On Set Consensus Numbers*

Eli Gafni[1] and Petr Kuznetsov[2]

[1] Computer Science Department, University of California, Los Angeles, USA
eli@ucla.edu
[2] TU Berlin/Deutsche Telekom Laboratories, Berlin, Germany
pkuznets@acm.org

Abstract. It is conjectured that the only way a failure detector (FD) can help solving n-process tasks is by providing k-set consensus for some $k \in \{1, \ldots, n\}$ among all the processes. It was recently shown by Zieliński that any FD that allows for solving a given n-process task that is unsolvable read-write wait-free, also solves $(n-1)$-set consensus.

In this paper, we provide a generalization of Zieliński's result. We show that any FD that solves a colorless task that cannot be solved read-write k-resiliently, also solves k-set consensus. More generally, we show that every colorless task $\mathcal{T}$ can be characterized by its *set consensus number*: the largest $k \in \{1, \ldots, n\}$ such that $\mathcal{T}$ is solvable $(k-1)$-resiliently. A task $\mathcal{T}$ with set consensus number k is, in the failure detector sense, equivalent to k-set consensus, i.e., a FD solves $\mathcal{T}$ if and only if it solves k-set consensus.

As a corollary, we determine the weakest FD for solving k-set consensus in every environment, i.e., for all assumptions on when and where failures might occur.

1 Introduction

One of the central challenges in distributed computing is characterizing the conditions under which a given problem is solvable. In this paper we focus on a class of problems, called *distributed tasks*. We consider a system of n processes, subject to the *crash* failures and communicating via reading and writing in the shared memory. Informally, the correctness of an algorithm solving a distributed task depends only on the inputs the processes receive in the beginning of the computation and the outputs they produce at the end of it.

An example of a distributed task is k-set consensus [7], in which the processes starts with inputs in $\{0, \ldots, k\}$ and the set of outputs must be a subset of inputs of size at most k. For all $0 < k < n$, there is no algorithm that solves k-set consensus tolerating k faulty processes and making no synchrony assumptions [17, 22, 2, 3]. In other words, the lack of synchrony and the presence of failures make k-resilient k-set consensus impossible. To circumvent the impossibility, assuming that we still want to tolerate failures, we need to introduce some synchrony into the system. But how much synchrony is enough?

Synchrony assumptions can be described using *failure detectors* (FDs) [5, 4], distributed oracles that provide processes with some (possibly inaccurate and incomplete) hints about failures. For example, Ω, the "eventual leader" failure detector [4], outputs, when queried, a process identifier and guarantees that, eventually, the same *correct* process identifier is forever output. Ω has been shown to be sufficient to solve any distributed task tolerating any number of failures. The exact amount of synchrony needed to circumvent an asynchronous impossibility is captured through the notion of the *weakest FD* [4]: $\mathcal{D}$ is the weakest FD for solving a problem $\mathcal{M}$ if $\mathcal{D}$ is both (1) *sufficient*

* This paper combines results appeared in the Proceedings of the 28th ACM Symposium on Principles of Distributed Computing (PODC 2009) and the 23rd International Symposium on Distributed Computing (DISC 2009).

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 FD that is sufficient to solve $\mathcal{M}$ provides at least as much information about failures as $\mathcal{D}$ does. For a very general definition of a distributed computing problem, it has been shown that every problem (and, specifically, every task) has a weakest FD [18]. Thus the question arises "how many weakest FDs are there?"

We conjecture that for tasks on n processes the answer is n, the weakest FDs to solve k-set consensus $0 < k \leq n$, i.e., including the "empty" FD, the FD that say nothing and corresponds to the wait-free solvable tasks and the Ω failure detector that solves any task. More precisely, for each task $\mathcal{T}$ on n processes, there is a k, such that $\neg\Omega_k$, the recently introduced anti-Ω-k FD, is the weakest to solve $\mathcal{T}$. $\neg\Omega_k$ [21, 24] outputs, when queried, a set of $n-k$ processes so that some correct process is output only finitely many times.

Zieliński showed recently that $\neg\Omega_{n-1}$ is necessary to any FD that allows for solving any n-process task not solvable $(n-1)$-resiliently (or *wait-free*) can be used for solving $(n-1)$-set consensus. In this paper, we generalize Zieliński's result and prove the conjecture for a large class of *colorless* tasks. Informally, in a solution of a colorless task, a process is free to adopt the input or output value of any other participating process.

We show that, for all $k \in \{1, \ldots, n\}$, $\neg\Omega_k$ is necessary to solve any n-process colorless task that cannot be solved k-resiliently. On the other hand, we also show that $\neg\Omega_k$ is sufficient to solve any colorless task that can be solved $(k-1)$-resiliently. Both results hold for all environments, i.e., for all possible assumptions on when and where failures might occur.

For a given task $\mathcal{T}$, let k be the largest integer in $\{1, \ldots, n)$ such that $\mathcal{T}$ is $(k-1)$-resiliently solvable. We say that k is the *set consensus number* of $\mathcal{T}$. Our results imply that in every environment, Ω_k is the weakest failure detector to solve any colorless task with set consensus number k. As a result, any colorless task $\mathcal{T}$ is equivalent to some k-set consensus in the sense that a failure detector $\mathcal{D}$ solves $\mathcal{T}$ if and only if $\mathcal{D}$ solves k-set consensus.

The rest of the paper is organized as follows. Section 2 describes our system model. Section 3 recalls the definition of $\neg\Omega_k$ and shows that $\neg\Omega_k$ is sufficient to solve k-set consensus. Section 4 proves that $\neg\Omega_k$ is necessary to solve any k-resilient unsolvable colorless task, and Section 5 presents a characterization criterion for distributed tasks. Section 6 overviews the related work. Section 7 concludes the paper by summarizing our results and listing some intriguing open questions.

2 Model

We consider a system of n processes $\Pi = \{p_1, \ldots, p_n\}$. Processes communicate via reading and writing in the shared memory and can query the failure detector. Processes are subject to *crash* failures.

2.1 Failure patterns and failure detectors

Processes can also query *failure detectors*, i.e., distributed oracles that provide the processes with information about failures of other processes [5, 4]. The local module of failure detector $\mathcal{D}$ at process p_i is denoted by $\mathcal{D}_i$. Informally, a failure detector is defined through a map from the current failure pattern (describing when and where failures occurred) to a set of failure detector histories (describing the information about failures the failure detector provides).

Formally, a *failure pattern* F is a function from the time range $\mathbb{T} = \mathbb{N}$ 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., $\forall t < t' : F(t) \subseteq F(t')$. We define $\mathit{faulty}(F) = \bigcup_{t \in \mathbb{T}} F(t)$, 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 is associated with a (possibly infinite) range $\mathcal{R}$ of values output by that failure detector. 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* $\mathcal{D}$ is a function that maps each failure pattern F to a (non-empty) set of failure detector histories with range $\mathcal{R}_\mathcal{D}$ (where $\mathcal{R}_\mathcal{D}$ denotes the range of failure detector outputs of $\mathcal{D}$). $\mathcal{D}(F)$ denotes the set of failure detector histories permitted by $\mathcal{D}$ for failure pattern F.

2.2 Algorithms

We define an *algorithm* $\mathcal{A}$ *using a failure detector* $\mathcal{D}$ as a collection of deterministic automata, one automaton $\mathcal{A}_p$ for each process p. In each step of $\mathcal{A}_p$, process p performs an atomic operation on a shared register or queries its failure detector module of $\mathcal{D}$ and receives a value, and then performs a state transition according to its automaton and the received values. A *step* of $\mathcal{A}$ is thus defined as a tuple (p, tp, x, v) where:

- p is a process id,
- tp is the type of the step (*read*, *write*, or *query*),
- x the accessed register if it is a memory (read or write) step or $\perp$ otherwise,
- v the read value if it is a read step, the written value if it is a write step, or the failure detector value if it is a query step.

