On the weakest failure detector ever

On the Weakest Failure Detector Ever

Rachid Guerraoui[1,2], Maurice Herlihy[3], Petr Kuznetsov[4], Nancy Lynch[1], and Calvin Newport[1]

[1] Computer Science and Artificial Intelligence Laboratory, MIT
[2] School of Computer and Communication Sciences, EPFL
[3] Computer Science Department, Brown University
[4] Technische Universität Berlin/Deutsche Telekom Laboratories

Abstract. Many problems in distributed computing are impossible to solve when no information about process failures is available. It is common to ask what information about failures is necessary and sufficient to circumvent some *specific* impossibility, e.g., consensus, atomic commit, mutual exclusion, etc. This paper asks what information about failures is necessary to circumvent *any* impossibility and sufficient to circumvent *some* impossibility. In other words, what is the *minimal* yet *non-trivial* failure information.

We present an abstraction, denoted Υ, that provides very little information about failures. In every run of the distributed system, Υ eventually informs the processes that *some* set of processes in the system cannot be the set of correct processes in that run. Although seemingly weak, for it might provide random information for an arbitrarily long period of time, and it eventually excludes only one set of processes (among many) that is not the set of correct processes in the current run, Υ still captures non-trivial failure information. We show that Υ is sufficient to circumvent the fundamental wait-free set-agreement impossibility. While doing so, we (a) disprove previous conjectures about the weakest failure detector to solve set-agreement and we (b) prove that solving set-agreement with registers is strictly weaker than solving $n+1$-process consensus using n-process consensus.

We show that Υ is the weakest *stable* non-trivial failure detector: any stable failure detector that circumvents some wait-free impossibility provides at least as much information about failures as Υ does.

Our results are generalized, from the wait-free to the f-resilient case, through an abstraction Υ^f that we introduce and prove minimal to solve any problem that cannot be solved in an f-resilient manner, and yet sufficient to solve f-resilient f-set-agreement.

1 Introduction

Fischer, Lynch, and Paterson's seminal result in the theory of distributed computing [11] says that the seemingly easy *consensus* task (a decision task where a collection of processes starts with some input values and needs to agree on one of the input values) cannot be deterministically solved in an asynchronous distributed system that is prone to process failures, even if processes simply fail by crashing, i.e., prematurely stop taking steps of their algorithm. Later, three independent groups of researchers [20, 14, 2] extended that result by proving the impossibility of *wait-free n-set-agreement* [5], a decision task where processes start with distinct input values and need to agree on up to n input values, in an asynchronous shared memory model of $n+1$-processes among which n can crash. This result was then extended to prove the asynchronous impossibility of *f-resilient* f-set agreement [2], i.e., f-set agreement among $n+1$ processes among which f can crash.

Asynchrony refers to the absence of timing assumptions on process speeds and communication delays. However, some timing assumptions can typically be made in most real distributed systems [9, 10]. In the best case, if we assume precise knowledge of bounds on communication delays and process relative speeds, then it is easy to show that known asynchronous impossibilities can be circumvented. Intuitively, such timing assumptions circumvent asynchronous impossibilities by providing processes with information about failures, typically through time-out (or heart-beat) mechanisms.

In general, although certain information about failures can indeed be obtained in distributed systems, it is nevertheless reasonable to assume that this information is only partial and might sometimes be inaccurate. Typically, bounds on process speeds and message delays hold only during certain periods of time, or only in certain parts of the system. Hence, the information provided about the failure of a process might not be perfect. It is common to ask what information about failures is necessary and sufficient to circumvent some *specific* impossibility, e.g., consensus [3], atomic commit [7], mutual exclusion [8], etc.

This paper asks, for the first time, what information about failures is *necessary* to circumvent *any* (asynchronous) impossibility and yet *sufficient* to circumvent *some* impossibility. In other words, we seek for the minimal *non-trivial* information about failures or, in the parlance of Chandra et al. [3], the weakest failure detector that cannot be implemented in an asynchronous system. By doing so, and assuming that this minimal information is sufficient to circumvent the impossibility of some problem T, we can derive that T, from the failure detection perspective, belongs to the equivalence class of the "weakest" impossible problems in asynchronous distributed computing.

We focus in this paper on the shared memory model. For presentation simplicity, we also consider first the n-resilient (wait-free) case and assume a system with $n+1$ processes among which n can crash. Then we move to the f-resilient case where $f \leq n$ processes can crash.

We define a new *failure detector* oracle, denoted by Υ. This oracle outputs, whenever queried by a process, a non-empty set of processes in the system. The output might vary for an arbitrarily long period. Eventually, however, the output set should:

(a) be the same at all correct processes and
(b) not be the exact set of correct processes.

Failure detector Υ provides very little information about failures: in each execution, it only excludes one (among many) set of processes that is not the set of correct processes, and it does so eventually. In particular, Υ does not say which set of processes are correct, and the set it outputs might never contain any correct (resp. faulty) process.

To illustrate Υ, consider for instance a system of 3 processes, p_1, p_2, p_3, and a run where p_1 fails while p_2 and p_3 are correct. Oracle Υ can output any set of processes for an arbitrarily long period, it can keep arbitrary changing this set and can output different sets at different processes. Eventually however, Υ should permanently output, at p_2 and p_3, either $\{p_1\}, \{p_2\}, \{p_3\}, \{p_1, p_3\}, \{p_1, p_2\}$ or $\{p_1, p_2, p_3\}$, i.e., any subset but $\{p_2, p_3\}$.

We prove that, although seemingly pretty weak, Υ is sufficient to solve n-set-agreement with read/write objects (registers), in a system of $n+1$ processes among which n might crash. In other words, Υ is sufficient to circumvent the seminal wait-free set-agreement impossibility.

This result (a) disproves the conjecture of [19] about the weakest failure detector to implement n-resilient n-set agreement and (b) proves that implementing n-resilient n-set agreement with read/write objects is strictly weaker than solving $n+1$-process consensus using n-process consensus.

We then extend our result to the f-resilient case, and propose an algorithm that solves f-set-agreement in a system of $n+1$ processes where $f \leq n$ processes can fail, using a generalization of Υ, which we denote Υ^f. This oracle outputs a set of processes of size at least $n+1-f$ such that (as for Υ) eventually: the same set is permanently output at all correct processes, and this set is not the exact set of correct processes.

We finally prove that Υ^f encapsulates, in a precise sense, minimal failure information to circumvent any impossibility in an asynchronous shared memory system where f processes can crash.

This minimality holds even if the shared memory contains any atomic object type, beyond just registers. Our notion of minimality relies on a restricted variant of the reduction notion of Chandra, Hadzilacos, and Toueg [3]. We show that any oracle that (1) eventually outputs a *permanent (stable)* value, as well as (2) helps circumvent *some* impossibility in an asynchronous system with f failures, can be used to compute a set of processes of size $n+1-f$ that is not the set of correct processes, i.e., can be used to emulate Υ^f. Our necessity result is very general: the minimal information about failures to solve a non-trivial problem is extracted from the impossibility of the problem itself, i.e., unlike the classical weakest failure detector result by Chandra et al [3], we do not explicitly use the problem semantics. Also, our necessity proof is non-constructive: we show that a reduction algorithm exists, but do not provide an explicit construction of it. As a result, the proof turns out to be much simpler than the proof of [3].

Roadmap. The rest of the paper is organized as follows. Section 2 discusses some related work on the weakest failure detector question. Section 3 gives some basic definitions needed to state and prove our results. Section 4 defines and discusses Υ. Section 5 describes our set-agreement algorithm using Υ. Section 6 proves the minimality of Υ and Υ^f in the class of stable failure detectors. Section 7 concludes the paper with some final remarks.

2 Related Work

Chandra, Hadzilacos, and Toueg established in [3] the weakest failure detector to solve consensus, in the form of a failure detector oracle, denoted by Ω. This oracle outputs, whenever queried by a process, a single leader process. Eventually, the outputs stabilize on the same correct leader at all processes. Ω is the weakest failure detector to solve consensus in the sense that (a) there is an algorithm that solves consensus using Ω, and (b) for every oracle $\mathcal{D}$ that provides (only) information about failures such that some algorithm solves consensus using $\mathcal{D}$, there is an algorithm that emulates Ω using $\mathcal{D}$. In short, every such $\mathcal{D}$ encapsulates at least as much information about failures as Ω. The motivation of our work is to address the general question of the necessary information about failures that is needed to circumvent *any* asynchronous impossibility, i.e., beyond consensus.

