Failure Detectors as Type Boosters ⋆

Rachid Guerraoui and Petr Kouznetsov

Distributed Programming Laboratory, EPFL
CH-1015, Switzerland

Abstract. The power of an object type T can be measured as the maximum number n of processes that can solve consensus using only objects of T and registers. This number, denoted $cons(T)$, is called the *consensus power* of T. This paper addresses the question of the weakest failure detector to solve consensus among a number $k > n$ of processes that communicate using shared objects of a type T with consensus power n. In other words, we seek for a failure detector that is sufficient and necessary to "boost" the consensus power of a type T from n to k. It was shown in [21] that a certain failure detector, denoted Ω_n, is sufficient to boost the power of a type T from n to k, and it was conjectured that Ω_n was also necessary. In this paper, we prove this conjecture for *one-shot deterministic* types. We show that, for any one-shot deterministic type T with $cons(T) \leq n$, Ω_n is necessary to boost the power of T from n to any $k > n$. Our result generalizes, in a precise sense, the result of the weakest failure detector to solve consensus in asynchronous message-passing systems [6]. As a corollary of our result, we show that Ω_n is the weakest failure detector to boost the resilience of a system of $(n-1)$-resilient objects of any types and wait-free registers with respect to the consensus problem.

1 Introduction

Background. Key agreement problems, such as consensus, are not solvable in an asynchronous system where processes communicate solely through registers (i. e., read-write shared memory), as long as one of these processes can fail by crashing [8, 10, 20]. Circumventing this impossibility has sparked off two research trends:

(1) Augmenting the system model with *synchrony* assumptions about relative process speeds and communication delays [9]. Such assumptions could be encapsulated within a *failure detector* abstraction [7]. In short, a failure detector uses the underlying synchrony assumptions to provide each process with (possibly unreliable) information about the failure pattern, i. e., about the crashes of other processes. A major milestone in this trend was the identification of the weakest failure detector to solve consensus in the asynchronous read-write model [6, 18]. This failure detector, denoted Ω, outputs one process at every process so that, eventually, all correct processes detect the same correct process. The very fact that Ω is the weakest to solve consensus means that (a) consensus can be solved using Ω, and (b) any failure detector that can be used

⋆ This paper is a revised and extended version of a paper in the Proceedings of the 17th International Symposium on Distributed Computing (DISC 2003), entitled "On failure detectors and type boosters".

to solve consensus can be transformed into Ω. In a sense, Ω encapsulates the minimum *amount of synchrony* needed to solve consensus among any number of processes communicating through registers.

(2) Augmenting the system model with more powerful communication primitives, typically defined through shared object types with sequential specifications [13, 20]. It has been shown, for instance, that consensus can be solved among any number of processes if objects of the compare&swap type can be used [13]. A major milestone in this trend was the definition of the power of an object type T, denoted $cons(T)$, as the maximum number n of processes that can solve consensus using only objects of T and registers. For instance, the power of the register type is simply 1 whereas the compare&swap type has power ∞. An interesting fact here is the existence of types with intermediate power, like test-and-set or queue, which have power 2 [13, 20].

Motivation. At first glance, the two trends appear to be fundamentally different. Failure detectors encapsulate synchrony assumptions and provide information about failure patterns, but cannot however be used to communicate information between processes. On the other hand, objects with sequential specifications can be used for inter-process communication, but they do not provide any information about failures. It is intriguing to figure out whether these trends can be effectively combined [21]. Indeed, in both cases, the goal is to augment the system model with abstractions that are powerful enough to solve consensus, and it is appealing to determine whether abstractions from different trends add up. For instance, one can wonder whether the weakest failure detector to solve consensus using registers and queues is strictly weaker than Ω.

One way to effectively combine the two trends is to determine a failure detector hierarchy, $\mathcal{D}_n$, $n \in \mathbb{N}$ such that $\mathcal{D}_n$ would be the weakest failure detector to solve consensus among $n+1$ processes using objects of any type T such that $cons(T) = n$. In the sense of [14], $\mathcal{D}_n$ would thus be the weakest failure detector to boost the power of T to higher levels of the consensus hierarchy.

A reasonable candidate for such a failure detector hierarchy was introduced by Neiger in [21]. This hierarchy is made of weaker variants of Ω, denoted Ω_n, $n \in \mathbb{N}$, where Ω_n is a failure detector that outputs, at each process, a set of processes so that all correct processes eventually detect the same set of at most n processes that includes at least one correct process. Clearly, Ω_1 is Ω. It was shown in [21] that Ω_n is sufficient to solve consensus among k processes ($k > n$) using *any* set of types T such that $cons(T) = n$ and registers. It was also conjectured in [21] that Ω_n is also necessary to boost the power of T to the level k of the consensus hierarchy. As pointed out in [21], the proof of this conjecture appears to be challenging and was indeed left open. The motivation of this work was to take up that challenge.

Contributions. In this paper, we assume that processes communicate using read-write shared memory (registers) and one-shot deterministic types [14]. Although these types restrict every process to

invoke at most one deterministic operation on each object, they include many popular types such as consensus and test-and-set, and they exhibit complex behavior in the context of the type booster question [4, 14, 16, 19].

We show that Ω_n is necessary to solve consensus using registers and objects of any one-shot deterministic type T such that $cons(T) \leq n$ and $2 \leq n$. As an interesting corollary of our result, we show that Ω_n is the weakest failure detector to boost the resilience of a system of $(n-1)$-resilient objects and registers solving consensus.

Our result is a strict generalization of the fundamental result of [6] where Ω was shown to be necessary to solve consensus in a message-passing system. We assume that, instead of reliable channels, processes communicate through registers and objects of a powerful sequential type T. The only information available on T is the fact that $cons(T) \leq n$ and T is one-shot and deterministic. The lack of information forced us to reconsider the proof of [6]. In particular, we revisit and generalize the notions of simulation tree, decision gadget and deciding process introduced in [6].

As a side result, we give a formal proof that any failure detector that implements consensus in the read-write memory model can be transformed to Ω. The result was first stated in [18] but, to our knowledge, its proof has never appeared in the literature.

Related work. The notion of consensus power was introduced by Herlihy [13] and then refined by Jayanti [16]. Chandra, Hadzilacos and Toueg [6] showed that Ω is the weakest failure detector to solve consensus in asynchronous message-passing systems with a majority of correct processes. Lo and Hadzilacos [18] showed that Ω can be used to solve consensus with registers and stated that any failure detector that can be used to solve consensus with registers can be transformed to Ω. Neiger [21] introduced the hierarchy of failure detectors Ω_n and showed that objects of consensus power n can solve consensus among any number of processes using Ω_n.

Roadmap. Section 2 presents necessary details of the model used in this chapter. Section 3 recalls the specification of consensus and recalls a few results from the literature used in this paper. Section 4 recalls the hierarchy of failure detectors Ω_n. Section 5 shows that Ω_n is necessary to boost the consensus power of one-shot deterministic objects. Section 6 applies our result to the question of boosting the resilience of a distributed system with respect to the consensus problem.

2 Model

Our model of processes communicating through shared objects is based on that of [15, 16] and our notion of failure detectors follows from [6]. Below we recall what is substantial to show our result.

Processes

We consider a set Π of k asynchronous processes $p_1, p_2, \ldots, p_k$ ($k \geq 2$) that communicate using shared objects. To simplify the presentation of our model, we assume the existence of a discrete global clock. This is a fictional device: the processes have no direct access to it. (More precisely, the information about global time can come *only* from failure detectors.) We take the range $\mathbb{T}$ of the clock's ticks to be the set of natural numbers and 0 ($\mathbb{T} = \{0\} \cup \mathbb{N}$).

2.1 Objects and types

An *object* is a data structure that can be accessed concurrently by the processes. Every object is an instance of a sequential *type* which is defined by a tuple $(Q, Q_0, O, n_p, R, \delta)$. Here Q is a set of *states*, $Q_0 \subseteq Q$ is a set of *initial states*, O is a set of *operations*, n_p is a positive integer denoting the number of *ports* (used as an interface between processes and objects), R is a set of *responses*, and δ is a relation known as the *sequential specification* of the type: it carries each state and operation to a set of response and state pairs. We assume that objects are *deterministic*: the set of initial states is a singleton $Q_0 = \{q_0\}$ and the sequential specification is a function $\delta : Q \times O \to Q \times R$.

The register type is defined as a tuple $(Q, \{q_0\}, O, k, R, \delta)$ where Q is a set of *values* that can be stored in a register, $q_0 \in Q$ is an initial value, $O = \{read(), \text{write}(v) : v \in Q\}$, $R = Q \cup \{ok\}$ and $\forall v, v' \in Q$, $\delta(v, write(v')) = (v', ok)$ and $\delta(v, read()) = (v, v)$.

A process accesses objects by invoking operations on the ports of the objects. A process can use at most one port of each object. A port can be used by at most one process. A port of an object of a *one-shot* type can be accessed at most once in any execution. If not explicitly specified, the number of ports of any object is k, and every object is connected to every process.

We consider here *linearizable* [15] objects: even though operations of concurrent processes may overlap, each operation takes effect instantaneously between its invocation and response. If a process invokes an operation on a linearizable object and fails before receiving a matching response, then the "failed" operation *may* take effect at any future time. Any execution on linearizable objects can thus be seen as a sequence of atomic invocation-response pairs.

Unless explicitly stated otherwise, we assume that the objects are *wait-free*: any operation invoked by a correct process on a wait-free object eventually returns, regardless of failures of other processes [13]. In contrast, *t-resilient* implementations of shared objects (considered in Section 6) guarantee that a process completes its operation, as long as no more than t processes crash. If more than t processes crash, no operation on a t-resilient object is obliged to return.

2.2 Failures and failure patterns

Processes are subject to *crash* failures. We do not consider Byzantine failures: a process either correctly executes the algorithm assigned to it, or crashes and stops forever executing any action.

A *failure pattern* F is a function from the global time range $\mathbb{T}$ 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 $\mathit{correct}(F) = \Pi - \cup_{t \in \mathbb{T}} F(t)$, the set of *correct* processes in F. Processes in $\Pi - \mathit{correct}(F)$ are called *faulty* in F. A process $p \notin F(t)$ is said to be *up* at time t. A process $p \in F(t)$ is said to be *crashed* at time t. We say that a subset $U \subseteq \Pi$ is *alive* if $U \cap \mathit{correct}(F) \neq \emptyset$. We consider here all failure *environments* [6], i.e., we make no assumptions on when and where failures might occur. However, we assume that there is at least one correct process in every failure pattern.

