Anonymity-Preserving Failure Detectors*

Zohir Bouzid and Corentin Travers

LaBRI, U. Bordeaux, France
name.surname@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* failure 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) if two or more processes are non-faulty, it eventually stabilizes.

1 Introduction

Anonymous computing The vast majority of the literature about distributed computing assumes that each process is provided with a 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 [23]. For example, for privacy reasons, a set of distributed processes may be willing to compute some function on their inputs without revealing their identity. Alternatively, the distributed computation might be performed on top of an anonymous communication system [14], 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 [5,23] 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 [1], the computational power of anonymous message passing system in the failure-free case has been investigated for particular or general graph topologies, e.g., [6,25].

* This work has been carried out with financial support from the French State, managed by the French National Research Agency (ANR) in the frame of the "Investments for the future" Program IdEx Bordeaux - CPU (ANR-10-IDEX-03-02).

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. Even if processes have unique identifiers, it is well known that asynchronous fault tolerant consensus is impossible as soon as at least one process may fail by crashing [24]. This impossibility trivially extends to anonymous systems.

Failure detectors A popular approach to circumvent impossibilities stemming from asynchrony and failures is to use *failure detectors* [13]. A failure detector is a distributed device that provides each processes with perhaps unreliable information about which other processes have crashed. In systems with identities, several classes of failure detectors have been defined [18]. 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 [12] in shared memory systems with identities.

Failure detectors in anonymous systems The study of failure detectors in anonymous message passing systems was initiated in [8]. In particular, *identity-free* counterparts of classical failure detectors including Ω and P are identified. $A\Omega$, an identity-free failure detector equivalent to Ω, outputs a Boolean value at each process such that eventually true is output only at a unique correct process. A consensus protocol that uses $A\Omega$ was also presented. In the shared memory model, an anonymous $A\Omega$-based protocol can be found in [15]. Bonnet and Raynal left open the following question: "Consensus in anonymous distributed systems: is there a weakest failure detector?"[7]. We answer 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 can be used 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 at process p, the same sequence might be output at every process without violating

the specification of the failure detector. An example of such failure detector is AP which provides each process with an eventually accurate estimation of the number of faulty 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 a failure occurs 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 strong enough to solve consensus while tolerating any number of failures (Section 4). Striving to not reinvent the wheel, the protocol relies on standard shared memory constructs, namely adopt-commit [19] and safe-agreement [10] objects.
3. It is then shown that C can be emulated using any anonymity-preserving failure detector powerful enough to solve consensus (Section 5). The extraction protocol reuses in part the techniques developed by Zieliński [26] for proving statement of this type in the shared memory model when processes are not anonymous. 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.

Due to space constraints, some proofs and additional results have been omitted. See [11] for a complete report on this work.

2 Computational Model

We consider an *asynchronous* and *crash-prone* shared-memory system consisting in a set $\Pi = \{p_1, \ldots, p_n\}$ of $n \geq 2$ processes, i is the *index* of p_i. Processes are *anonymous* in the sense that they run the same code and are not aware of their index. They communicate via a *shared memory* that consists in an unbounded number of *multi-writer/multi-reader atomic registers*. For modeling purpose we assumed the existence of global clock not accessible to the processes and whose range is the integers.

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 $\tau \in \mathbb{N}$. $\mathsf{faulty}(\mathcal{F}) = \bigcup_{\tau \geq 0} \mathcal{F}(\tau)$ denotes the set of processes that fail in $\mathcal{F}$. A process p is *faulty* in $\mathcal{F}$ if it belongs to $\mathsf{faulty}(\mathcal{F})$ and *correct* otherwise, that is $p \in \mathsf{correct}(\mathcal{F}) = \Pi \setminus \mathsf{faulty}(\mathcal{F})$. We assume the *wait-free environment* that contains every failure pattern in which at least one process is correct. A *failure detector* D with range $\mathcal{R}$ is a distributed device that provides each process with information about the failure pattern [13]. 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. The value returned by the

failure detector at process p_i at time τ is $H(p_i, \tau)$. D associates a non-empty set of histories $D(\mathcal{F})$ with each failure pattern $\mathcal{F}$.