Not surprisingly, in the case of two processes (i.e., the case where set agreement coincides with consensus), Ω and Υ are equivalent. Our minimality result is more restrictive: it is restricted to failure detectors that are stable. On the other hand, the proof is significantly simpler than that of [3]: in short, our approach extracts Ω from the fact of consensus impossibility [11] without having to go through valence arguments as in [3].

In the same vein, and in the general case of 2 or more processes, our approach extracts Υ directly from the fact of set-agreement impossibility, without having to go through topological arguments as in [20, 14, 2]. In this case, we prove that Υ is strictly weaker than failure detector Ω^n introduced in [18]. The latter failure detector outputs, whenever queried by a process, a subset of n processes such that, eventually, it is the same subset at all correct processes and it contains at least one correct process. Failure detector Ω^n was shown to be sufficient to solve (1) n-resilient n-set-agreement among $n+1$ processes using registers [18], and (2) $n+1$-process consensus using n-process consensus [21]. In fact, Ω^n was also shown to be necessary to implement $n+1$-process consensus using n-process consensus [13] and conjectured to be necessary to solve set-agreement [19]. It was our long quest to prove this conjecture that led us identify Υ and devise our set-agreement algorithm based on this oracle.

In a prior conference paper [12], we posed the question of the weakest failure detector *ever* and showed that Υ is the weakest non-trivial failure detectors among failure detectors that are stable and depend only on the set of correct processes (not on the finite prefix of a failure pattern). In this paper, we extend this result by getting rid of the second assumption. Chen et al [6] presented *unstable* failure detectors that are weaker than Υ but still strong enough to solve set-agreement.

We also conjectured in [12] that n-set agreement is the easiest problem that cannot be solved asynchronously by $n+1$ processes communicating via read-write shared-memory. Zielinski recently proved this conjecture, by introducing anti-Ω, an unstable failure that is strictly weaker than Υ, and showing that (1) anti-Ω is the weakest non-trivial *eventual* failure detector [22], and (2) anti-Ω is the weakest failure detector for solving set-agreement without any restrictions on failure detectors [23]. The minimality proof of [23] follows our approach of building upon the very fact that set-agreement is impossible to solve asynchronously which establishes that set-agreement is indeed the easiest unsolvable problem. However, the proof of [23] goes through a non-trivial simulation à la CHT, and it is unclear whether the proof can be generalized to the f-resilient case. In contrast, our minimality proof is much simpler, and it allows for a straightforward extension to f-resilient impossible problems.

3 Model

Our model of processes communicating through shared objects and using failure detectors is based on [15, 16, 3]. We recall below the details necessary for describing our results.

3.1 Processes and objects

The distributed system we consider is composed of a set Π of $n+1$ processes $\{p_1, \ldots, p_{n+1}\}$. Processes are subject to *crash* failures. A process that never fails is said to be *correct*. Process communicate through applying atomic operations on a collection of *shared objects*. We assume that the shared objects include *registers*, i.e., objects that export only base read-write operations. When presenting our algorithms (Section 5), we assume that only registers are available. The impossibility (Theorems 1 and 5) and necessity (Section 6) parts of our results do not restrict the types of shared objects that can be used in addition to registers.

3.2 Failure patterns and failure detectors

Besides accessing shared objects, processes can also make use of *failure detectors*, i.e., oracles that provide them with information about failures of other processes [4, 3]. The local module for process p_i of failure detector $\mathcal{D}$ is denoted by $\mathcal{D}_i$. Defining the notion of failure detector more precisely goes through defining the notions of *failure pattern* and *failure detector history*. A *failure pattern* F is a function from the time range $\mathbb{T} = \{0\} \cup \mathbb{N}$ to 2^{Π}, where $F(t)$ denotes the set of processes that have crashed by time t. Once a process crashes, it does not recover, i.e., $\forall t : F(t) \subseteq F(t+1)$. We define $\mathit{faulty}(F) = \cup_{t \in \mathbb{T}} F(t)$, the set of faulty processes in F. Processes in $\mathit{correct}(F) = \Pi - \mathit{faulty}(F)$ are called correct in F. A process $p \in F(t)$ is said to be *crashed* at time t. An *environment* is a set of failure patterns. Unless stated otherwise, we assume the environment that includes all failure patterns in which at least one process is correct, i.e., we assume that n or less processes can fail.

A *failure detector history H with range $\mathcal{R}$* is a function from $\Pi \times \mathbb{T}$ to $\mathcal{R}$. Informally, $H(p,t)$ is the value output by the failure detector module of process p at time t. A *failure detector $\mathcal{D}$* with

range $\mathcal{R}_\mathcal{D}$ is a function that maps each failure pattern to a nonempty set of failure detector histories with range $\mathcal{R}_\mathcal{D}$ (usually defined by a set of requirements that these histories should satisfy). $\mathcal{D}(F)$ denotes the set of possible failure detector histories permitted by $\mathcal{D}$ for failure pattern F. Note that we do not restrict possible ranges of failure detectors.

3.3 Algorithms

We define an *algorithm A using a failure detector $\mathcal{D}$* as a collection of deterministic automata, one for each process in the system, and an *initial memory state*, i.e., the initial states of all shared objects used by the algorithm. A_i denotes the automaton on which process p_i runs the algorithm A. Computation proceeds in atomic *steps* of A. In each step of A, process p_i

(i) invokes an operation on a shared object and receives a response from the object, *or* queries its failure detector module $\mathcal{D}_i$ and receives a value from $\mathcal{D}_i$ (in the latter case, we say that the step of p_i is a *query* step),
(ii) applies its current state, the response received from the shared object *or* the value output by $\mathcal{D}_i$ to the automaton A_i to obtain a new state, and
(iii) accepts an application *input* in I or produces (according to the automaton A_i) an *output* in O (I and O here are sets of all possible inputs and outputs, respectively).

A step of A is thus identified by a triple (p_i, x, y), where x is either the value returned by the invoked operation on a shared object and the resulting object state or, if the step is a query step, the failure detector value output at p during that step, and y is an input or an output. If no input is accepted and no output is produced in this step, then $y = \perp$.

A *run of algorithm A using a failure detector $\mathcal{D}$* is a tuple $R = \langle F, H, S, T\rangle$ where F is a failure pattern, $H \in \mathcal{D}(F)$ is a failure detector history, S is an *infinite* sequence of steps of A, and T is an *infinite* list of non-decreasing time values indicating when each step of S has occurred such that:

(1) For all $k \in \mathbb{N}$, if $S[k] = (p_i, x, y)$ (x and y denote here any legitimate values), then p_i has not crashed by time $T[k]$, i.e., $p_i \notin F(T[k])$;
(2) For all $k \in \mathbb{N}$, if $S[k] = (p_i, x, y)$ and $x \in \mathcal{R}_\mathcal{D}$, then x is the value of the failure detector module of p_i at time $T[k]$, i.e., $x = H(p_i, T[k])$;
(3) For all $k, \ell \in \mathbb{N}$, $k \neq \ell$, if $T[k] = T[l]$, then $S[k]$ and $S[l]$ are steps of different processes.
(4) S respects the specifications of all shared objects and all process automata, given their initial states and sequence of inputs that occur in S;
(5) Every process in $\mathit{correct}(F)$ takes infinitely many steps in S.

A *partial run* of an algorithm A is a finite prefix of a run of A.[5]

3.4 Traces and Problems

A *trace* is a tuple $\langle F, \sigma, T\rangle$ where F is a failure pattern, $\sigma \in (\Pi \times (I \cup O))^*$, and T is a sequence of non-decreasing time values, such that for all $k \in \mathbb{N}$, if $\sigma[k] = (p_i, x)$, then $p_i \notin F(T[k])$. We say that a run $R = \langle F, H, S, T\rangle$ *induces* a trace $\langle F, \sigma, \bar{T}\rangle$, if σ contains the sequence of all inputs and outputs that take place in S and $\bar{T}$ is the sequence of the corresponding times in T.

[5] A more formal definition of a run of an algorithm using a failure detector can be found in [3, 13].

A *problem* is a set of traces, usually defined by a set of properties traces must satisfy. A problem thus specifies the permitted sequences of inputs and outputs given the failure pattern and sequences of times when each input and output in the sequences takes place. In this paper, we consider problems that are closed under the *indistinguishability*: if a trace $\langle F, \sigma, T\rangle$ is in problem M, then any trace $\langle F', \sigma, T'\rangle$, such that $correct(F) = correct(F')$, is also in M.

An algorithm A solves a problem M *using a failure detector* $\mathcal{D}$, if the trace of every run of A using $\mathcal{D}$ is in M.