A *configuration* of $\mathcal{A}$ specifies the states of each automaton $\mathcal{A}_p$ and each register in the system. In an *initial configuration* I of $\mathcal{A}$, the state of each $\mathcal{A}_p$ is an initial state of the automaton. A step $s = (p, tp, x, v)$ of $\mathcal{A}$ is *applicable* to a configuration C if the next operation of p enabled by $\mathcal{A}_p$ in C is of type tp and if case $tp =$ *read*, then v is the value of register x in C. We denote by $s(C)$ the configuration resulting after s is applied to C: the state of $\mathcal{A}_p$ is changed based on C, tp and v and if s writes v' to x in C, then x contains v' in $s(C)$. If s is applicable to C, $s(C)$ is uniquely determined by C, s and $\mathcal{A}$.

If the steps of $\mathcal{A}$ do not depend on the failure detector values, i.e., the types of the steps of $\mathcal{A}$ are only *read* and *write*, we say $\mathcal{A}$ is *asynchronous*.

For each process p, the state of the automaton $\mathcal{A}_p$ includes a read-only input variable, denoted IN_p, and a write-once output variable, denoted OUT_p. In each initial state of $\mathcal{A}_p$, the input variable IN_p contains the input value of p or the special value $\perp$, and the output variable OUT_p is initialized to $\perp$ (to denote that it was not yet written by p). We assume that all initial states agree on the states of shared objects. Therefore, an initial state of $\mathcal{A}$ is unambiguously determined by the vector of IN: $\forall p \in \Pi$, $IN[p] = IN_p$. Therefore, we simply denote an initial state by IN.

2.3 Schedules

A *schedule* S of an algorithm $\mathcal{A}$ is a finite or infinite sequence of steps of $\mathcal{A}$. We denote by $\mathit{participants}(S)$ the set of processes that take at least one step in schedule S. The ith step in schedule S is denoted by $S[i]$. A schedule S is *applicable* to a state C if S is the empty schedule, or

$S[1]$ is applicable to C, $S[2]$ is applicable to $S[1](C)$, etc. If S is finite and is applicable to C, $S(C)$ denotes the configuration that results when we apply schedule S to configuration C.

Let S be a schedule applicable to an initial configuration I of an algorithm $\mathcal{A}$. We say that step $S[i]$ *causally precedes* step $S[j]$ in S if and only if one of the following holds [19]:

- $S[i]$ and $S[j]$ are steps of the same process and $i < j$;
- $S[i]$ is a step in which a process p writes a value v to a register x and $S[j]$ is a step in which a process q reads v from x;[3]
- there exists k such that $S[i]$ causally precedes $S[k]$, and $S[k]$ causally precedes $S[j]$ in S.

2.4 Runs

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 a schedule, and T is a list of values in $\mathbb{T}$ such that:

(1) S is applicable to I.
(2) S and T are both finite sequences of the same length, or are both infinite sequences.
(3) For all $1 \leq j \leq |S|$, if $S[j] = (i, tp, x, v)$, then $p_i \notin F(T[j])$ and if $tp = query$, then $v = H(p, T[j])$.
(4) For all $1 \leq i < j \leq |S|$, $T[i] \leq T[j]$.
(5) For all positive integers $i, j \leq |S|$, if step $S[i]$ causally precedes step $S[j]$ in the set of steps of S then $T[i] < T[j]$.

Property (3) states that a process does not takes steps after crashing, and that the failure detector value seen in a step is the one dictated by the failure detector history H. Property (4) states that the sequence of times when processes take steps in a schedule is non-decreasing, and property (5) states that these times respect causal precedence.

A run $R = \langle F, H, I, S, T\rangle$ is *infinite* if S is infinite. Otherwise, the run is *partial*. For an infinite run R, $inf(R)$ denote the set of processes that take infinitely many steps in S. We say that R is *fair* if $correct(F) = inf(R)$, and *k-resilient* if $|inf(R)| \geq n - k$.

Two runs of $\mathcal{A}$ that agree on the input configurations I and the schedules 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, a set $\mathcal{O}$ of output n-vectors and a total relation Δ that associates each input vector with a set of possible output vectors. If the value of p in a vector $I \in \mathcal{I}$ is $\perp$, then we say that p does not *participate* in I. If the value of p in a vector $O \in \mathcal{O}$ is $\perp$, then we say that p does not *decide* in O. We require that if $(I, O) \in \Delta$, then for all ℓ, $I[\ell] = \perp \Rightarrow O[\ell] = \perp$, and for each O' resulting after replacing some items in O with $\perp$, $(I, O') \in \Delta$. In this paper, we only consider tasks that have finite sets of inputs $\mathcal{I}$.

Let $val(U)$ denote the set of non-$\perp$ values in a vector U. A task $\mathcal{T} = (\mathcal{I}, \mathcal{O}, \Delta)$ is *colorless* [3] if, for all input vectors $I \in \mathcal{I}$ and $I' \in \mathcal{I}$ and all output vectors $O \in \mathcal{O}$ and $O' \in \mathcal{O}$, such that

[3] Without loss of generality, we assume that all values written to the shared memory are unique.

– $(I, O) \in \Delta$,
– $val(I) \subseteq val(I')$,
– $val(O') \subseteq val(O)$,
– and for all ℓ, $I'[\ell] = \perp \Rightarrow O'[\ell] = \perp$,

we have $(I', O') \in \Delta$. Thus, Δ can be defined as a relation in $val(\mathcal{I}) \times val(\mathcal{O})$, where $val(\mathcal{I})$ denotes the set of all possible input values and $val(\mathcal{O})$ denotes the set of all possible output values. In other words, in a solution to $\mathcal{T}$, a process is free to adopt the input and output value of any other participating process.

In the *n-process k-set consensus* task (we simply write k-set consensus), $\mathcal{I}$ and $\mathcal{O}$ consist of all vectors in which non-$\perp$ values are in $\{0, \ldots, k\}$, and $(I, O) \in \Delta$ if and only if $val(O) \subseteq val(I)$, $|val(U)| \leq k$. Obviously, k-set consensus is colorless.

2.6 Solving a task

Consider a task $\mathcal{T} = (\mathcal{I}, \mathcal{O}, \Delta)$ and an environment $\mathcal{E}$. We say that an algorithm $\mathcal{A}$ *solves* $\mathcal{T}$ *in* $\mathcal{E}$ *using* $\mathcal{D}$, if in every fair run $\langle F, H, IN, S, T\rangle$ of $\mathcal{A}$, where $F \in \mathcal{E}$, $IN \in \mathcal{I}$ and for each $p \in$ *participants*(S): $I[p] \neq \perp$, every process in *correct*(F) eventually *decides*, i.e., writes a non-$\perp$ output value to OUT_p, so that the resulting vector OUT satisfies $(IN, OUT) \in \Delta$. We say that $\mathcal{A}$ *weakly solves* $\mathcal{T}$ using $\mathcal{D}$ in $\mathcal{E}$ if it only guarantees that at least one process eventually decides.

It follows immediately from the definition that to solve k-set consensus (or, more generally, any colorless task), it is sufficient to weakly solve it. Indeed, the decided process can simply write its output value to the shared memory and the value can be adopted by every other process without affecting the correctness of decision.

A configuration of an algorithm in which some process decides is called *deciding*.

2.7 Resilience and active resilience

Let $\mathcal{E}_k$ denote the environment that consists of all failure patterns F such that $|faulty(F)| \leq k$ (in which any k processes may fail). A task can be (weakly) solved *k-resiliently* if there is an asynchronous algorithm that (weakly) solves the task in $\mathcal{E}_k$.