2.3 Failure detectors

A *failure detector history* H *with range* $\mathcal{R}$ is a function from $\Pi \times \mathbb{T}$ to $\mathcal{R}$. $H(p,t)$ is the value of the failure detector module of process p at time t. A *failure detector* $\mathcal{D}$ with range $\mathcal{R}_{\mathcal{D}}$ is a function that maps each failure pattern to a *set* of failure detector histories with range $\mathcal{R}_{\mathcal{D}}$ (usually defined by a set of requirements that these histories should satisfy). $\mathcal{D}(F)$ denotes the set of possible failure detector histories permitted by $\mathcal{D}$ for failure pattern F.

When any process p performs a step of computation, it can query its *failure detector module* of $\mathcal{D}$, denoted $\mathcal{D}_p$, and obtain a value $d \in \mathcal{R}_{\mathcal{D}}$ that encodes some information about failures. Note that we do not make any assumption a priori on the range of a failure detector.

The *leader failure detector* Ω outputs the id of a process at each process. There is a time after which it outputs the id of the same correct process at all correct processes [6]. Formally, $\mathcal{R}_{\Omega} = \Pi$ and, for each failure pattern F, $H \in \Omega(F) \quad \Leftrightarrow \exists t \in \mathbb{T}\, \exists q \in \mathit{correct}(F)\, \forall p \in \mathit{correct}(F)\, \forall t' \geq t : H(p,t') = q$

2.4 Algorithms

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

(i) performs an operation on a shared object *or* queries its failure detector module $\mathcal{D}_p$ (in the latter case, we say that the step of p is a *query* step), and
(ii) applies its current state together with the the response received from a shared object *or* the value received from $\mathcal{D}_p$ during that step to the automaton $\mathcal{A}(p)$ to obtain a new state.

A step of $\mathcal{A}$ is thus identified by a pair (p, x), where x is either λ (the empty value) or, if the step is a query step, the failure detector value output at p during that step.

2.5 Configurations, schedules and runs

A *configuration* of $\mathcal{A}$ defines the current state of each process and each object in the system. In an *initial configuration* of $\mathcal{A}$, every process p is in an initial state of $\mathcal{A}(p)$ and every object is an initial state specified by its object type.

The state of any process p in C determines whether in any step of p applied to C, p accesses a shared object or queries its failure detector module. Respectively, a step (p, x) is said to be *applicable* to C if and only if

(a) $x = \lambda$, and p invokes an operation o on a shared object X in its next step in C (we say that p *accesses* X *with* o in C), or
(b) $x \in \mathcal{R}_{\mathcal{D}}$, and p queries $\mathcal{D}_p$ in its next step in C. Here, x is the value obtained from $\mathcal{D}_p$ during that step.

For a step e applicable to C, $e(C)$ denotes the unique configuration that results from applying e to C.

A *schedule* S of algorithm $\mathcal{A}$ is a (finite or infinite) sequence of steps of $\mathcal{A}$. $S_\perp$ denotes the empty schedule. We say that *a schedule* S *is applicable to a configuration* C if and only if (a) $S = S_\perp$, or (b) $S[1]$ is applicable to C, $S[2]$ is applicable to $S[1](C)$, etc. For a finite schedule S applicable to C, $S(C)$ denotes the unique configuration that results from applying S to C.

Let S be any schedule applicable to a configuration C. We say that S *applied to* C *accesses* X if S has a prefix $S' \cdot (p, \lambda)$ where p accesses X in $S'(C)$.

For any $P \subseteq \Pi$, we say that S is a *P-solo schedule* if only processes in P take steps in S.

A *partial run of algorithm* $\mathcal{A}$ *using a failure detector* $\mathcal{D}$ is a tuple $R = \langle F, H_{\mathcal{D}}, I, S, T\rangle$ where F is a failure pattern, $H_{\mathcal{D}} \in \mathcal{D}(F)$ is a failure detector history, I is an initial configuration of $\mathcal{A}$, S is a *finite* schedule of $\mathcal{A}$, and $T \subseteq \mathbb{T}$ is a *finite* list of increasing time values such that $|S| = |T|$, S is applicable to I, and for all $1 \leq k \leq |S|$, if $S[k] = (p, x)$ then:

(1) Either p has not crashed by time $T[k]$, i.e., $p \notin F(T[k])$, or $x = \lambda$ and $S[k]$ is the last appearance of p in S, i.e., $\forall k < k' \leq |S|$: $S[k'] \neq (p, *)$ (the last condition takes care about the cases when an operation of p is linearized *after* p has crashed, and there can be at most one such operation in a run);
(2) if $x \in \mathcal{R}_{\mathcal{D}}$, then x is the value of the failure detector module of p at time $T[k]$, i.e., $d = H_{\mathcal{D}}(p, T[k])$.

A *run of algorithm* $\mathcal{A}$ *using a failure detector* $\mathcal{D}$ is a tuple $R = \langle F, H_{\mathcal{D}}, I, S, T\rangle$ where F is a failure pattern, $H_{\mathcal{D}} \in \mathcal{D}(F)$ is a failure detector history, I is an initial configuration of $\mathcal{A}$, S is an *infinite* schedule of $\mathcal{A}$, and $T \subseteq \mathbb{T}$ is an *infinite* list of increasing time values indicating when each step of S has occurred. In addition to satisfying properties (1) and (2) of a partial run, R should guarantee that

(3) every correct (in F) process takes an infinite number of steps in S.

2.6 Problems and solvability

A *problem* is a predicate on a set of runs (usually defined by a set of properties that these runs should satisfy). An algorithm $\mathcal{A}$ *solves a problem* $\mathcal{M}$ *using a failure detector* $\mathcal{D}$ if the set of all runs of $\mathcal{A}$ using $\mathcal{D}$ satisfies $\mathcal{M}$. We say that a failure detector $\mathcal{D}$ *solves problem* $\mathcal{M}$ if there is an algorithm $\mathcal{A}$ which solves $\mathcal{M}$ using $\mathcal{D}$.

2.7 A weakest failure detector

Informally, $\mathcal{D}$ is the weakest failure detector to solve a problem $\mathcal{M}$ if (a) $\mathcal{D}$ is *sufficient* to solve $\mathcal{M}$, i.e., $\mathcal{D}$ can be used to solve $\mathcal{M}$, and (b) $\mathcal{D}$ is *necessary* to solve $\mathcal{M}$, i.e., any failure detector $\mathcal{D}'$ that can be used to solve $\mathcal{M}$ can be transformed into $\mathcal{D}$.

More precisely, let $\mathcal{D}$ and $\mathcal{D}'$ be failure detectors. If, for failure detectors $\mathcal{D}$ and $\mathcal{D}'$, there is an algorithm $T_{\mathcal{D}'\rightarrow\mathcal{D}}$ that transforms $\mathcal{D}'$ into $\mathcal{D}$, we say that $\mathcal{D}$ *is weaker than* $\mathcal{D}'$, and we write $\mathcal{D} \preceq \mathcal{D}'$.

If $\mathcal{D} \preceq \mathcal{D}'$ but $\mathcal{D}' \not\preceq \mathcal{D}$, we say that $\mathcal{D}$ *is strictly weaker than* $\mathcal{D}'$, and we write $\mathcal{D} \prec \mathcal{D}'$. If $\mathcal{D} \preceq \mathcal{D}'$ and $\mathcal{D}' \preceq \mathcal{D}$, we say that $\mathcal{D}$ *and* $\mathcal{D}'$ *are equivalent*, and we write $\mathcal{D} \prec \mathcal{D}'$.

Algorithm $T_{\mathcal{D}'\rightarrow\mathcal{D}}$ that emulates histories of $\mathcal{D}$ using histories of $\mathcal{D}'$ is called a *reduction* algorithm. Note that $T_{\mathcal{D}'\rightarrow\mathcal{D}}$ does not need to emulate *all* histories of $\mathcal{D}$; it is required that all the failure detector histories it emulates be histories of $\mathcal{D}$.

We say that a failure detector $\mathcal{D}$ is *the weakest failure detector to solve a problem* $\mathcal{M}$ if the following conditions are satisfied:

(a) $\mathcal{D}$ is *sufficient* to solve $\mathcal{M}$, i.e., $\mathcal{D}$ solves $\mathcal{M}$, and
(b) $\mathcal{D}$ is *necessary* to solve $\mathcal{M}$, i.e., if a failure detector $\mathcal{D}'$ solves $\mathcal{M}$, then $\mathcal{D}$ is weaker than $\mathcal{D}'$.

There might be a number of distinct failure detectors satisfying these conditions. (Though all such failure detectors are in a strict sense equivalent.) With a slight abuse of grammar, it would be more technically correct to talk about *a* weakest failure detector to solve $\mathcal{M}$.

3 Preliminaries

In this section, we recall the specifications of consensus and related *team consensus* and *weak consensus* problems.

3.1 Consensus

The (binary) *m-process consensus* problem [10] consists for m processes to decide some final values (0 or 1) based on their initial proposed values in such a way that: (*Agreement*) no two processes decide different values, (*Validity*) every decided value is a proposed value of some process, and (*Termination*) every correct process eventually decides.

It is sometimes convenient to think of the consensus problem in terms of an object type. Formally, the m-process consensus type is specified as a tuple $(Q, Q_0, O, n_p, R, \delta)$, where $Q = \{\perp, 0, 1\}$, $Q_0 = \perp$, $O = \{propose(v) : v \in \{0, 1\}\}$, $n_p = m$, $R = \{0, 1\}$, and $\forall v, v' \in \{0, 1\}$, $\delta(\perp, propose(v)) = (v, v)$ and $\delta(v', propose(v)) = (v', v')$.

We say that a set $\mathcal{S}$ of types *solves m-process consensus* if there is an algorithm that solves consensus among m processes using registers and objects of types in $\mathcal{S}$.

The *consensus power* of an object type T, denoted $cons(T)$, is the largest number m of processes such that $\{T\}$ solves m-process consensus. If no such largest m exists, then $cons(T) = \infty$.

The test-and-set type is defined as a tuple $(Q, Q_0, O, k, R, \delta)$ where $Q = \{0, 1\}$, $Q_0 = \{0\}$, $O = \{test\text{-}and\text{-}set(), reset()\}$, $R = \{0, 1\}$ and $\delta(0, test\text{-}and\text{-}set()) = (1, 0)$, $\delta(1, test\text{-}and\text{-}set()) = (1, 1)$, and $\forall v \in \{0, 1\}$, $\delta(v, reset()) = (0, 0)$.

