The Weakest Failure Detector for Solving k-Set Agreement *

Eli Gafni† Petr Kuznetsov‡

Abstract

A failure detector is a distributed oracle that provides processes in a distributed system with hints about failures. The notion of a weakest failure detector captures the exact amount of synchrony needed for solving a given distributed computing problem.

In this paper, we determine the weakest failure detector for solving k-set agreement among n processes ($n > k$) using reads and writes in shared memory, regardless of the assumptions on when and where failures might occur. This failure detector is derived directly from the impossibility of wait-free $k+1$-process k-set agreement. Our approach can be viewed as an extension of the asynchronous BG-simulation technique to partially synchronous systems.

Keywords: k-set agreement, synchrony assumptions, failure detectors, BG-simulation

1 Introduction

The amount of synchrony is a crucial factor in reasoning about solvability of distributed computing problems in the presence of faults. A *synchronous* system, in which communication delays and relative processing speeds are bounded, and the bounds are known a priori, provides a fault-tolerant solution to almost any meaningful synchronization problem. On the other hand, in an *asynchronous* system, in which no synchrony assumptions can be made, even a basic form of non-trivial synchronization (*consensus*) is impossible if only one process may fail by crashing [12, 20].

Chandra and Toueg proposed *failure detectors* as a convenient and flexible language to describe synchrony assumptions in the presence of faults [7]. Informally, a failure detector is a distributed oracle that provides processes with hints about failures. The notion of a *weakest failure detector* captures the exact amount of synchrony needed for solving a given distributed computing problem [6]. $\mathcal{D}$ is the weakest failure detector for solving a problem $\mathcal{M}$ if $\mathcal{D}$ is both (1) *sufficient* to solve $\mathcal{M}$, i.e., there exists an algorithm that solves $\mathcal{M}$ using $\mathcal{D}$, and (2) *necessary* to solve $\mathcal{M}$, i.e., any failure detector that is sufficient to solve $\mathcal{M}$ provides at least as much information about failures as $\mathcal{D}$ does.

*The conference version of these results (PODC 2009) contains slight technical inconsistencies corrected in this technical report.

†Computer Science Department, University of California, Los Angeles. 3731F Boelter Hall, UCLA, LA. CA. 90095, USA, e-mail: eli@ucla.edu

‡TU Berlin/Deutsche Telekom Laboratories, Sekr. TEL 16, Ernst-Reuter-Platz 7, 10587 Berlin, e-mail: pkuznets@acm.org (contact author)

1.1 Background

In this paper, we address the weakest failure detector question in the following context. We consider a distributed system in which n crash-prone processes communicate using atomic reads and writes in shared memory. We focus on a class of distributed computing problems, called *tasks*, which are defined exclusively through processes' inputs and outputs. One example of a distributed task is binary consensus [12]: each process starts with a binary input and every correct (never-failing) process is supposed to output one of the inputs such that no two processes output different values. Chandra et al. [6] showed that the "eventual leader" failure detector Ω is the weakest failure detector to solve consensus. When queried, Ω outputs a process identifier, such that, eventually, the same *correct* (never failing) process identifier is always output at all processes.

In a generalization of consensus, k-set agreement [8], processes start with values in $\{0, \ldots, k\}$ and the set of outputs must be a subset of inputs of size at most k. In case $k = 1$, k-set agreement coincides with consensus. It has been established that $k+1$-process k-set agreement (also written as $(k+1, k)$-set agreement) is impossible to solve in a *wait-free* manner (tolerating up to k faulty processes) [16, 23, 3]. Moreover, for all $n \geq k+1$, there is no n-process algorithm that solves k-set agreement and tolerates k faulty processes [3].

Zieliński [24] determined the weakest failure detector for solving n-process $(n-1)$-set agreement. This failure detector (called anti-Ω) outputs, when queried, a process identifier and guarantees that eventually some correct process is *never* output.

Thus, while the question of the weakest failure-detector for the extremes, 1- and $(n-1)$-set agreements has been resolved, the general question of the weakest failure detector to solve k-set agreement for $1 < k < n-1$ has remained open until now.

Raynal [21] conjectured that a generalization of anti-Ω is the weakest failure-detector for n-process k-set agreement. This failure detector, denoted k-anti-Ω (or $\neg\Omega_k$ for short), outputs, when queried, a set of $n-k$ processes such that, eventually, at least one correct process is never output.

1.2 Contribution

This paper proves Raynal's conjecture. We show that for all $k = 1, \ldots, n-1$, $\neg\Omega_k$ is indeed the weakest failure detector to solve k-set agreement. The statement holds for all *environments*, i.e., for all possible assumptions on when and where failures may occur.

On the technical side, our proof that $\neg\Omega_k$ is necessary for solving k-set agreement extends the classical construction of Chandra et al. [6]. We use two key observations. First, every run of an algorithm $\mathcal{A}$ using failure detector $\mathcal{D}$ induces a directed acyclic graph (DAG) that samples the output of $\mathcal{D}$ in that run [6]. Second, this DAG can then be used for an *asynchronous* simulation of partial runs of $\mathcal{A}$ [24]. Our new insight here is that a k-resilient run of this asynchronous simulation can, in turn, be simulated, in a wait-free manner, by $k+1$ processes by applying the BG-simulation technique [3, 5]. Assuming that $\mathcal{A}$ uses $\mathcal{D}$ to solve k-set agreement, we apply the technique to eventually identify a "never-deciding" k-resilient run of the asynchronous simulation of $\mathcal{A}$. To emulate $\neg\Omega_k$, it is sufficient to output $n-k$ processes that appear the most in that run. At least one correct process is eventually never output: otherwise, the run would involve infinitely many steps of every correct process, and therefore would be deciding.

The argument above is a natural generalization of earlier results that derive a necessary failure detector for solving a wait-free impossible problem from the very fact that the problem is wait-free impossible, without explicitly using the problem semantics [13, 24]. The important difference is

that in our case, the original task (k-set agreement) is impossible to solve in the *k-resilient* way, and to reduce the derivation of the corresponding weakest failure detector to the wait-free impossibility of $(k+1)$-process k-set agreement we extend the wait-free BG-simulation technique to the world of failure detectors. The extension involves a novel technique of *corridor-based* depth-first simulation and is interesting in its own right.

As a byproduct, this paper gives an alternative (and seemingly simpler than in [6]) proof that Ω is the weakest failure detector for solving consensus in the read-write shared memory model.[1]

1.3 Roadmap

The rest of the paper is organized as follows. Section 2 describes our system model. Section 3 formally defines $\neg\Omega_k$ and shows that $\neg\Omega_k$ is sufficient to solve n-process k-set agreement. Sections 4 proves that $\neg\Omega_k$ is necessary to solve n-process k-set agreement task. Section 5 overviews the related work and Section 6 concludes the paper by discussing implications of our results.

2 Model

Our model in which a collection of processes communicate through read-write shared objects and use failure detectors is based on [6, 13, 15]. We describe below the details necessary for showing our results.

2.1 Processes and objects

We consider a distributed system composed of a set Π of n processes $\{p_1, \ldots, p_n\}$ $(n \geq 2)$. Processes are subject to *crash* failures. A process that never fails is said to be *correct*. Processes that are not correct are called *faulty*. 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 atomic read-write operations. The necessary parts of our results do not restrict the types of shared objects, but, for simplicity of presentation, we assume that the objects are *deterministic*, i.e., the value returned by an object is solely determined be the object's state and the applied operation.

2.2 Failure patterns and failure detectors

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. Respectively, $\mathit{correct}(F) = \Pi - \mathit{faulty}(F)$. A process $p \in F(t)$ is said to be *crashed* at time t. An *environment* is a set of failure patterns. By default, we assume that at least one process is correct in every failure pattern.

