Anonymity-Preserving Failure Detectors

Zohir Bouzid, Corentin Travers

Anonymity-Preserving Failure Detectors

Zohir Bouzid, U. Bordeaux, France*
zohir.bouzid@labri.fr

Corentin Travers, U. Bordeaux, France
travers@labri.fr

Abstract

The paper investigates the consensus problem in anonymous, failures prone and asynchronous shared memory systems. It introduces a new class of failure detectors, called *anonymity-preserving* failures detectors suited to anonymous systems. As its name indicates, a failure detector in this class cannot be relied upon to break anonymity. For example, the anonymous perfect detector AP, which gives at each process an estimation of the number of processes that have failed belongs to this class.

The paper then determines the weakest failure detector among this class for consensus. This failure detector, called C, may be seen as a loose failures counter: (1) after a failure occurs, the counter is eventually incremented, and (2) when at least two processes do not fail, it eventually stabilizes. Finally, the paper also introduces failure detector C_k, a simple generalization of C and shows that it can be used to solve k-set-agreement, a generalization of consensus in which up to k distinct values may be decided.

Keywords: failure detectors, shared memory, anonymous computing.

*Supported by the Idex Bordeaux CPU

1 Introduction

Anonymous computing The vast majority of the literature about distributed computing assumes that each process is provided with an unique identifier. We consider in this work *anonymous computing* in which processes have no identifiers and are programmed identically. Besides intellectual curiosity, anonymous computing might be of practical interest [27]. For example, for privacy reasons, a set of distributed processes may be willing to compute some function on their inputs without revealing their identity. Or the distributed computation might be performed on top of an anonymous communication system [16], and thus using ids is forbidden.

Specifically, we consider the *totally anonymous shared memory model* of distributed computing. The shared memory consists only in basic shared objects, namely read/write registers. We assume that there is no way to uniquely assign registers to the processes as this would provide a way to differentiate the processes. Previous works [6, 27] have shown that the lack of unique identifiers limits the computational power of the shared memory model. Similarly, starting from the pioneering work of Angluin [2], the computational power of anonymous message passing system in the failure-free case has been investigated for particular or general graph topologies, e.g., [8, 7, 31, 32].

Consensus, failure and asynchrony Besides the unavailability of unique identifiers, a major difficulty is coping with failures and asynchrony. Many simple distributed tasks cannot be solved in asynchronous and failures-prone distributed system. A prominent example is *consensus*, which is a cornerstone task in fault-tolerant distributed computing. Informally, the processes, each starting with a private value, are required to agree on one value chosen among their initial values. In systems with identities, it is well known that asynchronous fault tolerant consensus is impossible as soon as at least one process may fail by crashing [21, 29]. This impossibility trivially extends to anonymous systems.

Failure detectors *Failure detectors* [14] are a popular approach to circumvent impossibilities stemming from asynchrony and failures. A failure detector is a distributed device that provides each processes with perhaps unreliable information about which other processes have crashed. In non-anonymous systems, several classes of failure detectors have been defined [22]. In many cases, their specification involves processes identities. For example, the *perfect detector* P provides each process with a list of the identities of some of the processes that have crashed. The list is eventually complete in the sense that it eventually includes the identity of each crashed process. The *leader* failure detector Ω eventually outputs the same identity at every process, which is the identity of a non-faulty process.

Given a distributed task T, a natural question is to determine the *weakest failure detector* for T, that is a failure detector D which is both *sufficient* to solve the task – there is an asynchronous, fault tolerant protocol that uses D to solve T – and *necessary*, in the sense that any failure detector D' that can be used to solve T can also be used to emulate D. For example, is well-known that Ω is the weakest failure detector for consensus [13] in shared memory systems when processes are provided with unique identifiers.

Failures detector in anonymous systems Bonnet and Raynal initiated the study of failure detectors in anonymous message passing systems [10]. In particular, they identify *identity-free* counterpart of classical failure detectors including Ω and P. $A\Omega$, an identity-free failure detector

equivalent to Ω, outputs a Boolean value at each process and, eventually outputs at a single correct process true and false at every other process. A consensus algorithm that uses $A\Omega$ is also presented. In the shared memory model, an anonymous $A\Omega$-based protocol can be found in [17]. Bonnet and Raynal left open the following question: "Consensus in anonymous distributed systems: is there a weakest failure detector?"[9]. This paper answers this question positively.

Contributions of the paper Although the definition of the failure detector $A\Omega$ is useful for anonymous systems, as it does not involve processes identities, it allows to (eventually) break symmetry, as it eventually singles out one process. We are interested in failure detectors that preserve anonymity in the following sense: for any process p and any sequence of failure detector outputs $d = d[1], d[2], \ldots$ at process p, the same sequence might be output at every process whithout violating the specification of the failure detector. An example of such failure detector is AP which provides at each process an eventually accurate estimation of the number of faulty processes. Anonymity-preserving failure detectors cannot be relied upon to differentiate otherwise identical processes. Within this framework, we identify the weakest failure detector for consensus in the shared memory model. In more details the paper makes the following contributions:

1. It first defines (Section 3) the class of anonymity-preserving failure detectors and a new failure detector denoted C. Failure detector C might be seen as a shared loose failure counter. It guarantees that after each new failure the counter is eventually incremented, and in case two or more processes are non-faulty, the counter eventually stabilizes. Let us notice that even if several failures occur, the counter might be incremented only once. C is thus far from providing an accurate tally of failures.

2. The paper shows that C is powerful enough to solve consensus while tolerating any number of failures (Section 4). The algorithm is uniform, in the sense that it does not require the total number of processes n to be known. Striving to not reinvent the wheel, the protocol relies on standard shared memory constructs, namely adopt-commit [23] and safe-agreement [12] objects.

3. It is then shown that C can be emulated using any anonymity-preserving failure detector D powerful enough to solve consensus (Section 5). The extraction algorithm reuses in part the techniques developed by Zielinski [33] for proving statement of this type in the shared memory model with identities. Interestingly, the proof does not rely on the specifics of the impossibility of fault-tolerant consensus but rather on the fact this task cannot be solved non-anonymously wait-free among two processes.

4. Finally, it focuses on *k-set agreement* [15], a task that generalizes consensus by allowing up to k distinct values to be decided. A generalization denoted C_k of failure detector C is introduced and a C_k-based algorithm is presented (Section 6).

2 Computational Model

We recall in this section the main points of the asynchronous, anonymous and crash-prone model equipped with failure detectors.

Anonymous, asynchronous crash-prone shared memory model We consider an asynchronous and crash-prone shared-memory system consisting in $n \geq 2$ processes. Each process p has a unique *index* i in the range $[1, n]$. The indexes are used only for modeling purpose. The process with index i is denoted by p_i. Processes are *anonymous* in the sense that they run the same code and are not aware of their index. Hence, in an execution, processes may behave differently only as a result of the underlying schedule or because their initial states differ. Processes communicate via a *shared memory* that consists in an unbounded number of multi-writer/multi-reader atomic registers. Processes are *asynchronous*, e.g., each process runs at its own speed, independently of the other processes. Processes may fail by *crashing*. A process that crashes stops executing its code and never recovers.

Failure detector For modeling failure detectors, we assume the existence of a global clock whose values are the non-negative integers. The clock is not accessible to the processes. Let $\Pi = \{p_1, \ldots, p_n\}$ be the set of processes. A process may fail by *crashing*, i.e., prematurely stops executing its code. A process that has crashed never recover. A *failure pattern* is a function $\mathcal{F} : \mathbb{N} \to 2^{\Pi}$ that specifies the set of processes that have failed at each time $\tau \in \mathbb{N}$. Let $\mathsf{faulty}(\mathcal{F})$ denote the set of processes that fail in $\mathcal{F}$, i.e., $\mathsf{faulty}(\mathcal{F}) = \bigcup_{\tau \geq 0} \mathcal{F}(\tau)$. The set of processes that do not fail is $\mathsf{correct}(\mathcal{F}) = \Pi \setminus \mathsf{faulty}(\mathcal{F})$. When the failure pattern is clear from the context, we say that a process p_i is *correct* is $p_i \in \mathsf{correct}(\mathcal{F})$ and *faulty* otherwise. An *environment* is a set of failure patterns. Unless specified otherwise, we assume the *wait-free* environment that contains every failure pattern in which at least one process is correct.

A failure detector [14] provides at each process some information on the underlying failure pattern. Each process can *query* its local failure detector module. Such a query returns a value in some (possibly infinite) range $\mathcal{R}$. The outputs of the failure detector during an execution is described by a *failure detector history*. A failure detector history is a function $H : \Pi \times \mathbb{N} \to \mathcal{R}$ that maps each pair (process index, time) to a value in the failure detector range $\mathcal{R}$. The value returned by the failure detector at process p_i at time τ is $H(p_i, \tau)$. A failure detector D with range $\mathcal{R}$ associates a non-empty set of histories $D(\mathcal{F})$ to each failure pattern $\mathcal{F}$.

Protocol and executions A *distributed algorithm* or a *protocol* consists in n copies of a local algorithm $\mathcal{A}$, one per process. An *execution* or a *run* is a finite or infinite sequence of *steps*. In a step, a process (1) queries the failure detector or (2) reads or (3) writes a shared register and performs some local computation. A *configuration* describes the local state of each process and the state of each shared register.

An *execution* or a *run of a protocol $\mathcal{A}$ using failure detector D in environment $\mathcal{E}$* is a tuple $e = (\mathcal{F}, H, I, S, T)$ where $\mathcal{F}$ is a failure pattern in $\mathcal{E}$, H a failure detector history in $D(\mathcal{F})$, I is an initial configuration of $\mathcal{A}$, S a sequence of steps of $\mathcal{A}$ and T a non-decreasing sequence of times. S is called a *schedule* and the ith step $S[i]$ of S takes place at time $T[i]$. A tuple $e = (\mathcal{F}, H, I, S, T)$ represents an execution of $\mathcal{A}$ if and only if (1) S and T are both finite in which case they have the same length, or are both infinite, (2) no processes take a step after it has crashed, (3) if step $S[i]$ is a failure detector query by process p_j that returns d, then $d = H(p_j, T[i])$, i.e., the failure detector queries return values that are consistent with the history H, (4) the steps taken in S are consistent with $\mathcal{A}$, (5) the timings of read and write steps, together with the values written or read in these steps are consistent with the atomic semantic of the shared registers and, (6) if S is infinite, every correct process takes infinitely many steps in S. Condition (5) can be reformulated as follows [25]:

For any i, j, if step $S[i]$ causally precedes step $S[j]$, $T[i] < T[j]$. Formalizing these conditions is not difficult but tedious. We refer to prior work [18, 25, 33] on failure detector in the shared memory model for a formal treatment.