By combining the results of [1] and [5], it can be shown that test-and-set objects cannot "boost" the consensus power of any type:

Lemma 1 *Let $m \geq 2$ and $\mathcal{S}$ be any set of types. If $\mathcal{S} \cup \{$test-and-set$\}$ solve $(m+1)$-process consensus, then $\mathcal{S}$ solves $(m+1)$-process consensus.*

Assume that any set $\mathcal{S}$ of *deterministic* types can solve consensus among m processes when every object X is initialized to some state that results after applying a known sequence of operations on X (we say that the state is *reachable*). Then $\mathcal{S}$ solves consensus among m processes [4]:

Lemma 2 *Let $\mathcal{S}$ be any set of deterministic types. If $\mathcal{S}$ solves m-process consensus, when every object is initialized to some reachable state, then $\mathcal{S}$ solves m-process consensus.*

3.2 Team consensus

We also use a restricted form of consensus, called *team consensus*. This variant of consensus always ensures Validity and Termination, but Agreement is ensured only if the input values satisfy certain conditions. More precisely, assume that there exists a (known a priori) partition of the processes into two non-empty sets (teams). Team consensus requires Agreement only if all processes on a team have the same input value. Obviously, team consensus can be solved whenever consensus can be solved. Surprisingly, the converse is also true [21, 22]:

Lemma 3 *Let $\mathcal{S}$ be any set of types. If $\mathcal{S}$ solves team consensus among m processes, then $\mathcal{S}$ also solves consensus among m processes.*

Proof. We proceed by induction on m. For $m = 2$, team consensus is consensus. Assume that, (1) for some $m > 2$, $\mathcal{S}$ solves team consensus among m processes (for non-empty teams A and B), and (2) for all $2 \leq l < m$, $\mathcal{S}$ solves l-process consensus. Thus, A and B can use, respectively, $|A|$-process consensus and $|B|$-process consensus to agree on the teams' input values (A and B are non-empty, thus, $|A| < m$ and $|B| < m$). Once the team input value is known, the processes run the team consensus algorithm among m processes (with teams A and B). Since all processes on a team propose the same value, Agreement of m-process consensus is satisfied. □

3.3 Weak consensus

To prove our result, we also consider a weaker form of consensus, the *weak consensus* problem [10]. This variant of consensus always ensures Termination and Agreement, but Validity is replaced by a weaker *Non-Triviality* property: every algorithm that solves the weak consensus problem has a run in which 0 is decided and a run in which 1 is decided. Obviously, weak consensus can be solved whenever consensus can be solved. Surprisingly, if deterministic types are used, the converse is also true [12] (we give the proof of [12] here for self-consistency):

Lemma 4 *Let $\mathcal{S}$ be any set of deterministic types. If weak consensus among m processes can be solved using registers and objects of types in $\mathcal{S}$, then $\mathcal{S}$ solves m-process consensus.*

Proof. Let $\mathcal{A}$ be any algorithm that solves weak consensus among $m \geq 2$ processes using registers and objects of types in $\mathcal{S}$.

Let G be the execution graph of $\mathcal{A}$: the vertices of G are all possible states of $\mathcal{A}$ (defined by the states of the processes and all shared objects); vertices s and s' are connected with an edge directed from s to s' if and only if there is a step of $\mathcal{A}$ that, applied to s, results in s'.

A vertex s of G is assigned a tag $v \in \{0, 1\}$ if it has a successor s' in G (i.e., there exists a path in G from s to s') such that some process has decided v in s'. If a state has both tags 0 and 1, it is called *bivalent*. If a state has only one tag v, it is called *v-valent*. A state is univalent if it is 0-valent or 1-valent. The Termination property of weak consensus ensures that any state of $\mathcal{A}$ is either bivalent or univalent [10].

We show first that there exists a *critical* state in G, i.e., a bivalent state $\bar{s}$ such that every step of $\mathcal{A}$ applied to $\bar{s}$ results in a univalent state. Suppose not, i.e., every state in G has a bivalent successor. Then, starting from the initial state of $\mathcal{A}$, we can build an infinite run R of $\mathcal{A}$ that goes through bivalent states only. By the Agreement property of weak consensus, no process can decide

Initially:

all objects are initialized to their states in $\bar{s}$

Procedure TCPROPOSE(v): { *let* $p \in \Pi_i$, $i \in \{0,1\}$ }

1: $X_i \leftarrow v$ { write the proposal in the team's register }
2: let p be initialized to its state in $\bar{s}$
3: run $\mathcal{A}$ until it returns a value $j \in \{0,1\}$
4: return X_j

Fig. 1. A team consensus algorithm: process p

in a bivalent state. Thus, no process can ever decide in R — a contradiction with the Termination property of weak consensus.

Assume now that the system is in a critical state $\bar{s}$. Since protocol $\mathcal{A}$ and all objects that we use are deterministic, the step of any given process applied to $\bar{s}$ triggers exactly one transition in graph G. Thus, for any step of $\mathcal{A}$ applied to $\bar{s}$, the valence of the resulting state is defined by the identity of the process that takes that step. Now we partition Π into two teams Π_0 and Π_1: for each $i \in \{0,1\}$, Π_i consists of all processes whose steps applied to $\bar{s}$ result in i-valent states. Since $\bar{s}$ is bivalent and any step applied to $\bar{s}$ results in a univalent state, the two teams are non-empty.

The algorithm in Figure 1 solves team consensus among m processes for teams Π_0 and Π_1.

Let all objects used by $\mathcal{A}$ be initialized to their states in $\bar{s}$. For each $i \in \{0,1\}$, we associate team Π_i with register X_i. Every process writes its input value into its team's register and then runs $\mathcal{A}$ starting from its state in $\bar{s}$ until $\mathcal{A}$ returns a value $j \in \{0,1\}$. Then the process returns X_j.

Consider any run R of the algorithm in Figure 1. The Termination property of weak consensus ensures Termination of our algorithm. Assume that p returns a value of X_j in R, i.e. $\mathcal{A}$ returns j at p. By the definition of Π_0 and Π_1, the first step accessing X in R is by a process $q \in \Pi_j$. By the algorithm, q has previously written its input value in X_j. Thus, Validity of team consensus is ensured. The Agreement property of weak consensus ensures that $\mathcal{A}$ cannot return $1-j$ at any process q in R. Thus, no process can return a value of X_{1-j} in R. Assume now that all processes on a team propose the same value. Hence, Agreement is ensured, since the processes return the value previously written in X_j, and no two different values can be written in X_j.

Thus, $\mathcal{S}$ solves team consensus among m processes when objects are initialized to their states in $\bar{s}$. By the construction, these states are reachable. By Lemmas 2 and 3, $\mathcal{S}$ solves consensus among m processes. □

4 Hierarchy of failure detectors Ω_n

The hierarchy of failure detectors Ω_n ($n \in \mathbb{N}$) was introduced in [21]. Ω_n ($n \in \mathbb{N}$) outputs a set of *at most* n processes at each process so that, eventually, the same *alive* (including at least one correct process) set is output at all correct processes.

Formally, $\mathcal{R}_{\Omega_n} = \{P \subseteq \Pi : |P| \leq n\}$, and for each failure pattern F, $H \in \Omega_n(F) \quad \Leftrightarrow$

$$\exists t \in \mathbb{T}\, \exists P \in \mathcal{R}_{\Omega_n},\ P \cap \mathit{correct}(F) \neq \emptyset,\ \forall p \in \mathit{correct}(F)\, \forall t' \geq t :\ H(p, t') = P$$

Clearly, Ω_1 is equivalent to Ω. It was shown in [21] that, for all $k \geq 2$ and $1 \leq n \leq k-1$:

(a) $\Omega_{n+1} \prec \Omega_n$;
(b) for any type T such that $\mathit{cons}(T) = n$, Ω_n can be used to solve k-process consensus using registers and objects of type T.

5 Boosting consensus power

In this section, we show that Ω_n is necessary to solve consensus among k processes using registers and objects of type T a one-shot deterministic type T such that $\mathit{cons}(T) \leq n$ and $2 \leq n$. Our proof is a natural generalization of the proof that Ω is necessary to solve consensus in message-passing asynchronous systems [6].

5.1 An overview of the reduction algorithm

Let $\mathsf{Cons}_{\mathcal{D}}$ be any algorithm that solves consensus using a failure detector $\mathcal{D}$, registers and objects of a one-shot deterministic type T such that $\mathit{cons}(T) \leq n$ and $2 \leq n$. Our goal is to define a reduction algorithm $T_{\mathcal{D} \to \Omega_n}$ that emulates the output of Ω_n using $\mathcal{D}$ and $\mathsf{Cons}_{\mathcal{D}}$. The reduction algorithm should have all correct processes eventually agree on the same *alive* set of at most n processes.

$T_{\mathcal{D} \to \Omega_n}$ consists of two parallel tasks: a communication task and a computation task.

In the communication task, each process p periodically queries its failure detector module of $\mathcal{D}$ and exchanges the failure detector values with the other processes values using read-write memory. While doing so, p knows more and more of the other processes' failure detector outputs and temporal relations between them. All this information is pieced together in a single data structure, a directed acyclic graph (DAG) G_p.

In the computation task, p periodically uses its DAG G_p to simulate *locally*, for any initial configuration I and any set of processes $P \subseteq \Pi$, a number of finite runs $\mathsf{Cons}_{\mathcal{D}}$. These runs constitute an ever-growing *simulation tree*, denoted $\Upsilon_p^{P,I}$. Since registers provide reliable (though asynchronous) communication, all such $\Upsilon_p^{P,I}$ tend to the same *infinite* simulation tree $\Upsilon^{P,I}$.

It turns out that the processes can eventually detect the same set $P \subseteq \Pi$ such that P includes all correct processes, and either (a) there exists a correct *critical* process whose proposal value

Initially:

$G_p \leftarrow$ empty graph

$k_p \leftarrow 0$

1: **while** *true* **do**

2: **for all** $q \in \Pi$ **do** $G_p \leftarrow G_p \cup G_q$

3: $d_p \leftarrow$ query failure detector $\mathcal{D}$

4: $k_p \leftarrow k_p + 1$

5: add $[p, d_p, k_p]$ and edges from all vertices of G_p to $[p, d_p, k_p]$ to G_p

Fig. 2. Building a DAG: process p

in some initial configuration I defines the decision value in all paths in $\Upsilon^{P,I}$, or (b) some $\Upsilon^{P,I}$ has a finite subtree γ, called a *complete decision gadget*, that provides sufficient information to compute a set of at most n processes, called the *deciding set* of γ, that includes at least one correct process. Eventually, the correct processes either detect the same critical process or compute the same complete decision gadget and agree on its deciding set. In both cases, Ω_n is emulated.