A *failure detector history H with range $\mathcal{R}$* is a function from $\Pi \times \mathbb{T}$ to $\mathcal{R}$. $H(p_i, t)$ is interpreted as the value output by the failure detector module of process p_i at time t. A *failure detector $\mathcal{D}$* with range $\mathcal{R}_{\mathcal{D}}$ is a function that maps each failure pattern to a (non-empty) set of failure detector

[1] A self-contained version of this result that explicitly uses consensus instances in the extraction algorithm can be found in [18].

histories with range $\mathcal{R}_\mathcal{D}$. $\mathcal{D}(F)$ denotes the set of possible failure detector histories permitted by $\mathcal{D}$ for failure pattern F. We do not restrict possible ranges of failure detectors.

2.3 Algorithms

We define an *algorithm $\mathcal{A}$ using a failure detector $\mathcal{D}$* as a collection of deterministic automata, one for each process in the system. $\mathcal{A}_i$ denotes the automaton on which process p_i runs the algorithm $\mathcal{A}$. Computation proceeds in atomic *steps* of $\mathcal{A}$. In each step of $\mathcal{A}$, process p_i

(i) invokes an atomic 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 $\mathcal{D}$, and

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

A step of $\mathcal{A}$ is thus identified by a tuple (p_i, d), where d is the failure detector value output at p_i during that step if $\mathcal{D}$ was queried, and $\perp$ otherwise.

If the state transitions of all automata $\mathcal{A}_i$ do not depend on the failure detector values, we say that the algorithm $\mathcal{A}$ is *asynchronous*. Thus, for an asynchronous algorithm, a step is uniquely identified by the process id.

2.4 Runs

A *state* of $\mathcal{A}$ defines the state of each process and each object in the system. An *initial state* I of $\mathcal{A}$ specifies an initial state for every automaton $\mathcal{A}_i$ and every shared object.

A *run of algorithm $\mathcal{A}$ using a failure detector $\mathcal{D}$* in an environment $\mathcal{E}$ is a tuple $R = \langle F, H, I, S, T\rangle$ where $F \in \mathcal{E}$ is a failure pattern, $H \in \mathcal{D}(F)$ is a failure detector history, I is an initial state of $\mathcal{A}$, S is an *infinite* sequence of steps of $\mathcal{A}$ respecting the automata $\mathcal{A}$ and the sequential specification of shared objects, and T is an *infinite* list of increasing time values indicating when each step of S has occurred, such that for all $k \in \mathbb{N}$, if $S[k] = (p_i, d)$ with $d \neq \perp$, then $p_i \notin F(T[k])$ and $d = H(p_i, T[k])$.

Let $inf(R)$ denote the set of processes that appear infinitely often in S. We say that a run $R = \langle F, H, I, S, T\rangle$ is *fair* if $correct(F) = inf(R)$, and *k-resilient* if $|inf(R)| \geq n - k$. A *partial run* of an algorithm $\mathcal{A}$ is a finite prefix of a run of $\mathcal{A}$: a tuple $\langle F, H, I, S', T'\rangle$ where S' and T' are finite prefixes of the same length of some infinite sequences S and T such that $\langle F, H, I, S, T\rangle$ is a run of $\mathcal{A}$.

Two (partial) runs of $\mathcal{A}$ that agree on the initial state I and the sequence of steps S are indistinguishable to the processes. Therefore, in our reduction algorithm, a run is understood as an equivalence class of indistinguishable runs that agree on I and S.

2.5 Distributed tasks

A *task* is defined through a set $\mathcal{I}$ of input n-vectors (one input value for each process), a set $\mathcal{O}$ of output n-vectors (one input value for each process) and a total relation Δ that associates each input vector with a set of possible output vectors. An output may be $\perp$, which models *undecided* processes. We stipulate that if $(I, O) \in \Delta$, then, for each O' resulting after replacing some items in O with $\perp$, $(I, O') \in \Delta$. That is, the safety property of a task implementation depends only on

non-$\perp$ output values. An algorithm $\mathcal{A}$ solves a task $\mathcal{M} = (\mathcal{I}, \mathcal{O}, \Delta)$ if in each fair run of $\mathcal{A}$ with input vector I, every correct process decides on an output and the vector O of outputs satisfies $(I, O) \in \Delta$.

In the *n-process k-set agreement* task, each process takes a value in $\{0, \ldots, k\}$ as an input, and the set of non-$\perp$ output values is a subset of the input values of size at most k. Note that to solve k-set agreement it is sufficient to ensure that, in every fair run, at least one process decides (and writes the decided value in the shared memory). A (partial) run of a set agreement algorithm in which at least one process decides is thus called *deciding*.

2.6 Comparing failure detectors

We say that an algorithm $\mathcal{A}$ using $\mathcal{D}'$ *extracts the output of* $\mathcal{D}$ *in an environment* $\mathcal{E}$, if $\mathcal{A}$ implements a distributed variable $\mathcal{D}$-*output* such that for every run $R = \langle F, H', S, T\rangle$ of $\mathcal{A}$, where $F \in \mathcal{E}$, there exists $H \in \mathcal{D}(F)$ such that for all $p_i \in \Pi$ and $t \in \mathbb{T}$, $\mathcal{D}\text{-}output_i(t) = H(p_i, t)$ (i.e., the value of $\mathcal{D}$-*output* at p_i at time t is $H(p_i, t)$). We call $\mathcal{A}$ a *reduction* algorithm.[2]

If, for failure detectors $\mathcal{D}$ and $\mathcal{D}'$ and an environment $\mathcal{E}$, there is a reduction algorithm using $\mathcal{D}'$ that extracts the output $\mathcal{D}$ in $\mathcal{E}$, then we say that $\mathcal{D}$ *is weaker than* $\mathcal{D}'$ in $\mathcal{E}$. If $\mathcal{D}$ and $\mathcal{D}'$ are weaker than each other in $\mathcal{E}$, we say they are equivalent in $\mathcal{E}$.

$\mathcal{D}$ is the *weakest failure detector* to solve a task $\mathcal{M}$ in $\mathcal{E}$ if (i) there is an algorithm that solves $\mathcal{M}$ using $\mathcal{D}$ in $\mathcal{E}$ and (ii) $\mathcal{D}$ is weaker than any failure detector that can be used to solve $\mathcal{M}$ in $\mathcal{E}$. Every task can be shown to have a weakest failure detector [17].

3 The candidate failure detector

The failure detector $\neg\Omega_k$ outputs, at each process and each time, a set of $n - k$ processes. $\neg\Omega_k$ guarantees that there is a time after which some correct is never output:

$$\forall F,\ \forall H \in \neg\Omega_k(F),\ \exists p_i \in \mathit{correct}(F),\ t \in \mathbb{T},\\ \forall t' > t, \forall p_j \in \Pi : p_i \notin H(p_j, t').$$

By definition, $\neg\Omega_{n-1}$ is equivalent to anti-Ω [24]. Also, $\neg\Omega_1$ is equivalent to Ω [6]. To see this, we can simply output the complement of $\neg\Omega_1$ in Π: eventually, the same correct process will always be output at all processes.

We also consider the following "vector-Ω" failure detector, denoted $\overrightarrow{\Omega}_k$. This failure detector outputs a k-vector of process ids and guarantees that, eventually, at least one position in the vector stabilizes, at all processes, on the same correct process id:

$$\forall F,\ \forall H \in \overrightarrow{\Omega}_k(F),\ \exists p_i \in \mathit{correct}(F),\ \ell \in \{1, \ldots, k\},\ t \in \mathbb{T},\\ \forall t' > t, \forall p_j \in \Pi : H(p_j, t')[\ell] = p_i.$$

As conjectured in [24], $\neg\Omega_k$ and $\overrightarrow{\Omega}_k$ are equivalent. To obtain $\neg\Omega_k$ from $\overrightarrow{\Omega}_k$, it is sufficient to output, at every process, any set of $n - k$ processes that are not output by $\overrightarrow{\Omega}_k$. Eventually, some correct process will never be output at any process. The other direction is a simple generalization

[2]The notion of failure detector reduction (introduced in [6]) has been improved in [17] (in particular, by making it reflexive), but the difference between this notions of [6] and [17] does not affect our results.

of the reduction algorithm for the case $k = n - 1$ in [24] (similar, in turn, to the reduction of [10]) and is presented in the Appendix.