Let $\mathcal{E}_k^a$ denote the environment that consists of all failure patterns F such that $|faulty(F) - F(0)| \leq k$ (in which at most k *not initially faulty* processes may fail). A task can be (weakly) solved *actively k-resiliently* if an asynchronous algorithm (weakly) solves the task in $\mathcal{E}_k^a$.

Since $\mathcal{E}_k^a \subseteq \mathcal{E}_k$, if a task is (weakly) solvable actively k-resiliently, then it is also (weakly) solvable k-resiliently. It is shown in [2, 3] that k-set consensus is impossible to solve k-resiliently, and thus also actively k-resiliently.

2.8 Comparing failure detectors

Intuitively, a failure detector $\mathcal{D}'$ is weaker than a failure detector $\mathcal{D}$ if processes can use $\mathcal{D}$ to *emulate* $\mathcal{D}'$; so if a problem is solvable with $\mathcal{D}'$, it can also be solvable it with $\mathcal{D}$.

Intuitively, $\mathcal{D}$ can be used to emulate $\mathcal{D}'$ in an environment $\mathcal{E}$ if there is an algorithm that transforms $\mathcal{D}$ to $\mathcal{D}'$ in $\mathcal{E}$ as follows. The reduction algorithm, denoted $T_{\mathcal{D}\rightarrow\mathcal{D}'}$, uses $\mathcal{D}$ to maintain a variable $\mathcal{D}'$-$output_p$ at every process p. $\mathcal{D}'$-$output_p$ functions as the output of the emulated failure detector $\mathcal{D}'$ at p. For each run R of $T_{\mathcal{D}\rightarrow\mathcal{D}'}$, let O_R be the history of all the $\mathcal{D}'$-*output* variables in

R; i.e., $O_R(p,t)$ is the value of D'-$output_p$ at time t in R. Algorithm $T_{\mathcal{D}\to\mathcal{D}'}$ *transforms* $\mathcal{D}$ *to* $\mathcal{D}'$ *in* $\mathcal{E}$ if and only if for every fair run $R = (F, H, I, S, T)$ of $T_{\mathcal{D}\to\mathcal{D}'}$ using $\mathcal{D}$ in $\mathcal{E}$, $O_R \in \mathcal{D}'(F)$. We say that $\mathcal{D}'$ is *weaker than* $\mathcal{D}$ *in* $\mathcal{E}$ if there is an algorithm $T_{\mathcal{D}\to\mathcal{D}'}$ that transforms $\mathcal{D}$ to $\mathcal{D}'$ in $\mathcal{E}$. with D' in E can also be solved We say that two failure detectors are *equivalent* in $\mathcal{E}$ if each is weaker than the other 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 [18] in every environment.

3 Failure detectors and k-set consensus

In this section, we recall the definition of $\neg\Omega_k$ [21] and show that it is sufficient to solve k-set consensus.

$\neg\Omega_k$ 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 [23]. Also, $\neg\Omega_1$ is equivalent to Ω [4]. 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.

In every environment, $\neg\Omega_{n-1}$ is the weakest failure detector to solve $(n-1)$-set consensus [24].

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:

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

In every environment, $\neg\Omega_k$ and $\overrightarrow{\Omega}_k$ are equivalent [26]. 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 $\overrightarrow{\Omega}_k$. The other direction is a simple generalization of the reduction algorithm for the case $k = n-1$ in [24] (similar, in turn, to the reduction of [8]).

Solving k-set consensus with $\overrightarrow{\Omega}_k$ is straightforward [24]. Just run k instances of Ω-based consensus protocol [20], $C_1, \ldots, C_k$, 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 consensus. The first value to be returned by an instance of consensus is used as the output for k-set consensus. 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 consensus in all environments.*

4 $\neg\Omega_k$ is necessary to circumvent a k-resilient impossibility

This section shows that $\neg\Omega_k$ is necessary to solve any colorless task $\mathcal{T}$ that cannot be solved k-resiliently. We then use this result to characterize tasks based on their set consensus number.

4.1 Overview

Let $\mathcal{T} = (\mathcal{I}, \mathcal{O}, \Delta)$ be a colorless task that cannot be solved k-resiliently. Let $\mathcal{A}$ be an algorithm that solves $\mathcal{T}$ in $\mathcal{E}$ using a failure detector $\mathcal{D}$ in a given environment $\mathcal{E}$. Our goal is to construct a reduction algorithm that, in $\mathcal{E}$, extracts the output of $\neg\Omega_k$ using $\mathcal{A}$ and $\mathcal{D}$. Recall that to extract the output of $\neg\Omega_k$ means to output 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 algorithm using a failure detector 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 [4]. Given such a DAG G, we can construct an *asynchronous* algorithm $\mathcal{A}'$ that, instead of the failure detector, uses G to simulate (possibly finite or unfair) runs of $\mathcal{A}$ [24].

Our reduction algorithm uses the DAG to allow $k+1$ locally maintained *simulators* $q_1, \ldots, q_{k+1}$ to simulate multiple k-resilient runs of $\mathcal{A}'$ For this, the algorithm makes use of the BG-simulation technique [2, 3, 12]. Each simulator q_i accepts an input value in $val(\mathcal{I})$ and produces outputs in $val(\mathcal{O})$ such that the resulting input and output vectors satisfy $(val(I'), val(O')) \in \Delta$. To simulate the first step of a simulated process p'_j, the simulators use BG-agreement to agree on the input value of p'_j, where each simulator proposes its own input value. Since $\mathcal{T}'$ is colorless, the run can be considered *deciding* if at least one simulated process p_i returns an output of $\mathcal{T}$ at some simulator q_j.

Since $\mathcal{T}$ is not k-resiliently solvable and thus $\mathcal{T}'$ is not wait-free solvable, the described BG-simulation must produce at least one infinite non-deciding run that, in turn, corresponds to an infinite k-resilient non-deciding run of $\mathcal{A}'$. To emulate $\neg\Omega_k$, it is thus sufficient to output the set of $n-k$ processes that appear the latest in the *first* such run of $\mathcal{A}'$. By showing that every run of $\mathcal{A}'$ containing infinitely many steps of every correct process is deciding (Theorem 2), we derive that at least one correct process will eventually never be output. Thus, the output of $\neg\Omega_k$ is extracted.

However, there are two issues we have to address here. One issue is how to make sure that eventually each correct process forever selects ever-increasing non-deciding partial runs that are in a strict sense *close* to such an infinite non-deciding run. E.g., arguably the simplest way to order simulated schedules is to use the lexicographic order: schedule $q_1, q_2, ...$ is ordered before schedule $q'_1, q'_2, ...$ if there exists i such that $q_i < q'_i$ (assuming a deterministic order on process ids) and for all $1 \leq i < j$, $q_j = q'_j$. However, always choosing the first non-deciding run using the lexicographic order in the breadth-first exploration of the tree of all possible schedules, we cannot prevent the case of (temporarily) choosing a prefix of a deciding run that contains steps of arbitrary processes, thus causing arbitrary output of the emulated failure detector. We address the issue by making sure that the explored prefixes eventually only include steps of processes that appear infinitely often in some non-deciding run, and we achieve this by employing the *corridor-based* depth-first ordering of runs.

