On the Weakest Failure Detector Ever

Rachid Guerraoui
School of Computer and Communication Sciences, EPFL, and Computer Science and Artificial Intelligence Laboratory, MIT
rachid.guerraoui@epfl.ch

Maurice Herlihy
Computer Science Department, Brown University
mph@cs.brown.edu

Petr Kouznetsov
Max Planck Institute for Software Systems
pkouznet@mpi-sws.mpg.de

Nancy Lynch
Computer Science and Artificial Intelligence Laboratory, MIT
lynch@theory.csail.mit.edu

Calvin Newport
Computer Science and Artificial Intelligence Laboratory, MIT
cnewport@mit.edu

ABSTRACT

Many problems in distributed computing are impossible 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 *needed* 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 failure information. 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 only excludes one possibility of correct set among many, 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 prove that Υ is, in a precise sense, *minimal* non-trivial failure detector. Roughly, we show that Υ is the weakest *eventually stable* failure detector to circumvent any wait-free impossibility.

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

Categories and Subject Descriptors

C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design—*distributed networks*; C.2.4 [**Computer-Communication Networks**]: Distributed Systems; F.1.1 [**Computation by Abstract Devices**]: Models of Computation—*relations between models*

General Terms

Algorithms, Performance, Theory

Keywords

wait-free impossibilities, failure detectors, set-agreement, weakest failure detector ever

1. INTRODUCTION

Fischer, Lynch, and Paterson's seminal result in the theory of distributed computing [8] says that the seemingly easy *consensus* task (a decision task where a collection of processes start with some input values and need 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 algorithms. Later, three independent groups of researchers [15, 10, 2] extended that result by proving the impossibility of *wait-free set-agreement* [5], a decision task where processes need to agree on up to n input values, in an asynchronous shared

PODC'07, August 12–15, 2007, Portland, Oregon, USA.

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 when f processes can crash.

Asynchrony refers to the absence of timing assumptions on process speeds and communication delays. Some timing assumptions can typically be made in most real distributed systems however. 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, whereas 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 [6], mutual exclusion [7], etc.

This paper asks, for the first time. what information about failures is 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 would derive that T, from the failure detection perspective, is the weakest impossible problem in asynchronous distributed computing.

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

We introduce a *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 be varying 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: it basically only excludes one possibility of failures among many, and it does so only 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 $\{p_1, p_2, p_3\}$.

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. At the heart of our algorithm lies the following idea. We use the information provided by Υ to eventually partition the processes into two subsets: *gladiators* that are eventually permanently output by Υ, and *citizens* that are not. Very roughly speaking, our algorithm has the *gladiators* compete until they (1) eliminate one of their input values, which necessarily happens if one of them fails, or (2) escape from their condition if they see the initial value of a *citizen.* What Υ ensures is precisely that at least one of the *gladiators* fail or at least one of the *citizens* is correct.

We use the very existence of our algorithm to (a) disprove the conjecture of [14] about the weakest failure detector to implement n-resilient n-set agreement and (b) prove 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 show how our algorithm can be adapted to solve 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. To establish this minimality, we use here a restricted variant of the reduction notion of Chandra, Hadzilacos, and Toueg [3]. We show that any oracle that (1) might output any information (within its range) for an arbitrary period of time, and eventually outputs a permanent (stable) value that depends only on which processes failed (not on the computation), as well as (2) helps circumvent *any* 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 proof is very simple: basically, it extracts the minimal information about failures to solve a decision problem, from the impossibility of the problem itself.

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 failure 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, but the proof is significantly simpler than that of [3]: in short, our approach extracts Ω from the fact of consensus impossibility [8] without having to go through valence arguments as in [3]. As pointed out, our result is restricted to failure detectors that are eventually stable and depend only on which processes fail.

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 [15, 10, 2]. In this case, we prove that Υ is strictly weaker than failure detector Ω^n introduced in [13]. 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 [13], and (2) $n+1$-process consensus using n-process consensus [16]. In fact, Ω^n was also shown to be necessary to implement $n+1$-process consensus using n-process consensus [9] and conjectured to be necessary to solve set-agreement [14]. It was our long quest to prove this conjecture that led us identify Υ and devise our set-agreement algorithm based on this oracle.

Roadmap. The rest of the paper is organized as follows. Section 2 gives some basic definitions needed to state and prove our results. Section 3 defines and discusses Υ. Sections 4 describes our set-agreement algorithm using Υ. Section 5 proves the minimality of Υ and Υ^f. Section 6 concludes the paper by discussing open questions.

2. MODEL

Our model of processes communicating through shared objects and using failure detectors is based on [11, 12, 3]. We recall below the details necessary for describing our results.

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*. When presenting our algorithms, we assume that the shared objects are registers, i.e., they export only read-write operations. The impossibility and necessity parts of our results do not restrict the types of shared objects.