Solving k-set agreement with $\overrightarrow{\Omega}_k$ is straightforward [24]. Just run k instances of Ω-based consensus protocol [19], $C_1, \ldots, C_k$, where each C_ℓ uses position ℓ in the output of $\overrightarrow{\Omega}_k$. As an input in every instance of consensus, each process uses its input value for k-set agreement. The first value to be returned by an instance of consensus is used as the output for k-set agreement. By the agreement property of consensus, at most k distinct values can be output. Since, in at least one position, the output of $\overrightarrow{\Omega}_k$ stabilizes on the same correct process, at least one instance of consensus eventually returns at every process, and there are at most k different values can be returned. Thus:

Theorem 1 *$\neg\Omega_k$ is sufficient to solve k-set agreement in all environments.*

4 Necessity

Now we show that $\neg\Omega_k$ is not only sufficient but also necessary to solve k-set agreement. Let $\mathcal{A}$ be an algorithm that solves k-set agreement using $\mathcal{D}$. Our goal is to construct a reduction algorithm that extracts the output of $\neg\Omega_k$ using $\mathcal{D}$. Recall that to extract the output of $\neg\Omega_k$ means to output, at each time and at each process, a set of $n-k$ process identifiers and ensure that, eventually, some correct process is never included in the output sets.

Our reduction algorithm uses the observation that a run of any $\mathcal{D}$-based algorithm induces a *directed acyclic graph* (DAG). The DAG contains a sample of failure detector values output by $\mathcal{D}$ in the current run and captures some causal relations between them [6]. Given such a DAG G, we can construct an *asynchronous* algorithm $\mathcal{A}'$ that, instead of the "real" failure detector $\mathcal{D}$, uses G to simulate (possibly finite and unfair) runs of $\mathcal{A}$ [24]. Using BG-simulation [3, 5], $k+1$ processes can simulate k-resilient runs of $\mathcal{A}'$. The fact that $(k+1, k)$-set agreement is wait-free impossible implies that the simulation must produce at least one infinite "non-deciding" k-resilient run of $\mathcal{A}'$. This k-resilient run can be then used to extract the output of $\neg\Omega_k$: it is sufficient to output the set of $n-k$ processes that appear infinitely often in it. At least one correct process will eventually never be output, because, otherwise, the run would simulate a fair and thus deciding run of $\mathcal{A}$.

There are two issues we have to address here. One difficulty is how to make sure that eventually each correct process eventually forever selects ever-increasing non-deciding partial runs that are in a strict sense *close* to such an infinite non-deciding run. E.g., by ordering simulated runs lexicographically, and choosing the first non-deciding one, we cannot prevent the case of (temporarily) choosing a prefix of a deciding run that contains steps of arbitrary processes. We address the issue by employing the novel corridor-based depth-first ordering technique explained later in this section.

The second issue is that the DAGs constructed at different processes evolve in different ways. Thus, the non-deciding k-resilient runs located at different correct processes can be different. We resolve the issue by making sure that the correct processes eventually adopt the most "successful" simulation: whenever a process p_i observes that the "smallest" non-deciding simulated run is considered deciding by another process p_j, p_i adopts the set of simulated runs of p_j, and continues the simulation from there.

Our reduction algorithm consists therefore of two components that are running in parallel: the *communication component* and the *computation component*. In the communication component, every process p_i maintains the ever-growing directed acyclic graph (DAG) G_i by periodically querying its failure detector module and exchanging the results with the others through the shared memory.

Shared variables:
  for all $p_i \in \Pi$: V_i, initially $(\perp, \perp, \perp)$

1 $k_i := 0$
2 **while** *true* **do**
3 **for all** $p_j \neq p_i$ **do** $(G_j, \alpha_j, \beta_j) := V_j$; $G_i := G_i \cup G_j$
4 $d_i :=$ query failure detector $\mathcal{D}$
5 $k_i := k_i + 1$
6 add vertex $[p_i, d_i, k_i]$ to G_i
  for each vertex v of G_i, $v \neq [p_i, d_i, k_i]$:
    add edge $(v, [p_i, d_i, k_i])$ to G_i
7 $V_i := (G_i, \alpha_i, \beta_i)$

Figure 1: Building a DAG: the code for each process p_i

In the computation component, every process simulates a set of runs of $\mathcal{A}$ using the DAG and extracts the output of $\neg\Omega_k$.

4.1 DAGs

The communication component is presented in Figure 1. The component maintains, for each process p_i, an ever-growing DAG G_i that contains a sample of the current failure detector history. The DAG is stored in a register V_i which can be written by p_i and read by all processes.

In addition, V_i stores two elements, the set α_i of runs simulated by p_i so far and the *delay* map β_i, that specify the details of how exactly these runs were simulated by p_i. We explain how these mappings are maintained and used in Sections 4.2 and 4.5.

DAG G_i has some special properties which follow from its construction [6]. Let F be the current failure pattern, and $H \in \mathcal{D}(F)$ be the current failure detector history. Then for any correct process p_i and any time t a fair run of the algorithm in Figure 1 guarantees that (here $G_i(t)$ denotes the value of G_i at time t):

(1) The vertices of $G_i(t)$ are of the form $[p_j, d, \ell]$ where $p_j \in \Pi$, $d \in \mathcal{R}_\mathcal{D}$ and $\ell \in \mathbb{N}$. There is a map $\tau:$ vertices of $G_i(t) \mapsto \mathbb{T}$, such that:
  (a) For any vertex $v = [p_j, d, \ell]$, $p_j \notin F(\tau(v))$ and $d = H(p_j, \tau(v))$.
  (b) For any edge (v, v'), $\tau(v) < \tau(v')$.

(2) If $v' = [p_j, d, \ell]$ and $v'' = [p_j, d', \ell']$ are vertices of $G_i(t)$ and $\ell < \ell'$ then (v, v') is an edge of $G_i(t)$.

(3) $G_i(t)$ is transitively closed: if (v, v') and (v', v'') are edges of $G_i(t)$, then (v, v'') is also an edge of $G_i(t)$.

(4) For all correct processes p_j, there is a time $t' \geq t$, a $d \in \mathcal{R}_\mathcal{D}$ and an $\ell \in \mathbb{N}$ such that, for every vertex v of $G_i(t)$, $(v, [p_j, d, \ell])$ is an edge of $G_i(t')$, and $G_i(t) \subseteq G_j(t')$.

In a fair run, the ever-growing DAGs at correct processes tend to the same *limit* infinite DAG $\bar{G} = \cup_{t \in \mathbb{T}} G_i(t)$, and the set of processes that obtain infinitely many vertices in $\bar{G}$ is the set of correct processes [6]. A *subDAG* of $\bar{G}$ is any DAG that consists of a finite subset of vertices of $\bar{G}$ with the corresponding edges. Trivially, each subDAG satisfies properties (1)–(3) above.

4.2 Asynchronous simulation of $\mathcal{A}$

Let G be a DAG constructed as shown in Figure 1. Let β be any mapping from $\Pi \times \mathbb{N}$ to $\mathbb{N}$ such that $\beta(p_i, \ell) = 0$ if $\ell > 1$ and the latest vertex of p_i in G (if any) has the form $[p_i, d, \ell']$ where $\ell' < \ell - 1$. of the form $[p_i, d, \ell]$. We call β a *delay* map for G.

We show that G and β can be used to construct an *asynchronous* algorithm $\mathcal{A}^\beta$ that, for each input vector I simulates a run of $\mathcal{A}$ (Figure 2). In the algorithm, each process p_i starts with its input value in I and performs a sequence of simulated steps of $\mathcal{A}$. Each simulated step of $\mathcal{A}$ is associated with a vertex in G.