A difficult point here is that sometimes the deciding set is encoded in an object of type T. We cannot use the sequential specification of type T, and we hence cannot use the case analysis of [6] to compute the deciding set. Fortunately, in this case, it is possible to locate a special kind of a decision gadget, which we introduce here and which we call a *rake*, such that the deciding set is encoded in the states of the processes at the leaves of the rake and the state of an object of type T in the *pivot* of the rake. Using the assumptions that T is a one-shot deterministic type and $cons(T) \leq n$, we compute a set of at most n processes that include at least one correct process.

5.2 The communication task and DAGs

The communication task of algorithm $T_{\mathcal{D} \to \Omega}$ is presented in Figure 2. This task maintains an ever-growing DAG that contains a finite sample of the current failure detector history. (For simplicity, the DAG is stored in a register G_p which is periodically updated by p and read by all processes.)

Informally, every vertex $[q, d, k]$ of G_p is a failure detector value "seen" by q in its k-th query of its failure detector module. An edge $([q, d, k], [q', d', k'])$ can be interpreted as "q saw failure detector value d (in its k-th query) *before* q' saw failure detector value d' (in its k'-th query)".

DAG G_p has some special properties which follow from its construction [6]. Let F be the current failure pattern and H be the current failure detector history in $\mathcal{D}(F)$. Then:

(1) The vertices of G_p are of the form $[q, d, k]$ where $q \in \Pi$, $d \in \mathcal{R}_\mathcal{D}$ and $k \in \mathbb{N}$. There is a mapping $\tau :$ vertices of $G_p \mapsto \mathbb{T}$, associating a time with every vertex of G_p, such that:

(a) For any vertex $v = [q, d, k]$, $q \notin F(\tau(v))$ and $d = H(q, \tau(v))$. That is, d is the value output by q's failure detector module at time $\tau(v)$.

(b) For any edge (v, v') in G_p, $\tau(v) < \tau(v')$. That is, any edge in G_p reflects the temporal order in which the failure detector values are output.

(2) If $v' = [q, d, k]$ and $v'' = [q, d', k']$ are vertices of G_p and $k < k'$ then (v, v') is an edge of G_p.

(3) G_p is transitively closed: if (v, v') and (v', v'') are edges of G_p, then (v, v'') is also an edge of G_p.

(4) For all correct processes p and q and all times t, there is a time $t' \geq t$, a $d \in \mathcal{R}_\mathcal{D}$ and a $k \in \mathbb{N}$ such that for every vertex v of $G_p(t)$, $(v, [q, d, k])$ is an edge of $G_p(t')$. ($x(t)$ denotes the value of variable x at time t.)

Note that properties (1)–(4) imply that, for any time t and any set of vertices V of $G_p(t)$, there is a time $t' \geq t$ such that $G_p(t')$ contains a path g such that every correct process appears in g arbitrarily often and $\forall v \in V$, $v \cdot g$ is also a path of $G_p(t')$. Furthermore, every prefix of g is also a path in $G_p(t')$.

5.3 Simulation trees

Let I^l $(l = 0, \ldots, k)$ denote an initial configuration of $\mathsf{Cons}_\mathcal{D}$ in which processes $p_1, \ldots, p_l$ propose 1 and processes $p_{l+1}, \ldots, p_k$ propose 0. Let $P \subseteq \Pi$ be any set of processes, and $g = [q_1, d_1, k_1], [q_2, d_2, k_2], \ldots [q_s, d_s, k_s]$ be any path in G_p such that $\forall i \in \{1, 2, \ldots, s\} : q_i \in P$. Since algorithms and shared objects considered here are deterministic, g and I^l *induce* a unique schedule $S = (q_1, x_1), (q_2, x_2), \ldots, (q_s, x_s)$ of $\mathsf{Cons}_\mathcal{D}$ applicable to I^l such that:

$$\forall i \in \{1, 2, \ldots, s\} : x_i \in \{\lambda, d_i\}.$$

For each $P \subseteq \Pi$, the set of all P-solo schedules of $\mathsf{Cons}_\mathcal{D}$ induced by I_l and paths in G_p are pieced together in a tree $\Upsilon_p^{P,l}$, called the *simulation tree induced by P, I^l and G_p*, and defined as follows. The set of vertices of $\Upsilon_p^{P,l}$ is the set of *finite* P-solo schedules that are induced by I^l and paths in G_p. The root of $\Upsilon_p^{P,l}$ is the empty schedule $S_\perp$. There is an edge from a vertex S to a vertex S' whenever $S' = S \cdot e$ for some step e; the edge is labeled e. Thus, every vertex S of $\Upsilon_p^{P,l}$ is associated with a unique path $e_1 e_2, \ldots e_s$ in $\Upsilon_p^{P,l}$.

The construction implies that for any vertex S of $\Upsilon_p^{P,I}$, there exists a partial run $\langle F, H, I, S, T \rangle$ of $\mathsf{Cons}_\mathcal{D}$ where F is the current failure pattern and $H \in \mathcal{D}(F)$ is the current failure detector history.

We *tag* every vertex S of $\Upsilon_p^{P,l}$ according to the values decided in the descendants of S in $\Upsilon_p^{P,l}$: S is assigned a tag v if and only if it has a descendant S' such that p decides v in $S'(I^l)$. The set of all tags of S is called the *valence* of S and denoted *val*(S). If S has only one tag $u \in \{0, 1\}$, then S is called *u-valent*. A 0-valent or 1-valent vertex is called *univalent*. A vertex is called *bivalent* if it has both tags 0 and 1.

Thanks to reliable communication guarantees provided by registers, for any two correct processes p and q and any time t, there is a time $t' \geq t$ such that $\Upsilon_p^{P,l}(t) \subseteq \Upsilon_q^{P,l}(t')$. As a result, the simulation

trees $\Upsilon_p^{P,l}$ of correct processes p tend to the same *limit* infinite simulation tree which we denote $\Upsilon^{P,l}$.

Assume that *correct*$(F) \subseteq P$. By the construction, every vertex of $\Upsilon^{P,l}$ has an extension in $\Upsilon^{P,l}$ in which every correct process takes infinitely many steps. By the Termination property of consensus, this extension has a finite prefix S' such that every correct process has decided in $S'(I^l)$. Thus, every vertex S of $\Upsilon^{P,l}$ has a non-empty valence, i.e. S is univalent or bivalent.

More generally:

Lemma 5 *Let correct*$(F) \subseteq P \subseteq \Pi$, $0 \leq l \leq k$, $m \geq 1$, *and* $S_0, S_1, \ldots, S_m$ *be any vertices of* $\Upsilon^{P,l}$. *There exists a finite schedule* S' *containing only steps of correct processes such that*

(1) $S_0 \cdot S'$ *is a vertex of* $\Upsilon^{P,l}$ *and all correct processes have decided in* $S_0 \cdot S'(I^l)$, *and*
(2) for any $i \in \{1, 2, \ldots, m\}$, *if* S' *is applicable to* $S_i(I^l)$, *then* $S_i \cdot S'$ *is a vertex of* $\Upsilon^{P,l}$.

The following lemma will facilitate the proof of correctness of our reduction algorithm.

Lemma 6 *Let correct*$(F) \subseteq P \subseteq \Pi$. *Let* S_0 *and* S_1 *be two univalent vertices of* $\Upsilon^{P,l}$ *of opposite valence and* $V \subset \Pi$ *be a set of processes. If* $S_0(I^l)$ *and* $S_1(I^l)$ *differ only in the states of processes in* V, *then* V *includes at least one correct process.*

Proof. Since $S_0(I^l)$ and $S_1(I^l)$ differ only in the states of processes in V, any $(\Pi - V)$-solo schedule applicable to $S_0(I^l)$ is also applicable to $S_1(I^l)$. By contradiction, assume that V includes only faulty processes. By Lemma 5, there is a schedule S containing only steps of correct processes (and thus no steps of processes in V) such that all correct processes have decided in $S_0 \cdot S(I^l)$ and $S_1 \cdot S$ is a vertex of $\Upsilon^{P,l}$. Since no process in $\Pi - V$ can distinguish $S_0 \cdot S(I^l)$ and $S_1 \cdot S(I^l)$, the correct processes have decided the same values in these two configurations — a contradiction. □

5.4 Decision gadgets

A *decision gadget* γ is a finite subtree of $\Upsilon^{P,l}$ rooted at $S_\perp$ that includes a vertex $\bar{S}$ (called the *pivot* of the gadget) such that one of the following conditions is satisfied:

(*fork*) There are two steps e and e' of the same process q, such that $\bar{S} \cdot e$ and $\bar{S} \cdot e'$ are univalent vertices of $\Upsilon^{P,l}$ of opposite valence.

Note the next step of q in $\bar{S}(I^l)$ can only be a *query* step. Otherwise, $\bar{S} \cdot e(I^l) = \bar{S} \cdot e'(I^l)$ and thus $\bar{S} \cdot e$ and $\bar{S} \cdot e'$ cannot have opposite valence.

(*hook*) There is a step e of a process q and step e' of a process q' $(q \neq q')$, such that:

(i) $\bar{S} \cdot e' \cdot e$ and $\bar{S} \cdot e$ are univalent vertices of $\Upsilon^{P,l}$ of opposite valence.

(ii) q and q' do not access the same object of type T in $\bar{S}(I^l)$.

If for any $x \in \mathcal{R}_{\mathcal{D}} \cup \{\lambda\}$, $\bar{S} \cdot e \cdot (q', x)$ is not a vertex of $\Upsilon^{P,l}$, then q' is called *missing* in the hook γ. Clearly, if q' is correct, then it cannot be missing in γ.

(*rake*) There is a set $U \subseteq P$, $|U| \geq 2$, and an object X of type T such that each $q \in U$ accesses X in $\bar{S}(I^l)$ (U is called the *participating set* of γ). Let E denote the set of *all* vertices of $\Upsilon^{P,l}$ of the form $\bar{S} \cdot S$ where $S = (q_1, \lambda), (q_2, \lambda), \ldots, (q_{|U|}, \lambda)$ and $q_1, q_2, \ldots, q_{|U|}$ is a permutation of processes in U (E can be empty). $\bar{S}$, U and E satisfy the following conditions:

(i) There do not exist a $(\Pi - U)$-solo schedule S' and a process $q' \in \Pi - U$, such that $\forall S \in \{\bar{S}\} \cup E$, $S \cdot S' \cdot (q', \lambda)$ is a vertex of $\Upsilon^{P,l}$ and q' accesses X in $S \cdot S'(I^l)$.