Failure patterns and failure detectors. Besides accessing shared objects, processes can also make use of oracles that provide them with information about failures of other processes, i.e., *failure detectors* [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.

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. $A(p_i)$ denotes the automaton on which process p runs the algorithm A. Computation proceeds in atomic *steps* of A. In each step of A, process p

(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 is a *query* step), and

(ii) applies its current state, the response received from the shared object *or* the value output by $\mathcal{D}_i$ to the automaton $A(p)$ to obtain a new state.

A step of A is thus identified by a pair (p_i, x), 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.

Configurations and runs. A *configuration* of A defines the state of each process and each object in the system. In an *initial configuration I* of A, every process p_i is in an initial state of $A(p_i)$ and every object is in an initial state determined by its object type.

A *run of algorithm A using a failure detector $\mathcal{D}$* is a tuple $R = \langle F, H, I, S, T \rangle$ where F is a failure pattern, $H \in \mathcal{D}(F)$ is a failure detector history, I is an initial configuration of A, 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) S respects the specifications of all shared objects and all process automata, given their initial states in I;

(2) For all $k \in \mathbb{N}$, if $S[k] = (p_i, x)$ (x denotes here any legitimate value), then p_i has not crashed by time $T[k]$, i.e., $p_i \notin F(T[k])$;

(3) For all $k \in \mathbb{N}$, if $S[k] = (p_i, x)$ 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., $x = H(p_i, T[k])$;

(4) For all $k, l \in \mathbb{N}$, if $T[k] = T[l]$, then $S[k]$ and $S[l]$ are steps of different processes.

(5) Every correct process (in F) takes infinitely many steps in S.

A *partial run* of an algorithm A is a finite prefix of a run of A.[1]

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

Comparing failure detectors. If, for failure detectors $\mathcal{D}$ and $\mathcal{D}'$, there is an algorithm $T_{\mathcal{D}' \to \mathcal{D}}$ using $\mathcal{D}'$ that *extracts* the output of $\mathcal{D}$, i.e. implements a distributed variable $\mathcal{D}$-*output* such that in every run $R = \langle F, H', I, S, T\rangle$ of $T_{\mathcal{D}' \to \mathcal{D}}$, 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.

3. 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 [13], 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 our default 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 T 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 T in which the extracted failure detector output never stabilizes.

We consider the set of runs of T in which Υ permanently outputs $\{p_1, \dots, p_n\}$ 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 $\{p_1, \dots, p_n\}$.

Consider partial runs of T 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 $\{p_1, \dots, p_n\}$ at all processes and T outputs a process $p_{i_1} \in \{p_1, \dots, p_n\}$ 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 $\{p_1, \dots, p_n\}$ is faulty, and thus it is still legitimate for Υ to always output $\{p_1, \dots, p_n\}$. Thus, there exists a failure-free partial run R_2 extending R_1 in which Υ always outputs $\{p_1, \dots, p_n\}$ at all processes and T 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 $\{p_1, \dots, p_n\}$ at all processes and T 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 T in which Υ always outputs $\{p_1, \dots, p_n\}$ at all processes, but the extracted failure detector output never stabilizes — a contradiction. □

4. SET-AGREEMENT

4.1 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 $\bot \notin V$. 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 [15, 10, 2].

We show how to circumvent this impossibility using Υ: we describe a protocol that solves n-set-agreement using regis-

[1] A more formal definition of a run of an algorithm using a failure detector can be found in [3, 9].

ters 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 [16]. A process calls *k-converge* with an input value in V and gets back an output value 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. The *k-converge* routine can be implemented, for any k, using registers [16]. By definition, 0-*converge*(v) always returns $(v, false)$.

4.2 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.

```
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     $(v_i, c) := n\text{-}converge[r](v_i)$
5     if $c = true$ then
6       $D := v_i$; return $(v_i)$
7     $U :=$ query$(\Upsilon_i)$
8     if $p_i \notin U$ then
9       $D[r] := v_i$
10    else
11      $k := 0$
12      repeat
13        $k := k + 1$
14        $(v_i, c) := (|U| - 1)\text{-}converge[r][k](v_i)$
15        if $c = true$ then $D[r] := v_i$
16        if $U \neq query(\Upsilon_i)$ then $Stable[r] := false$
17      until $D \neq \perp$ or $D[r] \neq \perp$ or $\neg Stable[r]$
18      if $D[r] \neq \perp$ then
19        $v_i := D[r]$
20  until $D \neq \perp$
21  return $(D)$
```

Figure 1: Υ-based set agreement protocol.