The second issue is that the DAGs constructed at different processes evolve differently. Thus, the first non-deciding k-resilient runs located at different correct processes can also 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* (described in Section 4.2) and the *computation component* (described in

```
   Shared variables:
      for all p_i ∈ Π: V_i, initially (⊥, ⊥, ⊥)

1  ℓ_i := 0
2  while true do
3     for all p_j ≠ p_i do (G_j, α_j, β_j) := V_j; G_i := G_i ∪ G_j
4     d_i := query failure detector D
5     ℓ_i := ℓ_i + 1
6     add vertex [p_i, d_i, ℓ_i] to G_i
      for each vertex v of G_i, v ≠ [p_i, d_i, ℓ_i]:
         add edge (v, [p_i, d_i, ℓ_i]) to G_i
7     V_i := (G_i, α_i, β_i)
```

Fig. 1. Building a DAG: the program code for each process p_i

Section 4.5). 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. In the computation component, every process simulates a set of runs of $\mathcal{A}$ using the DAG and extracts the output of $\neg\Omega_k$.

4.2 Communication component and DAGs.

The communication component of our reduction algorithm 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 specifies the details of how exactly these runs were simulated by p_i. We explain how α_i and β_i are maintained and used in Sections 4.3 and 4.5.

DAG G_i has some special properties which follow from its construction [4]. 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'), $\tau(v) + 2 < \tau(v')$. values are output.
(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 $G_i(t) \subseteq G_j(t')$.

Condition 1b means that every edge in G_i reflects the temporal order in which the failure detector values in v and v' were output. Also, the inequality $\tau(v)+2 < \tau(v')$ means that at least two causally related events could take place between $\tau(v)$ and $\tau(v')$.

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 [4]. 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.

4.3 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 $\beta(p_i, \ell) = 0$ for all $\ell > \ell_i + 1$ where ℓ_i is the largest integer such that $[p_i, d, \ell']$ is in G (if any). 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 accepts no more steps 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.

Note that if G is fixed, then so is the delay map β. On the other hand, if G is concurrently maintained by the algorithm in Figure 1, then the delay map is mutable: the value of $\beta(p_i, \ell)$ may continue growing as long as ℓ-th vertex of p_i is not in G. However, if the vertex is located and p_i went through $\beta(p_i, \ell)$ waiting rounds (lines 13–16) before reaching line 17, $\beta(p_i, \ell)$ stops changing. This helps us in Section 4.5 when we simulate multiple runs of $\mathcal{A}^\beta$ in the presence of concurrently run communication component in Figure 1. In all such simulated runs of $\mathcal{A}^\beta$, whenever a simulated process p'_i reaches a given state of $\mathcal{A}$, it goes through the same number of waiting rounds before it simulates the next step of $\mathcal{A}$.

The following theorem shows that, given fixed G and β, 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, such that $inf(R_\mathcal{A}) = correct(F) \cap inf(R')$. 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. If the simulated state of $\mathcal{A}$ at p_i implies the next step of $\mathcal{A}$ to be a memory step, then the step is performed exactly as in $\mathcal{A}$. If the next step is a failure detector query, then the step is simulated using the parameters G and β.

Shared variables:
  $W_1, \ldots, W_n$, 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 $U := [W_1, \ldots, W_n]$
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 $W_i := [p_i, d, \ell]$
19 take the next step of $\mathcal{A}$ using d as the output of $\mathcal{D}$

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

Let $\mathcal{S}$ be the set of steps of $\mathcal{A}$ simulated in R'. We say that steps in $\mathcal{S}$ are related by the *causal order*, and write $(s, s') \in CO$, if (1) s and s' are both performed by the same process p_i and s is performed before s', or (2) s is a write step, s' is a read step, and s that returns the value written by s (recall that we assume that each written value is unique), or (3) there exists $s'' \in \mathcal{S}$ such that $(s, s'') \in CO$ and $(s'', s') \in CO$.

Consider a query step $s \in \mathcal{S}$, and let $[p_i, d, \ell]$ be the vertex of G that was used for simulating s. Let $\tau(s)$ denote the moment of time at which p_i queried the failure detector in R for the ℓ-th time. Therefore, τ imposes a partial order PO_τ on $\mathcal{S}$: s precedes s' in PO_τ if both are query steps and $\tau(s) < \tau(s')$.

Since PO_τ is defined on query steps in S only, it is sufficient to show that every two query steps $s_0, s_1 \in \mathcal{S}$ such that $(s_0, s_1) \in CO$ satisfy $(s_0, s_1) \in PO_\tau$. This would imply that the transitive closure of $CO \cup PO_\tau$ contains no loops and, thus, there exists a total order TO on steps in $\mathcal{S}$ that is consistent with both CO and PO_τ.

Indeed, consider two query steps s_0 and s_1 simulated by processes p_i and p_j, respectively, such that $(s_0, s_1) \in CO$. Thus, either $p_i = p_j$ and p_i simulated s_0 before s_1, or p_i simulated at least one write step s'_0 after s_0, and p_j simulated at least one read step s'_1 before s_1, such that the memory access of s'_1 took place before the memory access of s'_0 in R. By the algorithm, before simulating step s_1, p_j must have found $[p_i, d_i, \ell]$ or a later vertex in W_i (line 9) and, thus, the vertex of G used for simulating s_0 must be a descendant of $[p_j, d_j, \ell']$. By property (1) of DAGs, $\tau(s_0) < \tau(s_1)$, and thus $(s_0, s_1) \in PO_\tau$.

Thus, there exists TO, a total order on $\mathcal{S}$, which is consistent with both CO and PO_τ. Conformance with CO implies that the resulting schedule is indistinguishable from the schedule simulated by R'. Conformance with PO_τ implies that the resulting schedule indeed belongs to a run of $\mathcal{A}$. In other words, the sequence of steps of $\mathcal{A}$ simulated in R *could have happened* in some run $R_\mathcal{A}$ of $\mathcal{A}$ with failure pattern F and input vector I. The schedule of $R_\mathcal{A}$ is the steps in $\mathcal{S}$ ordered by TO and the increasing time values are of steps in this schedule are chosen consistently with τ. Note that

property (1b) of runs implies that if $(s_0, s_1) \in CO \cup PO_\tau$, then $\tau(s_0) + 2 < \tau(s_1)$ and thus, such a choice of time values is possible.

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

4.4 BG-simulation

BG-simulation is a 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 [2, 3]. Informally, the technique operates 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 $n-k$ 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 [2, 3]:

(BG1) Let σ' be the shortest prefix of σ that includes all steps the simulators in $faulty(\sigma)$ take in σ. Then for any infinite 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.

The BG-simulation technique allows for reducing the question of k-resilient solvability of an n-process colorless task $\mathcal{T} = (\mathcal{I}, \mathcal{O}, \Delta)$ defined for processes $p_1, \ldots, p_n$, to *wait-free* solvability of a $k+1$-process task $\mathcal{T}'$ defined for simulators $q_1, \ldots, q_{k+1}$. As an input in task $\mathcal{T}'$, each simulator q_i obtains a value in $val(\mathcal{I})$, and, as an output in $\mathcal{T}'$, q_i produces a value in $val(\mathcal{O})$ such that $(val(I'), val(O')) \in \Delta$, where I' and O' are the input and output vectors in the run, respectively. $\mathcal{T}$ is solvable k-resiliently if and only if $\mathcal{T}'$ is solvable wait-free [2, 3].

4.5 Extracting $\neg\Omega_k$: the reduction algorithm

In our reduction algorithm, every process simulates a system of $k+1$ BG-simulators $q_1, \ldots, q_{k+1}$. Every run of BG-simulation on $q_1, \ldots, q_{k+1}$ produced at process p_i simulates a k-resilient run of $\mathcal{A}^{\beta_i}$ using the concurrently maintained DAG G_i and delay map β_i. In turn, this run of $\mathcal{A}^{\beta_i}$ simulates a unique run of $\mathcal{A}$. The levels of simulation that takes place at every process p_i are summarized in

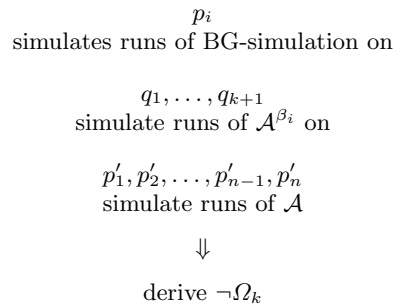

Fig. 3. Three levels of simulation.