Consensus and k-set agreement Consensus and k-set agreement are examples of *distributed tasks*. In the *consensus* task, each process starts with a value taken from some set $\mathcal{V}$ and is required to *decide* a value subject to the following requirements: (*Validity*) any decided value is the initial value of some process, (*Agreement*) no two distinct values are decided and (*Termination*) every non-faulty process decides. In *binary* consensus, the set of possible initial values is restricted to $\mathcal{V} = \{0, 1\}$. The *k-set agreement* task has the same validity and termination requirement of consensus but relaxes agreement as follows: (*k-agreement*) no more than k distinct values are decided.

Let $\mathcal{A}$ be a protocol that uses a failure detector D. We assume that each process p is equipped with two local variables, one, read-only in_p and the other, write-once out_p whose purpose is to store the input and decision values, respectively, of p. Let T be the consensus or k-set agreement tasks. We say that protocol $\mathcal{A}$ *solves* T *using failure detector* D *in environment* $\mathcal{E}$ if in any infinite execution $e = (\mathcal{F}, H, I, S, T)$ of $\mathcal{A}$ where $\mathcal{F} \in \mathcal{E}$ and for every process p, in_p is initialized to some value $v \in \mathcal{V}$, every process $q \in \mathsf{correct}(\mathcal{F})$ eventually *decides*, i.e. writes a value $\neq \perp$ to out_p and the values written satisfy the validity and (k)-agreement requirements of T.

Comparing failure detectors Let D and D' be two failure detectors. D is said to be *as least as weak as* D' in environment $\mathcal{E}$, denoted $D \leq_{\mathcal{E}} D'$ if there is a protocol $\mathcal{T}_{D' \to D}$ that *emulates* D *using* D'. Specifically, such a protocol maintains at each process p a local variable *out-*D_p whose successive values are in the range of D. $\mathcal{T}_{D' \to D}$ emulates D if for every failure pattern $\mathcal{F} \in \mathcal{E}$, in any execution with failure pattern $\mathcal{F}$, there exists an history $H \in D(\mathcal{F})$ such that for every process p and every time τ, $H(p, \tau) =$ *out-*D_p^τ where *out-*D_p^τ denotes the value of the variable *out-*D at p at time τ. D and D' are said to be *equivalent* in environment $\mathcal{E}$ if $D \leq_{\mathcal{E}} D'$ and $D \leq_{\mathcal{E}} D'$.

Finally, a failure detector D is said to be a *weakest failure detector* for a task T in environment $\mathcal{E}$ if (1) there is a protocol that solves T using D in $\mathcal{E}$ and (2) for every failure detector D' that can be used to solve T, $D \leq_{\mathcal{E}} D'$. For example, Ω is a weakest failure detector for consensus [13] in asynchronous systems with identities.

3 Anonymity-preserving failure detectors

This section introduces the class of anonymity preserving failure detectors and defines the new failure detector C.

The class of anonymity-preserving failure detectors Intuitively, a failure detector is anonymity preserving if it cannot be relied upon to break symmetry among the processes. In the context of anonymous systems, a failure detector history H is *anonymity-preserving* if for every time τ and every processes indexes i, j, $H(p_i, \tau) = H(p_j, \tau)$. That is, two queries at the same time by different processes return the same value. Hence, in such history, the value output by the failure detector only depends on the time at which the failure detector is queried, and does not depend on the querying process. An anonymity preserving history is thus a function $H : \mathbb{N} \to \mathcal{R}$ that maps time to values in the failure detector range.

A failure detector is *anonymity preserving* if for every failure pattern $\mathcal{F}$, for every $p_i \in \Pi$ and every history $H \in D(\mathcal{F})$, the anonymity-preserving history H':

$$\forall p_j \in \Pi, \forall \tau, H'(p_j, \tau) = H(p_i, \tau)$$

also belongs to $D(\mathcal{F})$. Intuitively, any sequence of values output by the failure detector at process p_i may have been returned at every other process. That is, if $d = d^1, d^2, \ldots$ is a legal sequence of output for process p_i for some failure pattern $\mathcal{F}$, then d is also a valid sequence at any process $p_j \neq p_i$, for the same failure pattern $\mathcal{F}$.

For instance, the failure detector $A\Omega$ [10] eventually distinguish a unique correct process. It provides to each process a single bit whose value eventually is 0 except for one correct process. More precisely, the range of $A\Omega$ is $\{0, 1\}$ and, for every failure pattern $\mathcal{F}$, the history $H : \Pi \times \mathbb{N} \to \{0, 1\}$ belongs to $A\Omega(\mathcal{F})$ if and only if:

Eventual leadership. there exists a time τ and a process $p_i \in \mathsf{correct}(\mathcal{F})$ such that for every $\tau' \geq \tau$ and every process p_j, $H(p_j, \tau') = 1 \iff j = i$.

Clearly, $A\Omega$ is not an anonymity-preserving failure detector. In every legal history, there is a time after which distinct values are output at least two processes.

An example of an anonymity-preserving failure detector is an identity-free variant [30] of the perfect failure detector, denoted AP in [10]. The range of AP is $\mathbb{N}$ and, for any failure pattern $\mathcal{F}$ the history $H : \Pi \times \mathbb{N} \to \mathbb{N}$ belongs to $AP(\mathcal{F})$ if and only if:

Accuracy. For every time τ and every process p_i, $H(p_i, \tau) \leq |\mathcal{F}(\tau)|$

Completeness. There exists a time τ such that for all $\tau' \geq \tau$, $H(p_i, \tau') = |\mathcal{F}(\tau')|$

AP is an anonymity-preserving failure detector. If for failure pattern $\mathcal{F}$ $d = f^1, f^2, \ldots$ is a valid sequence of outputs for process p_i, so it is for any process $p_j \neq p_i$.

Failure detector C Failure detector C is somewhat related to the *signaling failure* detector $\mathcal{FS}$ [26]. The range of failure detector $\mathcal{FS}$ is {*green*,*red*}. While no failures occur, the output of $\mathcal{FS}$ is *green*. Once a failure occurs, and only if it does, the failure detector must eventually output *red* at every correct process.

Failure detector C might be seen as an unreliable variant of $\mathcal{FS}$. The range of failure detector C is the integers. At each process, the sequence of integers output by C is non-decreasing, and after each new failure, the output of the failure detector is eventually increased. However, when at least two processes are correct in the underlying failure pattern, C output eventually stabilizes. That is, after some time, every query to C by process p_i returns the same value, for each process p_i. More formally, for every failure pattern $\mathcal{F}$, history $H : \Pi \times \mathbb{N} \to \mathbb{N}$ belongs to $C(\mathcal{F})$ if and only if:

1. *Monotonicity.* For every process p_i, for every times $\tau \leq \tau'$, $H(p_i, \tau) \leq H(p_i, \tau')$;
2. *Signaling* For every times $\tau, \tau' : \tau < \tau'$, for every processes p_i, p_j, if $|\mathcal{F}(\tau)| < |\mathcal{F}(\tau')|$, there exists a time $\tau'', \tau' \leq \tau''$ such that $H(p_i, \tau) < H(p_j, \tau'')$;
3. *Convergence.* If $|\mathsf{correct}(\mathcal{F})| > 1$, for every process p_i, there exists a time τ such that for every $\tau'\tau \leq \tau'$, $H(p_i, \tau) = H(p_i, \tau')$.

4 A C-based consensus protocol

This section presents a consensus protocol based on failure detector C. To simplify the presentation, we concentrate on *binary consensus* in which the set of possible inputs is $\{0, 1\}$. Multivalued consensus can be solved from binary consensus by successively agreeing on the bits of one of the proposed value. The protocol is depicted in Figure 1. Besides registers, it relies on standard shared memory constructs, namely adopt-commit [23] and safe agreement [11, 12] objects, that we describe next.

Base objects An *adopt-commit object* supports a single operation denoted propose(v) where v is a value from some finite set $\mathcal{V}$. Such an operation returns a couple (b, u) where b is either *adopt* or *commit* and u is a value in $\mathcal{V}$, subject to the following requirements [23, 30, 4]:

1. *Validity.* If an operation returns (d, v) then v is the input of a propose() operation.
2. *Agreement.* If an operation returns $(commit, v)$ then each output is either $(adopt, v)$ or $(commit, v)$.
3. *Convergence.* If the input of every operation is v, then every output is $(commit, v)$.
4. *Termination.* Each operation by a non-faulty process produces an output.

A shared-memory implementation of an adopt-commit object that tolerates an arbitrary number of crash-failures can be found in [4]. The implementation ([4], Algorithm 2) uses two multi-writer/multi-reader registers and a *conflict-detector*, which in turn can be implemented in a wait-free manner using only $\text{fact}^{-1}(|\mathcal{V}|)$ multi-writer/multi-reader registers ([4], Algorithm 3). Algorithms 2 and 3 do not use processes identities, and are thus suitable for the anonymous shared-memory model.

The *safe agreement* object, originally introduced by Borowsky and Gafni [11] allows the processes to propose values and to agree on a single value. It is at the heart of the BG-simulations [11, 24] in which it is used by simulators to agree on each step of the simulated processes. Different specifications of a safe agreement object can be found in the literature, e.g., [5, 12, 28]. Our specification below closely follows [5].

A safe agreement object supports two operations propose(v) where v is a value in $\{0, 1\}$[1] and read(). Both operations return either a value $u \in \{0, 1\}$ or $\perp$. Each process can invoke propose() at most once, while read() can be invoked arbitrarily many times. An execution is *well-formed* if (1) each process calls propose() at most once and, (2) no processes start a read() or propose() operation before its previous operation (if any) has returned. It is required that in any well-formed execution, the following properties are satisfied:

1. *Validity.* If an operation returns a value $v \neq \perp$, v is the input of a propose() operation.
2. *Agreement.* If values $v, v' \in \{0, 1\}$ are returned by some operation, $v = v'$.
3. *Termination.* Every operation performed by a non-faulty process terminates.

We say that a propose() operation is *successful* if it returns a value $\neq \perp$.

[1] More generally, v may belong to any finite set. Restricting to binary inputs is sufficient for our purpose, namely using failure detector C to solve binary consensus.

4. *Consistent reads.* Any read() operation that terminates and starts after a successful propose() operation has completed returns a non-$\perp$ value.

5. *Non-triviality.* Not every propose() operations return $\perp$.

Observe that the non-triviality property is satisfied in executions in which a process fails while performing a propose() operation. In the case in which no processes fail while performing propose(), it follows from the termination and non-triviality properties that at least one propose() operation is successful. Nevertheless, it is not guaranteed that every propose() operation is successful. However, the consistent reads property implies that if, after its propose() operation has returned, a process keeps reading the object, it eventually gets back a non-$\perp$ value.

When processes have unique identity, safe agreement objects can be implemented with registers, e.g., [11]. In anonymous systems, a safe agreement object implementation can be obtained by slightly modifying an anonymous binary consensus protocol by Attiya, Gorbach and Moran [6] designed for the asynchronous shared memory model with no failures. See appendix A for more details.