(ii) If $S \in E$, then S is univalent.

(iii) If $|E| = (|U|)!$, i.e., E includes *all* vertices $\bar{S} \cdot (q_1, \lambda) \cdot (q_2, \lambda) \cdots (q_{|U|}, \lambda)$ such that $q_1, q_2, \cdots, q_{|U|}$ is a permutation of processes in U, then there are at least one 0-valent vertex and at least one 1-valent vertex in E.

Note that if $|E| < (|U|)!$, then there is at least one process $q \in U$ such that for some $\{q_1, q_2, \ldots, q_s\} \subseteq U - \{q\}$, $\bar{S} \cdot (q_1, \lambda) \cdot (q_2, \lambda) \cdots (q_s, \lambda)$ is a vertex of $\Upsilon^{P,l}$, and $\bar{S} \cdot (q_1, \lambda) \cdot (q_2, \lambda) \cdots (q_s, \lambda) \cdot (q, \lambda)$ is *not* a vertex of $\Upsilon^{P,l}$. We call such processes *missing* in the rake. Clearly, every missing process is in *faulty*(F).

Examples of decision gadgets are depicted in Figure 3: (a) a fork where $e = (q, d)$ and $e' = (q, d')$, (b) a hook where $e = (q, x)$, $e' = (q', x')$, and q and q' do not access the same object of type T in $\bar{S}(I^l)$; (c) a rake with a participating set $U = \{q_1, q_2\}$ and a set of leaves $E = \{\bar{S} \cdot (q_1, \lambda) \cdot (q_2, \lambda), \bar{S} \cdot (q_2, \lambda) \cdot (q_1, \lambda)\}$, where q_1 and q_2 access the same object of type T in $\bar{S}(I^l)$.

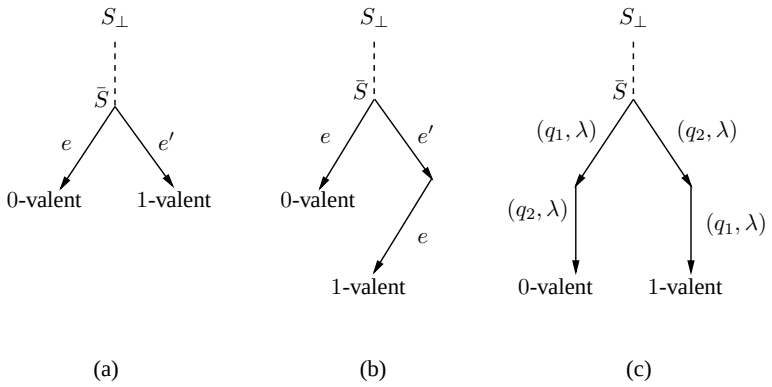

Fig. 3. A fork, a hook, and a rake

Lemma 7 *Let correct$(F) \subseteq P \subseteq \Pi$ and $l \in \{0, 1, \ldots, k\}$. If the root of $\Upsilon^{P,l}$ is bivalent, then $\Upsilon^{P,l}$ contains a decision gadget.*

Proof. Using arguments of Lemma 6.4.1 of [6], we can show that there exist a bivalent vertex S^* and a correct process p such that:

(*) For all descendants S' of S^* (including $S' = S^*$) and all $x \in \mathcal{R}_{\mathcal{D}} \cup \{\lambda\}$ such that $S' \cdot (p, x)$ is a vertex of $\Upsilon^{P,l}$, $S' \cdot (p, x)$ is univalent .

Moreover, one of the following conditions is satisfied:

(1) There are two steps e and e' of p, such that $S^* \cdot e$ and $S^* \cdot e'$ are vertices of $\Upsilon^{P,l}$ of opposite valence. That is, a fork is identified and we have the lemma.
(2) There is a step e of p and a step e' of a process q such that $S^* \cdot e$ and $S^* \cdot e' \cdot e$ are vertices of $\Upsilon^{P,l}$ of opposite valence.

Consider case (2). If $p = q$, then by condition (*), $S^* \cdot e'$ is a univalent vertex of $\Upsilon^{P,l}$, and a fork is identified.

Now assume that $p \neq q$. If p and q do not access the same object of type T in $S^*(I^l)$, we have a hook.

Thus, the only case left is when p and q access the same object X of type T in $S^*(I^l)$. The hypothetical algorithm of Figure 4 locates a rake in $\Upsilon^{P,l}$.

1: $U \leftarrow \{p, q\}$
2: $\bar{S} \leftarrow S^*$
3: **if** $\langle \bar{S} \cdot e \cdot e'$ is vertex of $\Upsilon^{P,l} \rangle$ **then**
4: $\quad E \leftarrow \{\bar{S} \cdot e' \cdot e, \bar{S} \cdot e \cdot e'\}$
5: **else**
6: $\quad E \leftarrow \{\bar{S} \cdot e' \cdot e\}$
7: **while** *true* **do**
8: $\quad$ **if** $\langle$ there exists a $(\Pi - U)$-solo schedule S' and a process $q' \in \Pi - U$
such that $\forall S \in \{\bar{S}\} \cup E$, $S \cdot S' \cdot (q', \lambda)$ is a vertex of $\Upsilon^{P,l}$
and q' accesses X in $S \cdot S'(I^l)$ $\rangle$
9: $\quad$ **then**
10: $\quad\quad$ let $S' \cdot (q', \lambda)$ be the shortest such schedule
11: $\quad\quad \bar{S} \leftarrow \bar{S} \cdot S'$
12: $\quad\quad U \leftarrow U \cup \{q'\}$
13: $\quad\quad E \leftarrow$ the set of all vertices $\bar{S} \cdot S$ of $\Upsilon^{P,l}$
such that $S = (q_1, \lambda), (q_2, \lambda), \ldots, (q_{|U|}, \lambda)$
where $q_1, q_2, \ldots, q_{|U|}$ is a permutation of processes in U
14: $\quad$ **else exit**

Fig. 4. Locating a rake in $\Upsilon^{P,l}$

We show first that the hypothetical algorithm terminates. Indeed, eventually either $U = \Pi$ and, trivially, there is no $q' \in \Pi - U$, or the algorithm terminates earlier with some $U \subset \Pi$.

Thus, we obtain a set U ($|U| \geq 2$) and a vertex $\bar{S} = S^* \cdot S''$ such that p and q take no steps in S'', S'' applied to $S^*(I^l)$ does not access X, and every $q' \in U$ accesses X in $\bar{S}(I^l)$. Furthermore:

(i) There do not exist a $(\Pi - U)$-solo schedule S' and a process $q' \in \Pi - U$, such that $\forall S \in \{\bar{S}\} \cup E$, $S \cdot S' \cdot (q', \lambda)$ is a vertex of $\Upsilon^{P,l}$ and q' accesses X in $S \cdot S'(I^l)$.

(ii) If $S \in E$, then S is univalent.
Indeed, take any $S \in E$. By the algorithm in Figure 4, $S = S^* \cdot S'$ such that every process in U takes exactly one step in S'. Since $p \in U$, p takes exactly one step in S'. By (*), S is univalent.

(iii) If $|E| = (|U|)!$, i.e., E includes all vertices $\bar{S} \cdot (q_1, \lambda) \cdot (q_2, \lambda) \cdots (q_{|U|}, \lambda)$ such that $q_1, q_2, \ldots, q_{|U|}$ is a permutation of processes in U, then there is at least one 0-valent vertex and at least one 1-valent vertex in E.
Indeed, assume that $|E| = (|U|)!$. By the algorithm, $S^* \cdot S'' \cdot e' \cdot e$, $S^* \cdot S'' \cdot e \cdot e'$, $S^* \cdot e' \cdot e \cdot S''$ and $S^* \cdot e' \cdot e \cdot S''$, where $e = (p, \lambda)$ and $e' = (q, \lambda)$, are vertices of $\Upsilon^{P,l}$. Since S'' applied to $S^*(I^l)$ does not access X, $S^* \cdot S'' \cdot e' \cdot e(I^l) = S^* \cdot e' \cdot e \cdot S''(I^l)$ and $S^* \cdot S'' \cdot e \cdot e'(I^l) = S^* \cdot e \cdot e' \cdot S''(I^l)$. But $S^* \cdot e \cdot e'$ and $S^* \cdot e' \cdot e$ are univalent vertices of opposite valence. Thus, $S^* \cdot S'' \cdot e \cdot e'$ and $S^* \cdot S'' \cdot e' \cdot e$ are also univalent vertices of opposite valence. Since E includes at least one descendant of $S^* \cdot S'' \cdot e \cdot e'$ and at least one descendant of $S^* \cdot S'' \cdot e' \cdot e$, there are at least one 0-valent vertex and at least one 1-valent vertex in E.

Hence, a rake with pivot $\bar{S}$ and participating set U is located. □

5.5 Complete decision gadgets

If a decision gadget γ has no missing processes, we say that γ is *complete*. If γ (a hook or a rake) has a non-empty set of missing processes, we say that γ is *incomplete*.

Lemma 8 *Let W be the set of missing processes of an incomplete decision gadget γ. Then $W \subseteq$ faulty(F).*

Proof. Let γ be an incomplete decision gadget of $\Upsilon^{P,l}$ and q be a missing process of γ. By definition, $q \in P$ and there is a vertex S of $\Upsilon^{P,l}$ such that for any $x \in \mathcal{R}_{\mathcal{D}} \cup \{\lambda\}$, $S \cdot (q, x)$ is not a vertex of $\Upsilon^{P,l}$. Thus, q is faulty in F. □

Lemmas 7 and 8 imply the following:

Corollary 9 *Let $C =$ correct(F). Every decision gadget of $\Upsilon^{C,l}$ is complete, and if the root of $\Upsilon^{C,l}$ is bivalent, then $\Upsilon^{C,l}$ contains at least one decision gadget.*

5.6 Confused processes

Lemma 10 *Let correct$(F) \subseteq P \subseteq \Pi$ and γ be a complete hook in $\Upsilon^{P,l}$ defined by a pivot $\bar{S}$, a step e of q, and a step e' of q' $(q \neq q')$. There exists a process $p \in \{q, q'\}$ and two vertices S_0 and S_1 in $\{\bar{S} \cdot e, \bar{S} \cdot e' \cdot e, \bar{S} \cdot e \cdot e'\}$ such that:*

(a) S_0 and S_1 are univalent vertices of $\Upsilon^{P,l}$ of opposite valence, and
(b) $S_0(I^l)$ and $S_1(I^l)$ differ only in the state of p.