Figure 3. To avoid confusion, for all $1 \leq j \leq n$, p'_j denotes here the process that represents p_j in the local simulation.

The computational component of our reduction algorithm is presented in Figure 4. The emulated output of $\neg\Omega_k$ at process p_i is maintained in a local variable $\neg\Omega_k$-$output_i$: every query of $\neg\Omega_k$ performed by p_i simply returns the current value of the variable. Each initial state I for task $\mathcal{T}'$ and each *schedule* σ, a sequence specifying the order in which simulators $q_1, \ldots, q_{k+1}$ take steps of BG-simulation, determine a unique run of $\mathcal{A}^{\beta_i}$, denoted by $\alpha_i(I, \sigma)$. For brevity, $dom_I(\alpha_i)$ denotes here all distinct (not related by containment) schedules σ explored (via recursive function *explore*) 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.

For all input vectors I of $\mathcal{T}$ (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 some infinite schedule $\tilde{\sigma}$ that corresponds to a non-deciding k-resilient run of $\mathcal{A}^{\beta_i}$. Otherwise, by Theorem 2 and the equivalence result in [2, 3], we would obtain a wait-free solution for $\mathcal{T}'$, and thus a k-resilient solution for task $\mathcal{T}$ — a contradiction.

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 simulators in $q_1, \ldots, q_{k+1}$. 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

```
20  for all I_0, input vectors of T' (in a deterministic order <) do      { For all possible inputs for q_1, ..., q_{k+1} }
21    explore(I_0, ⊥, {q_1, ..., q_{k+1}})

22  function explore(I, σ, S)
23    ¬Ω_k-output_i := n − k processes that appear the latest in α_i(I, σ)
                (if possible, any n − k processes otherwise, 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 schedules explored so far are deciding at some p_j }
25      α_i := α_j; β_i := β_j; G_i := G_j ∪ G_i        {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          add σ · q_j to dom_I(α_i); compute α_i(I, σ · q_j)
30          explore(I, σ · q_j, S')
```

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

maintain a "corridor" (the third parameter in function *explore* in Figure 4) — the set of simulators that can be 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). E.g., 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" extensions, etc. If all extensions within the chosen sub-corridor turn out to be deciding, the next sub-corridor is selected, etc.[4]

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.

Correctness. Consider any fair run of the algorithm in Figures 1, 2 and 4. Let F be the failure pattern of that run. First we prove the following auxiliary lemmas:

Lemma 1. *If the currently simulated schedule σ is found deciding by a process p_i with the current input vector I (line 23), then $\forall J \in \mathcal{I}$, $\forall \sigma' \in dom_J(\alpha_i)$, the execution of $\mathcal{A}^{\beta_i}$ determined by J and σ' is deciding.*

[4] Similar ordering was implicitly used in [24] for simulated executions on processes $p_1, ..., p_n$. Here we apply the ordering to simulators $q_1, \ldots, q_{k+1}$.

Proof. Suppose that, at some time t, a schedule σ' was witnessed deciding with an input vector J by some process p_j. Recall that the values of β_j stops changing for the vertices of G_j that were used in the simulated run of $\mathcal{A}^{\beta_j}$. Moreover, β_j records the number of waiting rounds that were made for vertices of that *were not* found in G_j used in the simulated run of $\mathcal{A}^{\beta_j}$ (line 15 in Figure 2). Any *extension* of β_j (a delay map maintained by any process that adopted β_j in line 25 in Figure 4) can only increase the values corresponding to these vertices. Thus, at any later time, σ', J, and extensions of β_j and G_j would result in the same simulated run of $\mathcal{A}$.

The domain of mapping α_i is constantly growing, and a process considers a new schedule σ in line 24 to be deciding with the current input vector I only if every distinct schedule σ' considered up to now was witnessed deciding with I by some process p_j that constructed the current value of β_i. Thus, p_i would find σ' deciding with the current value of β_i too.

Thus, even though the delay map β_i evolves, at any moment of time, all previously explored schedules and input vectors result in deciding executions of $\mathcal{A}$ with the *current* value of β_i.

Lemma 2. *If explore(I, σ, S) invoked by a correct process p_i in line 21 or 30 returns, then it returns at every correct process.*

Proof. Suppose *explore*(I, σ, 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 3. *There exists an input vector $\bar{I}$, such that each correct process p_i eventually invokes explore$(\bar{I}, \perp, \{q_1, \ldots, q_{k+1}\})$ in line 21 and the invocation never returns.*

Proof. Suppose that, at some correct process p_i, *explore*$(I, \perp, \{q_1, \ldots, q_{k+1}\})$ 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 1, 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 $\mathcal{A}^{\beta}$ 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, $q_1, \ldots, q_{k+1}$ can wait-free solve $\mathcal{T}'$ as follows. Each simulator q_i registers its input value of $\mathcal{T}'$ in the shared memory and then 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 proposes its own input value as the input value of p'_j in a BG-agreement protocol.

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 (which is sufficient for every other process to decide). 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 algorithm $\mathcal{A}$, and the inputs are provided by $q_1, \ldots, q_{k+1}$, the output satisfies the specification of $\mathcal{T}$ — a contradiction

Thus, there exists $\bar{I}$ such that *explore*$(\bar{I}, \perp, \{q_1, \ldots, q_{k+1}\})$ invoked by a correct process p_i never returns. By the algorithm, the invocation previously returned for all input vectors $J < \bar{I}$. By Lemma 2, *explore*$(\bar{I}, \perp, \{q_1, \ldots, q_{k+1}\})$ is also invoked by every other correct process and never returns.

Lemma 4. *There exists an infinite schedule $\tilde{\sigma}$ and an input vector I such that eventually, all correct processes perform the same infinite sequence of recursive invocations of explore with parameters (I, σ_0, S_0), (I, σ_1, S_1), …, $(I, \sigma_{\ell'}, \mathit{live}(\tilde{\sigma}))$, $(I, \sigma_{\ell'+1}, \mathit{live}(\tilde{\sigma}))$, … that never return.*

Proof. By Lemma 3, eventually, every correct process p_i invokes *explore*$(I, \perp, \{q_1, \ldots, q_{k+1}\})$ 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 2, 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 schedule that is never considered deciding with I by any process p_i. Now we show that for every such schedule $\tilde{\sigma}$, we have $\tilde{S} = \mathit{live}(\tilde{\sigma})$, the set of processes that appear infinitely often in $\tilde{\sigma}$.

By construction (line 28), $\mathit{live}(\tilde{\sigma}) \subseteq \tilde{S}$. Suppose, by contradiction, that $\mathit{live}(\tilde{\sigma})$ is a proper subset of $\tilde{S}$. Let $S'_0, S'_1, \ldots$ be the sequence of non-empty subsets of $\{q_1, \ldots, q_{k+1}\}$ such that $\forall 0 \leq \ell < \ell'$: $S'_\ell = S_\ell$ and $\forall \ell \geq \ell'$: $S_\ell = \mathit{live}(\tilde{\sigma})$. 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, σ_0, S_0), *explore*(I, σ_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, eventually, all correct processes perform the same infinite sequence of recursive invocations of *explore* with parameters (I, σ_0, S_0), (I, σ_1, S_1), …, $(I, \sigma_{\ell'}, \mathit{live}(\tilde{\sigma}))$, $(I, \sigma_{\ell'+1}, \mathit{live}(\tilde{\sigma}))$, ….

