weilian Su, Yogesh Sankarasubramaniam, and Erdal Cayirci. Wireless sensor networks: a survey. *Computer Networks*, 38(4):393–422, 2002.
2. Dana Angluin, James Aspnes, Zoë Diamadi, Michael J. Fischer, and René Peralta. Computation in networks of passively mobile finite-state sensors. *Distributed Computing*, 18(4):235–253, 2006.
3. Hamid R. Arabnia. A parallel algorithm for the arbitrary rotation of digitized images using process-and-data-decomposition approach. *J. Parallel Distrib. Comput.*, 10(2):188–192, 1990.
4. Hamid R. Arabnia. A distributed stereocorrelation algorithm. In *Proceedings of the 4th International Conference on Computer Communications and Networks (ICCCN '95), September 20-23, 1995, Las Vegas, Nevada, USA*, page 479, 1995.

5. Hamid R. Arabnia and Suchendra M. Bhandarkar. Parallel stereocorrelation on a reconfigurable multi-ring network. *The Journal of Supercomputing*, 10(3):243–269, 1996.
6. Hamid R. Arabnia and Martin A. Oliver. Arbitrary rotation of raster images with SIMD machine architectures. *Comput. Graph. Forum*, 6(1):3–11, 1987.
7. Hamid R. Arabnia and Martin A. Oliver. A transputer network for the arbitrary rotation of digitised images. *Comput. J.*, 30(5):425–432, 1987.
8. Hamid R. Arabnia and Martin A. Oliver. A transputer network for fast operations on digitised images. *Comput. Graph. Forum*, 8(1):3–11, 1989.
9. Sergio Arévalo, Antonio Fernández Anta, Damien Imbs, Ernesto Jiménez, and Michel Raynal. Failure detectors in homonymous distributed systems (with an application to consensus). In *2012 IEEE 32nd International Conference on Distributed Computing Systems, Macau, China, June 18-21, 2012*, pages 275–284, 2012.
10. Hagit Attiya and Jennifer Welch. *Distributed Computing: Fundamentals, Simulations, and Advanced Topics.* Wiley, 2004.
11. Roberto Baldoni, Stefano Cimmino, and Carlo Marchetti. A classification of total order specifications and its application to fixed sequencer-based implementations. *J. Parallel Distrib. Comput.*, 66(1):108–127, 2006.
12. Suchendra M. Bhandarkar and Hamid R. Arabnia. The hough transform on a reconfigurable multi-ring network. *J. Parallel Distrib. Comput.*, 24(1):107–114, 1995.
13. Suchendra M. Bhandarkar and Hamid R. Arabnia. The REFINE multiprocessor - theoretical properties and algorithms. *Parallel Computing*, 21(11):1783–1805, 1995.
14. Suchendra M. Bhandarkar, Hamid R. Arabnia, and Jeffrey W. Smith. A reconfigurable architecture for image processing and computer vision. *IJPRAI*, 9(2):201–229, 1995.
15. Kenneth P. Birman and Thomas A. Joseph. Reliable communication in the presence of failures. *ACM Trans. Comput. Syst.*, 5(1):47–76, 1987.
16. François Bonnet and Michel Raynal. Consensus in anonymous distributed systems: Is there a weakest failure detector? In *24th IEEE International Conference on Advanced Information Networking and Applications, AINA 2010, Perth, Australia, 20-13 April 2010*, pages 206–213, 2010.
17. François Bonnet and Michel Raynal. The price of anonymity: Optimal consensus despite asynchrony, crash, and anonymity. *TAAS*, 6(4):23, 2011.
18. François Bonnet and Michel Raynal. Anonymous asynchronous systems: the case of failure detectors. *Distributed Computing*, 26(3):141–158, 2013.
19. Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.
20. Jo-Mei Chang and Nicholas F. Maxemchuk. Reliable broadcast protocols. *ACM Trans. Comput. Syst.*, 2(3):251–273, 1984.
21. Tom Chothia and Konstantinos Chatzikokolakis. A survey of anonymous peer-to-peer file-sharing. In *Embedded and Ubiquitous Computing - EUC 2005 Workshops, EUC 2005 Workshops: UISW, NCUS, SecUbiq, USN, and TAUES, Nagasaki, Japan, December 6-9, 2005, Proceedings*, pages 744–755, 2005.
22. Xavier Défago, André Schiper, and Péter Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36(4):372–421, 2004.
23. GiuseppeAntonio Di Luna, Roberto Baldoni, Silvia Bonomi, and Ioannis Chatzigiannakis. Conscious and unconscious counting on anonymous dynamic networks. In Mainak Chatterjee, Jian-nong Cao, Kishore Kothapalli, and Sergio Rajsbaum, editors, *Distributed Computing and Networking*, volume 8314 of *Lecture Notes in Computer Science*, pages 257–271. Springer Berlin Heidelberg, 2014.
24. Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.
25. Hector Garcia-Molina and Annemarie Spauster. Ordered and reliable multicast communication. *ACM Trans. Comput. Syst.*, 9(3):242–271, 1991.
26. Rachid Guerraoui. Revistiting the relationship between non-blocking atomic commitment and consensus. In *Distributed Algorithms, 9th International Workshop, WDAG '95, Le Mont-Saint-Michel, France, September 13-15, 1995, Proceedings*, pages 87–100, 1995.
27. Vassos Hadzilacos and Sam Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical report, Cornell University, Ithaca (USA), TR #94–1425, 83 pages, 1994.
28. Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

29. Giuseppe Antonio Di Luna, Roberto Baldoni, Silvia Bonomi, and Ioannis Chatzigiannakis. Counting in anonymous dynamic networks under worst-case adversary. In *IEEE 34th International Conference on Distributed Computing Systems, ICDCS 2014, Madrid, Spain, June 30 - July 3, 2014*, pages 338–347, 2014.
30. Othon Michail, Ioannis Chatzigiannakis, and PaulG. Spirakis. Naming and counting in anonymous unknown dynamic networks. In Teruo Higashino, Yoshiaki Katayama, Toshimitsu Masuzawa, Maria Potop-Butucaru, and Masafumi Yamashita, editors, *Stabilization, Safety, and Security of Distributed Systems*, volume 8255 of *Lecture Notes in Computer Science*, pages 281–295. Springer International Publishing, 2013.
31. Achour Mostéfaoui, Sergio Rajsbaum, Michel Raynal, and Corentin Travers. On the computability power and the robustness of set agreement-oriented failure detector classes. *Distributed Computing*, 21(3):201–222, 2008.
32. ElMoustapha Ould-Ahmed-Vall, Douglas M. Blough, Bonnie S. Heck-Ferri, and George F. Riley. Distributed global ID assignment for wireless sensor networks. *Ad Hoc Networks*, 7(6):1194–1216, 2009.
33. Michel Raynal. *Communication and Agreement Abstractions for Fault-Tolerant Asynchronous Distributed Systems.* Morgan & Claypool, 2010.
34. M. Arif Wani and Hamid R. Arabnia. Parallel edge-region-based segmentation algorithm targeted at reconfigurable multiring network. *The Journal of Supercomputing*, 25(1):43–62, 2003.