Algorithm 1 C-based binary consensus protocol.

```
 1: init SA[1,...]                                          ▷ Array of safe agreement objects
 2:      AC[1,...]                                          ▷ Array of adopt-commit objects
 3:      D ← ⊥                                              ▷ Decision register, initially ⊥
 4: function propose(v)                                     ▷ v ∈ {0,1}
 5:     est ← v; start tasks T1, T2;
 6: task T1:
 7:     for r = 1,2,... do
 8:         repeat d ← C-query() until d ≥ r end repeat
 9:         aux ← SA[r].propose(est)                        ▷ aux ∈ {0,1,⊥}
10:         if aux = ⊥ then
11:             repeat aux ← SA[r].read(); d ← C-query()
12:             until (d > r) ∨ (aux ≠ ⊥)
13:         end if
14:         (b,u) ← AC[r].propose(aux)                      ▷ b ∈ {adopt,commit}, u ∈ {0,1,⊥}
15:         case b = commit ∧ u ∈ {0,1} then D ← u; return
16:              b = adopt  ∧ u ∈ {0,1} then est ← u
17:              default                then nop            ▷ u = ⊥
18:         end case
19:     end for
20: task T2:
21:     repeat u ← D until u ≠ ⊥ end repeat
22:     stop task T1; return u
```

Description of the protocol Algorithm. 1 consists in two tasks denoted T1 and T2, launched in parallel at each process p (line 5). In task T2, process p keeps reading a shared register D, whose initial value is $\perp$, until it sees some non-$\perp$ value u. u is then decided by p (line 22).

In task T1, processes proceed in asynchronous rounds aiming at writing a single non-$\perp$ value to D. An adopt-commit object and a safe agreement object denoted respectively $AC[r]$ and $SA[r]$ are associated with each round r. Following a standard design pattern, e.g., [3, 24], the processes that

enter round r first try to reach agreement by accessing the safe agreement object $SA[r]$ (line 9) and then check whether agreement has been achieved using the adopt-commit object $AC[r]$ (line 14).

In more details, each process p maintains an estimate *est* that contains the value it currently favors. In round r, process p proposes its estimate to $SA[r]$ (line 9) and, if its operation is unsuccessful (line 10), enters a loop in which it repeatedly reads the object (line 11). If no processes that enter round r fail, at least one of the invocations of propose() on $SA[r]$ is successful (non-triviality and termination properties of safe agreement) and thus by keeping reading the object, a process eventually obtains a non-$\perp$ value (consistent reads property of safe agreement). Hence, in the case in which no processes entering round r fail, every process that enters that round eventually obtains a non-$\perp$ value, either because its propose() operation is successful, or as a result of a read() operation. Note that this value is the same for every process (agreement property of safe agreement).

However, some of the processes that enter round r may fail. In this case, each propose() operation may be unsuccessful, and every read() operation may return $\perp$. We rely on failure detector C to ensure progress as follows:

- A process is allowed to enter a round r only if its local failure detector module output is larger than or equal to r (line 8);
- A process exits the loop in which it is trying to obtain a non-$\perp$ value from $SA[r]$ by performing read() operation whenever its local failure detector output is strictly larger than d_c (line 12).

This simple mechanism prevents processes from getting stuck in any round r in which a failure occurs. Indeed, a process p failing in round r must have obtained from C a value $d_c \geq r$ (line 7). Then, following the crash of p, due to the signaling property of C, C eventually outputs at every non-faulty processes values strictly larger than d_C, allowing these processes to exit the loop in which the safe agreement object $SA[r]$ is read (lines 11–12).

To reconcile processes that have obtained a non-$\perp$ value form $SA[r]$ and those to which C has signaled a failure, we use the adopt-commit object $AC[r]$ (line 14). Each process p keeps in its local variable *aux* the result of its operations (at lines 9 and 11) on $SA[r]$, e.g., $\perp$ or some value $v \in \{0, 1\}$. In the second part of round r, process p proposes the value stored in *aux* to $AC[r]$ (line 14). If it gets back $(adopt, u)$, where $u \neq \perp$ it changes its estimate to u (line 16). A process that receives $(commit, u)$ can thus safely write u to the decision register D, as it follows from the agreement of adopt-commit that every propose() operation returns $(commit, u)$ or $(adopt, u)$. Hence, every process either writes u to D or changes its estimate to u, thus preventing any value $u' \neq u$ to be written to D in subsequent rounds. Finally, if a process p receives $(*, \perp)$, then no process writes to D in the current round r, and p leaves its estimate unchanged (line 17).

As for termination, a process decides as soon as it reads a non-$\perp$ value from D (task T2). Let us observe that this happens if there is a round r in which (1) enters only one process, and this process is correct or (2) enter only correct processes, and at each of these processes, the largest output of C is r. Clearly, if only one process p enters round r, its propose() operation on $SA[r]$ returns a non-$\perp$ value u (non-triviality property of safe agreement). u is then the only value proposed to $AC[r]$. p thus receives $(commit, u)$ from $AC[r]$ (convergence property of adopt-commit) and then writes u to D. Condition (1) is satisfied in executions in which there is only one correct process.

For the second condition, if only correct process enters round r, at least one of the propose() operation on $SA[r]$ is successful. Moreover, no process can exit the reading loop (lines 11-12) without having obtained a non-$\perp$ value from $SA[r]$, as C never outputs a value larger than r to those processes. Since every non-$\perp$ value returned by operations on $SA[r]$ are the same, only one

value is proposed to $AC[r]$, from which we conclude that only $(commit, v)$, where $v \neq \perp$, is returned by each propose() operation performed on $AC[r]$ (convergence property of $AC[r]$.). Hence a value is written to D in round r. Condition (2) is met in every execution in which there is at least two correct processes, since in that case the output of C eventually stabilizes at every correct process, and the stabilization value is larger than every value output at faulty processes.

Proof of the protocol We consider an arbitrary execution e of the protocol. We assume that every correct process invokes propose() at line 4. Let $\mathcal{F}$ be the failure pattern, and $H \in C(\mathcal{F})$ the failure detector history in e. Consensus agreement and validity follow from the agreement and validity properties of the adopt-commit and the safe agreement objects (see the proof of Theorem 4.3). The proof of termination relies on the following Lemma, which stems from the signaling property of the underlying failure detector C.

Lemma 4.1. *Let τ_c be the time at which the last crash occurs in e (i.e., $\mathcal{F}(\tau) = \mathcal{F}(\tau_c)$, for every $\tau_c < \tau$). There exists $T \in \mathbb{N}$ such that: (i) for every faulty process p, for every time $\tau \leq \tau_c$, $H(p, \tau_c) \leq T$ and, (ii) there exists a time τ_T such that for every correct p, for every time $\tau, \tau_T \leq \tau, T < H(p, \tau)$.*

In other words, there is value T such that each query to C by a faulty processes returns a value smaller than T while the queries by the correct processes eventually return values strictly greater than T.

Proof. Let $T = \max\{H(p, \tau) : p \in \mathsf{faulty}(\mathcal{F}) \text{ and } \tau < \tau_c\}$. Item (i) is an immediate consequence of the definition of T. By the definition of T, there exists a faulty process q and a time $\tau_1 < \tau_c$ at which the output of C at q is T, i.e., $H(q, \tau_1) = T$. Let p be a correct process. As for every time $\tau, \tau_1 < \tau_c \leq \tau$, $\mathcal{F}(\tau_1) < \mathcal{F}(\tau)$, there exists a time τ_p, $\tau_1 < \tau_p$ such that $T = H(q, \tau_1) < H(p, \tau_p)$ (Signaling property of C). Let $\tau_T = \max\{\tau_p :$ p is correct$\}$. By the monotonicity property of C, for every time $\tau, \tau_T \leq \tau$ and every correct process p, $T < H(p, \tau_p) \leq H(p, \tau)$, which proves item (ii). □

We say that a process *decides* when it returns a value at line 22 in task T2.

Lemma 4.2 (Termination)**.** *Every correct process decides.*

Proof. Assume for contradiction that no correct processes decide. As a process decides as soon as it reads a non-$\perp$ value from the decision register D (task T2), no non-$\perp$ value is ever written to D. Hence no process exits task T1 by executing the **return** statement of line 15.

We say that a process *enters round* r is it exists the **repeat** loop (line 8) of round r. Note that this occurs when a query to C returns a value $d, r \leq d$. A process *is blocked in round* r if it enters round r and never enters round $r + 1$.

Let p denote a correct process and let d be a value returned to p by a query to C. Observe that p enters round d. Indeed, for every round $r < d$, p is not blocked in round r. First, as the successive values returned by the queries to C form a non-decreasing sequence, the condition for entering round r is eventually satisfied at p (line 8). Second, in round r, as p does not fail, its calls to propose() on the safe agreement and adopt-commit objects $SA[r]$ and $AC[r]$ terminate. Finally, the **repeat** loop at lines 11–12 eventually ends since C eventually outputs a value greater than r at process p.

We consider two cases according to the number of correct processes in the execution.

- $|\mathsf{correct}(\mathcal{F})| = 1$. Let p be the correct process of the execution. By Lemma 4.1 and the monotonicity property of C, there exists an integer T such that (1) after some time every query to C by p returns a value greater than T and (2) every query by a faulty process returns a value $\leq T$. It follows that there exists a round $r, T \leq r$ such that no process but p enters round r (Faulty processes cannot enter round r since they do not obtain large enough values from C. On the contrary, p does enter round r as a query by p to C returns r.).

  Therefore, p is the only process that calls propose() (on line 9) on the safe agreement object $SA[r]$. By the non-triviality property of safe agreement, it must get back a value $v \neq \perp$ from that call, which it then proposes to the adopt-commit object $AC[r]$ (line 14). This operation returns $(commit, v)$ as p is the only process that accesses the object. It thus follows from the code that p writes v to D. This is a contradiction.

- $|\mathsf{correct}(\mathcal{F})| > 1$. In this case, at each correct process p the output of C eventually stabilizes to a value denoted d_p. Let $R = \max\{d_p : p \in \mathsf{correct}(\mathcal{F})\}$ and let P be the set of processes at which the failure detector output is eventually R. As seen above, each process $p \in P$ eventually enters round R, and it follows from Lemma 4.1 that no faulty process enters rounds R. Hence every call to propose() on the safe agreement object $SA[R]$ returns. By the non-triviality property, at least one of these calls is successful, i.e., returns a non-$\perp$ value. By the agreement property of safe agreement, all non-$\perp$ values returned by successful propose() operations are the same. Let v denote this value.

  Let $p \in P$. Suppose that p propose() operation on $SA[R]$ returns $\perp$. p cannot exit the **repeat** loop on lines 11-12 by getting a value $d > R$ from C, as by definition R is the largest value output by C at p. However, as there is a successful propose() operation on $SA[R]$, one of p read() operations on $SA[R]$ returns a non-$\perp$ value. By the agreement property of safe agreement objects, this value is v.

  Therefore, every propose() operation on the adopt-commit object $AC[R]$ has v as input. Each such operation thus returns $(commit, v)$, from which we conclude that v is written to the decision register D. This is a contradiction. □

Theorem 4.3. *Protocol 1 solves binary consensus using failure detector C.*

Proof. Validity directly follows from the validity properties of safe agreement and adopt-commit object. Termination follows from Lemma 4.2. For agreement, we prove that all values written to the decision register D are the same.