Theorem 3. *Let $\mathcal{E}$ be any environment and $\mathcal{T}$ be any colorless task that cannot be solved k-resiliently. Let $\mathcal{D}$ be a failure detector that solves $\mathcal{T}$ in $\mathcal{E}$. Then $\neg\Omega_k$ is weaker than $\mathcal{D}$ in $\mathcal{E}$.*

Proof. By Lemma 4, eventually, every correct process goes through the same infinite sequence of recursive never-returning invocations of *explore* with parameters (I, σ_0, S_0), (I, σ_1, S_1), …, $(I, \sigma_{\ell'}, \mathit{live}(\tilde{\sigma}))$, $(I, \sigma_{\ell'+1}, \mathit{live}(\tilde{\sigma}))$, …. Here $\tilde{\sigma}$ is an infinite schedule such that no process ever considers a prefix of $\tilde{\sigma}$ to be deciding with I.

Let σ' be the shortest prefix of $\tilde{\sigma}$ that contains *all* appearances of the simulators that appear only finitely often in $\tilde{\sigma}$, $\mathit{faulty}(\tilde{\sigma})$. Eventually, all correct processes simulate extensions of σ' in which simulators in $\mathit{faulty}(\tilde{\sigma})$ do not appear. Let W be the set of simulated processes in $\{p'_1, \ldots, p'_n\}$ that are *blocked* in $\tilde{\sigma}$ (Section 4.4). We argue that W contains at least one process p'_j such that $p_j \in \mathit{correct}(F)$.

Consider $p_j \in \mathit{correct}(F)$. The communication component described in Figure 1 makes sure that, for all $\ell \in \mathbb{N}$, the DAG G_i maintained at each correct process p_i eventually contains a vertex $[p_j, d, \ell]$. Thus, in simulating runs of $\mathcal{A}^{\beta_i}$ using the algorithm in Figure 2, every correct process p_i eventually finds the vertex $[p_j, d, \ell]$ in G_i and stops incrementing $\beta_i(p_j, \ell)$. Hence, for all $\ell \in \mathbb{N}$, $\beta_i(p_j, \ell)$ maintained at each correct process p_i eventually stops growing.

Also, each faulty process p_i eventually stops updating its α_i and β_i in the shared memory. Thus, there is a time after which no correct process adopts delay maps of faulty processes in line 25.

Therefore, there exists a delay map β defined on the limit infinite DAG $\tilde{G}$ such that, eventually, every finite prefix $\tilde{\sigma}$ stabilizes on simulating the same finite run of $\mathcal{A}^{\beta}$.

By contradiction, suppose that no process in $correct(F)$ is blocked in $\tilde{\sigma}$. Thus, every correct p_i simulates a run of $\mathcal{A}^{\beta}$ in which every p'_j such that $p_j \in correct(F)$ appears arbitrarily often. But by Theorem 2, eventually, the run of $\mathcal{A}^{\beta}$ 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 with the assumption that no prefix of $\tilde{\sigma}$ is ever considered deciding with I.

Thus, there exists $p'_j \in W$ such that $p_j \in correct(F)$. 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.

The fact that k-set consensus cannot be solved k-resiliently, combined with Theorem 1 and 3, implies:

Corollary 1. *For all environments $\mathcal{E}$, $\neg\Omega_k$ is the weakest failure detector to solve k-set consensus in $\mathcal{E}$.*

5 Set consensus number: categorizing distributed tasks

This section presents the sufficiency part of our result: every $(k-1)$-resiliently solvable task can be solved (in every environment) if, in addition to read-write registers, we are allowed to use $\neg\Omega_k$.

Let a task $\mathcal{T}$ be actively $(k-1)$-resilient solvable, and let $\mathcal{A}$ be the corresponding algorithm. Intuitively, if k or less processes participate in the computation, we can simply use $\mathcal{A}$ to solve the task. However, we also need to account for the case when $k+1$ or more processes participate and more than k of them can fail. This is exactly the case when we use $\neg\Omega_k$ (or, more precisely, the equivalent failure detector $\overrightarrow{\Omega}_k$). Our goal is to *simulate* an execution of $\mathcal{A}$ in which at most $k-1$ participating processes fail, and thus some process must eventually decide.

First we describe an abstract simulation technique that uses $\overrightarrow{\Omega}_k$ to simulate, in a system of n processes, a run of an arbitrary asynchronous k-process algorithm $\mathcal{A}'$.

Then we apply this technique to show that, in every environment, we can use $\overrightarrow{\Omega}_k$ to simulate an execution e of $\mathcal{A}$ such that (1) e only contains steps of participating processes, (2) at least one simulated process takes infinitely many steps, (3) at most $k-1$ participating processes fail. Therefore, the simulated execution of $\mathcal{A}$ must be deciding, which is sufficient to solve the task.

5.1 Simulating k codes using $\neg\Omega_k$

Suppose we are given a read-write algorithm $\mathcal{A}'$ on k processes, $p'_1, \ldots, p'_k$. Suppose that $\mathcal{A}'$ requires no inputs. Assuming that $\overrightarrow{\Omega}_k$ is available, the algorithm in Figure 5 describes how n *simulators*, $p_1, \ldots, p_n$ can simulate an infinite execution of $\mathcal{A}'$.

The simulation is similar in spirit to BG-simulation [2, 3]. Every simulator p_i first registers its participation in the shared memory and then tries to advance simulated processes $p'_1, \ldots, p'_{\min(k,m)}$, where m is the number of simulators that p_i has witnessed participating.

```
Shared variables:
    R_j, j = 1,...,n, initially ⊥
    V_j, j = 1,...,k, initially the initial state of p'_j
Local variables:
    Leader_j, j = 1,...,k, initially p_1
    ℓ_j, j = 1,...,k, initially 1
    v_j, j = 1,...,k, initially ⊥

Task 1:
31  R_i := 1
32  for j = 1,...,k do v_j := {V_1,...,V_k}
33  while undecided do
34     for j = 1,...,min(|pars|, k) do
35        perform one more step of Cons_{j,ℓ_j}(v_j) using Leader_j as a leader
36        if Cons_{j,ℓ_j}(v_j) returns v then          { The next state of p'_j is decided }
37           V_j := v          { Adopt the decided state of p'_j }
38           simulate the next step of p'_j in A'
39           if v allows p_i to decide then          { The simulator can depart }
40              undecided := false
41              R_i := ⊥
42           v_j := {V_1,...,V_k}          { Evaluate the next state of p'_j }
43           ℓ_j := ℓ_j + 1

Task 2:
44  while true do
45     pars := {p_j, R_j ≠ ⊥}
46     if |pars| ≤ k then
47        for j = 1,...,|pars| do Leader_j := the j-th smallest process in pars
48     else
49        for j = 1,...,k do Leader_j := Ω⃗_k[j]
```

Fig. 5. Simulating k codes using vector-Ω_k: the program code for simulator p_i

To simulate a step of p'_j, simulators agree on the view of the process after performing the step. However, instead of the BG-agreement protocol of [2, 3], we use here a leader-based consensus algorithm [4]. This algorithm terminates under the condition that all processes eventually agree on the same correct leader. The instance of the consensus algorithm used to simulate ℓ-th step of process p'_j is denoted by $Cons_{j,\ell}$.

As long as the number of participating simulators is k or less, the participating simulator with the j-th smallest id acts as a leader for simulating steps of p'_j. When the number of participating simulators exceeds k, the leader for simulating steps of p'_j is given by the output of $\overrightarrow{\Omega}[j]$.

In both cases, at least one simulated process is eventually associated with the same correct leader. Thus, at least one simulated process makes progress in the simulation.

The algorithm also assumes that a simulator p_i may decide to leave the simulation if the simulated execution produced a desired output (line 41). We use this option in the next section.