A *protocol* consists in n copies of a local algorithm $\mathcal{A}$, one per process. In a *step* a process (1) queries the failure detector or (2) reads or (3) writes a shared register, and then performs some local computation. A *run of a protocol $\mathcal{A}$ using failure detector D* is a tuple $e = (\mathcal{F}, H, I, S, T)$ where $\mathcal{F}$ is a failure pattern, $H \in D(\mathcal{F})$, I and S are respectively an initial configuration and 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 $T[i]$. $e = (\mathcal{F}, H, I, S, T)$ represents an execution of $\mathcal{A}$ if and only if (1) S and T are both infinite or $|S| = |T|$, (2) no processes take a step after it has crashed, (3) if step $S[i]$ is a failure detector query by process p that returns d, then $d = H(p, T[i])$, (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.

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.

A failure detector D is said to be *as least as weak as* a failure detector D', denoted $D \leq D'$ if there is a protocol $\mathcal{T}_{D' \to D}$ that *emulates D using D'*. Failure detector D is said to be the *weakest failure detector* for a task T 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, $D \leq D'$. In systems with identities, Ω is the weakest failure detector for consensus [12].

3 Anonymity-Preserving Failure Detectors

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. 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 for process $p_j \neq p_i$, for the same failure pattern $\mathcal{F}$.

For instance, the failure detector $A\Omega$ [8] eventually distinguishes a unique correct process. It provides to each process a single bit whose value eventually is 0 except for one correct process. $A\Omega$ is thus not an anonymity-preserving failure detector. An example of an anonymity-preserving failure detector is the identity-free variant of the perfect failure detector, denoted AP in [8]. 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, $H(p_i, \tau) \leq |\mathcal{F}(\tau)|$, and (*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 $\mathcal{F}$ $f_1, f_2, \ldots$ 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 might be seen as an unreliable variant of the *signaling failure* detector $\mathcal{FS}$ [22]. 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.

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. Moreover, 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 $|\mathsf{correct}(\mathcal{F})| > 1$, there exists a time τ : for every process p_i, $\forall \tau' \tau \leq \tau'$, $H(p_i, \tau) = H(p_i, \tau')$.

4 A *C*-based Consensus Protocol

This section presents a consensus protocol (Protocol 1) based on failure detector C. To simplify the presentation, we concentrate on *binary consensus* in which the set of possible inputs is $\{0, 1\}$. Besides registers, it relies on standard shared memory constructs, namely adopt-commit [19] and safe agreement [9,10] objects, that we describe next.

Base objects An *adopt-commit* object has a single operation denoted $\mathsf{propose}(v)$ 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 [3,19]: (*Validity*) If an operation returns (d, v) then v is the input of

a propose() operation; (*Agreement*) If an operation returns $(commit, v)$ then each output is either $(adopt, v)$ or $(commit, v)$; (*Convergence*) If the input of every operation is v, then every output is $(commit, v)$; (*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 [3]. The implementation ([3], 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 $\mathrm{fact}^{-1}(|\mathcal{V}|)$ multi-writer/multi-reader registers ([3], Algorithm 3). These algorithms do not use identities, and are thus suitable for the anonymous shared-memory model.

The *safe agreement* object, introduced by Borowsky and Gafni in [9] allows processes to propose values and to agree on a single value. It is at the heart of the BG-simulations [9] 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., [4,10]. Our specification below closely follows [4].

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. We say that a propose() operation is *successful* if it returns a value $\neq \perp$. 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: (*Validity*) If an operation returns a value $v \neq \perp$, v is the input of a propose() operation; (*Agreement*) If values $v, v' \in \{0, 1\}$ are returned by some operation, $v = v'$; (*Termination*) Every operation performed by a non-faulty process terminates; (*Consistent reads*) Any read() operation that terminates and starts after a successful propose() operation has completed returns a non-$\perp$ value; (*Non-triviality*) Not every propose() operation returns $\perp$.

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.

In systems with identities, safe agreement objects can be implemented with registers, e.g., [9]. In anonymous systems, a safe agreement object implementation can be obtained by slightly modifying an anonymous binary consensus protocol by Attiya, Gorbach and Moran [5] designed for the asynchronous shared memory model with no failures.

[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.

Protocol 1 C-based binary consensus.