Proof. By the definition of γ, $\bar{S} \cdot e$ and $\bar{S} \cdot e' \cdot e$ are univalent vertices of $\Upsilon^{P,l}$ of opposite valence, q and q' do not access the same object of type T, and there is a vertex $\bar{S} \cdot e \cdot (q', x)$ in $\Upsilon^{P,l}$ for some $x \in \mathcal{R}_{\mathcal{D}} \cup \{\lambda\}$.

Assume that $e' = (q', \lambda)$, and q and q' do not access the same register in $\bar{S}(I^l)$. Thus, $\bar{S} \cdot e \cdot e'$ is a vertex of $\Upsilon^{P,l}$ such that $\bar{S} \cdot e \cdot e'(I^l) = \bar{S} \cdot e' \cdot e(I^l)$. But $\bar{S} \cdot e$ and $\bar{S} \cdot e' \cdot e$ have opposite valences — a contradiction. Thus either (1) e' is a query step in $\bar{S}(I^l)$, or (2) q and q' access the same register in $\bar{S}(I^l)$.

(1) If e' is a query step in $\bar{S}(I^l)$, then $S_0 = \bar{S} \cdot e$ and $S_1 = \bar{S} \cdot e' \cdot e$ are univalent vertices of $\Upsilon^{P,l}$ of opposite valence such that $S_0(I^l)$ and $S_1(I^l)$ differ only in the state of q'.
(2) Assume now that e and e' access the same register X in $\bar{S}(I^l)$. Thus, $e = (q, \lambda)$, $e' = (q', \lambda)$, and $\bar{S} \cdot e \cdot e'$ is a univalent vertex of $\Upsilon^{P,l}$.
   - If q writes in X in $\bar{S}(I^l)$, then $S_0 = \bar{S} \cdot e$ and $S_1 = \bar{S} \cdot e' \cdot e$ are univalent vertices of $\Upsilon^{P,l}$ of opposite valence such that $S_0(I^l)$ and $S_1(I^l)$ differ only in the state of q'.
   - If q reads X in $\bar{S}(I^l)$, then $S_0 = \bar{S} \cdot e \cdot e'$ and $S_1 = \bar{S} \cdot e' \cdot e$ are univalent vertices of $\Upsilon^{P,l}$ of opposite valence such that $S_0(I^l)$ and $S_1(I^l)$ differ only in the state of q.

In each case, we obtain a process $p \in \{q, q'\}$ and two vertices S_0 and S_1 in $\{\bar{S} \cdot e, \bar{S} \cdot e' \cdot e, \bar{S} \cdot e \cdot e'\}$ such that (a) S_0 and S_1 are univalent vertices of $\Upsilon^{P,l}$ of opposite valence, and (b) $S_0(I^l)$ and $S_1(I^l)$ differ only in the state of p. □

The following lemma uses the assumptions that type T is *deterministic*, *cons*$(T) \leq n$ and $2 \leq n$.

Lemma 11 *Let correct$(F) \subseteq P \subseteq \Pi$ and γ be a complete rake in $\Upsilon^{P,l}$ with a pivot $\bar{S}$ and a participating set U such that $|U| \geq n + 1$. Let E be the set of leaves of γ. There exist a set $W \subset U$ and two univalent vertices $\bar{S} \cdot S_0$ and $\bar{S} \cdot S_1$ in E such that*

(a) $|W| = |U| - n$,
(b) val$(\bar{S} \cdot S_0) \neq$ val$(\bar{S} \cdot S_1)$, and
(c) processes in W have the same states in $\bar{S} \cdot S_0(I^l)$ and $\bar{S} \cdot S_1(I^l)$.

Proof. Let γ be a complete rake with a pivot $\bar{S}$ and a participating set U such that $|U| = m \geq n+1$. Let X be the object of type T such that each process $q \in U$ accesses X in $\bar{S}(I^l)$. Let σ_X be the state of X in $\bar{S}(I^l)$.

Construct a graph $\mathcal{K}$ as follows. The set of vertices of $\mathcal{K}$ is E. Two vertices $\bar{S} \cdot S$ and $\bar{S} \cdot S'$ of $\mathcal{K}$ are connected with an edge if at least $m - n$ processes p have the same states in $\bar{S} \cdot S(I^l)$ and $\bar{S} \cdot S'(I^l)$. Now we color each vertex $\bar{S} \cdot S$ of $\mathcal{K}$ with $val(\bar{S} \cdot S)$.

Claim 12 *Vertices of $\mathcal{K}$ are colored* 0 *or* 1*, and $\mathcal{K}$ has at least one vertex of color* 0 *and at least one vertex of color* 1*.*

Proof of Claim 12. Immediate from the definition of $\mathcal{K}$. □

Now we show that $\mathcal{K}$ is connected. By contradiction, assume that $\mathcal{K}$ consists of two or more connected components. Let $\mathcal{K}_0$ be any connected component of $\mathcal{K}$ and $\mathcal{K}_1 = \mathcal{K} - \mathcal{K}_0$. Note that for any two vertices $S \in \mathcal{K}_0$ and $S' \in \mathcal{K}_1$, there are at most $m - n - 1$ processes that have the same states in $S(I^l)$ and $S'(I^l)$. We establish a contradiction by showing consensus can be solved among $n + 1$ processes using registers and objects of type T.

Claim 13 *There is an algorithm that solves* n-resilient weak consensus *among m processes using registers and one object of type T, initialized to σ_X.*

Proof of Claim 13. Let X, an object of type T, be initialized to σ_X. Every process $p \in U$ executes one step of $\mathsf{Cons}_{\mathcal{D}}$ determined by p's state in $\bar{S}(I^l)$ (by the definition of γ, in this step, p accesses X). Process p writes its resulting state (a *view*) in register Y_p and then keeps collecting registers $\{Y_q\}_{q \in \Pi}$ until at least $m - n$ distinct views are collected. If the collected views belong to a state $S(I^l)$ such that $S \in \mathcal{K}_0$, then p decides 0. Otherwise, p decides 1.

Termination is ensured as long as not more than n processes fail. Since any $m - n$ distinct views identify to which component the resulting state belongs, Agreement is satisfied. Non-Triviality follows from the fact that $\mathcal{K}_0$ and $\mathcal{K}_1$ are not empty. □

Claim 14 *There is an algorithm that solves weak consensus among $n+1$ processes using registers, test-and-set objects, and one object of type T, initialized to σ_X.*

Proof of Claim 14. Let $\mathcal{A}'$ be an algorithm that solves *n-resilient* weak consensus among set U of m processes, $q_1, q_2, \ldots, q_m$, using registers and one objects of type T, initialized to σ_X (by Claim 13, such an algorithm exists). In Figure 5, we describe an algorithm that solves weak consensus *among $n + 1$ processes* $p_1, p_2, \ldots, p_{n+1}$ using registers, test-and-set objects and one object of type T, initialized to σ_X.

Initially:

$\forall q_j \in U$, Z_j contains the initial state of $\mathcal{A}'(q_j)$

X and $\{Y_j\}_{q_j \in U}$ are as in the initial state of $\mathcal{A}'$

Procedure WEAKCONSENSUS():

1: **repeat**
2: $q_j \leftarrow$ ⟨ the next process in U in a fair order ⟩
3: ⟨ the current state of q_j ⟩ $\leftarrow Z_j$
4: **if** ⟨ T_j.*test-and-set*() $= 0$ ⟩
5: **then** { p *wins* T_j }
6: perform one step of q_j defined by $\mathcal{A}'$
7: $Z_j \leftarrow$ ⟨ the new state of q_j ⟩
8: T_j.*reset*() { *reset* T_j }
9: **until** ⟨ q_j decides a value v in its current state ⟩
10: return v

Fig. 5. Simulating m processes with registers and test-and-set objects: process p_i

In the algorithm, by employing the technique of [5], we make processes $p_1, p_2, \ldots, p_{n+1}$ *simulate* processes $q_1, q_2, \ldots, q_m$ running algorithm $\mathcal{A}'$. For this, each process p_i attempts to perform steps of all m processes $q_1, q_2, \ldots, q_m$ in a "fair" fashion, so that every simulated process q_j gets infinitely many chances to get its steps of $\mathcal{A}'$ performed.

To prevent multiple processes from performing the same step of a simulated process q_j, we require that a process gain exclusive access to q_j before performing q_j's steps. This is implemented by associating with q_j a test-and-set object T_j. When a process p_i is about to perform a step of $\mathcal{A}'$ prescribed for a simulated process q_j, it first performs a test-and-set operation on T_j. If p_i wins T_j (i.e., T_j.*test-and-set*() returns 0), then p_i gets the current simulated state of q_j stored in register Z_j, performs the next step of q_j defined by $\mathcal{A}'$ and updates the state of q_j in Z_j. Then p_i resets T_j so that some other process could perform the next step of q_j. If p_i loses T_j (i.e., T_j.*test-and-set*() returns 1), then p_i moves on to the next simulated process in a fair fashion. If in its simulated state, q_j decides v, then p_i returns v.

Note that at any time, p_i is preventing at most one simulated process from being accessed by other processes. Hence, a crash of p_i can make at most one simulated process inaccessible. Thus, even if n processes crash, the remaining process will still be able to simulate infinitely many steps of at least $m-n$ simulated processes. Thus, the algorithm of Figure 5 simulates runs of $\mathcal{A}'$ in which at most n processes are faulty. Since $\mathcal{A}'$ solves n-resilient weak consensus, our algorithm satisfies the Termination and Agreement properties of weak consensus.

Now we show that our algorithm satisfies Non-Triviality. For this, we put some additional restrictions on the fair order in which processes p_1 and p_2 simulate the steps of processes in U. Let

R_0 be a run of $\mathcal{A}'$ in which 0 is decided and R_1 be a run of $\mathcal{A}'$ in which 1 is decided (these runs exist, since $\mathcal{A}'$ satisfies Non-Triviality of weak consensus). In our algorithm, p_1 cyclically simulates the steps of processes in U in the order determined by R_0. Respectively, p_2 cyclically simulates the steps of processes in U in the order determined by R_1. Hence, if p_1 is the only correct process, then p_1 eventually decides 0. Respectively, if p_2 is the only correct process, then p_2 decides 1. Thus, the Non-Triviality property of weak consensus is ensured. □

By Claim 14 and Lemma 4, we can solve consensus among $n+1$ processes using registers, test-and-set objects, and one object of type T, initialized to σ_X. By Lemma 1, we can solve consensus among $n+1$ processes using only registers and objects of type T, initialized to σ_X. By the construction, σ_X (the state of X in $\bar{S}(I^l)$) is reachable. By Lemma 2, $\{T\}$ solves consensus among $n+1$ processes — a contradiction with the assumption that $cons(T) \leq n$.