Theorem 4. *In every environment, the protocol in Figure 5 simulates an infinite execution of any k-process algorithm $\mathcal{A}'$. Moreover, if m simulators participate, i.e., $|pars| = m$, then at most $\min(k, m)$ processes participate in the simulated execution.*

Proof. Consider an infinite run of the algorithm. Since every next state of each simulated process p'_j is decided using a consensus algorithm, every simulator observes exactly the same evolution of states for every simulated process. Thus, the simulated schedule indeed belongs to a run of $\mathcal{A}'$

Now consider the construction of variables $Leader_1$, ..., $Leader_k$ used by the consensus algorithms $Cons_{1,\ell}$, ..., $Cons_{k,\ell}$ (lines 44-49). Let m be the number of participating simulators.

If $m \leq k$, the simulator with the j-th smallest id in *pars* is assigned to be the leader of exactly one simulated process p'_j. Since at least one simulator is correct, there exists p'_j ($j = 1, \ldots, |pars|$) such that all instances $Cons_{j,\ell_j}$ using L_j eventually terminate. Thus, p'_j accepts infinitely many steps in the simulated execution.

If $m > k$, at least one L_j ($j = 1, \ldots, k$) eventually stabilizes on some correct process id, as guaranteed by the properties of $\overrightarrow{\Omega}_k$. Again, p'_j takes infinitely many steps in the simulated execution.

In both cases, at most $\min(m, k)$ simulated processes appear in the produced execution of $\mathcal{A}'$, and at least one simulated process takes infinitely many steps.

5.2 Active k-resilience and $\neg\Omega_k$

Theorem 5. *Let $\mathcal{T}$ be any actively $(k-1)$-resiliently solvable task. In every environment $\mathcal{E}$, $\mathcal{T}$ can be solved using $\neg\Omega_k$.*

Proof. Let $\mathcal{A}$ be the algorithm that solves $\mathcal{T}$ actively $(k-1)$-resiliently. We simply employ the simulation protocol in Figure 5 (Theorem 4), and suppose that the simulated algorithm $\mathcal{A}'$ is BG-simulation [2, 3]. More precisely, $\mathcal{A}'$ simulates algorithm $\mathcal{A}$ of participating processes in $p_1, \ldots, p_n$.

The double simulation is built as follows. Every process p_i writes its input value of $\mathcal{T}$ to the shared memory and starts the simulation of k processes $p'_1, \ldots, p'_k$ using the algorithm in Figure 5 running, in turn, BG-simulation of $\mathcal{A}$ on n processes $p''_1, \ldots, p''_n$.

Each simulated process p''_j is simulated only if the corresponding p_j has written its input of $\mathcal{T}$ in the shared memory and p''_j has not yet obtained an output in the simulated execution.

When p''_j obtains an output, the corresponding simulator p_j considers itself "decided" (line 39) and writes $\perp$ in R_i (line 41) and stops taking steps in the algorithm in Figure 5 (but continues to take steps in algorithms $Cons_{s,r}$ in case it is elected $Leader_r$).

Thus, as long as m processes $\{p_{\ell_1}, \ldots, p_{\ell_m}\}$ participate, only m processes $\{p''_{\ell_1}, \ldots, p''_{\ell_m}\}$ take steps in the resulting simulated execution.

Moreover, as long as m processes in $\{p_1, \ldots, p_n\}$ participate, at most $\min(m, k)$ BG-simulators in $\{p'_1, \ldots, p'_k\}$ take steps and at least one of them takes infinitely many steps. The double simulation results in an execution of $\mathcal{A}$ in which at most $\min(k, m) - 1$ out of m participating processes fail.

Recall that $\mathcal{A}$ solves $\mathcal{T}$ actively k-resiliently. Since in the simulated execution at most k participating simulated processes fail and at least one simulated process takes infinitely many steps, some process p_i must eventually decide. As soon as the decided process p_i departs by writing $\perp$ to R_i, we have one simulator p_i and one simulated process p''_i less. The argument is repeated, as long as there is at least one correct participating simulator with an output. Thus, we obtain an algorithm that, in every environment, solves $\mathcal{T}$.

5.3 Set consensus number

The *set consensus number* of a task $\mathcal{T}$ is the largest $k \in \{1, \ldots, n\}$ such that $\mathcal{T}$ can be solved $(k-1)$-resiliently.

First, we observe that a colorless task with set consensus number k can be solved *actively* $(k-1)$-resiliently.

Theorem 6. *Let $\mathcal{T}$ be any colorless task. If $\mathcal{T}$ can be solved $(k-1)$-resiliently, then $\mathcal{T}$ can be solved actively-$(k-1)$-resiliently.*

Proof. Now let $\mathcal{A}$ be an algorithm that solves $\mathcal{T}$ $(k-1)$-resiliently. We build an algorithm $\mathcal{A}'$ that solves $\mathcal{T}$ actively $(k-1)$-resiliently as follows. In $\mathcal{A}'$, each process p_i acts as a BG-simulator of the system of n processes p'_1, ..., p'_n running $\mathcal{A}$. When simulating the first step of p'_j, processes agree on its input value in the simulated execution, each proposing its own input values. Since $\mathcal{T}$ is colorless, every input value can be adopted by any participating process, and thus the algorithm indeed simulates an execution of $\mathcal{A}$. Since in the resulting execution, at most $k-1$ simulated processes take only finitely many steps, eventually some simulated process p'_j decides, and, since $\mathcal{T}$ is colorless, the value can be returned by every participating process p_i.

Now Theorems 3, 5 and 6 imply:

Corollary 2. *Let $\mathcal{T}$ be any colorless task with set consensus number k. Then, in every environment $\mathcal{E}$, the weakest failure detector to solve $\mathcal{T}$ in $\mathcal{E}$ is $\neg\Omega_k$.*

Hence, in any environment $\mathcal{E}$, any colorless task $\mathcal{T}$ is equivalent to some k-set consensus in the failure detector sense: a failure detector $\mathcal{D}$ solves $\mathcal{T}$ in $\mathcal{E}$ if and only if $\mathcal{D}$ solves k-set consensus in $\mathcal{E}$.

6 Related work

The notion of the weakest failure detector was introduced by Chandra et al [4] who showed that Ω, the failure detector that eventually outputs the identifier of the same correct process, is the weakest failure detector to solve consensus (1-set consensus) in the message-passing model with a majority of correct processes. An extension of this result to the read-write shared memory model appears in [20, 15].

The task of k-set consensus was introduced in [6], and shown to be k-resilient impossible in [17, 22, 2, 3].

Zieliński [23, 24] introduced anti-Ω ($\neg\Omega_{n-1}$ in our notation) and proved that it is the weakest failure detector to solve $(n-1)$-set consensus, for every environment. The result has been generalized to the case of $\neg\Omega_k$ and k-set consensus in [14]. Delporte et al. [10] claimed to have concurrently derived the same result. Anta et al.[1] showed that $\neg\Omega_k$ is the weakest failure detector for solving k-set consensus in the special case of k-resilient environment $\mathcal{E}_k$.

Our reduction algorithm employed the BG-simulation technique [2, 3]. The DAG-based simulation framework follows the strategy proposed in [4]. Unlike [4], however, the computation component of our algorithm does not use the task specification explicitly and bases solely on the fact that the given task is k-resilient impossible.

This paper combines results appeared previously in [14, 13]. $\neg\Omega_k$ was originally shown to be the weakest failure detector for k-set agreement in [14]. In [13], the result was extended to a larger class of tasks using unpublished work [12]. More precisely, the result in [13] relies on the equivalence of k-set agreement and active k-resilience informally stated in [12]. To make the paper self-consistent, here we derive a strictly weaker result that any actively $(k-1)$-resiliently solvable task can be solved using $\neg\Omega_k$ (Theorem 5). Our constructions explicitly use the properties of $\neg\Omega_k$, and not a solution to k-set consensus as a "black box" as in [12].