To perform the next step of $\mathcal{A}$, p_i first scans the shared memory (line 9 in Figure 2) to get the list of vertices associated with the latest simulated steps of $\mathcal{A}$ performed by other processes (every simulated step is registered in the shared memory). Then p_i chooses the *earliest* vertex $[p_i, d, \ell]$ of G such that all simulated steps of $\mathcal{A}$ currently observed by p_i are associated with vertices of G that precede $[p_i, d, \ell]$ in G. Then p_i takes the next step specified by the automaton $\mathcal{A}_i$. In case the next step is a query step, p_i uses d as the corresponding failure detector value. Thus, instead of querying $\mathcal{D}$, processes use the sample of $\mathcal{D}$'s output contained in G.

To locate ℓ-th vertex of p_i in G, the simulation first takes $\beta(p_i, \ell)$ (in case $\beta(p_i, \ell) > 0$) "waiting" rounds (lines 13–16 in Figure 2). If $\beta(p_i, \ell) = 0$ (which means that ℓ-th vertex of p_i has not yet been used in the simulation), then p_i waits until the vertex arrives (through the concurrently run algorithm in Figure 1). If such a vertex never appears in G, then p_i waits forever, and therefore appears no more in the currently simulated run of $\mathcal{A}$. If the vertex arrives after r waiting rounds, then $\beta(p_i, \ell)$ is set to r.

Intuitively, memorizing the number of waiting rounds for every considered vertex in the delay map β ensures that a given *finite* run of $\mathcal{A}^\beta$ repeated multiple times will simulate the same run of $\mathcal{A}$ even when the underlying DAG G is concurrently growing. Indeed, fix a finite run $\mathcal{R}$ of $\mathcal{A}^\beta$ using a finite graph G. Now consider any extension $\mathcal{R}'$ of $\mathcal{R}$ and suppose that $\mathcal{R}'$ is using β constructed in $\mathcal{R}$ and a super-graph G' of G that now contains vertices and edges that were possibly missing in G. We observe that the run of $\mathcal{A}$ simulated by $\mathcal{R}'$ extends the run of $\mathcal{A}$ simulated by $\mathcal{R}$: each "new" vertex appearing in G' may only contribute to a simulated step of $\mathcal{A}$ appearing *after* all steps of $\mathcal{A}$ simulated by $\mathcal{R}$ took place. This implies that $\mathcal{A}^\beta$ is not affected by the way the partial DAGs are constructed in the algorithm in Figure 1, and depends only on β and the limit infinite DAG. The concurrent maintenance of β will be used in Section 4.5 for multiple simulations of runs of $\mathcal{A}$ using ever-growing DAGs G.

The following theorem shows that the sequence of simulated steps produced by $\mathcal{A}^\beta$ indeed belongs to a (possibly unfair) run of $\mathcal{A}$.

Theorem 2 *Let G be a DAG produced by the algorithm in Figure 1 in a fair run R with a failure pattern F. Let β be any delay map for G. Let R' be any run of $\mathcal{A}^\beta$ using G and an input vector I (Figure 2). Then the sequence of steps simulated by $\mathcal{A}^\beta$ in R' belongs to a run of $\mathcal{A}$, $R_\mathcal{A}$, with input vector I and failure pattern F, $inf(R_\mathcal{A}) = correct(F) \cap inf(R')$. Specifically, if $correct(F) \subseteq inf(R')$, then $R_\mathcal{A}$ is fair.*

Proof. Recall that a step of $\mathcal{A}$ of a process p_i can be either a *memory* step in which p_i accesses shared memory or a *query* step in which p_i queries the failure detector. Since memory steps are performed in $\mathcal{A}^\beta$ as in $\mathcal{A}$, to check whether the algorithm produces a run of $\mathcal{A}$ with failure pattern F, it is enough to make sure that the sequence of failure detector queries in the simulated run (using vertices of G) *could* have been observed in a real run of $\mathcal{A}$ with F.

Shared variables:
  $V_1, \ldots, V_n$, initially $\perp, \ldots, \perp$
  Shared variables of $\mathcal{A}$

Initially: assign processes $p_1, \ldots, p_n$ with states of $\mathcal{A}$ using I

To simulate the next step of p_i:
8 $\ell := 0$
9 $U := [V_1, \ldots, V_n]$
10 **repeat**
11 $\ell := \ell + 1$
12 $r := 0$
13 **repeat**
14 $r := r + 1$
15 **if** $r > \beta(p_i, \ell)$ **then** $\beta(p_i, \ell) := r$
16 **until** G includes $[p_i, d, \ell]$ for some d **and** $r = \beta(p_i, \ell)$
17 **until** $\forall j$, $U[j] \neq \perp$: $(U[j], [p_i, d, \ell]) \in G$
18 $V_i := [p_i, d, \ell]$
19 take the next step of $\mathcal{A}$ using d as the output of $\mathcal{D}$

Figure 2: $\mathcal{A}^\beta$: an asynchronous simulation of $\mathcal{A}$ with input vector I

Consider two simulated query steps s_i and s_j are associated with vertices $[p_i, d_i, \ell]$ and $[p_j, d_j, \ell']$, respectively. Suppose that in $\mathcal{R}_\mathcal{A}$, s_i occurs before s_j.

If $[p_i, d_i, \ell]$ precedes $[p_j, d_j, \ell']$ in G, i.e., $([p_i, d_i, \ell], [p_j, d_j, \ell'])$ is an edge of G, then, by property (1) of DAGs, $\tau([p_i, d_i, \ell]) < \tau([p_j, d_j, \ell'])$, and s_i and s_j could have taken place in $\mathcal{R}_\mathcal{A}$ with F in that order.

Now suppose that $[p_i, d_i, \ell]$ does not precede $[p_j, d_j, \ell']$ in G. We show that s_j does not causally precede s_i in the simulated run, and, thus, the simulated run is indistinguishable from a run in which s_i takes place before s_j. Suppose not, i.e., p_j simulated at least one memory step s'_j after s_j, and p_i simulated at least one memory step s'_i before s_i, such that the memory access of s'_j took place before the memory access of s'_i in R. But then p_i must have found $[p_j, d_j, \ell']$ or a later vertex in V_j before simulating step s_i (line 9) and, thus, the vertex of G used for simulating s_i must be a descendant of $[p_j, d_j, \ell']$ — a contradiction. Thus, the sequence of steps of $\mathcal{A}$ simulated in R *could have happened* in a run $R_\mathcal{A}$ of $\mathcal{A}$ with failure pattern F and input vector I.

Since in $\mathcal{A}^\beta$, a simulated step of p_i can only be performed by p_i itself, $\mathit{inf}(R_\mathcal{A}) \subseteq \mathit{inf}(R')$. Also, since each faulty in F process contains only finitely many vertices in G, each process in $\mathit{inf}(R') - \mathit{correct}(F)$ is eventually blocked forever in lines 13–16 in Figure 2, and, thus, $\mathit{inf}(R_\mathcal{A}) \subseteq \mathit{correct}(F)$. By property (4) of DAGs, for every finite set V of vertices in G, every process in $\mathit{correct}(F)$ obtains infinitely many vertices in G that succeed every vertex in V. Thus, no process in $\mathit{correct}(F) \cap \mathit{inf}(R')$ can be blocked forever in lines 13–16. Hence, every process in $\mathit{correct}(F) \cap \mathit{inf}(R')$ simulates infinitely many steps of $\mathcal{A}^\beta$, and, thus, $\mathit{inf}(R_\mathcal{A}) = \mathit{correct}(F) \cap \mathit{inf}(R')$. Specifically, if $\mathit{correct}(F) \subseteq \mathit{inf}(R')$, then the set of processes that appear infinitely often in $R_\mathcal{A}$ is $\mathit{correct}(F)$, and the run is fair. □

4.3 The BG-simulation technique

Borowsky and Gafni proposed a simulation technique by which $k+1$ processes $q_1, \ldots, q_{k+1}$, called *simulators*, can wait-free simulate a k-resilient execution of any asynchronous n-process protocol [3, 5]. Informally, the simulation works as follows. Every simulator q_i tries to simulate steps of all n processes $p_1, \ldots, p_n$ in a round-robin fashion. The simulation guarantees that the next step of every process p_j is either agreed on by all simulators, or one less simulator participates further in the simulation for each step which is not agreed on. Consequently, as long as there is at least one live simulator, at most k simulated processes may be blocked and at least $n-k$ simulated processes in $\{p_1, \ldots, p_n\}$ accept infinitely many simulated steps. A sequence of steps σ of the simulators $q_1, \ldots, q_{k+1}$ determines the unique sequence $BG(\sigma)$ of processes in $p_1, \ldots, p_n$ that specifies the order in which the processes take steps in the corresponding simulated k-resilient execution.