3.5 Comparing failure detectors

If, for failure detectors $\mathcal{D}$ and $\mathcal{D}'$, there is a *reduction* algorithm using $\mathcal{D}'$ that *extracts* the output of $\mathcal{D}$, i.e. implements a distributed variable $\mathcal{D}$-*output* such that in every run $R = \langle F, H', S, T\rangle$ of the reduction algorithm, there exists $H \in \mathcal{D}(F)$ such that for all $p_i \in \Pi$ and $t \in \mathbb{T}$, $H(p_i, t) = \mathcal{D}\text{-}output_i(t)$ (i.e., the value of $\mathcal{D}$-*output* output at p_i at time t), then we say that $\mathcal{D}$ *is weaker than* $\mathcal{D}'$. If $\mathcal{D}$ is weaker than $\mathcal{D}'$ but $\mathcal{D}'$ is *not* weaker than $\mathcal{D}$, then we say that $\mathcal{D}$ *strictly weaker than* $\mathcal{D}'$. If $\mathcal{D}$ and $\mathcal{D}'$ are weaker than each other, we say they are equivalent.

If $\mathcal{D}$ is weaker than $\mathcal{D}'$, then $\mathcal{D}'$ provides at least as much information about failures as $\mathcal{D}$: every problem that can be solved using $\mathcal{D}$ can also be solved using $\mathcal{D}'$. $\mathcal{D}$ is the weakest failure detector to solve a problem M if there is an algorithm that solves M using $\mathcal{D}$ and $\mathcal{D}$ is weaker than any failure detector that can be used to solve M. If the weakest failure detector to solve a problem A is strictly weaker than the weakest failure detector to solve a problem B, then we say that A is strictly weaker than B, i.e., A requires strictly less failure information than B.

4 A Very Weak Failure Detector

We introduce failure detector Υ, which outputs a non-empty set of processes ($\mathcal{R}_\Upsilon = 2^\Pi - \{\emptyset\}$), such that for every failure pattern F and every failure detector history $H \in \Upsilon(F)$, eventually:

(1) the same set $U \in 2^\Pi - \{\emptyset\}$ is permanently output at all correct processes.
(2) this set U is not the set of correct processes in F, i.e., $U \neq \text{correct}(F)$.

In a system of 2 processes, Υ and Ω [3] are equivalent. (Recall that Ω outputs a *leader* process so that eventually the same correct leader is output at all correct processes). Basically, to get Υ from Ω, every process outputs the complement of Ω in Π. On the other hand, to get Ω from Υ, every process outputs the complement of Υ if this is a singleton, and outputs the process identifier otherwise.

Ω was generalized to a failure detector Ω^n [18], which outputs a set of processes of size n so that, eventually, the same set containing at least one correct process is permanently output at all correct processes. (Clearly, Ω^1 is Ω.) The complement of Ω^n in Π is a legal output for Υ. Hence, Υ is weaker than Ω^n. The converse is however not true in the environment where n processes can fail, as we show below.

Theorem 1 *Υ is strictly weaker than Ω^n if $n \geq 2$.*

Proof. We just discussed how to transform Ω^n into Υ, so it remains to show that Υ cannot be transformed into Ω^n.

Assume, by contradiction, that we can extract the output of Ω^n from Υ. Extracting the output of Ω^n is equivalent to eventually identifying, in every run and at every correct process, the same process p_c that is *not the only* correct process in that run. Thus, our assumption implies that there exists an algorithm A that, using Υ, eventually outputs the same p_c at every correct process and $\Pi - \{p_c\}$ contains at least one correct process. To establish a contradiction, we construct a run of A in which the extracted failure detector output never stabilizes.

We consider the set of runs of A in which Υ permanently outputs $\{p_1, \ldots, p_n\}$ at all processes. Recall that this is a legitimate output of Υ if either p_{n+1} is correct or there is at least one faulty process in $\{p_1, \ldots, p_n\}$.

Consider partial runs of A in which no process fails but p_{n+1} is the only process that takes steps. Note that these partial runs are indistinguishable for p_{n+1} from partial runs in which every process but p_{n+1} is faulty. Thus, there exists a sufficiently long such partial run R_1 in which Υ always outputs $\{p_1, \ldots, p_n\}$ at all processes and A outputs a process $p_{i_1} \in \{p_1, \ldots, p_n\}$ at p_{n+1}.

Now consider partial runs extending R_1 in which (1) no process fails, and (2) every process takes exactly one step after R_1 after which p_{i_1} is the only process that takes steps. Again, these partial runs are indistinguishable for p_{i_1} from partial runs in which every process but p_{i_1} is faulty. Note that, since $n \geq 2$, if p_{i_1} is the only correct process in a run, then at least one process in $\{p_1, \ldots, p_n\}$ is faulty, and thus it is still legitimate for Υ to always output $\{p_1, \ldots, p_n\}$. Thus, there exists a failure-free partial run R_2 extending R_1 in which Υ always outputs $\{p_1, \ldots, p_n\}$ at all processes and A outputs a process $p_{i_2} \in \Pi - \{p_{i_1}\}$ at p_{i_1} *after* R_1 (i.e., after the last step of R_1 in R_2).

Now consider partial runs extending R_2 in which (1) no process fails, and (2) every process takes exactly one step after R_2 and then p_{i_2} is the only process that takes steps. Similarly, there exists a sufficiently long such partial run in which Υ always outputs $\{p_1, \ldots, p_n\}$ at all processes and A outputs a process $p_{i_3} \in \Pi - \{p_{i_2}\}$ at p_{i_2} after R_2.

By repeating this procedure, we obtain a failure-free run R of A in which Υ always outputs $\{p_1, \ldots, p_n\}$ at all processes, but the extracted failure detector output never stabilizes — a contradiction. □

5 Set-Agreement

5.1 The problem

In the k-set-agreement problem, processes need to agree on at most k values out of a possibly larger set of values. Let V be the value domain such that $\perp \notin V$. Every process p_i starts with an initial value v in V (we say p_i *proposes* v), and aims at reaching a state in which p_i irrevocably commits on a decision value v' in V (we say p_i *decides* on v'). Every run of a k-set-agreement algorithm satisfies the following properties: (1) *Termination:* Every correct process eventually decides on a value; (2) *Agreement:* At most k values are decided on; (3) *Validity:* Any value decided is a value proposed.

In the following, we first focus on solving n-set-agreement in a system of $n+1$ processes. We sometimes also talk about implementing n-resilient n-set agreement. This problem is impossible if processes can only communicate using registers, n processes can crash, and no information about failures is available [20, 14, 2].

We show how to circumvent this impossibility using Υ: we describe a protocol that solves n-set-agreement using registers and Υ, while tolerating the failure of n processes. Basically, implementing

set-agreement aims at *excluding* at least one proposed value among the $n+1$ possible ones. Our protocol achieves this by using the output of Υ to eventually split the processes into two non-overlapping subsets: those in the subset output by Υ, and which we call *gladiators*, and those outside that subset, and which we call *citizens*. Intuitively, *gladiators* do not decide on any value until either they make sure one of them gives up its value, which is guaranteed to happen if one of them crashes, or they see a value of a *citizen*, in which case they simply decide on that value. The property eventually ensured by Υ is that either at least one of the *gladiators* crash or at least one of the *citizens* is correct.

Besides putting this intuition to work, technical difficulties handled by our protocol include coping with the facts that (1) Υ might output random sets for an arbitrarily long periods of time, providing divergent and temporary information about who is *gladiator* and who is *citizen*, and (2) *citizens* might be faulty. A key procedure we use to handle these difficulties is the *k-converge* routine, introduced in [21]. A process calls *k-converge* with an input value in V and gets back an output value $v \in V$ and a boolean c. We say that the process *picks* v and, if $c = true$, we say that the process *commits* v. The *k-converge* routine ensures the following properties: (1) *C-Termination*: every correct process picks some value; (2) *C-Validity*: if a process picks v then some process invoked *k-converge* with v; (3) *C-Agreement*: If some process commits to a value, then at most k values are picked; (4) *Convergence*: If there are at most k different input values, then every process that picks a value commits. For any $k \in \{1, \ldots, n+1\}$, the *k-converge* routine can be implemented using registers in an asynchronous system where any number of processes may fail [21]. By definition, 0-*converge*(v) always returns $(v, false)$.

5.2 The protocol

The abstract pseudo-code of the protocol that solves n-set agreement using Υ and registers is described in Figure 1.

The protocol proceeds in rounds. In every round r, the processes first try to reach agreement using n-convergence (line 4). If a process p_i commits to a value v, then p_i writes v in register D and returns v. If p_i fails to commit (which can only happen if all $n+1$ processes take part in the n-convergence instance), then p_i queries Υ. Let U be the returned value.