7 Concluding remarks

Viewed collectively, our results imply that colorless n-process distributed tasks can be categorized into n equivalence classes, $C(1), \ldots, C(n)$. For each $k = 1, \ldots, n$, the weakest failure detector for solving any task in class $C(k)$ is $\neg\Omega_k$. Class $C(1)$ consists of *universal* tasks: whenever a universal task is solvable, any other task is solvable [16]. Class $C(n)$ consists of *trivial* tasks that can be solved asynchronously. More generally, a task $\mathcal{T}$ is in class k is equivalent to k-set consensus: any failure detector that solves $\mathcal{T}$ can solve k-set consensus, and vice versa. The classes are totally ordered in decreasing strength: any failure detector that solves a task $\mathcal{T}$ in class $C(k)$ solves any task in classes $C(k')$, $k' \geq k$.

The techniques presented in this paper can be used to show a more general result: $\neg\Omega_k$ is the weakest failure detector for solving every task that can be solved actively-$(k-1)$-resiliently but not weakly k-resiliently. Indeed, the necessity of Section 4 can be extended to general tasks that cannot be solved weakly-(k-resiliently using Extended BG-simulation [12]. The sufficiency part in Section 5 works for all tasks (not necessarily colorless) that can be solved actively $(k-1)$-resiliently. However we could not find a convincing example of a "colored" (not colorless) task $\mathcal{T}$ for which such a "threshold" k exists.

Characterizing solvability of "colored" tasks is generally an open question. Consider, for instance, a task $\mathcal{T}^{k+1,k}$ which requires each subset of $k+1$ processes to solve $(k+1,k)$-set consensus ($\mathcal{T}^{2,1}$, the consensus variant of this task, was considered in [9]). This task is weakly solvable k-resiliently. Moreover, there is an algorithm which makes at least $n-k-1$ processes decide in every k-resilient run: every process tries to solve, one by one, k-set consensus for each set of $k+1$ processes it belongs to. For each k-set consensus, the process uses the k-BG-agreement protocol, a variant of BG-agreement [2, 3] that returns at most k proposed values and guarantees termination under the condition that at most $k-1$ processes that invoke the protocol fail. Since at most k processes can fail, at most one of these protocols (for some set S of $k+1$ processes) can block forever, and thus, at least $n-k-1$ processes that do not belong to S will output.

Note that the algorithm described above also solves $\mathcal{T}^{k,k+1}$ actively $(k-1)$-resiliently: if at most $k-1$ participating processes fail, no k-BG-agreement will block and every participating process will eventually output. On the other hand, $\mathcal{T}^{k,k+1}$ cannot be solved actively k-resiliently: otherwise, we obtain a wait-free solution to $(k+1)$-process k-set consensus, contradicting [17, 22, 2]. Thus, $\mathcal{T}^{k+1,k}$ does not belong to any of the classes $C(1), \ldots, C(n)$.

Therefore, the techniques developed in this paper do not allows us to show that $\neg\Omega_k$ is necessary to solve $\mathcal{T}^{k+1,k}$. This, once proved, would give a natural generalization of the fundamental equivalence result [9] that solving consensus among any pair of processes requires exactly the same amount of synchrony as solving consensus among all processes (Ω).

The conjecture that the weakest FD for solving any given task is in $\{\neg\Omega_1, \ldots, \neg\Omega_n\}$ and, thus, every task is equivalent to some form of set consensus does not, unfortunately, hold. There are many tasks that cannot be solved with $\neg\Omega_k$ but do not require $\neg\Omega_{k-1}$ to be solved [25], e.g., a task which only requires to solve k-set agreement among a given subset of $k+1$ processes. Characterizing tasks for which the conjecture holds is therefore an interesting open question.

So we have here a number of very interesting open questions which, once answered, can imply a more general categorization of distributed tasks.

References

1. A. F. Anta, S. Rajsbaum, and C. Travers. Weakest failure detectors via an egg-laying simulation (brief announcement). In *PODC*, 2009.
2. E. Borowsky and E. Gafni. Generalized FLP impossibility result for t-resilient asynchronous computations. In *STOC*, pages 91–100. ACM Press, May 1993.
3. E. Borowsky, E. Gafni, N. A. Lynch, and S. Rajsbaum. The BG distributed simulation algorithm. *Distributed Computing*, 14(3):127–146, 2001.
4. T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *J. ACM*, 43(4):685–722, July 1996.
5. T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, Mar. 1996.
6. S. Chaudhuri. Agreement is harder than consensus: Set consensus problems in totally asynchronous systems. In *PODC*, pages 311–324, Aug. 1990.
7. S. Chaudhuri. More *choices* allow more *faults*: Set consensus problems in totally asynchronous systems. *Information and Computation*, 105(1):132–158, 1993.
8. F. Chu. Reducing Ω to $\Diamond W$. *Information Processing Letters*, 67(6):298–293, Sept. 1998.
9. C. Delporte-Gallet, H. Fauconnier, and R. Guerraoui. Tight failure detection bounds on atomic object implementations. *J. ACM*, 57(4), 2010.
10. C. Delporte-Gallet, H. Fauconnier, R. Guerraoui, and A. Tielmann. The disagreement power of an adversary (brief announcement). In *PODC*, 2009.
11. E. Gafni. The extended BG-Simulation. In *STOC*, 2009. Available at: http://www.cs.ucla.edu/∼eli/eli/230-gafni1.pdf.
12. E. Gafni and R. Guerraoui. Simulating few by many: Limited concurrency=set consensus. Unpublished manuscript, available at http://www.cs.ucla.edu/∼eli/eli/kconc.pdf, May 2009.
13. E. Gafni and P. Kuznetsov. On set consensus numbers. In *DISC*, pages 35–47, 2009.
14. E. Gafni and P. Kuznetsov. The weakest failure detector for solving k-set agreement. In *PODC*, 2009. Full version: http://www.net.t-labs.tu-berlin.de/∼petr/pubs/wfd-kset.pdf.
15. R. Guerraoui and P. Kuznetsov. Failure detectors as type boosters. *Distributed Computing*, 20(5):343–358, 2008.
16. M. Herlihy. Wait-free synchronization. *ACM Trans. Prog. Lang. Syst.*, 13(1):123–149, Jan. 1991.
17. M. Herlihy and N. Shavit. The asynchronous computability theorem for t-resilient tasks. In *STOC*, pages 111–120, May 1993.
18. P. Jayanti and S. Toueg. Every problem has a weakest failure detector. In *PODC*, pages 75–84, 2008.
19. L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.
20. W.-K. Lo and V. Hadzilacos. Using failure detectors to solve consensus in asynchronous shared memory systems. In *WDAG*, LNCS 857, pages 280–295, Sept. 1994.
21. M. Raynal. K-anti-Omega, August 2007. Rump session at PODC 2007.
22. M. Saks and F. Zaharoglou. Wait-free k-set agreement is impossible: The topology of public knowledge. In *STOC*, pages 101–110. ACM Press, May 1993.
23. P. Zieliński. Automatic classification of eventual failure detectors. In *DISC*, 2007.
24. P. Zieliński. Anti-omega: the weakest failure detector for set agreement. In *PODC*, Aug. 2008.
25. P. Zieliński. Sub-consensus hierarchy is false (for symmetric, participation-aware tasks). https://sites.google.com/site/piotrzielinski/home/symmetric.pdf?attredirects=0, November 2009.
26. P. Zieliński. Anti-*omega*: the weakest failure detector for set agreement. *Distributed Computing*, 22(5-6):335–348, 2010.