Let r be the first round in which a value is written to D. That is, a process writes to D after entering round r and before leaving that round. Let v be the value written by that process. By the code (line 15), p has previously called propose() on $AC[r]$ and has received $(commit, v)$ from that call. By adopt-commit agreement, every other propose() operation on $AC[r]$ that terminates returns $(adopt, v)$ or $(commit, v)$. It thus follows that (1) every value written to D in round r is v and, (2) every process that does not return in round r (at line 15) sets its estimate *est* to v before leaving round r (line 16). Therefore, for any round $r' \geq r$, the estimate of any process entering round r' is v, from which we conclude that no value $\neq v$ can be written to D after round r. □

5 C is necessary to solve consensus

Let X be an anonymity-preserving failure detector, and assume that there is a protocol $\mathcal{A}$ that solves consensus using X. We present (Algorithm 2) a protocol $\mathcal{T}_{X \to C}$ that emulates C using X in the wait-free environment.

Overview As in previous protocols that emulate weakest failure detectors [13, 19, 25, 33], in $\mathcal{T}_{X \to C}$ each process locally simulates many possible runs of algorithm $\mathcal{A}$. According to the output of these runs, information on the failure pattern is inferred and the desired weakest failure detector emulated.

Let $\mathcal{F}$ denote the failure pattern underlying the execution of $\mathcal{T}_{X \to C}$. In order to simulate valid runs of $\mathcal{A}$, e.g., runs indistinguishable from reals runs of $\mathcal{A}$, samples from the underlying failure detector X have to be collected. Those samples are then used in the simulation of each step in which a query to failure detector X occurs. Hence, each process p must collect samples from its failure detector module, but also from other processes. Precedence relationships between samples should also be maintained to order to simulate valid runs of $\mathcal{A}$. For example, the simulation must avoid using a sample from some faulty process q if a sample taken after the failure of q has already been used. In systems with identities, this is usually achieved by maintaining a DAG, where each vertex v contains failure detector sample d and a process id, and for any successor v' of v, the sample d' associated with v' has been taken after d. In the anonymous shared memory model, the lack of identifiers make tracking precedence relationships difficult and the standard technique [13] does not apply. However, in the case of anonymity-preserving failure detectors, the samples taken by each process p from its local failure detector module are sufficient to simulate runs of $\mathcal{A}$, even with more than one participating process. This is because the sequence of samples obtained by p might have been also obtained by every other processes in some execution with the same failure pattern $\mathcal{F}$.

In $\mathcal{T}_{X \to C}$, each process p simulates executions of $\mathcal{A}$ in which at most two processes, denoted q_0 and q_1, participate with input 0 and 1 respectively. On one hand, for such an execution e by adding *clones* of q_0 and q_1 one may construct an indistinguishable execution e' in which the number of participating processes matches the number of correct processes in $\mathsf{correct}(\mathcal{F})$. It thus can be shown that execution e is indistinguishable from the point of view of q_0 and q_1 from some real execution of $\mathcal{A}$ with failure pattern $\mathcal{F}$. On the other hand, there must exist an interleaving of the steps of q_0 and q_1 such that the corresponding emulated execution of $\mathcal{A}$ does not decide. Otherwise, algorithm $\mathcal{A}$ together with the sequence of failure detector samples collected by p can be used to solve binary consensus wait-free and without failure detector by two non-anonymous processes q_0 and q_1, contradicting the impossibility of consensus.

Operationally, process p explores every possible two-processes schedules of $\mathcal{A}$ in a particular, *corridor*-based order, as in [25, 33]. Whenever a decision occurs in the execution simulated by p, a shared counter is incremented, and the output of C at p is set to the new value of the counter. We prove (1) that following any (real) process failure, p eventually simulates an execution of $\mathcal{A}$ in which a decision occurs, and due to the order in which schedules are explored, (2) that eventually p keeps simulating one infinite execution in which no processes decide. The correctness of the emulation C then follows (1) and (2).

Algorithm $\mathcal{A}'$ Let MEM denote the (not necessarily finite) array of registers used by $\mathcal{A}$. Recall that in a step of $\mathcal{A}$, a process performs a read() or write() operation on some register $MEM[\ell]$ or a query() operation, from which it gets back an output from the underlying failure detector X. It may then perform some local computation. Instead of simulating runs of $\mathcal{A}$, we are going to simulate runs of a slightly modified version of $\mathcal{A}$, called $\mathcal{A}'$, defined as follows. The purpose of the modification is to help tracking causality relations between steps of the algorithms.

In algorithm $\mathcal{A}'$, each process has an extra local counter η whose initial value is 1. Each register $MEM[\ell]$ is divided in two fields, *data* and *ctr*. $MEM[\ell].data$ is initialized as specified by $\mathcal{A}$ while the initial value of $MEM[\ell].ctr$ is 0. For each integer ℓ and value v, each operation $MEM[\ell].\mathsf{write}(v)$ in $\mathcal{A}$ is replaced in $\mathcal{A}'$ by $MEM[\ell].\mathsf{write}(\langle v, \eta\rangle)$, i.e., v and the current value of the local variable η are written to the *data* and *ctr* components, respectively, of $MEM[\ell]$. Similarly, each instruction of the form $v \leftarrow MEM[\ell].\mathsf{read}()$ in $\mathcal{A}$ is replaced in $\mathcal{A}'$ by $\langle v, \eta'\rangle \leftarrow MEM[\ell].\mathsf{read}()$; $\eta \leftarrow \max(\eta, \eta' + 1)$. Finally, after each write(), read() or query() operation η is incremented ($\eta \leftarrow \eta + 1$). For each step s of the modified algorithm $\mathcal{A}'$, we define $\eta(s)$ as the value of η immediately before it is incremented (e.g., immediately before $\eta \leftarrow \eta + 1$ is performed). Obviously, these modifications do not affect the correctness of $\mathcal{A}'$, i.e., $\mathcal{A}'$ solves consensus using X.

Causality Let r be a run of $\mathcal{A}'$ with two processes q_0, q_1, where the input of $q_i, i \in \{0, 1\}$ is i. Note that in these particular executions, although the processes are anonymous, we can assume that the values written are unique, as they can be tagged with the process input and a sequence number. For any two steps s, s' in r, s *causally precedes* s', denoted $s \preceq s'$ if and only if :

1. s and s' are performed by the same process in that order or,
2. in s a value v is written to some register $MEM[\ell]$, and in s' v is read from $MEM[\ell]$ or,
3. there exists a steps s'' such that $s \preceq s''$ and $s'' \preceq s'$.

The following Lemma follows from the management of the variables η in $\mathcal{A}'$.

Lemma 5.1. *Let r be a run of $\mathcal{A}'$ by two processes q_0, q_1 with input 0 and 1 respectively. For every steps s, s' of r, $s \preceq s' \implies \eta(s) < \eta(s')$.*

Proof. If s and s' are two steps by the same process q_i, occurring in that order, the lemma follows from the fact that η is incremented at the end of each step of q_i. Suppose value v is written to some register in s and read in s'. Then $\eta(s') = \max(old, \eta(s)+1) > \eta(s)$ where *old* is the value of η at the beginning of s' and the lemma follows. Otherwise, there exists steps $s_0(= s), s_1, \ldots, s_m(= s')$ such that for every $j, 0 \leq j \leq m-1$, steps s_j and s_{j+1} are performed by the same process in that order or a value written in s_j is read in s_{j+1}. In that case, the lemma follows from an easy induction on m. □

Collecting failure detector X samples In order to simulate a run r of $\mathcal{A}'$, each process p must be able to select appropriate outputs from the failure detector X for each query() steps in r. Since failure detector X is anonymity-preserving, p does not need to know outputs of X at other processes p'. Indeed, for any failure pattern $\mathcal{F}$ and any finite or infinite sequence $x = x_1, x_2, \ldots$ of outputs of X collected by p in a run with failure pattern $\mathcal{F}$, there is a run with the same failure pattern in which every process see the same sequence x of outputs of X. Therefore, in order to provide failure detector values for the simulation of runs of $\mathcal{A}'$, p simply builds an ever growing

sequence of failure detector outputs $x[1], x[2], \ldots$ by repeatedly querying its local failure detector module.

Induced schedules of $\mathcal{A}'$ Each process p simulates runs of $\mathcal{A}'$ in which at most two processes, denoted q_0 and q_1, take steps with initial values 0 and 1 respectively. We next describe how a binary sequence and a sequence of failure detector X outputs induce a schedule S of $\mathcal{A}'$, that is a sequence of steps of $\mathcal{A}'$.

Let x denote a sequence of failure detector outputs, obtained from X at increasing times, and let λ denote a binary sequence. Intuitively, λ describes in which order processes take steps in S and x supplies failure detector outputs for simulating query(). A difficulty is to choose output in x for each query() step of S in such a way that there is a real execution of $\mathcal{A}'$ indistinguishable to q_0 and q_1 from S.

Schedule S is defined inductively. Recall that a configuration c, in the context of a two processes schedule consists in a triplet (s_0, s_1, MEM) where $s_i, i \in \{0,1\}$ is the local state of q_i and the array MEM contains the current value of each register used by $\mathcal{A}'$. In the initial configuration c_0 of S, $c_0.s_i, i \in \{0,1\}$ reflects the fact that the initial value of q_i is i and $c_0.MEM$ is initialized as specified by $\mathcal{A}'$. The ith step of S is taken by process $q_{\lambda[i]}$ and is deduced from $\mathcal{A}'$ applied to the local state of $q_{\lambda[i]}$ in configuration c_{i-1}. If this step is a read() or write() step, it is simulated by reading or writing a value to/from MEM. If the step is a query() operation, it is simulated by taking $x[\eta_{\lambda[i]}]$ as its result, where $\eta_{\lambda[i]}$ is the value of the variable η at process $q_{\lambda[i]}$ in configuration c_{i-1}. Configuration c_i is then derived from c_{i-1} in the obvious way.

The choice of output for each simulated query() preserves causality in the following sense: Let s and s' be steps of S in which X is queried and assume that $s \preceq s'$. Let j, j' the indices in x of the values returned by these queries in the simulation. Then $x[j]$ is obtained from X before $x[j']$, i.e., $j < j'$, as one would expect. Indeed, let q_i and $q_{i'}$ be respectively the processes that perform s and s', and let $\eta(s)$ and $\eta(s')$ be the value of η at process q_i and $q_{i'}$ in the configurations that immediately precede s and s', respectively. The results of the queries in s and s' are $x[\eta(s)]$ and $x[\eta(s')]$. By Lemma 5.1, as $s \preceq s'$, $\eta(s) < \eta(s')$.