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

Description of the protocol Protocol 1 consists in two tasks denoted T1 and T2, launched in parallel at each process p (line 3). 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 20).

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., [2,20], the processes that enter round r first try to reach agreement by accessing the safe agreement object $SA[r]$ (line 7) and then check whether agreement has been achieved using the adopt-commit object $AC[r]$ (line 12).

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 7) and, if its operation is unsuccessful (line 8), it enters a loop in which it repeatedly reads the object (line 9). 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 6);
- A process exits the loop in which it is trying to obtain a non-$\perp$ value by performing read() operations on $SA[r]$ when its local failure detector output is strictly larger than d_c (line 10).

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 5). 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 9–10).

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 12). Each process p keeps in its local variable aux the result of its operations (at lines 7 and 9) 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 12). If it gets back $(adopt, u)$, where $u \neq \perp$ it changes its estimate to u (line 14). 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 15).

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 processes enter round r, at least one of the propose() operations on $SA[r]$ is successful. Moreover, no process can exit the reading loop (lines 9-10) without having obtained a non-$\perp$ value from $SA[r]$, as C never outputs a value larger than r to those processes. Since all non-$\perp$ values 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 are 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. The proof of correctness can be found in the full version [11].

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 (Protocol 2) a protocol $\mathcal{T}_{X \to C}$ that emulates C using X in the wait-free environment.

Overview As in previous protocols [12,16,21,26] that emulate weakest failure detectors, in $\mathcal{T}_{X \to C}$ each process locally simulates many possible runs of protocol $\mathcal{A}$. 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 a 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 [12] 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 $\mathcal{F}$.

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 the 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 $\mathcal{F}$. It thus can be shown that, from the point of view of q_0 and q_1, execution e is indistinguishable 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, protocol $\mathcal{A}$ 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 [21,26]. Whenever a decision occurs in

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

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 that (1) 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, that (2) eventually p keeps simulating one infinite execution in which no processes decide. The correctness of the emulation C then follows from (1) and (2).

Protocol $\mathcal{A}'$ 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 to 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 protocols.

In protocol $\mathcal{A}'$, 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 $MEM[\ell]$.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 $\mathsf{write}(), \mathsf{read}()$ or $\mathsf{query}()$ operation η is incremented ($\eta \leftarrow \eta+1$). For each step s of the modified protocol $\mathcal{A}'$, 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 step 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 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, $s \preceq s' \implies \eta(s) < \eta(s')$.*

Collecting failure detector X samples As X is anonymity-preserving, for any failure pattern $\mathcal{F}$ and any finite or infinite sequence $x = x_1, x_2, \ldots$ of outputs of X collected by some process 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 (specifying the order in which q_0 and q_1 takes steps) 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 $\mathsf{query}()$. A difficulty is to choose an output in x for each $\mathsf{query}()$ step of S in such a way that there is a real execution of $\mathcal{A}'$ indistinguishable from S to both q_0 and q_1.

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 $\mathsf{read}()$ or $\mathsf{write}()$ step, it is simulated by reading or writing a value to/from MEM. If the step is a $\mathsf{query}()$ 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 $\mathsf{query}()$ 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 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 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.*

Run r may however 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 $inf(\lambda) \subseteq \{0, 1\}$ be the bits that appear infinitely 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 induced by a binary sequence and a sequence of failure detector outputs λ, x.

Lemma 3. *Let λ, x be infinite sequences of respectively bits and failure detector X outputs. Suppose that there exists a failure pattern $\mathcal{F}$, a sequence of times $\tau_1 < \tau_2 < \ldots$ and a 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* [17] 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.

C emulation Protocol 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 Zieliński [26]. 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 8-18). 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 16).

For any arbitrary run of protocol 2 with failure pattern $\mathcal{F}$, we first show (see [11]) that each correct process p simulates at least one schedule in which a decision occurs after the time of the last crash. 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. We then establish that if $|\mathsf{correct}(\mathcal{F})| \geq 2$, the exploration procedure is eventually stuck simulating a non-deciding schedule. As the output of C is modified each time a simulated schedule decides, the output of C eventually stabilizes at each process, hence,

Theorem 1. *Protocol 2 emulates C.*

6 Conclusion

The paper has defined the class of anonymity-preserving failure detectors and has shown that within this class, at least for consensus a weakest failure C exists in the anonymous shared-memory model.