Thus, $\mathcal{K}$ is connected. By Claim 12, there are at least two vertices $\bar{S} \cdot S$ and $\bar{S} \cdot S'$ in E of different colors, connected with an edge. Thus, there is a set W of $|U| - n$ process that have the same states in $\bar{S} \cdot S(I^l)$ and $\bar{S} \cdot S'(I^l)$, and we have the lemma. □

5.7 Critical index

We say that index $l \in \{1, 2, \ldots, k\}$ is *critical* in P if *either* $\Upsilon^{P,l}$ contains a decision gadget *or* the root of $\Upsilon^{P,l-1}$ is 0-valent, and the root of $\Upsilon^{P,l}$ is 1-valent. In the first case, we say that l is *bivalent critical*. In the second case, we say that l is *univalent critical*.

Lemma 15 *Let correct$(F) \subseteq P \subseteq \Pi$. There exists a critical index in P.*

Proof. By Validity of consensus, $\Upsilon^{P,0}$ is 0-valent and $\Upsilon^{P,k}$ is 1-valent. Hence, there exists $l \in \{1, 2, \ldots, k\}$ such that the root of $\Upsilon^{P,l-1}$ is 0-valent and the root of $\Upsilon^{P,l}$ is either 1-valent or bivalent. If the root of $\Upsilon^{P,l}$ is 1-valent, l is univalent critical. If the root of $\Upsilon^{P,l}$ is bivalent, by Lemma 7, $\Upsilon^{P,l}$ contains a decision gadget. Thus, l is critical. □

5.8 Deciding sets

Instead of the notion of a *deciding process* used in [6], we introduce the notion of a *deciding set* $V \subset \Pi$. The deciding set V of a *complete* decision gadget γ is defined as follows:

(1) Let γ be a fork defined by pivot $\bar{S}$ and steps e and e' of the same process q, such that $\bar{S} \cdot e$ and $\bar{S} \cdot e'$ are univalent vertices of $\Upsilon^{P,l}$ of opposite valence.
Then $V = \{q\}$.

(2) Let γ be a complete hook defined by a pivot $\bar{S}$, a step e of q, and a step e' of q' ($q \neq q'$).
By Lemma 10, there exists a process $p \in \{q, q'\}$ and two vertices S_0 and S_1 in $\{\bar{S}\cdot e, \bar{S}\cdot e'\cdot e, \bar{S}\cdot e\cdot e'\}$ such that (a) S_0 and S_1 are univalent vertices of opposite valence, and (b) $S_0(I^l)$ and $S_1(I^l)$ differ only in the state of p. Then we define the deciding set of γ as $V = \{p'\}$ where p' is the smallest such process..

(3) Let γ be a complete rake defined by a pivot $\bar{S}$, a participating set U, and a set of leaves E.

- If $|U| \leq n$, then we define the deciding set of γ as $V = U$.
- If $|U| \geq n+1$, then by Lemma 11 there is a set $W \subset U$ of $|U| - n$ "confused" processes such that, for some $\bar{S} \cdot S$ and $\bar{S} \cdot S'$ in E, processes in W have the same states in $\bar{S} \cdot S(I^l)$ and $\bar{S}\cdot S'(I^l)$, and $val(\bar{S}\cdot S) \neq val(\bar{S}\cdot S')$. Then we define the deciding set of γ as $V = U - W'$ where W' is the *smallest* such set (it is well-defined, since subsets U of can be totally ordered).

By the construction, in each case, V is a set of at most n processes. The following lemma uses the assumption that type T is *one-shot*.

Lemma 16 *The deciding set of a complete decision gadget contains at least one correct process.*

Proof. There are two cases to consider:

(1) Let γ be a fork with leaves S_0 and S_1 and a deciding set $\{p\}$. The difference between $S_0(I^l)$ and $S_1(I^l)$ consists only in the state of p. By Lemma 6, $V = \{p\}$ includes exactly one correct process.

(2) Let γ be a hook with a deciding set $V = \{p\}$. By Lemma 6, p is correct.

(3) Let γ be a complete rake defined by a pivot $\bar{S}$, a participating set U, and a set of leaves E. Let X be the object of type T accessed by steps of processes in U in $\bar{S}(I^l)$. The following cases are possible:

(3a) $|U| \leq n$.

Assume, by contradiction, that all processes in deciding set $V = U$ are faulty.
There exist two vertices $\bar{S} \cdot S_0$ and $\bar{S} \cdot S_1$ in E such that $val(\bar{S} \cdot S_0) = 0$ and $val(\bar{S} \cdot S_1) = 1$.
Since only processes in U take steps in S_0 and S_1 and each process $p \in U$ accesses X in $\bar{S}(I^l)$, the difference between $\bar{S}(I^l)$, $\bar{S} \cdot S_0(I^l)$ and $\bar{S} \cdot S_1(I^l)$ consists only in the states of processes in U and object X.
By Lemma 5, there is a schedule S containing only steps of correct processes (and thus no steps of processes in U) such that all correct processes have decided in $\bar{S} \cdot S(I^l)$ and, for any $S' \in E$, if S is applicable to $S'(I^l)$, then $S' \cdot S$ is a vertex of $\Upsilon^{P,l}$.
Suppose that S applied to $\bar{S}(I^l)$ accesses X, i.e., S' has a prefix $S'' \cdot (q, \lambda)$ such that S'' applied to $\bar{S}(I^l)$ does not access X and q accesses X in $\bar{S}(I^l)$. Let $S' \in E$. Since S' and $\bar{S}$ differ only in the states of processes in U and X, and S'' includes no steps of processes in

X and, applied to $\bar{S}(I^l)$, does not access X, $S'' \cdot (q, \lambda)$ is applicable to S', and q accesses X in $S'(I^l)$. This contradicts the definition of γ.

Thus, S applied to $\bar{S}(I^l)$ does not access X. Thus, S is also applicable to $\bar{S} \cdot S_0(I^l)$ and $\bar{S} \cdot S_1(I^l)$. Thus, $\bar{S} \cdot S_0 \cdot S$ and $\bar{S} \cdot S_1 \cdot S$ are vertices of $\Upsilon^{P,l}$.

But no process in $\Pi - U$ can distinguish $\bar{S} \cdot S(I^l)$, $\bar{S} \cdot S_0 \cdot S(I^l)$ and $\bar{S} \cdot S_1 \cdot S(I^l)$. Thus, the correct processes have decided the same values in these configurations — a contradiction.

(3b) $|U| \geq n + 1$, i.e., $U = \Pi$. Let $V = U - W$ be the deciding set of γ, i.e., processes in W cannot distinguish $\bar{S} \cdot S_0(I^l)$ and $\bar{S} \cdot S_1(I^l)$, the vertices of $\Upsilon^{P,l}$ of opposite valence. Thus, the difference between $\bar{S} \cdot S_0(I^l)$ and $\bar{S} \cdot S_1(I^l)$ consists only in the states of processes in V and object X.

By Lemma 5, there is a schedule S containing only steps of correct processes (and thus no steps of processes in V) such that all correct processes have decided in $\bar{S} \cdot S_0 \cdot S(I^l)$ and if S is applicable to $\bar{S} \cdot S_1 \cdot S(I^l)$, then $\bar{S} \cdot S_1 \cdot S$ is a vertex of $\Upsilon^{P,l}$.

Again, by the definition of γ, S applied to $\bar{S} \cdot S_0(I^l)$ does not access X. Hence, S is also applicable to $\bar{S} \cdot S_1(I^l)$. Thus, $\bar{S} \cdot S_1 \cdot S$ is a vertex of $\Upsilon^{P,l}$.

But no process in $\Pi - V$ can distinguish $\bar{S} \cdot S_0 \cdot S(I^l)$ and $\bar{S} \cdot S_1 \cdot S(I^l)$. Thus, the correct processes have decided the same values in these configurations — a contradiction.

In each case, the deciding set V is of size at most n and contains at least one correct process. □

5.9 The reduction algorithm

Theorem 17 *Let T be any one-shot deterministic type, such that cons(T) $\leq n$ and $2 \leq n$. If a failure detector $\mathcal{D}$ solves consensus using only registers and objects of type T, then $\Omega_n \preceq \mathcal{D}$.*

Proof. The communication task presented in Figure 2 and the computation task presented in Figure 6 constitute the reduction algorithm $T_{\mathcal{D} \to \Omega_n}$. The current estimate of Ω_n at process p is stored in variable Ω_n-*output*$_p$.

In the communication task described in Figure 2, every process p maintains an ever-growing DAG G_p.

In the computation task described in Figure 6, for each $P \subseteq \Pi$ and each $l \in \{0, \ldots, k\}$, process p constructs a finite simulation tree $\Upsilon_p^{P,l}$ induced by P, I^l and G_p and tags each vertex S of $\Upsilon_p^{P,l}$ with a set of decisions taken in all $S'(I^l)$ such that S' is a descendant of S in $\Upsilon_p^{P,l}$. Initially $P = \Pi$. Let P have a critical index and let l be the smallest critical index in P. If l is univalent, then p outputs $\{p_l\}$ (line 12). If l is bivalent, and the smallest decision gadget in $\Upsilon_p^{P,l}$, denoted γ, is complete, then p outputs the deciding set of γ (line 16). If l is bivalent, and γ is incomplete, then p removes missing (in γ) processes from P and proceeds to the next iteration (line 19).