Indistinguishability of induced schedules from real runs Given a binary sequence λ and a sequence x of outputs of X, the schedule $S_{\lambda,x}$ induced by λ and x may not correspond to a real execution of $\mathcal{A}'$. More precisely, for the simulation of S to be meaningful, we need that there exists a real run r of $\mathcal{A}'$ that is indistinguishable from S to q_0 and q_1. The schedule in r may differ from S, but the successive states of q_i must be the same in S and r, for each $i \in \{0,1\}$. Next Lemma establishes the existence of r.

Lemma 5.2. *Let λ be a binary sequence. Let x denote a (finite or infinite) sequence of outputs of X and let S denote the schedule induced by λ and x. Assume that there exists a failure pattern $\mathcal{F}$, a history $H \in X(\mathcal{F})$ and an increasing sequence of times $\tau_1 < \tau_2 < \ldots$ such that for every $i, x[i] = H(p, \tau_i)$ for some process p. If for every i, $|\mathcal{F}(\tau_i)| \leq n-2$, there exists a run of $\mathcal{A}$ indistinguishable from S to q_0 and q_1.*

Proof. Let $r = (\mathcal{F}, H, I, S', T)$ where $\mathcal{F}$ and H as are in the Lemma statement, and I is the initial configuration in which process q_i input is i, for each $i \in \{0,1\}$. S' is a schedule with exactly the same steps as S and T is a non-decreasing sequence of times, such that for every i, step $S[i]$ occurs

at time $T[i]$. To define S' and T, we associate with each step s in S a time $\theta(s)$ as follows: Let $prec(s)$ be the set of steps in S that causally precede s, i.e., $prec(s) = \{s' \in S : s' \preceq s\}$.

$$\theta(s) = \begin{cases} \tau_{\eta(s)} & \text{if } s \text{ is a } \mathsf{query}() \text{ step} \\ \max\{\theta(s') : s' \in prec(s)\} + 1 & \text{otherwise.} \end{cases}$$

S' is the sequence of steps in S, order according to θ. If two steps s, s' are such that $\theta(s) = \theta(s')$, they are performed by two distinct processes and we order the step performed by q_0 first. T is then the sequence $\theta(S'[1]), \theta(S'[2]), \ldots$ We next verify that r defines a run of $\mathcal{A}'$.

We check that for any two steps s_1, s_2, if $s_1 \preceq s_2$, then $\theta(s_1) < \theta(s_2)$. Suppose first that s_1 is a query step. If s_2 is also a query step, since $s_1 \preceq s_2$, it follows from Lemma 5.1 that $\eta(s_1) < \eta(s_2)$. Hence $\theta(s_1) = \tau_{\eta(s_1)} < \tau_{\eta(s_2)} = \theta(s_2)$. If s_2 is a read or write step, since $s_1 \in prec(s_2)$, $\theta(s_1) < \theta(s_2)$.

Suppose now that s_1 is read or write step. If s_2 is also read or write step, $\theta(s_1) < \theta(s_2)$ since $s_1 \in prec(s_2)$. Otherwise s_2 is a query. Observe that $\ell \leq \theta(s_1)$, where ℓ is the length of the longest causal chain of steps ending in s_1. By the management of the η variables in $\mathcal{A}'$, $\ell = \eta(s_1)$. If $\ell = \theta(s_1)$, as $s_1 \preceq s_2$, $\eta(s_1) < \eta(s_2)$ (Lemma 5.1) and thus, since $\eta(s_2) \leq \tau_{\eta(s_2)}$, $\theta(s_1) < \theta(s_2)$. Otherwise, there exists a query step $s \in prec(s_1)$ such that $\theta(s_1) = \tau_{\eta(s)} + \delta$, where δ is the length of the causal chain from s to s_1. Since $s \preceq s_1 \preceq s_2$, $\eta(s_2) = \eta(s) + \delta'$ where $\delta < \delta'$. Hence, as $\tau_{\eta(s)} + \delta \leq \tau_{\eta(s)+\delta} < \tau_{\eta(s)+\delta'} = \theta(s_2)$, $\theta(s_1) < \theta(s_2)$. Therefore, for any i, j, if $S'[i]$ causally precedes $S'[j]$ then $T[i] < T[j]$. Property (5) of a run is satisfied.

Clearly S' and T have the same length (property (1)). By assumption, at least two processes, say p and p' do not fail in $\mathcal{F}$. Algorithm $\mathcal{A}'$ being anonymous, and failure detector X anonymity preserving, we can think of run r as a run in which p and p' take the steps in lieu of q_0 and q_1, respectively. Hence, no process takes a step in r after it has failed (property (2)). By definition, if $s = S'[i]$ is a query step, $T[i] = \tau_{\eta(s)}$ and the value returned by the query is $d = x[\eta(s)] = H(p, \tau_{\eta(s)})$ for some process p. Since X is an anonymity preserving, d is also a valid output at time $\tau_{\eta(s)}$ for the process performing the query (property (3)). S and S' contain the same set of steps. For each process q_i, the steps of q_i occur in the same order in S and S'. Hence S and S' are indistinguishable to q_i. Finally, as the order of read/write operations on each register $MEM[\ell]$ is preserved in S', each read returns last value written. Hence property (4) of runs holds. □

Run r however may not be fair. A infinite run $r = (\mathcal{F}, H, I, S, T)$ is *fair* if every process in $\mathsf{correct}(\mathcal{F})$ take infinitely many steps in r. Given an infinite binary sequence λ, let $inf(\lambda) \subseteq \{0, 1\}$ the bits that appear infinitely many often in λ. Next Lemma expresses a sufficient condition for the existence of a fair run indistinguishable to q_0 and q_1 from the schedule induce by binary sequence and a sequence of failure detector outputs λ, x.

Lemma 5.3. *Let λ be an infinite binary sequence and x an infinite sequence of failure detector X outputs. Suppose that there exists a failure pattern $\mathcal{F}$, a sequence of times $\tau_1 < \tau_2 < \ldots$ and an history $H \in X(\mathcal{F})$ such that for every $i \geq 1$, $x[i] = H(p, \tau)$ for some process p.*

If $|\mathsf{correct}(\mathcal{F})| \geq 2$ and $\inf(\lambda) = \{0, 1\}$ then the schedule $S_{\lambda,x}$ induced by λ, x is indistinguishable from the schedule in a fair run r of $\mathcal{A}$.

In the induced schedule $S_{\lambda,x}$, only two processes take step. However, more than two processes may be correct in the failure pattern $\mathcal{F}$. We resolve this difficulty by adding clones of q_0 and q_1. A *clone* [20] of process q_i is a process that has the same input and the same code as q_i. p is scheduled in lock-step with q_i: it reads and writes the same values as p, and each of its queries to

X returns the same output as the queries by p. The latter is made possible by the fact that X is anonymity-preserving. The outputs of X at q_i are also valid outputs at any other processes.

Proof. Let $r = (\mathcal{F}, H, I, S', T)$ be the run indistinguishable to q_0 and q_1, as defined in the proof of Lemma 5.2. If $|\mathsf{correct}(\mathcal{F})| > 2$, we construct fair run r' by adding $|\mathsf{correct}(\mathcal{F})| - 2$ clones of q_0 to r. Each clone p has the same input as q_0, writes and reads the same value as q_0. The queries by p occur at the same times as the queries by q_0 and return the same values. Since X is an anonymity-preserving failure detector, there exists and history $H' \in X(\mathcal{F})$ such that for any time τ and any clone p, $H'(p, \tau) = H'(q_0, \tau) = H(q_0, \tau)$. Steps by clones are not seen by neither q_0 nor q_1. Thus r' is indistinguishable from r to q_0 and q_1. Clearly r' is a fair run of $\mathcal{A}'$. □

C emulation Algorithm 2 emulates failure detector C from any anonymity preserving failure detector X that can be used to solve consensus. It closely follows the emulation technique of Zielinski [33]. At each process p, the emulation consists in two tasks T and T' that run in parallel. In task T, p collects outputs of X by querying its local failure detector module. The outputs are stored in the array x. In task T', p recursively simulates every possible schedule of $\mathcal{A}'$ (lines 9-19). An infinite array A of registers is used to implement a weak shared counter. Each register $A[i]$ initial value is $\perp$. The counter is incremented by changing to $\top$ the value of the register with the smallest index containing $\perp$. The value of the counter is thus the largest index i of A such that $A[i] = \top$. Each time a process decides in a simulated schedule, the counter is incremented and the output of C is set to the counter new value (line 17).

Algorithm 2 $\mathcal{T}_{X \to C}$, where X can be used to solve consensus.

```
 1: init A[1...] ← [⊥,...]                                   ▷ Array of registers with initial value ⊥
 2: procedure C-emulation
 3:     x[1...] ← [⊥,...]                                    ▷ Array for storing outputs of X
 4:     c_0 ← initial configuration: q_i, i ∈ {0,1} input is i, MEM is initialized as prescribed by A'
 5:     P_0 ← {q_0, q_1}; λ_0 ← ϵ; OUT-C ← 0
 6:     start tasks T and T' where task T' is explore(λ_0, c_0, P_0);
 7: function explore(λ, c, P)
 8:     let U be the set of processes still undecided in c
 9:     for each P' ⊆ P ∩ U in an order consistent with ⊆ do
10:         for each q_i ∈ P' do
11:             let step be the next step of q_i in configuration c according to A'   ▷ simulate next step of q_i
12:             case step = read() from ℓth register then read c.MEM[ℓ]
13:                  step = write(v) to ℓth register then write v to c.MEM[ℓ]
14:                  step = X-query() then take x[ℓ] as the output of X, where ℓ is the value of η in c.s_i;
15:             end case
16:             perform local computation; update c.s_i
17:             if q_i has decided in c then let m ← min{ℓ : A[ℓ] = ⊥}; A[m] ← ⊤; C-OUT ← m
18:             end if                                    ▷ update (emulated) failure detector C output
19:             λ ← λ · i; explore(λ, c, P')
20:         end for
21:     end for
22: task T:                                                 ▷ Failure detector X sampling
23:     for i = 1, 2, ... do x[i] ← X-query() end for
```

In the following, we consider an arbitrary run of algorithm 2. Let $\mathcal{F}$ denote the failure pattern

of this run and $H \in X(\mathcal{F})$ the failure detector history. We denote by f the total number of failures, i.e., $f = |\mathsf{faulty}(\mathcal{F})|$. We first show that each correct process p simulates at least one schedule in which a decision occurs after the time of the last crash in $\mathcal{F}$. (Lemma 5.4). Consequently, the output of C at p is incremented at least once after the last time a process fails, as required by the signaling property. The second Lemma (Lemma 5.5) states that if the underlying failure pattern includes at least two correct processes, the exploration procedure is eventually stuck simulating a non-deciding schedule. As the output of C is modified each time a simulated schedule decides, it follows that eventually the emulation of C eventually stabilizes at each process.

Lemma 5.4. *Let τ_c be the time of the last crash and let $m = \max\{\ell : A[\ell] = \top$ at a time $\tau_c\}$. There exist a time τ following the time of the last crash such that for every correct process p, $C\text{-OUT}_p^\tau > m$.*

Proof. Let p be a correct process. There exists an index i such that for every $j, i < j$, $x[j]$ is a value output by X after the last crash occurs. Let i_c denote the smallest such i.