Let σ be any infinite execution of simulators $q_1, \ldots, q_{k+1}$. A process p_i is said to be *blocked in* σ if p_i appears only finitely often in $BG(\sigma)$. Let $live(\sigma)$ denote the set of simulators that appear infinitely often in σ and $faulty(\sigma)$ be the complement to $live(\sigma)$ in $\{q_1, \ldots, q_{k+1}\}$. In our reduction algorithm, we are going to use the following property of the BG-simulation technique [3, 5]:

(BG1) Let σ' be the shortest prefix of σ that includes all steps the processes in $faulty(\sigma)$ take in σ. Then for any extension $\sigma' \cdot \zeta$ such that ζ includes only steps of simulators in $live(\sigma)$, every process which is blocked in σ is also blocked in $\sigma' \cdot \zeta$, and $BG(\sigma)$ and $BG(\sigma' \cdot \zeta)$ agree on the shortest prefix which contains all steps of the blocked in σ processes.

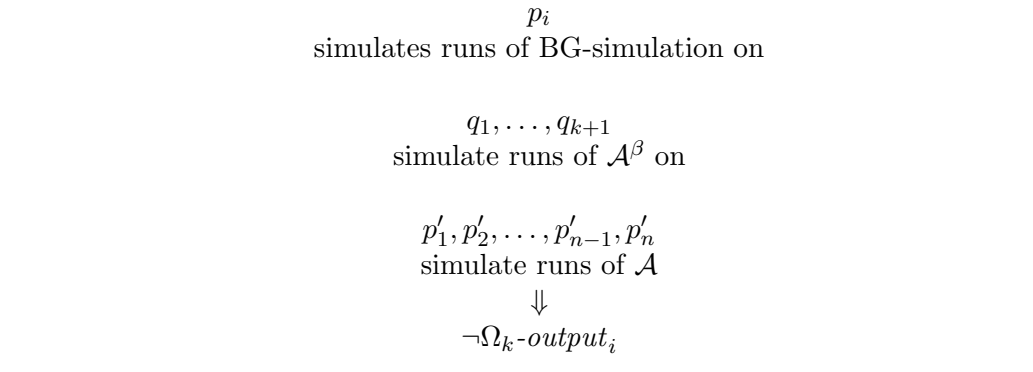

Figure 3: Three levels of simulation.

4.4 Hierarchical simulation

Our reduction algorithm employs *triple* simulation (summarized in Figure 3). For a given DAG G and a delay map β defined on G, a process p_i locally simulates multiple partial runs of a system of $k+1$ simulators $\mathcal{Q} = \{q_1, \ldots, q_{k+1}\}$ that use the BG-simulation technique to collectively produce partial runs of $\mathcal{A}^\beta$ for n simulated processes $\Pi' = \{p'_1, \ldots, p'_n\}$. In turn, each run of $\mathcal{A}^\beta$ on $p'_1, \ldots, p'_n$ simulates a run of the original algorithm $\mathcal{A}$ using, instead of failure detector $\mathcal{D}$, the sample of $\mathcal{D}$'s output encoded in G (as shown in Figure 2). To avoid confusion, here we use p'_j to denote the process that models p_j in a run of $\mathcal{A}^\beta$ simulated by a "real" process p_i. There is a trivial bijection between p_j and p'_j.

To simulate steps of p'_j in a set agreement algorithm, simulators $q_1, \ldots, q_{k+1}$ should agree first on the its input value. Therefore, in the first simulated step of p'_j, every simulator proposes its own input value in I as the input value of p'_j. Once the input value of p'_j is agreed upon, the simulation moves forward as before. Note that the output of the simulated run of $\mathcal{A}$ on $p'_1, \ldots, p'_n$ complies with the inputs of the simulators and thus the specification of k-set agreement is not violated.

The "waiting" cycle in lines 13–16 is simulated by $q_1, \ldots, q_{k+1}$ as *one* local step. Thus, before p_j performs a step of $\mathcal{A}$ using a vertex $[p_j, d, \ell]$ in DAG G_i (line 19 in Figure 2), it first performs $\beta_i(p_j, \ell)$ local "waiting" steps. If a vertex of the form $[p_j, d, \ell]$ never appears in G_i, then p_j performs infinitely many local steps, and drops out from the simulated run of $\mathcal{A}$.

4.5 Extracting $\neg\Omega_k$

The computational component of our reduction algorithm is presented in Figure 4. Each initial state I for $k+1$-process k-set agreement, each *schedule* σ, a sequence specifying the order in which simulators $q_1, \ldots, q_{k+1}$ take steps of BG-simulation, and the DAG G_i, determine a run of $\mathcal{A}^{\beta_i}$, denoted $\alpha_i(I, \sigma)$. For brevity, $dom_I(\alpha_i)$ denotes here all distinct (not related by containment) schedules σ explored so far by p_i with input vector I, i.e., used for evaluating $\alpha_i(I, \sigma)$ in line 23.

We say that $\alpha_i(I, \sigma)$ is *deciding* if it simulates a run of $\mathcal{A}$ in which at least one process decides. Respectively, a schedule σ is called *deciding at p_i with input vector I* if $\alpha_i(I, \sigma)$ is deciding.

4.5.1 Overview

For all input vectors I (chosen in some deterministic order $<$), the reduction algorithm simulates longer and longer executions of $\mathcal{A}^{\beta_i}$, using longer and longer schedules of steps of simulators. The schedules are selected following the depth-first-search strategy: every next simulated step extends the longest currently observed non-deciding schedule. Each process p_i periodically registers in the shared memory all currently simulated runs in the form of the simulation map α_i and the delay map β_i (Figure 2), and scans the memory to get the latest update on the maps of other processes. If p_i finds out that, at some process p_j, all distinct schedules simulated so far by p_i (including the currently simulated schedule σ) are deciding (line 24), then p_i adopts all simulations of p_j, rolls back and continues the simulation from the longest non-deciding prefix of σ.

Periodically, p_i evaluates the set of $n-k$ processes that appear the latest in the currently simulated run of $\mathcal{A}^{\beta_i}$ as the output of $\neg\Omega_k$. The intuition is that, since the task has no wait-free solution for $k+1$ processes, eventually, all processes will proceed with longer and longer prefixes of the same schedule that corresponds to a non-deciding k-resilient runs of $\mathcal{A}^{\beta_i}$. Otherwise, by Theorem 2, we would obtain a wait-free algorithm for $k+1$-process k-set agreement.

4.5.2 Maintaining the simulation corridors

The conventional depth-first-search technique allows the simulation to temporarily go along a finite deciding "branch" of the ever-growing non-deciding schedule, and such branches may involve steps of arbitrary processes in $p_1, \ldots, p_n$. As a result, the set of $n-k$ process that appear the latest in the currently simulated run of $\mathcal{A}^{\beta_i}$ may infinitely often include arbitrary processes.