Now p_i cyclically executes the following procedure (lines 12–17). If p_i does not belong to U (p_i believes it is a *citizen*), then p_i writes its value in a shared register $D[r]$ and proceeds to the next round. Otherwise (p_i believes it is a *gladiator*), p_i takes part in the $(|U|-1)$-convergence protocol trying to eliminate one of the values concurrently proposed by processes in U. (Recall that, by definition, 0-*converge*(v) always returns $(v, false)$.) The procedure is repeated as long as none of the conditions in line 17 is satisfied, i.e., (a) no process participating in the current round r reports that the output Υ has not yet stabilized, (b) $(|U|-1)$-convergence does not commit to a value, and (c) no non-$\perp$ value is found in $D[r]$ or D (line 17). If p_i finds $D[r] \neq \perp$, then p_i adopts the value in $D[r]$ and proceeds to round $r+1$. If p_i finds $D \neq \perp$ then p_i returns D.

Remember that there is a time after which Υ permanently outputs, at all correct processes, the same set U that is not the set of correct processes: U either contains a faulty process or there is a correct process outside U. Thus, no process can be blocked in round r by repeating forever the procedure described above: eventually, either some process outside U writes its value in $D[r]$, or some process is faulty in U and $(|U|-1)$-convergence returns a committed value.

As a result, eventually, there is a round in which at least one input value is eliminated: either some process in U adopts a value from outside U, or processes in U commit to at most $|U|-1$

Shared abstractions:
Registers D, $D[\,]$, initially $\perp$
Binary registers *Stable*$[\,]$, initially *true*
Convergence instances: n-*converge*$[\,]$,
j-*converge*$[\,][\,]$, for all $j = 0, \ldots, n$

Code for every process p_i:

```
1   v_i := the input value of p_i; r := 0
2   repeat
3     r := r + 1
4     (v_i, c) := n-converge[r](v_i)
5     if c = true then
6       D := v_i; return (v_i)
7     U := query(Υ_i)
8     if p_i ∉ U then      { p_i is a citizen }
9       D[r] := v_i
10    else      { p_i is a gladiator }
11      k := 0
12      repeat
13        k := k + 1
14        (v_i, c) := (|U| − 1)-converge[r][k](v_i)
15        if c = true then D[r] := v_i
16        if U ≠ query(Υ_i) then Stable[r] := false
17      until D ≠ ⊥ or D[r] ≠ ⊥ or ¬Stable[r]
18      if D[r] ≠ ⊥ then
19        v_i := D[r]      { Adopt a value from a citizen or a committed gladiator }
20  until D ≠ ⊥
21  return (D)
```

Fig. 1. Υ-based set agreement protocol.

input values. In both cases, every process that participates in n-convergence in round $r+1$ (line 4) commits one of at most n "survived" values.

Theorem 2 *The algorithm in Figure 1 solves n-set agreement using Υ and registers.*

Proof. Consider an arbitrary run R of the algorithm in Figure 1.

Validity immediately follows from the protocol and the C-Validity property of k-converge.

Agreement is implied by the fact that every decided value is first committed by n-convergence (line 4). Indeed, let r be the first round in which some process p_i commits to a value after invoking n-*converge*$[r]$. By the C-Agreement property of n-convergence, every process that invoked n-*converge*$[r]$ picked at most n different values. Thus, no more than n different values can ever be written in register D. Since a process is allowed to decide on a value only if the value was previously written in D (lines 6 and 21), at most n different values can be decided on.

Now consider Termination. We observe first that no process can decide unless D contains a non-$\perp$ value, and if $D \neq \perp$, then every correct process eventually decides. This is because the converge instances are non-blocking and every correct process periodically checks whether D contains a non-$\perp$ value and, if there is one, returns the value (lines 20 and 17). Assume now, by contradiction, that $D = \perp$ forever and, thus, no process ever decides in R.

Let U be the stable output of Υ in R, i.e., at every correct process, Υ eventually permanently outputs U. Whenever a process observes that the output of Υ is not stable in round r, it sets register

Stable[r] to *true* (line 16) and proceeds to the next round. Further, if a process finds $D[r] \neq \perp$, then eventually every correct process finds $D[r] \neq \perp$ and proceeds to the next round. Moreover, by our assumption, no process ever writes in D and returns in line 6. Thus, there exists a round r such that every correct process reaches r, and the observed output of Υ at every process that reached round r has stabilized on U.

Recall that U is a non-empty set of processes that is *not* the set of correct processes in R, i.e., $U \neq \emptyset$ and $U \neq C$, where C is the set of correct processes in R. Thus, two cases are possible: (1) C is a proper subset of U, and (2) $C - U \neq \emptyset$.

In case (1), there is at least one faulty process in U. Since every faulty process eventually crashes, there exists $k \in \mathbb{N}$, such that at most $|U| - 1$ values are proposed to $(|U| - 1)$-*converge*$[r][k]$. By the Convergence property of the $(|U| - 1)$-*converge* procedure, every correct process eventually commits to a value, writes it in $D[r]$ and proceeds to round $r + 1$.

In case (2), there is at least one correct process p_j outside U. Thus, p_j eventually reaches round r and writes its current value in $D[r]$. Thus, every correct process eventually reads the value, adopts it and proceeds to round $r + 1$.

In both cases, every correct process reaches round $r + 1$. By the algorithm, every process that reaches round $r + 1$ adopted a value previously written in $D[r]$.

A process is allowed to write a value in $D[r]$ only if (a) the process is in $\Pi - U$, or (b) a process is in U and the value is committed in $(|U| - 1)$-*converge*$[r][k]$ for some k.

If (b) does not hold, then at most $n + 1 - |U| \leq n$ distinct values can be found in $D[r]$. Otherwise, consider the first sub-round k such that some process in U has committed a value in $(|U| - 1)$-*converge*$[r][k]$. By the C-Agreement property of $(|U| - 1)$-convergence, at most $|U| - 1$ distinct values are picked by processes in U in $(|U| - 1)$-*converge*$[r][k]$. If a process does not commit on a value picked in $(|U| - 1)$-*converge*$[r][k]$, it uses the value in $(|U| - 1)$-*converge*$[r][k+1]$ (line 14). By the Convergence, C-Agreement, and C-Validity properties of $(|U| - 1)$-convergence, every correct process in U commits on one of at most $|U| - 1$ distinct proposed values in sub-round k or $k + 1$.

In both cases, at most $n+1-|U|+|U|-1 = n$ distinct values can ever be found in $D[r]$. Hence, at most n distinct values can be proposed to n-convergence (line 4) in round $r+1$. By the Convergence property of n-convergence, every correct process commits and decides — a contradiction.

Thus, eventually, every correct process decides. □

Remark. Our algorithm actually solves a stronger version of set-agreement that terminates even if not every correct process *participates*, i.e., proposes a value and executes the protocol. Indeed, assume (by slightly changing the model) that some (possibly correct) process does not participate in a given run of the algorithm in Figure 1. Thus, in round 1, at most n different values are proposed to n-*converge* (line 4) and, by the Convergence property of n-*converge*, every correct participant commits to a value. Thus, every correct *participant* returns in line 6 of round 1.

As a corollary to Theorems 1 and 2, we disprove the conjecture of [19] by showing that:

Corollary 3 *For all $n \geq 2$, Ω^n is not the weakest failure detector to implement n-resilient n-set-agreement among $n + 1$ processes using registers.*

As a corollary to Theorems 1 and 2, and the fact that Ω^n is the weakest failure detector to implement $n + 1$-process consensus using n-process consensus [13], we obtain that implementing n-set-agreement using registers is *strictly easier* than solving consensus using n-process consensus objects and registers:

Corollary 4 *For all $n \geq 2$, in a system of $n+1$ processes where up to n can fail, every failure detector that can be used to solve consensus using n-consensus objects and registers can also be used to solve $n+1$-set-agreement using registers, but not vice versa.*

5.3 f-Resilient Set-Agreement

For pedagogical purposes, we focused so far on the environment where n out of $n+1$ processes can crash, i.e., on the "wait-free" case. In this section, we consider the more general environment where f processes can crash, and $0 < f < n+1$. More specifically, we consider the environment $\mathcal{E}^f$ that consists of all failure patterns F such that $\mathit{faulty}(F) \leq f$.

By reduction to the impossibility of wait-free set agreement, Borowsky and Gafni showed that f-set agreement is impossible in $\mathcal{E}^f$ [2]. We present a failure detector, which generalizes Υ, and which circumvents this impossibility. This failure detector, which we denote by Υ^f, outputs a set of processes of size at least $n+1-f$ ($\mathcal{R}_{\Upsilon^f} = \{U \subseteq \Pi : |U| \geq n+1-f\}$), such that, for every failure pattern $F \in \mathcal{E}^f$ and every failure detector history $H \in \Upsilon^f(F)$, eventually (as for Υ): (1) the same set is permanently output at all correct processes, and (2) this set is not the set of correct processes in F. Clearly, Υ^n is Υ.