$\mathsf{explore}(\lambda_0, c_0, P_0)$ recursively simulates schedules of $\mathcal{A}'$. Recall that simulating a query step consists in selecting an output in x. For any time τ, let i_e^τ be the largest index of x such that $x[i_e^\tau]$ has been used to simulate a query step.

Claim: For any time τ, if $i_e^\tau \leq i_c$, $\mathsf{explore}(\lambda_0, c_0, P_0)$ has not terminated by time τ.
Proof of the Claim. Assume for contradiction that $\mathsf{explore}(\lambda_0, c_0, P_0)$ has terminated by time τ. Essentially, this implies that $\mathcal{A}'$ together with $x[1, \ldots, i_e^\tau]$ can be used to solve consensus wait-free using only registers.

Assume for contradiction that $\mathsf{explore}(\lambda_0, c_0, P_0)$ terminates. Then for each possible schedule S of $\mathcal{A}'$ for two processes q_0, q_1, there is a finite prefix S' at the end of which each participating process has decided. This schedule is induced by some finite binary sequence λ' and the sequence of outputs of X $x' = x[1..i_e^\tau]$, i.e., $S' = S_{\lambda', x'}$. There exists times $\tau'_1 < \ldots < \tau'_{i_e^\tau}$ such that for every $i, 1 \leq i \leq i_e^\tau, x'[i] = x[i] = H(p, \tau'_i)$. As $i_e^\tau \leq i_c$, $\tau'_i < \tau_c$ where τ_c is the time of the last crash. Therefore, $|\mathcal{F}(\tau'_i)| < f \leq n-1$, for every $i, 1 \leq i \leq i_e^\tau$. It thus follows from Lemma 5.2 that $S_{\lambda', x'}$ is indistinguishable for its participating processes from a real run of $\mathcal{A}'$. Therefore, the decisions in $S_{\lambda, x'}$ satisfy the validity and agreement requirements of consensus. Moreover, in the simulation of $S_{\lambda', x'}$, the simulation of $\mathsf{query}()$ steps for q_i depends only on the local state of q_i in the simulation (line 23).

Therefore algorithm $\mathcal{A}'$ together with the finite array x' can be used by two non-anonymous processes q_0 and q_1 to solve wait-free using only registers the following election task: Each non-faulty process is required to decide the id of one of the participating processes such that all decisions are the same. This task cannot be solved wait-free with registers [11], as it can then be used to solve consensus. *end of the proof of the Claim.*

The recursion tree of $\mathsf{explore}(\lambda_0, c_0, P_0)$ has bounded degree. It thus follows from the claim that a schedule is simulated in which an output $x[j]$ for some $j > i_c$, is used to simulate a query step. That is, $x[j]$ is a value output by X after the time of the last crash. Let $\mathsf{explore}(\lambda, c, P)$ be the first call to $\mathsf{explore}$ in which a value output by X after the last crash is used to simulate a query step. Without loss of generality, let us assume that q_0 is the simulated process in this step. As the output of the query depends solely on λ and c, the next calls to $\mathsf{explore}$ following the simulation of this step are of the form $\mathsf{explore}(*, *, \{q_0\})$ (line 9). That is, $x[j]$ is used for the first time while simulating the schedule induced by the sequence $\lambda \cdot 0 \cdot 0 \cdot 0 \ldots$.

Let r the real run of $\mathcal{A}'$ that is indistinguishable from $S_{\lambda,x}$. By Lemma 5.2, r does exist since every failure detector output used in $S_{\lambda,x}$ is taken before the last crash, that is at times τ such that $|\mathcal{F}(\tau)| \leq n-2$. Run r' is obtained by extending and adding clones to r. In r', after r, only q_0 and his clones take steps. Those steps are the same as in the simulated schedule $S_{\lambda \cdot 0 \cdot 0 \cdot 0 \ldots, x}$. The number of clones of q_0 is $|\mathsf{correct}(\mathcal{F})| - 1$. q_1 has no clones. r' is a real execution of $\mathcal{A}$, as we can identify q_0 and his clones to the set of correct processes in $\mathcal{F}$ and q_1 with one the faulty processes. Thus no process takes step after it has failed in r'. Moreover, r' is fair. Therefore, q_0 decides in $S_{\lambda \cdot 0 \cdot 0 \cdot 0 \ldots, x}$. This happens at some time $\tau_d > \tau_c$. By the code (line 17), the value of a new register $A[m']$ with $m' > m$ is changed to $\top$ and the emulated output of C is set to $m' > m$. □

Lemma 5.5. *If $|\mathsf{correct}(\mathcal{F})| \geq 2$, there exist an integer D such that for every process p and every time τ, C-OUT$_p^\tau \leq D$.*

Proof. Let p be a correct process and x the array of samples of X at p. We assume that there is at least two correct processes in the underlying failure pattern $\mathcal{F}$. Therefore, according to Lemma 5.2, every simulated schedule $S_{\lambda,x}$ is indistinguishable from a real execution of $\mathcal{A}'$. Hence, as seen in the proof of Lemma 5.4, $\mathsf{explore}(\lambda_0, c_0, P_0)$ does not terminate at each correct process p, as it would otherwise imply a wait-free protocol for two-processes election using only registers.

The remaining of the proof is essentially the same as the proof of Theorem 3 in [33]. The key ingredient is the particular order in which schedules are simulated. Let $\lambda_0 \cdot \lambda_1 \cdot \ldots$, $c_0 \cdot c_1 \cdot \ldots$ and $P_0 \cdot P_1 \ldots$ be defined as having $\mathsf{explore}(\lambda_0 \cdot \lambda_1 \cdots \lambda_{k+1}, c_{k+1}, P_{k+1})$ be the first call in $\mathsf{explore}(\lambda_0 \cdot \lambda_1 \cdots \lambda_k, c_k, P_k)$ that does not terminate. Let $\lambda = \lambda_0 \cdot \lambda_1 \cdots$. At least one process takes infinitely many steps but does not decide in $S_{\lambda,x}$. As $S_{\lambda,x}$ is indistinguishable from a real run of $\mathcal{A}'$, it must be the case that $inf(\lambda) \subsetneq \{0,1\}$ (Lemma 5.3). Without loss of generality, let $inf(\lambda) = \{0\}$, that is $\lambda = \lambda' \cdot 0 \cdot 0 \cdots$.

For every k, k', $k \leq k'$, $P_k \supseteq P_{k'} \supseteq \{\lambda_{k'}\}$ (lines 9-10). Moreover, the order in which set P are chosen (line 9) implies that for each k, $P_k = \cup_{k' \ geq k} \{\lambda_{k'}\}$. Therefore, we have $P_{k'} = \{q_0\}$ for each $k' \geq k$ for some $k > 0$. Hence, eventually process p simulates solely schedule $S_{\lambda,x}$, in which process q_0 never decides and takes infinitely many steps. It then follows that p eventually stops modifying the output of the emulated failure detector C. □

Theorem 5.6. *Algorithm 2 emulates C.*

Proof. By the code, the values of the variable OUT-C never decrease (Monotonicity property). The convergence property directly follows from Lemma 5.5.

Lemma 5.4 implies that the signaling property holds: Let $\tau, \tau', \tau < \tau'$ be two times such that $\mathcal{F}(\tau) \subsetneq \mathcal{F}(\tau')$. Let p and p' be two processes, p' being a correct process. As a crash occurs after time τ, OUT-$C_p^\tau \leq m$. This follows from the facts that at the time τ_c of the last crash, only the registers $A[1], \ldots, A[m]$ have their value equal to $\top$, and whenever OUT-C is set to d, the value of register $A[d]$ has previously been changed to $\top$ (line 17). Hence OUT-$C_p^\tau \leq m$. By lemma 5.4 and the monotonicity property, there exists a time $\tau'', \tau' \leq \tau''$ such that OUT-$C_p^\tau \leq m <$ OUT-$C_{p'}^{\tau''}$, as desired. □

6 The case of k-set agreement

This section defines the failure detector C_k and presents a C_k-base algorithm for k-set agreement.

6.1 The failure detector family $C_k, 1 \leq k \leq n$

For every $k, 1 \leq k \leq n$, failure detector C_k generalizes C as follows. For every failure pattern $\mathcal{F}$, history $H : \Pi \times \mathbb{N} \to \mathbb{N}$ belongs to $C_k(\mathcal{F})$ if and only if

1. *Monotonicity.* For every process p_i, for every times $\tau \leq \tau'$, $H(p_i, \tau) \leq H(p_i, \tau')$;
2. *k-Signaling* For every times $\tau \leq \tau'$, for every pair of processes p_i, p_j, if $k \leq |\mathcal{F}(\tau') \setminus \mathcal{F}(\tau)|$, there exists a time $\tau'', \tau' \leq \tau''$ such that $H(p_i, \tau) < H(p_j, \tau'')$;
3. *k-Convergence.* If $k < |\mathsf{correct}(\mathcal{F})|$, there exists a time τ such that for every $\tau', \tau \leq \tau'$, for every processes p_i, p_j, $H(p_i, \tau) = H(p_j, \tau')$.

C_1 is identical to C, while in an n-processes system, C_n provides no information on failures. Intuitively, failure detector C_k behaves in a similar way as C, but has a coarser view on failures. In any time interval in which less than k failures occur, the output of C_k may or may not change. However, as soon as the number of failures is at least k, and provided that the interval is large enough, it is guaranteed that the output of C_k is incremented.

6.2 C_k-based k-set agreement protocol

This section presents a protocol (Algorithm 3) that solves the k-simultaneous binary consensus task using failure detector C_k. In the *k-binary simultaneous consensus* task [1] (k-BSC for short), each process is initially provided with a binary vector $\overrightarrow{v} \in \{0,1\}^k$ and is required to decide a pair (i, b) where $i, 1 \leq i \leq k$ is an integer and b a binary value subject to the following requirements:

1. *Validity.* If (i, b) is decided then b is the ith bit of some input vector.
2. *Agreement.* For every $i, 1 \leq i \leq k$, if pairs (i, b) and (i, b') are decided then $b = b'$.
3. *Termination.* Every correct process eventually decides.

Intuitively, the k-BSC task might be seen as trying to solve k instance of binary consensus in parallel. Each process has initially a binary of proposal $\overrightarrow{v}[i]$ for each instance i, and is required to decide in at least one instance. Decisions (if any) in each instance i must satisfy the validity and agreement of consensus. k-set agreement can be implemented wait-free from k-BSC objects and registers. The protocol presented in [1] does not use ids but relies on snapshot objects, which can be implemented with registers without using ids [27]. Therefore, k-set agreement can be solved using k-SBC in the anonymous shared memory model.

Description of the protocol Algorithm 3 shares a similar structure with the binary consensus protocol (Algorithm 1). It consist in two tasks $T1$ and $T2$ that run in parallel. A process decides at line 27 in task $T2$ when it reads a pair (i, v) from the decision register D.