To overcome this issue, we put an additional restriction on the order in which we choose the next simulator to extend the current non-deciding schedule σ. At each point in the simulation, we maintain a "corridor" (the third parameter in function *explore*) — the set of simulators that can be

```
20  for all inputs I_0 (in a deterministic order <) do      { For all possible inputs for q_1, ..., q_{k+1} }
21    explore(I_0, ⊥, Π)

22  function explore(I, σ, S)
23    ¬Ω_k-output_i := n − k processes that appear the latest in α_i(I, σ)
                (where each p'_j is replaced with p_j)
24    if ∃p_j ∈ Π: ∀σ' ∈ dom_I(α_i), ∃σ'', a prefix of σ': α_j(I, σ'') is deciding then
                        { If all explored schedules are deciding at some p_j with I }
25      α_i := α_j; β_i := β_j       {adopt p_j's simulation}
26    else
27      for all non-empty S' ⊆ S (in a deterministic order consistent with ⊆) do
28        for all q_j ∈ S' (in a deterministic order) do
29          explore(I, σ · q_j, S')
```

Figure 4: Computational component of the reduction algorithm: code for each process p_i.

used for further extensions of the current schedule. The extended schedule can only use simulators in a sub-corridor of the current corridor, and the sub-corridors are selected in a deterministic order, consistent with the $\subseteq$ relation (lines 27 and 28 in Figure 4). Thus, the algorithm first explores all "solo" corridors consisting of solo extensions of σ, then all "duet" corridors consisting of extensions including steps of two given processes, then "trio", etc. If all extensions within the chosen sub-corridor turn out to be deciding, the next sub-corridor is selected, etc.[3]

As a result, eventually, only simulators that appear infinitely often in some never-deciding run will be output: otherwise, the simulation already operates in proper superset of a more narrow corridor that contains a never-deciding schedule $\tilde{\sigma}$, and that contradicts the order in which corridors are chosen (consistently with $\subseteq$).

At least one simulated process p'_j, such that $p_j \in \mathit{correct}(F)$, must be blocked in $\tilde{\sigma}$. Otherwise, the schedule would simulate a fair run of $\mathcal{A}$ and thus would be deciding. Moreover, since all correct processes extend longer and longer prefixes of $\tilde{\sigma}$, by property (BG1) of BG-simulation, p'_j eventually stops taking steps in runs simulated at correct processes. All simulated runs of $\mathcal{A}^{\beta_i}$ extend ever-growing prefixes of a k-resilient run, and hence the set of $n-k$ latest processes in them will eventually never include p_j. Thus, the output of $\neg\Omega_k$ updated in line 23 at each correct process will eventually never include p_j.

Our reduction algorithm therefore outputs, at each time and at every process, a set of $n-k$ processes, such that, eventually, some correct process is never output — $\neg\Omega_k$ is extracted.

4.5.3 Correctness

Consider any fair run of the algorithm in Figures 1 and 4. Let F be the failure pattern of that run. First we prove the following auxiliary lemmas:

Lemma 3 *If the currently simulated schedule σ is deciding at a process p_i with the current input vector I (in line 23), then (i) $\forall\sigma' \in \mathit{dom}_I(\alpha_i)$, $\alpha_i(I,\sigma')$ is deciding, and (ii) $\forall\sigma'$, $\forall J < I$, $\exists\sigma''$, a prefix of σ' such that $\alpha_i(J,\sigma'')$ is deciding.*

[3]Here we apply the technique proposed in [24] to simulators.

Proof. The domain of mapping α_i is constantly growing, and a process considers a new schedule σ in line 23 to be deciding with the current input vactor I only if all distinct schedules σ considered up to now were deciding with I.

Furthermore, for all previously considered input vectors $J < I$ all schedules have deciding prefixes: the invocation of $explore(J, \perp, \Pi)$ in line 21 returns only if all possible schedules σ have deciding with J prefixes.

This property is inductively preserved if p_i adopts α_j from another process p_j in line 25. □

Lemma 4 *If $explore(I, \sigma, S)$ invoked by a correct process p_i in line 21 or 29 returns, then it returns at every correct process.*

Proof. Suppose $explore(I, \sigma, S)$ invoked by a correct process p_i returns. In other words, every sufficiently long extension of σ (within the corridor S) is considered deciding with I at p_i. Thus, eventually, p_i registers α_i in the shared memory (line 7 in Figure 1) and it will be read (line 3 in Figure 1) and adopted by every correct process p_j in line 25: thus every sufficiently long extension of σ (within the corridor S) will be considered deciding with I at p_j. □

Lemma 5 *There exists an input vector I, such that each correct process p_i eventually invokes $explore(I, \perp, \Pi)$ in line 21 and the invocation never returns.*

Proof. Suppose that, at some correct process p_i, $explore(I, \perp, \Pi)$ returns for all input vectors I. Let β be the value of β_i and α be the value of α_i when the last such invocation returns. By Lemma 3, for all input vectors I and for all schedules σ, there exists σ', a prefix of σ, such that $\alpha(I, \sigma')$, i.e., the run of BG-simulation with input vector I and schedule σ' simulates a deciding run of $\mathcal{A}^\beta$.

Note that $\mathcal{A}^\beta$ uses vertices of a DAG G instead of the failure detector output. For all possible input vectors I, consider the tree of all schedules σ of steps of the simulators that are deciding with I. All such trees have finite branching (each vertex has at most $k+1$ descendants) and contain no infinite paths. By König's lemma, the trees have finitely many vertices. Thus, the set of vertices of G used by the runs of $\mathcal{A}'$ simulated by deciding schedules of $BG(\mathcal{A}'_\beta)$ is also finite. Let $\bar{G}$ be a finite subgraph of G that includes all vertices used by these runs.

Thus, we can construct a wait-free k-set agreement algorithm for $q_1, \ldots, q_{k+1}$ as follows. Each process q_i runs BG-simulation of $\mathcal{A}^\beta$ using the finite DAG $\bar{G}$ as a parameter. To simulate the first step of a process p_j, q_i uses its own input value as the input value of p_j. When q_i simulates a step of $\mathcal{A}^\beta$ in which a simulated process p'_j decides, q_i writes the decided value in the shared memory and returns the value. Since every simulated run is deciding, each q_i eventually simulates a deciding step or finds a decided value in the shared memory. Since the decided values are coming from a run of the failure-detector-based k-set agreement algorithm $\mathcal{A}$, and the inputs are provided by $q_1, \ldots, q_{k+1}$, the set of decided val;ues is a subset of the inputs of size at most k. But this contradicts [16, 23, 3].

Thus, there exists I such that $explore(I, \perp, \Pi)$ invoked by a correct process p_i never returns. By the algorithm, the invocation previously returned for all input vectors $J < I$. By Lemma 4, $explore(I, \perp, \Pi)$ is also invoked by every other correct process and never returns. □

Theorem 6 *In all environments $\mathcal{E}$, if a failure detector $\mathcal{D}$ solves n-process k-set agreement, then $\neg\Omega_k$ is weaker than $\mathcal{D}$.*

Proof. By Lemma 5, there exists I, such that for all correct process p_i, p_i invokes $explore(I, \perp, \Pi)$ in line 21 and the invocation never returns. Thus, every correct process p_i makes an infinite sequence of recursive invocations of *explore* with parameters (I, σ_0, S_0), (I, σ_1, S_1), ..., where (line 27) $\forall \ell \in \mathbb{N}$: $S_\ell \neq \emptyset$, $S_{\ell+1} \subseteq S_\ell$. By Lemma 4, all correct processes agree on the sequence (I, σ_0, S_0), (I, σ_1, S_1), ...