Failure detector Ω^f can also be used to solve f-resilient f-set agreement [17, 18]. It is easy to see that Υ^f is weaker than Ω^f in $\mathcal{E}^f$: to emulate Υ^f, every process simply outputs the complement of Ω^f in Π. Eventually the correct processes obtain the same set of $n+1-f$ processes that is not the set of correct processes: the output of Ω^f eventually includes at least one correct process.

It is also straightforward to extract $\Omega^1 = \Omega$ from Υ^1 in $\mathcal{E}^1$. In the reduction algorithm, every process p_i periodically writes ever-growing timestamps in the shared memory. If Υ^1_i outputs a proper subset of Π (of size n), then p_i elects the process $p_\ell = \Pi - \Upsilon_i$, otherwise, if Υ^1 outputs Π (i.e., exactly one process is faulty), then p_i elects the process with the smallest id among n processes with the highest timestamps. Eventually, the same correct process is elected by the correct processes — the output of Ω is extracted. However, in general, Υ^f is strictly weaker than Ω^f:

Theorem 5 *Υ^f is strictly weaker than Ω^f in $\mathcal{E}^f$ if $2 \leq f \leq n$.*

Proof. We generalize the proof of Theorem 1. By contradiction, assume there exists an algorithm A using Υ^f that, in every run with at least $n+1-f$ correct processes, eventually outputs at every correct process the same set of processes L such that $|L| = f$ and L contains at least one correct process. To establish a contradiction, we construct a run of A in which the extracted output never stabilizes.

We consider the set of runs of A in which Υ^f permanently outputs $U = \{p_1, \ldots, p_n\}$ at all processes. Recall that this is a legitimate output if either p_{n+1} is correct or there is at least one faulty process in $\{p_1, \ldots, p_n\}$.

Let R_1 be any partial run of A in which no process fails and Υ^f always outputs U. Let L_1 be the set output by A at some process in run R_1.

Now consider partial runs that extend R_1 in which (1) no process fails, and (2) every process takes exactly one step after the last step of R_1 and then only processes in $\Pi - L_1$ take steps. These partial runs are indistinguishable for processes in $\Pi - L_1$ from partial runs in which the processes in L_1 are faulty. Note that, since $2 \leq f \leq n$, $U \neq \Pi - L_1$, and it is thus legitimate for Υ^f to output U in any run in which every process in L_1 is faulty. Thus, there exists such a partial run R_2 in which Υ^f always outputs U and A outputs a set $L_2 \neq L_1$ at some process after R_1.

Now consider partial runs that extend R_2 in which (1) no process fails, and (2) every process takes exactly one step after the last step of R_2 and then only processes in $\Pi - L_2$ take steps. Similarly, there exists such a partial run R_3 in which Υ^f always outputs U and A outputs a set $L_3 \neq L_2$ at some process after R_2.

Following this procedure, we obtain a failure-free run of A in which Υ^f always outputs $U = \{p_1, \ldots, p_n\}$ but the extracted output of Ω^f never stabilizes — a contradiction. □

A generalized f-resilient f-set-agreement algorithm using Υ^f is presented in Figure 2. The algorithm essentially follows the lines of our "wait-free" algorithm described in Figure 1, except that now the set U of $n+1-f$ or more *gladiators* (processes that are eventually permanently output by Υ^f) have to be able to eventually commit on at most $|U| + f - n - 1$ distinct values, so that, together with at most $n + 1 - |U|$ values chosen by the *citizens*, there would eventually be at most f distinct values in the system. To achieve this, we add a simple mechanism based on the use of *atomic snapshots* [1].

An atomic snapshot object has $n+1$ *positions* and exports two atomic operations: *update* and *snapshot*. Operation *update*(i, v) writes value v in position i, and *snapshot*() returns the content of the object. Note that the results of every two snapshots are related by containment, i.e., one of them contains, in each position, the same or more recently written value than the other. Atomic snapshots can be implemented in an asynchronous system using registers [1].

In our algorithm, the use atomic snapshots ensures that, if at least one gladiator is faulty and all citizens are faulty, then the correct gladiators eventually eliminate at least $n+1-f$ values, and, thus, at most f values will eventually be decided. In each iteration (r, k) (lines 15-30 in Figure 2), every gladiator (process in U) first updates its value in atomic snapshot object $A[r][k]$, and then repeatedly takes snapshots of $A[r][k]$ until a snapshot with at least $n + 1 - f$ non-$\perp$ values is obtained (line 19). Since all snapshots of $A[r][k]$ are related by containment, and assuming that each resulting snapshot contains at least $n+1-f$ and at most $|U|-1$ values (at least one process does not access $A[r][k]$), at most $|U| + f - n - 1$ distinct snapshots of $A[r][k]$ can be obtained by processes in U. Every process in U adopts the minimal value in its latest snapshot of $A[r][k]$ (line 25), and, thus, at most $|U| + f - n - 1$ distinct values can be adopted. As a result, gladiators commit on at most $|U| + f - n - 1$ values using the $(|U| + f - n - 1)$-*converge*$[r][k]$ procedure (line 26).

Theorem 6 *There is an algorithm that implements f-set agreement using Υ^f and registers in $\mathcal{E}^f$.*

Proof. Consider an arbitrary run of the protocol in Figure 2. The Agreement and Validity properties are immediate from the algorithm. Termination is shown along the lines of the proof of our "wait-free" algorithm described in Figure 1, except that now we have a new potentially blocking loop (in lines 17–19).

Suppose, by contradiction, that some correct process is blocked in the loop of lines 17–19, while executing a sub-round k of a round r (let k and r be the earliest sub-round and round, respectively, in which this happens). It is easy to see that, if a correct process exits the loop (by evaluating the condition of line 19 to true), then eventually every correct process is freed too. Further, since no correct process was blocked in the loop of lines 17–19 before sub-round k of round r, using the arguments presented in the proof of Theorem 2, we observe that every correct process also reached round r. Note that in round r, no process has written a non-$\perp$ value in $D[r]$ (line 11): otherwise, every correct process that is blocked in lines 17–19 would eventually read the value and escape.

Shared abstractions:
Registers D, $D[\,]$, initially $\perp$
Atomic snapshot objects $A[\,][\,]$, initially $\perp$
Binary registers $Stable[\,]$, initially *true*
Convergence instances: n-*converge*$[\,]$,
j-*converge*$[\,][\,]$, for all $j = 0, \ldots, n$

Code for every process p_i:

```
1   v_i := the input value of p_i
2   r := 0
3   repeat
4     r := r + 1
5     (v_i, c) := f-converge[r](v_i)
6     if c = true then
7       D := v_i
8       return (v_i)
9     U := query(Υ_i)
10    if p_i ∉ U then
11      D[r] := v_i
12    else
13      k := 0
14      repeat
15        k := k + 1
16        A[r][k].update(i, v_i)
17        repeat
18          V := A[r][k].snapshot()
19        until D ≠ ⊥ or D[r] ≠ ⊥ or ¬Stable[r]
                or V contains ≥ n + 1 − f non-⊥ entries
20        if D ≠ ⊥ then
21          return(D)
22        else if D[r] ≠ ⊥ then
23          v_i := D[r]
24        else if Stable[r] then
25          v_i := min non-⊥ value in V
26          (v_i, c) := (|U| + f − n − 1)-converge[r][k](v_i)
27          if c = true then
28            D[r] := v_i
29          if U ≠ query(Υ_i) then
30            Stable[r] := false
31      until D ≠ ⊥ or D[r] ≠ ⊥ or ¬Stable[r]
32      if D[r] ≠ ⊥ then
33        v_i := D[r]
34  until D ≠ ⊥
35  return (D)
```

Fig. 2. Υ^f-based f-resilient f-set agreement protocol.

Thus, *every* correct process is eventually blocked in lines 17–19, while executing sub-round k of round r. Previously, every correct process p_i has written a non-$\perp$ value in $A[r][k][i]$ (line 16). But since there are at least $n+1-f$ correct processes in R, $A[r][k]$ eventually contains at least $n+1-f$ non-$\perp$ entries and, thus, the condition in line 19 is eventually satisfied — a contradiction. Thus, no correct process can be blocked forever, while executing lines 17–19.

Suppose, by contradiction, that there is a run R of our algorithm in which some correct process never decides. By repeating the arguments presented in the proof of Theorem 2, D always contains $\perp$ in R, and there exists a round r such that every correct process reached round r, and the observed output of Υ^f at every process that reached round r has stabilized on some set U in round r. By the properties of Υ^f, U is of size at least $n+1-f$ and U is not the set of correct processes in R.