In task $T1$, each $i, 1 \leq i \leq k$ is associated with an infinite alternating sequence $\delta_i = SA_i[1], AC_i[1], SA_i[2], AC_i[2]$ of safe agreement and adopt-commit objects, in a way similar to the extended BG-simulation protocol [24]. Indeed, tasks $T1$ might be seen as a variant of the extended BG-simulation adapted to the anonymous settings. The purpose of each sequence δ_i is to reach agreement on a common value, which is one the proposal $\overrightarrow{v}[i]$ of some process. As in Algorithm 1, in each round r, processes first access safe agreement $SA_i[r]$ trying to pick a common non-$\perp$ value (lines 9-17). Then, again as in

Algorithm 1 object $AC_i[r]$ is used to reconcile the two possible outputs returned by the operations on $SA_i[r]$, namely $\perp$ and some value $v \in \{0, 1\}$ (lines 18-24), and ensure agreement and validity.

For termination, some care should be taken as:

- On one hand, when the number of failures f is small ($f \leq k-1$) or there are few correct processes ($|\mathsf{correct}| \leq k$), the output of C_k is arbitrary;
- On the other hand, when a large number $f, f \geq k$ of processes fail, every safe agreement objects $SA_1[r], \ldots, SA_k[r]$ of a given round r may be "blocked", in the sense that no propose() operations on each of the objects are successful. In that case, the output of C_k should allow the processes to move to the next round.

To resolve this difficulty, each process accesses the safe agreement objects $SA_1[r], \ldots, SA_2[r]$ of round r in that order until it receives a non-$\perp$ response or reaches the end of the sequence (line 9-12). If at most k processes enter the round, each non-faulty process thus performs a successful propose() operation on one the safe agreement of the round object. Moreover, as in Algorithm 1, a process enters round r only after its local failure detector module has output a value $d \geq r$ (line 8). In the case where $f \geq k$, there thus exists due to the k-signaling property of C_k a round R such that no more than $k-1$ faulty processes enter any round $r, r \geq R$. In any such round r, at most $k-1$ safe agreement objects are blocked (by processes failing while performing a propose() operation).

After having invoked propose() on each safe agreement $SA_1[r], \ldots, SA_k[r]$, a process waits until either (1) a read() operation returns a non-$\perp$ value or (2) the output of C_k is larger than r (line 16). When $|\mathsf{correct}| > k$, the k-convergence property ensures that eventually the output of C_k is the same value $D \geq R$ at each process. Hence, in round D condition (1) cannot be satisfied, but as seen above, at least one the safe agreement of round R is not blocked. Eventually, every read() operations performed on that object returns a non-$\perp$ value. To summarize, the protocol ensures that every non-faulty process receives a non-$\perp$ value from one of the safe agreement object of round r if (i) at most k processes enter round r or (ii) C_k output eventually stabilizes on D and $r = D$. Note that condition (i) or (ii) is satisfied in every execution, as if the failure detector output does not stabilize, $|\mathsf{correct}| \leq k$. Finally, when a process p has obtained a non-$\perp$ value v from safe agreement $SA_\ell[r]$, it first accesses the corresponding object $AC_\ell[r]$ to try to commit value v (line 18). If round r satisfies conditions (i) or (ii) above, this ensure that only v is proposed to $AC_\ell[r]$, and thus p writes to the decision register D (line 20) and eventually decides.

Proof of the protocol We consider an arbitrary run of the protocol. Let $\mathcal{F}$ and $H \in C_k(\mathcal{F})$ be respectively the failure pattern and the failure detector history in the run.

Lemma 6.1. *Let p be a correct process. There is an integer D and at time τ such that the number of faulty processes at which C_k outputs values larger than or equal to D is strictly smaller than k: $|\{q : q \in \mathsf{faulty}(\mathcal{F}) \wedge \exists \tau_q : H(q, \tau_q) \geq D\}| < k$.*

Proof. Let f denote the number of failures in $\mathcal{F}$ and let $\tau_1 \leq \ldots \leq \tau_f$ denote the times at which the failures occur. If $f < k$, the Lemma is trivially satisfied.

Let us assume that $f \geq k$. Let p be a correct process. Let d_f be the largest output of C_k at faulty processes before τ_{f-k+1}. Formally, $d_f = \max\{H(q, \tau) : q \in \mathsf{faulty}(\mathcal{F}) \wedge \tau \leq \tau_{f-k+1}\}$. As exactly $k-1$ crashes occur after τ_{f-k+1}, C_k output may be strictly larger than d_f for at most $k-1$ faulty processes.

Algorithm 3 C_k-based k-simultaneous binary consensus protocol.

```
1: init SA_{i,1≤i≤k}[1,...]                          ▷ k infinite arrays of safe agreement objects
2:      AC_{i,1≤i≤k}[1,...]                          ▷ k inifinite arrays of adopt-commit objects
3:      D ← ⊥                                        ▷ Decision register
4: function propose(v⃗)                               ▷ v⃗ ∈ {0,1}^k
5:     est⃗ ← v⃗; start tasks T1,T2
6: task T1:
7:     for r = 1, 2, ... do
8:         repeat d ← C-query() until d ≥ r end repeat
9:         for i = 1, ..., k do
10:            aux⃗[i] ← SA_i[r].propose(est⃗[i])      ▷ aux⃗ ∈ {0,1,⊥}^k
11:            if aux⃗[i] ≠ ⊥ then break end if
12:        end for
13:        if ∃j : aux⃗[j] ≠ ⊥ then ℓ ← j such that aux⃗[j] ≠ ⊥
14:        else ℓ ← 0
15:            repeat ℓ ← [ℓ mod k] + 1; aux⃗[ℓ] ← SA_ℓ[r].read(); d ← C-query()
16:            until (d > r) ∨ (aux⃗[ℓ] ≠ ⊥)
17:        end if
18:        for i = ℓ, 1, ..., ℓ−1, ℓ+1, ..., k do
19:            (b_i, u_i) ← AC_i[r].propose(aux⃗[i])   ▷ b_i ∈ {adopt, commit}, u_i ∈ {0,1,⊥}
20:            case b_i = commit ∧ u_i ∈ {0,1} then D ← (i, u_i); return
21:                 b_i = adopt  ∧ u_i ∈ {0,1} then est⃗[i] ← u_i
22:                 default                    then nop          ▷ u_i = ⊥
23:            end case
24:        end for
25:    end for
26: task T2:
27:    repeat d ← D until d ≠ ⊥ end repeat; return d
```

Let $\tau \leq \tau_{f-k+1}$ be a time at which d_f is output by C_k at some faulty process q. Since $|\mathcal{F}(\tau_f) \setminus \mathcal{F}(\tau)| \geq k$, it follows from the signaling property of C_k that there exists a time $\tau'' \geq \tau_{f-k+1}$ and an integer $D > d_f$ such that $H(p, \tau'') = D$. Since $D > d_f$, the output of C_k can be $\geq D$ at no more than $k-1$ faulty processes. □

A process decides when a pair is returned at line 27.

Lemma 6.2 (Termination)**.** *Every correct process decides.*

Proof. Assume for contradiction that no correct processes decide. Therefore no processes write to the decision register D at line 20.

We distinguish two cases according to whether the outputs of C_k is bounded or not.

- The output of C_k is unbounded. As in the consensus protocol (Algorithm 1), a correct process cannot get stuck in round r if its local failure detector module eventually outputs a value $> r$. Indeed, the only blocking part of the code of round r is the **repeat** loop of lines 15-16. For each process p, the loop terminates as soon a value $d > r$ is output by C_k. It thus follows that there is a round R in which only correct processes enter.

  Let p be a correct process that enters round R. As no processes invoking propose() on the safe agreement objects of round R fail, at least one of the propose() operations performed

on $SA_i[R]$ is successful, for each $i, 1 \leq i \leq k$. By the code (lines 9-12), each process invoke propose() on the objects $SA_1[R], SA_2[R], \ldots, SA_k[R]$ in that order on until it receives a non-$\perp$ response or reach the last object in the sequence. As the output of C_k never stabilizes, the k-convergence property of C_k implies that $|\mathsf{correct}(\mathcal{F})| \leq k$, and thus at most k processes enters round R. Hence, at most $k - i + 1$ processes invoke propose() on each safe agreement object $SA_i[R]$. It thus follows that p receives a non-$\perp$ response from one of its propose() operations. Let b_i be this successful response and i the index of the object $SA_i[R]$ on which the corresponding propose() operation is performed

It follows from the code (line 13 and lines 18-19) that p invokes propose() on $AC_i[R]$ with input b_i. Due to the order in which objects $AC_j[R], 1 \leq j \leq k$ are accessed (line 18), every process that performs a successful propose() on $SA_i[R]$ does the same, while processes that did not access the object or receives a $\perp$ response from $SA_i[R]$ do not access $AC_i[R]$. As all the successful propose() operations on $SA_i[R]$ return the same value, the propose() invocation by p on $AC_i[R]$ returns $(commit, b_i)$, with $b_i \neq \perp$ from which we conclude that p writes to D. This is a contradiction.

- The output of C_k is bounded. Let R be the largest value output by C_k. As seen in the previous case, every correct process eventually enters round R. By Lemma 6.1, at most $k-1$ faulty processes enter R, since R can be the output of C_k at at most $k-1$ faulty processes. Therefore, at most $k-1$ processes fail after entering round R. Consequently, for at least one of the objects $SA_1[R], \ldots, SA_k[R]$, no processes fail while performing a propose() operation on that object, and thus one of these operations is successful.

  Let p be a process that enters round R. As R is the largest output of C_k, p cannot exit the **repeat** loop of lines 15-16 before having read a non-$\perp$ value from one the safe agreement object of round R. As at least one propose() operation on the objects $SA_1[R], \ldots, SA_k[R]$ is successful, this eventually happens. Let ℓ and $b_\ell \neq \perp$ such that $\overrightarrow{aux}[\ell] = b_\ell$ when p exits the **repeat** loop.

  In the second part of round R, observe that for every process q that accesses object $AC_\ell[R]$, we have $\overrightarrow{aux}[\ell] = b_q \neq \perp$. Note that by the agreement property of safe agreement, for every process q, q', if $\overrightarrow{aux}[\ell] = b_q \neq \perp$ at q and $\overrightarrow{aux}[\ell] = b_{q'} \neq \perp$ at q' then $b_q = b_{q'}$. Hence, the input of every propose() operation on $AC_\ell[R]$ is the same value $b_\ell \neq \perp$. By the convergence property of adopt-commit object, each of the operation returns $(commit, b_\ell)$. Therefore, by the code (line 20) p writes $b_\ell \neq \perp$ to D : a contradiction. □

Theorem 6.3. *Algorithm 3 solves k-binary simultaneous consensus using failure detector C_k.*

Proof. Validity directly follows from the validity properties of adopt-commit and safe agreement objects. Termination follows from Lemma 6.2. For agreement, it is sufficient to prove that if (i, b) and (i, b') are written to D we have $b = b'$, for every $i, 1 \leq i \leq k$. The proof is similar to the proof of agreement of the consensus algorithm (Theorem 4.3).