Initially:

$\Omega_n\text{-}output_p \leftarrow \{p\}$

1: **while** true **do**
2: **for all** $P \subseteq \Pi$ and $l \in \{0, 1, \ldots, k\}$ **do**
3: $\Upsilon_p^{P,l} \leftarrow$ simulation tree induced by P, I^l and G_p
4: $V \leftarrow \emptyset$
5: $P \leftarrow \Pi$
6: **repeat**
7: **if** P has no critical index **then**
8: $V \leftarrow \{p\}$
9: **else**
10: let l be *the smallest* critical index of P
11: **if** l is univalent critical **then**
12: $V \leftarrow \{p_l\}$
13: **else**
14: $\gamma \leftarrow$ *the smallest* decision gadget in $\Upsilon_p^{P,l}$
15: **if** γ is complete **then**
16: $V \leftarrow$ the deciding set of γ
17: **else**
18: let W be the set of missing processes in γ
19: $P \leftarrow P - W$
20: **until** $V \neq \emptyset$ **or** $P = \emptyset$
21: **if** $P = \emptyset$ **then** $V \leftarrow \{p\}$
22: $\Omega_n\text{-}output_p \leftarrow V$

Fig. 6. Extracting Ω_n: process p

Note that the "repeat-until" cycle in lines 6–20 is non-blocking. Indeed, p eventually sets V to a non-empty value (in lines 8, 12 or 16), or sets P to $\emptyset$ in line 19. In both cases, p eventually exits the "repeat-until" cycle.

Recall that finite simulation trees $\Upsilon_p^{P,l}$ at all correct processes p tend to the same *infinite* simulation tree $\Upsilon^{P,l}$. Let F be the current failure pattern.

Claim 18 *There exist $P^* \subseteq \Pi$, $correct(F) \subseteq P^*$, such that there is a time after which every correct process p has $P = P^*$ in line 21.*

Proof of Claim 18. By Lemma 15, every $P \subseteq \Pi$ such that $correct(F) \subseteq P$ has a critical index. Thus, there is a time after which the correct processes compute the same critical index l in every such P, and if l is bivalent, then the correct processes locate the same smallest (complete or incomplete) decision gadget in $\Upsilon^{P,l}$.

By Lemma 8, there is a time after which whenever a correct process p reaches line 19, $W \subseteq faulty(F)$. Thus, there is a time after which one of the following cases always holds:

(a) p exits the "repeat-until" cycle in line 12 after having located a univalent critical index in some P such that $correct(F) \subseteq P$.
(b) p exits the "repeat-until" cycle in line 16 after having located a complete decision gadget in univalent critical index in $\Upsilon^{P,l}$ where $P \subseteq \Pi$ and $correct(F) \subseteq P$.
(c) p reaches line 14 with $P = correct(F)$.

In case (c), by Corollary 9, there is a time after which the smallest decision gadget in $\Upsilon^{P,l}$ is complete and p exits the "repeat-until" cycle in line 16. In all cases, there exists $P^* \subseteq \Pi$ such that $correct(F) \subseteq P$, and there is a time after which every correct process has $P = P^*$ in line 21. □

Thus, there exist $P \subseteq \Pi$ and $V^* \neq \emptyset$ such that every correct process eventually reaches line 21 with $P = P^*$ and $V = V^*$. Let l be the smallest critical index in P^*. By the algorithm, the following cases are possible:

(1) l is univalent critical. That is, the root of $\Upsilon^{P^*,l-1}$ is 0-valent and the root of $\Upsilon^{P^*,l}$ is 1-valent. In this case, eventually, every correct process p permanently outputs $V^* = \{p_l\}$. I^{l-1} and I^l differ only in the state of process p_l. By Lemma 5, p_l is correct.
(2) l is bivalent critical. Moreover, the smallest decision gadget in $\Upsilon^{P^*,l}$ is complete. In this case, eventually, every correct process p permanently outputs the deciding set V^* (of size at most n) of the complete decision gadget. By Lemma 16, the deciding set of γ includes at least one correct process.

In both cases, eventually, the correct processes agree on a set of at most n processes that include at least one correct process. Thus, our reduction algorithm (Figures 2 and 6) emulates the output

of Ω_n. □

Theorem 17 and the algorithm of [21] imply the following:

Theorem 1. *Let T be any one-shot deterministic type such that cons$(T) = n$ and $n \geq 2$. Then Ω_n is the weakest failure detector to solve consensus using registers and objects of type T.*

As a corollary of Theorem 1, assuming that only registers are available, we obtain the following result, stated in [18].

Corollary 19 *Ω is the weakest failure detector to solve consensus using only registers.*

6 Boosting object resilience

So far we considered systems in which processes communicate through *wait-free* linearizable implementations of shared objects. Every process can complete every operation on a wait-free object in a finite number of its own steps, regardless of the behavior of other processes. In contrast, in this section we consider *t-resilient* linearizable implementations (we will simply call these t-resilient objects). These implementations only guarantee that a process completes its operation, as long as no more than t processes crash, where t is a specified parameter. If more than t processes crash, no operation on a t-resilient implementation is obliged to return.

It is impossible to solve $(t+1)$-resilient consensus among $n > t+1$ processes using *only* wait-free registers and t-resilient objects. An indirect proof that of this statement, based on the results of [5, 13, 17], appeared in [11]. A direct self-contained proof of this statement appeared in [2], and then it was extended to more general classes of *distributed services* in [3].

Not surprisingly, this impossibility can be circumvented by augmenting the system with a failure detector abstraction. Assume that k processes communicate through wait-free registers and t-resilient objects. In this section, we show that Ω_{t+1} is the weakest failure detector to solve consensus in this system.

The following two lemmas are restatements in our terminology of the "necessity" part and the "sufficiency" part of Theorem 4.1 in [5], respectively.

Lemma 1. *Let t and n be integers, $0 \leq t$, $1 \leq n$. Then there exists a t-resilient n-process implementation of consensus from wait-free $(t+1)$-process consensus objects and wait-free registers.*[1]

Lemma 2. *Let t and n be integers, $2 \leq t < n$. Then there exists a wait-free $(t+1)$-process implementation of consensus from t-resilient n-process consensus objects and wait-free registers.*

The following result follows easily from Herlihy's universal construction [13]:

[1] Theorem 4.1 in [5] assumes $2 \leq t$. However, the necessity part of the theorem requires only $0 \leq t$.

Lemma 3. *Let t and n be integers, $0 \leq t$, $1 \leq n$. Let T be any type. Then there exists a t-resilient n-process implementation of an atomic object of type $\mathcal{T}$ from t-resilient n-process consensus objects and wait-free registers.*

The following result is shown in [17]:

Lemma 4. *Let n be integer, $n \geq 0$. There does not exist a wait-free $(n+1)$-process implementation of consensus from wait-free n-process consensus objects and wait-free registers.*

Our result on boosting the consensus power of one-shot deterministic types implies the following result:

Theorem 20 *Let t be any integer, $t \geq 2$. Let T be any type (not necessarily one-shot deterministic), such that registers and t-resilient objects of type T solve t-resilient consensus. Ω_{t+1} is then the weakest failure detector to solve consensus using wait-free registers and t-resilient objects of type T.*

Proof. By Lemma 2, $(t+1)$-process consensus can be implemented from wait-free registers and t-resilient objects of type T. The algorithm of [21] implements wait-free consensus using registers, $(t+1)$-process consensus objects and Ω_{t+1}. This gives the sufficient part of the theorem.

Assume now that a failure detector $\mathcal{D}$ solves consensus using registers and t-resilient objects of type T. By Lemmas 1, 2 and 3 any t-resilient object can be implemented from wait-free registers and $(t+1)$-process consensus objects. By Lemma 4, $cons((t+1)\text{-process consensus}) \leq t+1$. Furthermore, $(t+1)$-process consensus is a one-shot deterministic type.

Thus, $\mathcal{D}$ solves consensus using registers and objects of one-shot deterministic of consensus power $t+1$. By Theorem 17, $\Omega_{t+1} \preceq \mathcal{D}$. This gives the necessary part of the theorem. □

References

1. Y. Afek, D. Greenberg, M. Merritt, and G. Taubenfeld. Computing with faulty shared memory. In *Proceedings of the 11th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, 1992.
2. P. Attie, N. A. Lynch, and S. Rajsbaum. Boosting fault-tolerance in asynchronous message passing systems is impossible. Technical report, MIT Laboratory for Computer Science, MIT-LCS-TR-877, 2002.
3. P. C. Attie, R. Guerraoui, P. Kouznetsov, N. A. Lynch, and S. Rajsbaum. The impossibility of boosting distributed service resilience. In *Proceedings of the 25th IEEE International Conference on Distributed Computing Systems (ICDCS'05)*, June 2005. Available at http://theory.lcs.mit.edu/tds/papers/Attie/boosting-tr.ps.
4. E. Borowsky, E. Gafni, and Y. Afek. Consensus power makes (some) sense! In *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 363–372, August 1994.
5. T. D. Chandra, V. Hadzilacos, P. Jayanti, and S. Toueg. Wait-freedom vs. t-resiliency and the robustness of wait-free hierarchies. In *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 334–343, August 1994.
6. T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722, July 1996.

7. T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.
8. D. Dolev, C. Dwork, and L. J. Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM*, 34(1):77–97, January 1987.
9. C. Dwork, N. A. Lynch, and L. J. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288 – 323, April 1988.
10. M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(3):374–382, April 1985.
11. R. Guerraoui and P. Kouznetsov. On failure detectors and type boosters. In *Proceedings of the 17th International Symposium on Distributed Computing (DISC'03)*, October 2003.
12. R. Guerraoui and P. Kouznetsov. The gap in circumventing the consensus impossibility. Technical report, EPFL, ID:IC/2004/28, January 2004. Available at http://icwww.epfl.ch/publications/.
13. M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.
14. M. Herlihy and E. Ruppert. On the existence of booster types. In *Proceedings of the 41st IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 653–663, 2000.
15. M. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, June 1990.
16. P. Jayanti. Robust wait-free hierarchies. *Journal of the ACM*, 44(4):592–614, 1997.
17. P. Jayanti and S. Toueg. Some results on the impossibility, universability and decidability of consensus. In *Proceedings of the 6th International Workshop on Distributed Algorithms (WDAG'92)*, volume 647 of *LNCS*. Springer Verlag, 1992.
18. W.-K. Lo and V. Hadzilacos. Using failure detectors to solve consensus in asynchronous shared-memory systems. In *Proceedings of the 8th International Workshop on Distributed Algorithms (WDAG'94)*, volume 857 of *LNCS*, pages 280–295. Springer Verlag, 1994.
19. W.-K. Lo and V. Hadzilacos. All of us are smarter than any of us: Nondeterministic wait-free hierarchies are not robust. *SIAM Journal of Computing*, 30(3):689–728, 2000.
20. M. C. Loui and H. H. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. *Advances in Computing Research*, pages 163–183, 1987.
21. G. Neiger. Failure detectors and the wait-free hierarchy. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 100–109, August 1995.
22. E. Ruppert. Determining consensus numbers. *SIAM Journal of Computing*, 30(4):1156–1168, 2000.