Hence, $\Pi - U$ contains no correct process: otherwise, some correct process in $\Pi - U$ would eventually write a non-$\perp$ value in $D[r]$ in line 11, and every correct process would eventually exit the loop. Thus, every correct process p_i belongs to U and eventually writes a non-$\perp$ value in $A[r][k][i]$ (line 16). But since there are at least $n+1-f$ correct processes in R, $A[r][k]$ eventually contains at least $n+1-f$ non-$\perp$ entries and, thus, the condition in line 19 is eventually satisfied — a contradiction.

In every sub-round k of round r, each correct process eventually exits the loop in lines 17–19 and, since D is never $\perp$, reaches line 23 (if $D[r] \neq \perp$) or line 26 (otherwise).

Note that at most $n+1-|U|$ different non-$\perp$ values can be written in $D[r]$ by processes not in U. On the other hand, a process in U is allowed to write v in $D[r]$ only if it has committed on v in some instance of $(f+|U|-n-1)$-*converge*$[r][k]$. By the C-Agreement and Validity properties of $(f+|U|-n-1)$-convergence and the fact that every value returned by $(f+|U|-n-1)$-*converge*$[r][k]$ is adopted, at most $f+|U|-n-1$ distinct values can ever written by processes in U. Thus, at most $n+1-|U|+f+|U|-n-1=f$ distinct values can ever be written in $D[r]$.

Suppose that $D[r] \neq \perp$ at some point in R. Thus, eventually every process either fails or adopts one of at most f values written in $D[r]$ (line 23 or 33), and then proceeds to round $r+1$. Hence, by the Convergence property of f-convergence, every correct process commits a value after invoking f-*converge*$[r+1]$ and decides — a contradiction.

Now suppose that $D[r] = \perp$ forever. By the algorithm, there are no correct processes outside U and, thus, there is at least one faulty process in U (otherwise, U would be the set of correct processes, violating the properties of Υ^f). Let k be a sub-round of round r in which no faulty process participates (every faulty process fails before starting the sub-round). Since there is at least one faulty process in U, at most $|U|-1$ values can be written in $A[r][k]$.

Now consider all sets that can be returned by $A[r][k]$*snapshot*() at different processes right before the process exits the repeat-until loop in lines 17–31. Every such set contains at least $n+1-f$ and at most $|U|-1$ non-$\perp$ values. Moreover, by the properties of atomic snapshot [1], all these sets are related by containment. Thus, there can be at most $|U|-1-(n+1-f)+1 = |U|+f-n-1$ distinct sets, and, thus, at most $|U|+f-n-1$ different values can be computed by the processes in line 25. Hence, by the Convergence property of $(|U|+f-n-1)$-*converge*$[r][k]$, every correct process that invokes the operation, commits on a value and writes it in $D[r]$ — a contradiction.

Thus, eventually, every correct process decides. □

6 The Necessity of Υ^f

In this section we show that, in a certain sense, Υ^f is minimal in systems where up to f processes can crash. Our minimality result holds within the class of *stable* failure detectors that eventually stick to the same information about failures, say after all faulty processes have crashed. More specifically, we show that Υ^f is weaker than any stable f-*non-trivial* failure detector, i.e., any stable failure

detector that cannot be implemented in an f-resilient asynchronous system. This implies that Υ is also minimal when up to n processes can crash.

6.1 Intuition

To get intuition about our minimality result, let us consider the case $f = n$ and focus on a restricted class of "faithful" non-trivial failure detectors that, in every run, output the same value at every correct process, and the output value depends only on the set of correct processes. The immediate observation is that for each faithful failure detector $\mathcal{D}$, and for each value $d \in \mathcal{R}_\mathcal{D}$, there exists $C \in 2^\Pi - \{\emptyset\}$ such that, for all F with $correct(F) = C$, $\mathcal{D}$ cannot output d for F. Indeed, if there is a value that can be output by $\mathcal{D}$ in every failure pattern, then $\mathcal{D}$ can be implemented from the "dummy" failure detector that always outputs d. But this would contradict the assumption that $\mathcal{D}$ is non-trivial. Thus, in every run, by observing the output of a "faithful" failure detector $\mathcal{D}$, we can deterministically choose a non-empty set of processes that cannot be the set of correct processes in that run — this is sufficient for emulating Υ.

Note that the sketched necessity proof is non-constructive. Indeed, determining the set of processes C that is "incompatible" with d is in general undecidable. However, to show that a reduction algorithm exists, it is sufficient to show that there exists a deterministic map from $\mathcal{R}_\mathcal{D}$

In the following, we extend this intuition to the class of stable failure detectors.

6.2 Stable failure detectors

Establishing our most general necessity result goes through delimiting the scope of failure detectors within which Υ^f is minimal. We consider the class of *stable* failure detectors. We say that a failure detector, $\mathcal{D}$, is stable if the same value is eventually permanently output by $\mathcal{D}$ at all correct processes. Formally, for every failure pattern F and every $H \in \mathcal{D}(F)$, there exists a value $d \in \mathcal{R}_\mathcal{D}$ and $t \in \mathbb{N}$ such that for all $t' \geq t$ and $p_i \in correct(H)$, $H(p_i, t') = d$ (we say that d is stable in H).[6] Most failure detectors proposed in the literature for solving decision problems in the shared memory model [4, 3, 18, 7] are stable or equivalent to some stable failure detectors. Some failure detectors are nevertheless unstable and cannot be shown equivalent to a stable one [22, 23].

6.3 Minimality

Before we proceed with the minimality proof, we introduce some auxiliary notions.

Let $\mathcal{D}$ be a failure detector with range $\mathcal{R}$. Let σ be an element in $(\Pi \times \mathcal{R})^*$, i.e., a sequence $(q_1, d_1), (q_2, d_2), \ldots$, where for all $k \in \mathbb{N}$, $q_k \in \Pi$ and $d_k \in R$. We denote by $correct(\sigma)$ the set of processes that appear infinitely often in σ. We say that σ is an *f-resilient sample* of $\mathcal{D}$ if $|correct(\sigma)| \geq n + 1 - f$ and there exist a failure pattern $F \in \mathcal{E}_f$, a history $H \in \mathcal{D}(F)$ and a list T of non-decreasing time values such that,

(i) for all $k \in \mathbb{N}$, $q_k \notin F(T[k])$,
(ii) for all $k \in \mathbb{N}$, $d_k = H(q_k, T[k])$, and
(iii) for all $k, \ell \in \mathbb{N}$, $k \neq \ell$, if $T[k] = T[l]$, then $q_k \neq q_\ell$.

[6] Our lower bound proofs actually work also for "locally stable" failure detectors that eventually permanently output a "stable" value at every correct process (the stable values output at different correct processes can be different though).

Intuitively, $\sigma = (q_1, d_1), (q_2, d_2), \ldots$ is an f-resilient sample of $\mathcal{D}$ if failure-detector values $d_1, d_2, \ldots$ could have been observed (in this order) by processes $q_1, q_2, \ldots$ in a run of some algorithm using $\mathcal{D}$ in $F \in \mathcal{E}_f$. The following observation is immediate from the definition.

Lemma 7 *Let $\sigma \in (\Pi \times \mathcal{R})^*$ be an f-resilient sample of $\mathcal{D}$, and σ' be a subsequence of σ such that $correct(\sigma) = correct(\sigma')$. Then σ' is also an f-resilient sample of $\mathcal{D}$.*

We also introduce the notion of a *dummy* failure detector, which always outputs the same value (i.e., its range is a singleton $\{d\}$). Clearly, a dummy failure detector $\mathcal{D}$ can be *emulated* in an asynchronous system. If a problem can be solved in $\mathcal{E}^f$ using a *dummy* failure detector, then we say that the problem is *f-resilient solvable*. Otherwise, we say that the problem is *f-resilient impossible*. We say that a failure detector is *f-non-trivial* if it can be used to solve an f-resilient impossible problem in $\mathcal{E}^f$. By definition, an n-resilient impossible problem is wait-free impossible.

First we observe that the output of an f-non-trivial failure detector can be associated with a sequence in $(\Pi \times \mathcal{R})^*$ that is "incompatible" with the current run:

Lemma 8 *Let $\mathcal{D}$ be an f-non-trivial failure detector. Let $\mathcal{R}$ be the range of $\mathcal{D}$. Then, for all $d \in \mathcal{R}$, there exists a sequence $\sigma \in (\Pi \times \{d\})^*$ such that $|correct(\sigma)| \geq n+1-f$ and σ is not an f-resilient sample of $\mathcal{D}$.*

Proof. Let A be an algorithm that solves an f-resilient impossible problem M using $\mathcal{D}$.