Suppose that processes p and p' write respectively (i, b) and (i, b') to D in that order, where $i, 1 \leq i \leq k$. Without loss of generality, assume that p writes to D in round r. By the code (line 20), before writing to D, p invokes propose() on $AC_i[r]$ and receives $(commit, b)$ as a response. It follows from the agreement property of adopt-commit object that every value written to D in round r is b.

Moreover, for every process q that does not return in round r at line 20, we have $\overrightarrow{est}[i] = b$ (agreement property of adopt-commit) when q leaves round r. Hence in any subsequent round r',

only b can be proposed to $SA_i[r']$ and thus the input of every propose() operation on $AC_i[r']$ is b or $\perp$. Therefore, if (i, b') is written to D in round r', $b' = b$. □

As k-set agreement can be implemented anonymously from registers and k-binary simultaneous consensus objects, we have:

Corollary 6.4. *There is an algorithm that solves k-set agreement using registers and failure detector C_k.*

7 Conclusion

The paper has defined the new failure detector C and has shown that, within the class of anonymity-preserving failure detectors, it is the weakest failure detector for consensus in the anonymous shared memory model in any environment. The paper has also shown that failure detector C_k, a natural extension of C, can be used to solve k-set agreement tolerating any number of failures. Obvious questions for future work include (dis)proving that C_k is the weakest anonymity preserving failure detector for k-set agreement and extending weakest failure detector results in anonymous systems outside the domain of anonymity preserving failure detectors.

References

[1] Yehuda Afek, Eli Gafni, Sergio Rajsbaum, Michel Raynal, and Corentin Travers. The k-simultaneous consensus problem. *Distributed Computing*, 22(3):185–195, 2010.

[2] Dana Angluin. Local and global properties in networks of processors (extended abstract). In *Proceedings of the Twelfth Annual ACM Symposium on Theory of Computing of Conference (STOC)*, pages 82–93. ACM, 1980.

[3] James Aspnes. A modular approach to shared-memory consensus, with applications to the probabilistic-write model. *Distributed Computing*, 25(2):179–188, May 2012.

[4] James Aspnes and Faith Ellen. Tight bounds for adopt-commit objects. *Theory Comput. Syst.*, 55(3):451–474, 2014.

[5] Hagit Attiya. Adapting to point contention with long-lived safe agreement. In *Proceedings of the 13th International Colloquium on Structural Information and Communication Complexity(SIROCCO)*, volume 4056 of *Lecture Notes in Computer Science*, pages 10–23. Springer, 2006.

[6] Hagit Attiya, Alla Gorbach, and Shlomo Moran. Computing in totally anonymous asynchronous shared memory systems. *Inf. Comput.*, 173(2):162–183, 2002.

[7] Hagit Attiya and Marc Snir. Better computing on the anonymous ring. *J. Algorithms*, 12(2):204–238, 1991.

[8] Hagit Attiya, Marc Snir, and Manfred K. Warmuth. Computing on an anonymous ring. *J. ACM*, 35(4):845–875, 1988.

[9] 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)*, pages 206–213, 2010.

[10] François Bonnet and Michel Raynal. Anonymous asynchronous systems: the case of failure detectors. *Distributed Computing*, 26(3):141–158, 2013.

[11] Elizabeth Borowsky and Eli Gafni. Generalized FLP impossibility result for t-resilient asynchronous computations. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing (PODC)*, pages 91–100. ACM, 1993.

[12] Elizabeth Borowsky, Eli Gafni, Nancy A. Lynch, and Sergio Rajsbaum. The BG distributed simulation algorithm. *Distributed Computing*, 14(3):127–146, 2001.

[13] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. *J. ACM*, 43(4):685–722, 1996.

[14] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.

[15] Soma Chaudhuri. More choices allow more faults: Set consensus problems in totally asynchronous systems. *Inf. Comput.*, 105(1):132–158, 1993.

[16] George Danezis and Claudia Diaz. A survey of anonymous communication channels. Technical report MSR-TR-2008-35, Microsoft Research, 2008.

[17] Carole Delporte-Gallet and Hugues Fauconnier. Two consensus algorithms with atomic registers and failure detector omega. In *Proceedings of the 10th International Conference on Distributed Computing and Networking*, ICDCN '09, pages 251–262, Berlin, Heidelberg, 2009. Springer-Verlag.

[18] Carole Delporte-Gallet, Hugues Fauconnier, Eli Gafni, and Petr Kuznetsov. Wait-freedom with advice. *Distributed Computing*, 28(1):3–19, 2015.

[19] Carole Delporte-Gallet, Hugues Fauconnier, and Rachid Guerraoui. Tight failure detection bounds on atomic object implementations. *J. ACM*, 57(4), 2010.

[20] Faith Ellen Fich, Maurice Herlihy, and Nir Shavit. On the space complexity of randomized synchronization. *J. ACM*, 45(5):843–862, 1998.

[21] Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.

[22] Felix C. Freiling, Rachid Guerraoui, and Petr Kuznetsov. The failure detector abstraction. *ACM Comput. Surv.*, 43(2):9, 2011.

[23] Eli Gafni. Round-by-round fault detectors: Unifying synchrony and asynchrony (extended abstract). In *Proceedings of the 17th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 143–152. ACM, 1998.

[24] Eli Gafni. The extended bg-simulation and the characterization of t-resiliency. In *Proceedings of the 41st Annual ACM Symposium on Theory of Computing (STOC)*, pages 85–92. ACM, 2009.

[25] Eli Gafni and Petr Kuznetsov. On set consensus numbers. *Distributed Computing*, 24(3-4):149–163, 2011.

[26] Rachid Guerraoui, Vassos Hadzilacos, Petr Kuznetsov, and Sam Toueg. The weakest failure detectors to solve quittable consensus and nonblocking atomic commit. *SIAM J. Comput.*, 41(6):1343–1379, 2012.

[27] Rachid Guerraoui and Eric Ruppert. Anonymous and fault-tolerant shared-memory computing. *Distributed Computing*, 20(3):165–177, 2007.

[28] Damien Imbs and Michel Raynal. Visiting gafni's reduction land: From the BG simulation to the extended BG simulation. In *Proceedings of the 11th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, volume 5873 of *Lecture Notes in Computer Science*, pages 369–383. Springer, 2009.

[29] MC Loui and HH Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. *Advances in Computing Research*, 4:163–183, 1987.

[30] Achour Mostéfaoui, Sergio Rajsbaum, Michel Raynal, and Corentin Travers. The combined power of conditions and information on failures to solve asynchronous set agreement. *SIAM J. Comput.*, 38(4):1574–1601, 2008.

[31] Masafumi Yamashita and Tsunehiko Kameda. Computing on anonymous networks: Part i-characterizing the solvable cases. *IEEE Trans. Parallel Distrib. Syst.*, 7(1):69–89, 1996.

[32] Masafumi Yamashita and Tsunehiko Kameda. Computing on anonymous networks: Part ii-decision and membership problems. *IEEE Trans. Parallel Distrib. Syst.*, 7(1):90–96, 1996.

[33] Piotr Zielinski. Anti-*Omega*: the weakest failure detector for set agreement. *Distributed Computing*, 22(5-6):335–348, 2010.

A Anonymous safe agreement

For completeness, this section presents an anonymous implementation of a safe agreement object using registers. The implementation is an almost verbatim copy of an anonymous binary consensus protocol [6], designed for asynchronous, anonymous but failure-free shared-memory systems.

Attiya, Gorbach and Moran protocol (Algorithm 4 in [6]) consists in two parts, a wait-free section and a final wait statement. The processes first execute the wait-free code. Agreement is eventually reached in the wait-free section by evicting slow processes and, in case of conflicting proposals, favoring value 0. That is, the number of processing willing to decide 1 progressively decreases. Processes dropped from the wait-free section simply wait until the remaining processes are able to agree. In the safe agreement implementation (Algorithm 4 below), the propose() operation is implemented by the wait-free code, while a read() operation consists in checking whether the exit condition of the wait statement is satisfied.

Algorithm 4 Safe agreement (Attiya et. al failure-free consensus protocol [6]).

```
1: init ∀i ∈ {0,1}, j ∈ {1,2,...} : A_j[i] = false for each i ∈ {0,1}   ▷ A_j[i] register, initially false
2:     D ← ⊥                                                            ▷ Decision register, initially ⊥
3: function propose(v)                                                  ▷ v ∈ {0,1}
4:     est ← v;
5:     for j = 1, 2, ... do
6:         u ← A_j[¬est]                                                ▷ ¬est = 1 − est
7:         if u = true then return ⊥ end if
8:         est ← reduceone(j, est)
9:         if est = ⊥ then return ⊥ end if
10:        if j > 1 then u ← A_{j−1}[¬est]
11:            if u = false then D ← est; return est end if
12:        end if
13:    end for

14: function read()
15:    d ← D; return d

16: procedure reduceone(j, v)                                           ▷ v ∈ {0,1}
17:    A_j[v] ← true; w ← A_j[v̄]
18:    if w = false then return v
19:    else
20:        if v = 0 then return ⊥
21:        else return 0
22:        end if
23:    end if
```

In more details, the protocol relies on a procedure reduceone() (lines 16-23) that aims at decreasing disagreement among the processes. Each process invokes reduceone() with input 0 or 1 and obtains a response in $\{\perp, 0, 1\}$ with the following properties:

Lemma A.1 ([6], Lemma 4.1)**.** Non-triviality*: The output of at least one process is not* $\perp$*;* Validity*: If the output of* p *is* $v = \perp$*, then the input of some process is* v*.* Monotonicy*: If some process outputs* 0 *and another process outputs* 1*, then the number of processes with output* 1 *is strictly smaller than the number of processes with input* 1*.*

By the code, reduceone() is wait-free. Suppose that n processes invoke propose() and none of them fails. Essentially, in each iteration j of the **for** loop (lines 5-13) (1) at least one process receives a non-$\perp$ value from its invocation of reduceone$(j, *)$ and (2) the number of processes whose estimate *est* is 1 is strictly smaller than in the previous iteration. Hence, after sufficiently many operation, only the value of $A_j[0]$ is changed from false to true, which then allow some process to write 0 to D and return 0 as the result of its propose() operation. Indeed, we have

Lemma A.2 ([6], Lemma 4.6)**.** *If* n *processes invoke* propose() *and none of them fail, some process write a non-*$\perp$ *value to* D *in iteration* $j \leq n + 1$

When a successful propose() returns, a value $\neq \perp$ has been written to the shared register D. Hence, every read() operation that follows returns a non-$\perp$ value (consistent reads property). Validity and agreement of the safe agreement implementation follow from the same properties of the original consensus protocol. Indeed, as it is designed for asynchronous systems, failures may impair progress but do not cause violation of safety properties. For details about the underlying principle and the proof of Algorithm 4, we refer to [6].