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 $|U|-1$ 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*$[r]$ adopted 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) $C \subsetneq U$, 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. By the C-Agreement and C-Validity properties of $(|U| - 1)$-convergence and because every value returned by an instance of $(|U|-1)$-*converge*$[r][k]$ is adopted (line 14), at most $|U|-1$ distinct values can be written in $D[r]$ by processes in U. Thus, 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 [14] 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 [9], we get the following:

Corollary 4. *For all $n \geq 2$, implementing n-resilient n-set-agreement among $n + 1$ processes using registers is strictly easier than implementing $n + 1$-process consensus using n-process consensus.*

4.3 f-Resilient Set-Agreement

For pedagogical purposes, we focused so far on the environment where n out of $n + 1$ processes can crash. 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 *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 (a simple variation of the consensus algorithm in [13] will work). 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 that there exists an algorithm T 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 T in which the extracted output of never stabilizes.

We consider the set of runs of T 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 T in which no process fails and Υ^f always outputs U. Let L_1 be the set output by T 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 T 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 T outputs a set $L_3 \neq L_2$ at some process after R_2.

Following this procedure, we obtain a failure-free run of T in 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] which ensures that, whenever U contains at least one faulty processes, at most $|U| + f - n - 1$ values are eventually chosen by the members of U.

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 proved 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 17–17). Suppose, by contradiction, that there is a run R of our algorithm in which some correct process never decides. Using the arguments presented in the proof of Theorem 2, we can show that D always contains $\bot$. Further, 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.

Suppose, by contradiction, that some correct process is blocked in the loop of lines 17–19, while executing sub-round k of round r. It is easy to see that, if a correct process exits the loop, then eventually every correct process is freed too. Thus, our assumption implies that every correct process is blocked in the loop of lines 17–19, while executing sub-round k of round r.

Hence, $\Pi - U$ contains no correct process: otherwise, some correct process in $\Pi - U$ would eventually write a non-$\bot$ 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-$\bot$ 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-$\bot$ entries and, thus, the condition in line 19 is eventually satisfied — a contradiction.

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

```
Shared abstractions:
    Registers $D$, $D[\,]$, initially $\bot$
    Vectors of registers $A[\,][\,]$, initially $\bot$
    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$(\Upsilon_i)$
10       if $p_i \notin U$ then
11          $D[r] := v_i$
12       else
13          $k := 0$
14          repeat
15             $k := k + 1$
16             $A[r][k][i] := v_i$
17             repeat
18                $V :=$ atomic-snapshot$(A[r][k])$
19             until $D \neq \bot$ or $D[r] \neq \bot$ or $\neg Stable[r]$ or
                        $V$ contains $\geq n + 1 - f$ non-$\bot$ entries
20             if $D \neq \bot$ then
21                return$(D)$
22             else if $D[r] \neq \bot$ then
23                $v_i := D[r]$
24             else if $Stable[r]$ then
25                $v_i :=$ min non-$\bot$ 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 $U \neq$ query$(\Upsilon_i)$ then
30                   $Stable[r] := false$
31          until $D \neq \bot$ or $D[r] \neq \bot$ or $\neg Stable[r]$
32          if $D[r] \neq \bot$ then
33             $v_i := D[r]$
34    until $D \neq \bot$
35    return $(D)$
```

Figure 2: Υ^f-based f-resilient f-set agreement protocol.

Note that at most $n + 1 - |U|$ different non-$\bot$ 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 $D[r] \neq \bot$ 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 $D[r] = \bot$ 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 *atomic-snapshot*($A[r][k]$) in line 18. 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. □

5. THE NECESSITY OF Υ^F

We establish here that Υ^f is, in a certain sense, minimal in systems where up to f processes can crash, implying that Υ is also minimal when up to n processes can crash.

We 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*. 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 $\mathcal{E}^f$.

Establishing our minimality result goes through delimiting the scope of failure detectors within which Υ^f is minimal. We say that a failure detector, $\mathcal{D}$ with range $\mathcal{R}_\mathcal{D}$, is *eventually stable* if it satisfies the following properties:

- 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).[2]

- The stable output of $\mathcal{D}$ depends only on the set of correct processes. Formally, for all failure patterns F and $H \in \mathcal{D}(F)$, if d is stable in H, then for all F' such that *correct*(F') = *correct*(F), there exists $H' \in \mathcal{D}(F')$ such that d is also stable in H'.

- $\mathcal{D}$ is allowed to output any value in its range in every finite prefix of every history of $\mathcal{D}$. Formally, for all failure patterns F, histories $H \in \mathcal{D}(F)$, values $d \in \mathcal{R}_\mathcal{D}$ and times $t \in \mathbb{T}$, there exists $H' \in \mathcal{D}(F)$, such that, for all $p_i \in \Pi$ and $t' \in \mathbb{T}$, $H'(p_i, t') = d$ if $t' \leq t$ and $H'(p_i, t') = H(p_i, t)$.