Since all S_ℓ are non-empty, there exist $\tilde{S} \neq \emptyset$ and $\ell' \in \mathbb{N}$, such that $\forall \ell \geq \ell'$: $S_\ell = \tilde{S}$. Also, each σ_ℓ is a prefix of some infinite *non-deciding* schedule $\tilde{\sigma}$. Now we show that $\tilde{S} = live(\tilde{\sigma})$, the set of processes that appear infinitely often in $\tilde{\sigma}$.

By construction (line 28), $live(\tilde{\sigma}) \subseteq \tilde{S}$. Suppose, by contradiction, that $live(\tilde{\sigma})$ is a proper subset of $\tilde{S}$. Let $S'_0, S'_1, \ldots$ be the sequence of non-empty subsets of Π such that $\forall 0 \leq \ell < \ell'$: $S'_\ell = S_\ell$ and $\forall \ell \geq \ell'$: $S_\ell = live(\tilde{\sigma})$. Thus, the non-deciding schedule $\tilde{\sigma} = q_{i_0}, q_{i_1}, \ldots$ *fits* the corridor specified by $S'_0, S'_1, \ldots$, i.e., $\forall \ell \in \mathcal{N} : q_{i_\ell} \in S'_0$. By the algorithm, before making the infinite sequence of invocations $explore(I, \sigma_0, S_0)$, $explore(I, \sigma_1, S_1)$, ..., p_i has previously explored prefixes of all schedules that fit $S'_0, S'_1, \ldots$, including a prefix of $\tilde{\sigma}$, and found all of them deciding — a contradiction.

Thus, all correct processes p_i perform the same infinite sequence of recursive invocations of *explore* with parameters (I, σ_0, S_0), (I, σ_1, S_1), ..., $(I, \sigma_{\ell'}, live(\tilde{\sigma}))$, $(I, \sigma_{\ell'+1}, live(\tilde{\sigma}))$, ...

Let σ' be the shortest prefix of $\tilde{\sigma}$ that contains *all* appearances of the simulators that are faulty in $\tilde{\sigma}$, $faulty(\tilde{\sigma})$. Eventually, all correct processes simulate extensions of σ' in which processes in $faulty(\tilde{\sigma})$ do not appear. Let W be the set of simulated processes in $\{p'_1, \ldots, p'_n\}$ that are *blocked* in $\tilde{\sigma}$ (Section 4.3). We argue that W contains at least one process p'_j such that $p_j \in correct(F)$.

By contradiction, suppose that no process in $correct(F)$ is blocked in $\tilde{\sigma}$. Let $p_j \in correct(F)$. Thus, p_i simulates a run of $\mathcal{A}^{\beta_i}$ in which p'_j appears infinitely often. Moreover, map β_i maintained at p_i ensures that for all $\ell \in \mathbb{N}$, $\beta(p_j, \ell)$ eventually stops growing. This is because every correct process p_s eventually finds a vertex $[p_j, d, \ell]$ in G_s and stops incrementing $\beta(p_j, \ell)$ and there is a time after which p_i does not adopt delay maps of faulty processes in line 25. Thus, by Theorem 2, the run of $\mathcal{A}^{\beta_i}$ with input vector I simulated by $q_1, \ldots, q_{k+1}$ in schedule $\tilde{\sigma}$ produces a fair and thus deciding run of $\mathcal{A}$ — a contradiction.

Now, by property (BG1) of BG-simulation, p'_j eventually stops participating in all runs of $\mathcal{A}_{\beta_i}$ simulated at every correct process p_i. Moreover, since p_i simulates extensions of longer and longer prefixes of some k-resilient run R', eventually, the latest $n-k$ processes seen in every run of $\mathcal{A}^{\beta_i}$ simulated by p_i will include only processes in $inf(R)$ and, thus, p_j will eventually never be output in line 23.

To summarize, we have an algorithm that outputs, at each time and at every process, a set of $n-k$ processes, such that, eventually, some correct process is never output — $\neg\Omega_k$ is extracted. □

A combination of Theorem 1 and Theorem 6 implies:

Theorem 7 *In all environments $\mathcal{E}$, $\neg\Omega_k$ is the weakest failure detector to solve k-set agreement.*

5 Related Work

Chandra et al. [6] derived the first "weakest failure detector" result by showing that Ω is necessary and sufficient to solve consensus in the message-passing model. The result was later generalized to the read-write shared memory model.[4]

Guerraoui et al. [13] proposed the notion of the weakest failure detector *ever*: a failure detector that is sufficient to circumvent *some* asynchronous impossibility and necessary to circumvent *any* asynchronous impossibility. A candidate failure detector, denoted Υ was proposed in [13] and shown to be strong enough to solve wait-free $(n-1)$-set agreement. Υ was shown to be weaker than any *stable* failure detector that circumvents an asynchronous impossibility. The result easily extends to the case of n-process k-set agreement through failure detector Υ^k. [5]

Zieliński [24] introduced anti-Ω and proved that it is the weakest failure detector to solve wait-free $(n-1)$-set agreement in the whole universe of failure detectors (including unstable ones). However, generalizing [24, 13] to k-set agreement for all k without restricting the space of considered failure detectors turned out to be non-trivial and has remained unresolved until now.

The necessity part of this paper builds atop of two fundamental results. The first is the celebrated BG-simulation [3, 5] that allows $k+1$ processes simulate, in a wait-free manner, a k-resilient run of any n-process asynchronous algorithm. The second is a brilliant observation made by Zieliński [24] that any run of an algorithm $\mathcal{A}$ using a failure detector $\mathcal{D}$ induces an asynchronous algorithm that simulates (possibly finite and unfair) runs of $\mathcal{A}$. Unlike [24], however, our reduction algorithm assumes the conventional read-write shared memory model without using immediate snapshots [4]. Also, instead of growing "precedence" and "detector" maps of [24], we use directed acyclic graphs á la [6].

Concurrently and independently, two papers [11, 2] claimed to have shown that $\neg\Omega_k$ is the weakest failure detector for solving k-set agreement, [11] — for all environments, and [2] — for all environments in which at most k processes fail.

6 Discussion

In this paper, we show that, for all $k = 1, \ldots, n-1$, $\neg\Omega_k$ is the weakest failure detector to solve n-process k-set agreement, in all environments, i.e., regardless of the assumptions on when and where failure may occur. The paper fills the gap between the proof that Ω (equivalent to $\neg\Omega_1$) is the weakest failure detector to solve consensus [6] and the proof that anti-Ω (equivalent to $\neg\Omega_{n-1}$) is the weakest failure detector to solve $(n-1)$-set agreement. Therefore, this paper closes the long-standing quest for finding the weakest failure detector for general k-set agreement [6, 22, 9, 13, 24].

Given that k-set agreement is equivalent to k simultaneous consensuses of which one is guaranteed to return [1], one may be tempted to try determining the weakest failure detector for k-set agreement through running k parallel CHTs (reduction algorithms of Chandra et al. [6] that derive the weakest failure detector for consensus). But such a derivation may be tricky to find: CHT re-implements crucial elements of the FLP consensus impossibility proof [12], and thus extending it to set agreement may be similar to extending FLP to proving the impossibility of set agreement,

[4] The result was stated in [19], but the only published proof of it (we are aware of) appears in [15].

[5] The conference version of the paper [13] assumes that failure detectors are stable and their output depends only on the set of correct processes, and not on the timing of failures. The second assumption is eliminated in the full version of this paper [14].

and such an extension stays elusive for many years.

Instead, our proof derives a necessary failure detector for solving k-set agreement from the very fact that wait-free $k+1$-process k-set agreement is impossible [16, 23, 3]. Borowsky and Gafni showed in [3] that the impossibility of solving a task in a k-resilient manner can be derived from the impossibility of solving the task in a wait-free manner among $k+1$ processes. In this paper, we extend this observation to the failure detector world, and we derive the weakest failure detector to circumvent an impossibility of solving a task from the impossibility itself, ignoring the exact specification of the task. One interesting feature of our necessity proof is that it does not restrict the space of available shared objects. We show that $\neg\Omega_k$ is necessary for solving k-set agreement in any system and any environment in which the task cannot be solved asynchronously.