In the full version [11], a natural generalization denoted C_k of failure detector C is introduced and a C_k-based protocol for k-set agreement is presented. 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. D. Angluin. Local and global properties in networks of processors (extended abstract). *STOC'80*, pp. 82–93, ACM.
2. J. Aspnes. A modular approach to shared-memory consensus, with applications to the probabilistic-write model. *Distributed Computing*, 25(2):179–188, 2012.

3. J. Aspnes and F. Ellen. Tight bounds for adopt-commit objects. *Theory Comput. Syst.*, 55(3):451–474, 2014.
4. H. Attiya. Adapting to point contention with long-lived safe agreement. *SIROCCO'06*, LNCS #4056, pp. 10–23, Springer.
5. H. Attiya, A. Gorbach, and S. Moran. Computing in totally anonymous asynchronous shared memory systems. *Inf. Comput.*, 173(2):162–183, 2002.
6. H. Attiya and M. Snir. Better computing on the anonymous ring. *J. Algorithms*, 12(2):204–238, 1991.
7. F. Bonnet and M. Raynal. Consensus in anonymous distributed systems: Is there a weakest failure detector? *AINA'10*, pp. 206–213, IEEE.
8. F. Bonnet and M. Raynal. Anonymous asynchronous systems: the case of failure detectors. *Distributed Computing*, 26(3):141–158, 2013.
9. E. Borowsky and E. Gafni. Generalized FLP impossibility result for t-resilient asynchronous computations. *PODC'93*, pp. 91–100, ACM.
10. E. Borowsky, E. Gafni, N. A. Lynch, and S. Rajsbaum. The BG distributed simulation algorithm. *Distributed Computing*, 14(3):127–146, 2001.
11. Z. Bouzid and C. Travers. Anonymity preserving failure detectors. Tech. Rep., LaBRI, July 2016. `https://hal.archives-ouvertes.fr/hal-01344446`.
12. T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *J. ACM*, 43(4):685–722, 1996.
13. T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.
14. G. Danezis and C. Diaz. A survey of anonymous communication channels. Tech. Rep., MSR-TR-2008-35, Microsoft Research, 2008.
15. C. Delporte-Gallet and H. Fauconnier. Two consensus algorithms with atomic registers and failure detector omega. *ICDCN'09*, LNCS #5408 pp. 251–262, Springer.
16. C. Delporte-Gallet, H. Fauconnier, and R. Guerraoui. Tight failure detection bounds on atomic object implementations. *J. ACM*, 57(4), 2010.
17. F. E. Fich, M. Herlihy, and N. Shavit. On the space complexity of randomized synchronization. *J. ACM*, 45(5):843–862, 1998.
18. F. C. Freiling, R. Guerraoui, and P. Kuznetsov. The failure detector abstraction. *ACM Comput. Surv.*, 43(2):9, 2011.
19. E. Gafni. Round-by-round fault detectors: Unifying synchrony and asynchrony (extended abstract). *PODC'98*, pp. 143–152, ACM.
20. E. Gafni. The extended BG-simulation and the characterization of t-resiliency. *STOC'09*, pp. 85–92, ACM.
21. E. Gafni and P. Kuznetsov. On set consensus numbers. *Distributed Computing*, 24(3-4):149–163, 2011.
22. R. Guerraoui, V. Hadzilacos, P. Kuznetsov, and S. Toueg. The weakest failure detectors to solve quittable consensus and nonblocking atomic commit. *SIAM J. Comput.*, 41(6):1343–1379, 2012.
23. R. Guerraoui and E. Ruppert. Anonymous and fault-tolerant shared-memory computing. *Distributed Computing*, 20(3):165–177, 2007.
24. M. Loui and H. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. *Advances in Computing Research*, 4:163–183, 1987.
25. M. Yamashita and T. Kameda. Computing on anonymous networks: Part I-characterizing the solvable cases. *Trans. Parallel Distrib. Syst.*, 7(1):69–89, 1996.
26. P. Zieliński. Anti-Ω: the weakest failure detector for set agreement. *Distributed Computing*, 22(5-6):335–348, 2010.