Theorem 7. *Υ^f is weaker than any f-non-trivial eventually stable failure detector.*

Proof. Let $\mathcal{D}$ be any eventually stable failure detector that can be used to solve an f-resilient impossible problem. Let $\mathcal{R}_\mathcal{D}$ be the range of $\mathcal{D}$, and let d be any value in $\mathcal{R}_\mathcal{D}$.

First we show that there exists a set of processes $U \in \mathcal{R}_{\Upsilon^f}$ (i.e., $|U| \geq n+1-f$) such that for all failure patterns F where *correct*$(F) = U$ and all histories $H \in \mathcal{D}(F)$, d is not the stable value in H.

Suppose not, i.e., there exists a value $d \in \mathcal{R}_\mathcal{D}$ such that, for all $U \in \mathcal{R}_{\Upsilon^f}$, there exists a failure pattern F and a history $H \in \mathcal{D}(F)$ such that *correct*$(F) = U$ and d is the stable value of H. Since $\mathcal{D}$ is stable, for every F, $\mathcal{D}(F)$ contains a history in which d is always output at every process.

But then every history of a dummy failure detector that always outputs d is a history of $\mathcal{D}$, i.e., the dummy failure detector can implement $\mathcal{D}$. This contradicts the assumption that $\mathcal{D}$ is f-non-trivial.

Thus, for any $d \in \mathcal{R}_\mathcal{D}$, there exists $U \in \mathcal{R}_{\Upsilon^f}$ such that d cannot be the stable value whenever U is the set of correct processes. Since the elements of $\mathcal{R}_{\Upsilon^f}$ can be totally ordered, we can define a function σ that maps every $d \in \mathcal{R}_\mathcal{D}$ to the smallest $U \in \mathcal{R}_{\Upsilon^f}$ such that whenever d is stable, U that cannot be the current set of correct processes.

The reduction algorithm $T_{\mathcal{D}\rightarrow\Upsilon^f}$ works as follows. Every process periodically queries its module of $\mathcal{D}$ and for every returned value d outputs $U = \sigma(d)$. In every run of $T_{\mathcal{D}\rightarrow\Upsilon^f}$, the produced output eventually stabilizes at a set $U \in \mathcal{R}_{\Upsilon^f}$, that is not the set of correct processes. That is, the output of Υ^f is extracted. □

6. CONCLUDING REMARKS

We stated in this paper that Υ (resp. Υ^f) is weaker than any eventually stable failure detector that circumvents a wait-free (resp. f-resilient) impossibility.

Generalizing this result by determining the weakest non-trivial among a wider class of failure detectors is left for future research. Nevertheless, and interestingly, 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 eventually stable or have eventually stable equivalents [4, 3, 13, 9].

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.

7. REFERENCES

[1] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit. Atomic snapshots of shared memory. *J. ACM*, 40(4):873–890, 1993.

[2] E. Borowsky and E. Gafni. Generalized FLP impossibility result for t-resilient asynchronous computations. In *Proceedings of the 25th ACM Symposium on Theory of Computing*, pages 91–100, May 1993.

[3] T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *J. ACM*, 43(4):685–722, July 1996.

[2] 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).

[4] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, Mar. 1996.

[5] S. Chaudhuri. More choices allow more faults: Set consensus problems in totally asynchronous systems. *Information and Computation*, 105(1):132–158, 1993.

[6] C. Delporte-Gallet, H. Fauconnier, R. Guerraoui, V. Hadzilacos, P. Kouznetsov, and S. Toueg. The weakest failure detectors to solve certain fundamental problems in distributed computing. In *PODC*, pages 338–346, 2004.

[7] C. Delporte-Gallet, H. Fauconnier, R. Guerraoui, and P. Kouznetsov. Mutual exclusion in asynchronous systems with failure detectors. *J. Parallel Distrib. Comput.*, 65(4):492–505, 2005.

[8] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, Apr. 1985.

[9] R. Guerraoui and P. Kouznetsov. On failure detectors and type boosters. In *Proceedings of the 17th International Symposium on Distributed Computing*, pages 292–305. Springer-Verlag, 2003.

[10] M. Herlihy and N. 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.

[11] M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[12] P. Jayanti. Robust wait-free hierarchies. *Journal of the ACM*, 44(4):592–614, 1997.

[13] G. Neiger. Failure detectors and the wait-free hierarchy. In *14th ACM Symposium on Principles of Distributed Computing*, 1995.

[14] M. Raynal and C. Travers. In search of the holy grail: Looking for the weakest failure detector for wait-free set agreement. In *OPODIS*, pages 3–19, 2006.

[15] M. Saks and F. Zaharoglou. Wait-free k-set agreement is impossible: The topology of public knowledge. In *Proceedings of the Twenty fifth ACM Symposium on Theory of Computing*, pages 101–110, May 1993.

[16] J. Yang, G. Neiger, and E. 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.