By contradiction, suppose that there exists a value $d \in \mathcal{R}$ such that each $\sigma \in (\Pi \times \{d\})^*$ with $|correct(\sigma)| \geq n + 1 - f$ is an f-resilient sample of $\mathcal{D}$, i.e., there exist a failure pattern $F \in \mathcal{E}_f$, a history $H \in \mathcal{D}(F)$ and a list T of increasing time values such that for all $k \in \mathbb{N}$, (i) $q_k \notin F(T[k])$, (ii) $d = H(q_k, T[k])$, and (iii) for all $\ell \in \mathbb{N}$, $k \neq \ell$, if $T[k] = T[l]$, then $q_k \neq q_\ell$.

Consider algorithm A' that is defined exactly like A except that, instead of $\mathcal{D}$, A' uses a dummy failure detector $\mathcal{I}_d$ that always outputs d: each time a process is expected (according to its state in A) to query $\mathcal{D}$ we substitute $\mathcal{D}$ with $\mathcal{I}_d$.

For each run $R' = \langle F', H', S, T' \rangle$ of A' where $F' \in \mathcal{E}_f$, there exists a run $R = \langle F, H, S, T \rangle$ of A. Indeed, let $q_1, q_2, \ldots$ be the sequence of process ids such that $\forall k \in \mathbb{N}$, $S[k] = (q_k, -, -)$ and $H'(q_k, T'[k]) = d$. Let F be a failure pattern in $\mathcal{E}_f$, H be a history in $\mathcal{D}(F)$, and T be a list of non-decreasing time values such that $correct(F) = correct(F')$, and $\forall k \in \mathbb{N}$, $q_k \notin F(T[k])$, $H(q_k, T[k]) = d$, and for all $k, \ell \in \mathbb{N}$, $k \neq \ell$, if $T[k] = T[\ell]$, then $q_k \neq q_\ell$. By construction, $R = \langle F, H, S, T \rangle$ is a run of A, and, thus, the trace of R is in M. But the traces of R and R' are indistinguishable, and, thus, the trace of R' is also in M. Hence, A' solves M using $\mathcal{I}_d$ - a contradiction to the assumption that M is f-resilient impossible.

Thus, for all $d \in \mathcal{R}$, there exists a sequence $\sigma \in (\Pi \times \{d\})^*$, with $|correct(\sigma)| \geq n + 1 - f$, that is not an f-resilient sample of $\mathcal{D}$. □

For a sequence $\sigma \in (\Pi \times \{d\})^*$, let $w(\sigma)$ denote the length of the shortest prefix of σ that includes all steps that processes in $\Pi - correct(\sigma)$ take in σ; if $correct(\sigma) = \Pi$, then $w(\sigma) = 0$. The following corollary follows immediately from Lemma 8:

Corollary 9 *For each f-non-trivial failure detector $\mathcal{D}$ with range $\mathcal{R}$, there exists a map $\varphi_{\mathcal{D}}$ that carries each $d \in \mathcal{R}$ to a tuple $(correct(\sigma), w(\sigma))$, where $\sigma \in (\Pi \times \{d\})^*$, $|correct(\sigma)| \geq n + 1 - f$, and σ is not an f-resilient sample of $\mathcal{D}$.*

Note that we do not *construct* the map $\varphi_{\mathcal{D}}$ here: it is sufficient for us to know that such a map exists for each f-non-trivial failure detector.

Now we are ready to prove the necessity part of our result. Roughly, we extract the output of Υ from the output of an f-non-trivial failure detector $\mathcal{D}$ as follows. Processes periodically query their modules of $\mathcal{D}$ and *report* the obtained values by writing the values equipped with increasing timestamps in the shared memory. Each output failure detector value d is associated with an sequence $\sigma \in (\Pi \times \mathcal{R})^*$ ($|correct(\sigma)| \geq n+1-f$) that is suspected to be "incompatible" with the current run (here we use the map $\varphi_{\mathcal{D}}$ the existence of which is guaranteed by Corollary 9).

For a stabilized value d, if the shortest prefix of σ that includes all steps of processes appearing only finitely often in σ was *observed* in the current run (could have taken place with the current failure pattern), then the processes evaluate the extracted output of Υ_f as the set of processes that appear *infinitely* often in σ. Indeed, $correct(\sigma)$ cannot be the set of correct processes: otherwise, by Lemma 7, σ would be an f-resilient sample of $\mathcal{D}$. Here to make sure that a given finite schedule of length r could have happened with the current failure pattern, it is sufficient for some process to observe r consecutive batches of steps such that, in every batch, *every* process queried its failure detector module and obtained d at least once.

On the other hand, as long as the finite prefix of σ is not observed, the processes evaluate the extracted output of Υ_f as Π. Note that a given finite prefix of σ is never observed only if some process is faulty. In that case, Π cannot be the set of correct processes.

In both cases, every process evaluates an extracted value of Υ^f. If a value different from d is reported, then the extraction procedure is restarted. Eventually, the values will stabilize, and the extracted output will conform with the specification of Υ^f.

Theorem 10 *Υ^f is weaker than any f-non-trivial stable failure detector.*

Proof. Let $\mathcal{D}$ be any stable failure detector that can be used to solve an f-resilient impossible problem M. Let A be the corresponding algorithm. Let $\mathcal{R}$ be the range of $\mathcal{D}$.

The reduction algorithm that transforms $\mathcal{D}$ into Υ^f is presented in Figure 3. In the algorithm, every process p_i runs two parallel tasks, Task 1 and Task 2. Here $\varphi_{\mathcal{D}}$ denotes a map that carries each $d \in \mathcal{R}$ to a tuple $(correct(\sigma), w(\sigma))$, where $\sigma \in (\Pi \times \{d\})^*$, $|correct(\sigma)| \geq n+1-f$, and σ is not an f-resilient sample of $\mathcal{D}$. (By Corollary 9, such a map exists.)

In Task 1, p_i periodically queries its module of $\mathcal{D}$ and writes the returned value, equipped with an ever-increasing timestamp, in a register $R[i]$ that is periodically read by all. The ever-increasing timestamps allows p_i to detect when a given process p_j reports a new failure detector value: it is sufficient to wait until p_j increases its timestamp (i.e., writes in $R[j]$) at least twice.

In Task 2, p_i proceeds in rounds. In every round, consisting of steps described in lines 7-21 of Figure 3, p_i tries to compute the stable output of Υ^f, as long as the observed output of $\mathcal{D}$ at every process does not change (lines 15 and 21). When some process reports that its output of $\mathcal{D}$ has not stabilized yet, p_i proceeds to the next round. Since $\mathcal{D}$ eventually outputs the same value at every correct process, every correct process is eventually blocked forever in line 15 or line 21.

Let d be the output of $\mathcal{D}$ at p_i in a given round of Task 2. In the beginning of the round, p_i sets Υ^f-$output_i$ to Π (line 8), and deterministically evaluates (S, r) as $(correct(\sigma), w(\sigma))$, where $\sigma \in (\Pi \times \{d\})^*$, $|correct(\sigma)| \geq n+1-f$, and σ is not an f-resilient sample of $\mathcal{D}$ (line 10). If $S = \Pi$, then p_i simply waits until some process reports that its module of $\mathcal{D}$ outputs a value different from d (line 21).

If $S \neq \Pi$, then p_i first waits until any process observes $r = w(\sigma)$ batches of steps such that, in every batch, every process took at least one new query step in which $\mathcal{D}$ returned d (line 15). If r

Shared variables

Registers: $R[1,\ldots,n+1]$, $D[1,\ldots,n+1]$

$\forall i = 1,\ldots,n+1$, initially $\perp$, written by p_i and read by all

Local variable at every process p_i

$\mathcal{R}_\Upsilon$: Υ^f-$output_i$, initially Π

Boolean: Stable, initially *true*

Code for every process p_i:

Task 1:

```
1   r := 0
2   repeat forever
3     r := r + 1
4     d := query(D_i)
5     R[i] := [d, r]      {Report a new step with failure detector value d}
```

Task 2:

```
6   repeat forever
7     Stable := true
8     Υ^f-output_i := Π
9     d := query(D_i)
10    (S, r) := φ_D(d)      {Compute S = correct(σ) and r = w(σ)}
                            {where σ is defined as in Corollary 9}
11    if S ≠ Π then
12      k := 0
13      repeat
14        k := k + 1
15        wait until every process in Π reports d
                    or ∃j: D[j] = d or some process reports d' ≠ d
16        if some process reported d' ≠ d then Stable := false
17      until k = r or ¬Stable
18      if ¬Stable then proceed to line 7
19      D[i] := d
20      Υ^f-output_i := S
21    wait until some process reports d' ≠ d
```

Fig. 3. Transforming $\mathcal{D}$ into Υ^f.

such batches are observed, then p_i concludes that S cannot be the set of correct processes in the current run, sets Υ^f-$output_i$ to S, and blocks in line 21.