As a byproduct, this paper gives an alternative (and seemingly simpler than in [6, 15]) proof that Ω is the weakest failure detector for solving consensus using reads and writes among $n > 2$ processes (the case $n = 2$ is covered in [24]). The proof only uses the fact that wait-free 2-process consensus is impossible, and unlike [6, 15], does not involve elements of consensus impossibility [12], such as bivalency, critical configurations, etc. The proof is however not self-contained, since it relies heavily on BG simulation.

Finally, this paper provides an evidence for the "folklore" hierarchy of n-process "sub-consensus" symmetric distributed tasks, based on the amount of synchrony needed to solve them using reads and writes. (A task is symmetric if the input-output relation withstands an arbitrary permutation of process identifiers.) The bottom level (level 0) in this conjectured hierarchy is populated by *trivial* tasks, tasks that can be solved asynchronously (e.g., $(2n-1)$-renaming). The top level (level $n-1$) is populated by *universal* tasks (e.g., consensus): if a failure detector solves a universal task, then it solves any task. The weakest failure detector to solve a universal task is Ω [6, 15]. Now level ℓ ($\ell = 1, \ldots, n-2$) is defined iteratively as follows. A task $\mathcal{T}$ belongs to level ℓ if and only if it does not belong to level $\ell - 1$ and any failure detector that solves a task that does not belong to level $\ell - 1$ also solves $\mathcal{T}$.

Level 1 of the conjectured task hierarchy is characterized in [24] where $(n-1)$-set agreement is shown to be the easiest (in the failure detector sense) non-trivial task and $\neg\Omega_{n-1}$ to be the matching weakest failure detector. Our results suggest that level ℓ ($\ell = 2, \ldots, n-1$) is populated by $(n-\ell)$-set agreement and the corresponding failure detector is $\neg\Omega_{n-\ell}$. Proving this conjecture would boil down to showing that $\neg\Omega_k$ is necessary to solve any symmetric task that cannot be solved using $\neg\Omega_{k-1}$ which is more general than the necessity result of our paper.

Acknowledgements

The conjectured hierarchy of tasks described in Section 6 was conjectured concurrently by several researchers, including Eli Gafni, Rachid Guerraoui, Achour Mostefaoui, and Michel Raynal.

References

[1] Yehuda Afek, Eli Gafni, Sergio Rajsbaum, Michel Raynal, and Corentin Travers. Simultaneous consensus tasks: A tighter characterization of set-consensus. In *ICDCN*, pages 331–341, 2006.

[2] Antonio Fernández Anta, Sergio Rajsbaum, and Corentin Travers. Weakest failure detectors via an egg-laying simulation (brief announcement). In *PODC*, 2009.

[3] Elizabeth Borowsky and Eli Gafni. Generalized FLP impossibility result for t-resilient asynchronous computations. In *STOC*, pages 91–100. ACM Press, May 1993.

[4] Elizabeth Borowsky and Eli Gafni. Immediate atomic snapshots and fast renaming. In *PODC*, pages 41–51, New York, NY, USA, 1993. ACM Press.

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

[6] 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.

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

[8] Soma Chaudhuri. Agreement is harder than consensus: Set consensus problems in totally asynchronous systems. In *PODC*, pages 311–324, August 1990.

[9] Wei Chen, Jialin Zhang, Yu Chen, and Xuezheng Liu. Weakening failure detectors for -set agreement via the partition approach. In *DISC*, pages 123–138, 2007.

[10] F. Chu. Reducing Ω to $\Diamond W$. *Information Processing Letters*, 67(6):298–293, September 1998.

[11] Carole Delporte-Gallet, Hugues Fauconnier, Rachid Guerraoui, and Andreas Tielmann. The disagreement power of an adversary (brief announcement). In *PODC*, 2009.

[12] 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.

[13] Rachid Guerraoui, Maurice Herlihy, Petr Kouznetsov, Nancy A. Lynch, and Calvin C. Newport. On the weakest failure detector ever. In *PODC*, pages 235–243, August 2007.

[14] Rachid Guerraoui, Maurice Herlihy, Petr Kuznetsov, Nancy A. Lynch, and Calvin C. Newport. On the weakest failure detector ever. *Distributed Computing*, 21(5):353–366, February 2009.

[15] Rachid Guerraoui and Petr Kuznetsov. Failure detectors as type boosters. *Distributed Computing*, 20(5):343–358, 2008.

[16] Maurice Herlihy and Nir Shavit. The asynchronous computability theorem for t-resilient tasks. In *STOC*, pages 111–120, May 1993.

[17] Prasad Jayanti and Sam Toueg. Every problem has a weakest failure detector. In *PODC*, pages 75–84, 2008.

[18] Petr Kuznetsov. Simle CHT: A new derivation of the weakest failure detector for consensus. Technical report, 2009.

[19] Wai-Kau Lo and Vassos Hadzilacos. Using failure detectors to solve consensus in asynchronous shared memory systems. In *WDAG*, LNCS 857, pages 280–295, September 1994.

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

```
   Shared variables:
      for all p_j ∈ Π: Counters_j, initially 0, ..., 0

30 while true do
31    output := query ¬Ω_k
32    for all p_ℓ ∈ output do
33          Counters_i[ℓ] := Counters_i[ℓ] + 1
34    for ℓ = 1, 2, ..., n do
35          total[ℓ] := Counters_1[ℓ] + ... + Counters_n[ℓ]
36    p_{i_1}, ..., p_{i_n} := deterministic permutation of p_1, ..., p_n
         w.r.t. increasing total[ℓ]
37    Ω⃗_k-output := (p_{i_1}, ..., p_{i_k})
```

Figure 5: Transforming $\neg\Omega_k$ into vector-Ω_k: the code for each process p_i

[21] Michel Raynal. K-anti-Omega, August 2007. Rump session at PODC 2007.

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

[23] Michael Saks and Fotios Zaharoglou. Wait-free k-set agreement is impossible: The topology of public knowledge. In *STOC*, pages 101–110. ACM Press, May 1993.

[24] Piotr Zieliński. Anti-omega: the weakest failure detector for set agreement. In *PODC*, August 2008.

$\neg\Omega_k$ and vector-Ω_k are equivalent

Theorem 8 *For all $\mathcal{E}$ and all $k \in \{1, \ldots, n-1\}$, $\overrightarrow{\Omega}_k$ is weaker than $\neg\Omega_k$ in $\mathcal{E}$.*

Proof. The algorithm transforming $\neg\Omega_k$ into $\overrightarrow{\Omega}_k$ is presented in Figure 5. Periodically, every process p_i queries $\neg\Omega_k$, and for each p_j output by $\neg\Omega_k$, increments the shared counter register $Counters_i[j]$ and calculates k processes $p_{i_1}, \ldots, p_{i_k}$ which are least output by $\neg\Omega_k$ so far (summed over all processes). Then the k-vector of the extracted output of $\overrightarrow{\Omega}_k$ is updated to $[p_{i_1}, \ldots, p_{i_k}]$.

Let $U \subseteq \Pi$ be the set of processes p_ℓ that are output only finitely many times by $\neg\Omega_k$, and, thus, $total[\ell]$ eventually stabilizes on the same finite value at each correct process. Since at least one correct process is eventually never output by $\neg\Omega_k$, U includes at least one correct process. On the other hand, since $\neg\Omega_k$ always outputs $n-k$ processes, there are at least $n-k$ processes p_ℓ such that $total[\ell]$ grows without bound at each correct process, and thus $|U| \leq k$. Thus, there is a time after which all correct processes agree on the first $|U|$ processes $p_{i_1}, \ldots p_{i_{|U|}}$ in the produced output of $\overrightarrow{\Omega}_k$ and at least one of them is correct — the output of $\overrightarrow{\Omega}_k$ is extracted. □