Now we show that, in every run of our algorithm, variables $\{\Upsilon^f\text{-}output_j\}$ satisfy the properties of Υ^f, i.e., there is a time after which the correct processes output the same non-empty set of processes that is not the current set of correct processes.

Consider any run of the reduction algorithm. Let F be the failure pattern and H be the failure detector history in that run. Consider the time after which no process ever observes that a new value other than d is output by $\mathcal{D}$ at any process, i.e., no process ever reports a new step with a value $d' \neq d$. Let $(S, r) = \varphi_{\mathcal{D}}(d)$, $S = correct(\sigma)$, $r = w(\sigma)$, where $\sigma \in (\Pi \times \{d\})^*$, $|correct(\sigma)| \geq n+1-f$, and σ is not an f-resilient sample of $\mathcal{D}$.

If $correct(\sigma) = \Pi$, then every correct process eventually sets Υ^f-$output_i$ to Π in line 8 and blocks forever in line 21. Note that σ is a subsequence of every $\sigma' \in (\Pi \times \{d\})^*$ such that $correct(\sigma') = \Pi$.

Suppose, by contradiction, that $correct(F) = \Pi$. Since d is the failure detector value that every process eventually obtains, there exists $\sigma' \in (\Pi \times \{d\})^*$ such that $correct(\sigma') = \Pi$ and σ' is an f-resilient sample of $\mathcal{D}$. By Lemma 7, σ is also an f-resilient sample of $\mathcal{D}$ — a contradiction. Hence, $correct(F) \neq \Pi$, and Π is a legitimate stable output of Υ^f in this case.

Now assume that $correct(\sigma) \neq \Pi$. Suppose that a correct process is blocked forever in line 15. Thus, eventually, every correct process is also blocked in line 15. Indeed, since we assume that no process observes a failure detector value other than d, no process can proceed to the next round in line 18. Thus, when a correct process p_j exits the wait clause, it sets $D[j] = d$ in line 19 and blocks in line 21. Every correct process blocked in line 15 will eventually find $D[j] = d$ and exit the wait clause.

Hence, we only need to consider two cases: (1) all correct processes are blocked forever in the wait clause in line 15, and (2) all correct processes exit the wait clause in line 15 and block forever in line 21.

In case (1), every correct process p_i sets Υ^f-$output_i$ to Π and blocks forever waiting for some process p_j to take one more step in Task 1. This can only happen if p_j is faulty. Thus, $correct(F) \neq \Pi$, and Π is a legitimate stable output of $\mathcal{D}$.

In case (2), some process previously observed $w(\sigma)$ batches of steps such that, in every batch, every process p_j reported, at least twice, that its module of $\mathcal{D}$ output d by writing new values in register $R[j]$. By our algorithm (Task 1), between these two writing steps, p_j queried $\mathcal{D}$ and obtained d. Thus there exists σ', an f-resilient sample of $\mathcal{D}$, where $correct(\sigma') = correct(F)$, that begins with $w(\sigma)$ batches of the form $(q_1, d), (q_2, d), \ldots, (q_{n+1}, d)$, where $\{q_1, q_2, \ldots, q_{n+1}\} = \Pi$.

Suppose now that $correct(\sigma) = correct(F) = correct(\sigma')$. Note that the prefix of σ of length $w(\sigma)$ is a subsequence of the prefix of σ' of length $w(\sigma)(n+1)$. Since the prefix of σ of length $w(\sigma)$ includes all steps that processes in $\Pi - correct(F)$ take in σ, σ is a subsequence of σ'. By Lemma 7, σ is an f-resilient sample of $\mathcal{D}$ — a contradiction. Thus, $correct(\sigma) \neq correct(F)$, and $correct(\sigma)$ is a legitimate stable output of Υ^f.

Thus, in every run, the correct processes eventually set their emulated outputs of Υ^f to the same non-empty set of at least $n + 1 - f$ processes that is not the set of correct processes in that run — the output of Υ^f is extracted. □

7 Concluding Remarks

We established in this paper that Υ (resp. Υ^f) is weaker than any stable failure detector that circumvents a wait-free (resp. f-resilient) impossibility. Most failure detectors (we are aware of) that have been proposed to capture minimal information to circumvent asynchronous impossibilities in the shared memory model are stable or have stable equivalents [4, 3, 18, 7].

An interesting aspect of our minimality result is that it holds regardless of which shared objects are used to circumvent an impossibility. Indeed, the only fact we use to extract the output of Υ^f is the very impossibility to solve a given problem in a given model. On the other hand, our $\Upsilon(\Upsilon^f)$-based algorithms work in the "weakest" shared memory model where processes communicate through registers.

References

1. Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic snapshots of shared memory. *Journal of the ACM*, 40(4):873–890, 1993.
2. Elizabeth Borowsky and Eli Gafni. Generalized FLP impossibility result for t-resilient asynchronous computations. In *Proceedings of the 25th ACM Symposium on Theory of Computing*, pages 91–100. ACM Press, May 1993.
3. Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722, July 1996.
4. Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.
5. Soma Chaudhuri. More choices allow more faults: Set consensus problems in totally asynchronous systems. *Information and Computation*, 105(1):132–158, 1993.
6. Wei Chen, Jialin Zhang, Yu Chen, and Xuezheng Liu. Weakening failure detectors for k-set agreement via the partition approach. In *Proceedings of the 21st International Symposium on Distributed Computing*, pages 123–138, 2007.
7. Carole Delporte-Gallet, Hugues Fauconnier, Rachid Guerraoui, Vassos Hadzilacos, Petr Koutnetzov, and Sam Toueg. The weakest failure detectors to solve certain fundamental problems in distributed computing. In *Proceedings of the 23th ACM Symposium on Principles of Distributed Computing*, 2004.
8. Carole Delporte-Gallet, Hugues Fauconnier, Rachid Guerraoui, and Petr Kouznetsov. Mutual exclusion in asynchronous systems with failure detectors. *J. Parallel Distrib. Comput.*, 65(4):492–505, 2005.
9. Danny Dolev, Cynthia Dwork, and Larry Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM*, 34(1):77–97, January 1987.
10. Cynthia Dwork, Nancy A. Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, April 1988.
11. Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.
12. Rachid Guerraoui, Maurice Herlihy, Petr Kouznetsov, Nancy A. Lynch, and Calvin C. Newport. On the weakest failure detector ever. In *Proceedings of the 26th ACM Symposium on Principles of Distributed Computing*, pages 235–243, 2007.
13. Rachid Guerraoui and Petr Kouznetsov. Failure detectors as type boosters. *Distributed Computing*, 20(5):343–358, 2008.
14. Maurice Herlihy and Nir Shavit. The asynchronous computability theorem for t-resilient tasks. In *Proceedings of the 25th ACM Symposium on Theory of Computing*, pages 111–120, May 1993.
15. Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.
16. Prasad Jayanti. Robust wait-free hierarchies. *Journal of the ACM*, 44(4):592–614, 1997.
17. Achour Mostéfaoui, Michel Raynal, and Corentin Travers. Exploring Gafni's reduction land: From *omega* to wait-free adaptive (2p-[p/k])-renaming via k-set agreement. In *Proceedings of the 20th International Symposium on Distributed Computing*, pages 1–15, 2006.
18. Gil Neiger. Failure detectors and the wait-free hierarchy. In *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*, 1995.
19. Michel Raynal and Corentin Travers. In search of the holy grail: Looking for the weakest failure detector for wait-free set agreement. In *Proceedings of the 10th International Conference on Principles of Distributed Systems*, pages 3–19, 2006.
20. Michael Saks and Fotios Zaharoglou. Wait-free k-set agreement is impossible: The topology of public knowledge. In *Proceedings of the 25th ACM Symposium on Theory of Computing*, pages 101–110. ACM Press, May 1993.
21. Jiong Yang, Gil Neiger, and Eli Gafni. Structured derivations of consensus algorithms for failure detectors. In *Proceedings of the 17th ACM Symposium on Principles of Distributed Computing*, pages 297–306, 1998.
22. Piotr Zielinski. Automatic classification of eventual failure detectors. In *Proceedings of the 21st International Symposium on Distributed Computing*, pages 465–479, 2007.
23. Piotr Zielinski. Anti-Omega: the weakest failure detector for set agreement. In *Proceedings of the 27th ACM Symposium on Principles of Distributed Computing*, 2008.