Distrib. Comput. (2008) 20:343–358
DOI 10.1007/s00446-007-0043-z

Failure detectors as type boosters

Rachid Guerraoui · Petr Kouznetsov

Received: 1 June 2004 / Accepted: 26 July 2007 / Published online: 9 October 2007

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 Neiger (Proceedings of the 14th annual ACM symposium on principles of distributed computing (PODC), pp. 100–109, 1995) 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 first 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 $n+1$. Then we go a step further and show that Ω_n is also the weakest to boost the power of $(n+1)$-*ported* one-shot deterministic types 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 (Chandra et al. in J ACM 43(4):685–722, 1996). As a corollary, we show that Ω_t is the weakest failure detector to boost the *resilience* level of a distributed shared memory system, i.e., to solve consensus among $n > t$ processes using $(t-1)$-resilient objects of consensus power t.

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

R. Guerraoui (✉) · P. Kouznetsov
Distributed Programming Laboratory, EPFL,
1015 Lausanne, Switzerland
e-mail: rachid.guerraoui@epfl.ch

P. Kouznetsov
Max Planck Institute for Software Systems,
Stuhlsatzenhausweg 85, 66123 Sarbrücken, Germany

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 [9,11,21]. Circumventing this impossibility has sparked off two research trends:

(1) Augmenting the system model with *synchrony* assumptions about relative process speeds and communication delays [10]. Such assumptions could be encapsulated within a *failure detector* abstraction [8]. 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 an asynchronous message-passing system [7]. The result was extended later to the read–write shared memory model [19]. 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 any failure detector that solves consensus can emulate the output of 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 as shared object types with sequential specifications [14,21]. 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 [14]. 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 [14,21].

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 [23]. 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 [15], $\mathcal{D}_n$ 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 [23]. 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 [23] 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 [23] that Ω_n is the weakest failure detector to boost the power of T to the level $n+1$ of the consensus hierarchy. As pointed out in [23], 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 [15]. 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 [5,15,17,20].

We show that Ω_n is *necessary* to solve consensus in a system of $k = n+1$ processes using objects of any one-shot deterministic type T ($cons(T) = n$) and registers. Then we generalize the result by showing that Ω_n is the weakest failure detector to solve consensus in a system of k ($k > n$) processes using registers and $(n+1)$-ported objects of type T.

Our result is a strict generalization of the fundamental result of [7] 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 $cons(T) = n$ and T is one-shot and deterministic. This lack of information forced us to reconsider the proof of [7]. In particular, we reuse and generalize the notions of simulation tree, decision gadget and deciding process introduced in [7].

As a side effect, we get a formal proof that any failure detector that can be used to solve consensus among $k \geq 2$ processes in the read-write memory model can be transformed to Ω. The result was first stated in [19] but, to our knowledge, its proof has never appeared in the literature.

As another interesting corollary of our result, we identify the weakest failure detector to boost the *resilience* level of a distributed shared memory system. More precisely, consider a systems of n processes that communicate through read-write registers and t-*resilient* objects. Informally, an object implementation is called t-resilient if any operation on the object terminates unless more than t processes fail, where t is a specified parameter. We show that Ω_{t+1} is necessary and sufficient to solve consensus among $n > t+1$ processes using t-resilient objects of consensus power $t+1$.

Related work. The notion of consensus power was introduced by Herlihy [14] and then refined by Jayanti [17]. Chandra et al. [7] showed in that Ω is the weakest failure detector to solve consensus in asynchronous message-passing systems with a majority of correct processes. Lo and Hadzilacos showed that Ω can be used to solve consensus with registers and outlined a proof that any failure detector that can be used to solve consensus with registers can be transformed to Ω [19]. Neiger [23] 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. Neiger also conjectured in [23] that Ω_n was actually necessary for solving consensus using objects of consensus power n and gave a high-level outline of a potential proof of this conjecture. An indirect proof that it is impossible to boost the resilience of atomic objects without using failure detectors appeared first in [13] (this fact was independently and concurrently observed by P. Jayanti (private communication, 2003). A direct self-contained proof of this result appeared in [2], and then it was extended to more general classes of distributed services in [3].

Roadmap. Section 2 presents necessary details of the model used in this chapter. We also recall here the hierarchy of failure detectors Ω_n. Section 3 shows that Ω_n is necessary to boost the consensus power of one-shot deterministic objects one level up. Section 4 generalizes the result to any number of levels. Section 5 applies our result to the question of boosting the resilience of a distributed system with respect to the consensus problem. Section 6 concludes the paper with a discussion on how our proof relates to the Neiger's original outline of how to approach such a proof [23].

2 Model

Our model of processes communicating through shared objects is based on that of [16,17] and our notion of failure detectors follows from [7]. Below we recall what is substantial to show our result.

Processes

We consider a set Π of k asynchronous processes $p_1, \ldots, p_k$ ($k \geq 2$) 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 *type* which is defined by a tuple (Q, O, m, R, δ). Here Q is a set of *states*, O is a set of *operations*, m is a positive integer denoting the number of *ports* (used as the 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 port, state and operation to a set of response and state pairs. We assume that objects are *deterministic*: the sequential specification is a function $\delta : \mathbb{N}_m \times Q \times O \to Q \times R$. A type is said to be *m-ported* if it has m ports. If not explicitly specified, we assume that the number of ports of any object is k, i.e., the object is connected to every process.

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.

We say that type $T = (Q, O, m, R, \delta)$ is *one-shot* if $\perp \in R$, $Q = 2^{\mathbb{N}_m} \times Q'$, for some Q', such that for all $(S, q) \in Q$, $j \in \mathbb{N}_m$, and $op \in O$, if $j \notin S$, then $\delta(j, (S, q), op) = ((S \cup \{j\}, q'), r)$ where $q' \in Q'$ and $r \in R$, otherwise (if $j \in S$), $\delta(j, (S, q), op) = ((S, q), \perp)$. Informally, a port of a one-shot object can return meaningful (non-$\perp$) response at most once in any execution: every subsequent operation applied on the port does not impact the object and returns $\perp$.

We consider here *linearizable* [4,16] 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, where the last operation invoked by a failed process may be *linearized* (appointed to take effect) at any time after the invocation.

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.

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 $correct(F) = \Pi - \cup_{t \in \mathbb{T}} F(t)$, the set of *correct* processes in F. Processes in $\Pi - correct(F)$ 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 $U \cap correct(F) \neq \emptyset$. We consider here all failure *environments* [7], 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. Note that we do not make any assumption a priori on the range of a failure detector. When any process p performs a step of computation, it can *query* its module of $\mathcal{D}$, denoted $\mathcal{D}_p$, and obtain a value $d \in \mathcal{R}_\mathcal{D}$ that encodes some information about failures.

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 [7]. Formally, for each failure pattern F, $H \in \Omega(F)$ if and only if $\exists t \in \mathbb{T}\, \exists q \in correct(F)\, \forall p \in correct(F)\, \forall t' \geq t : H(p, t') = q$.

2.4 Algorithms

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

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

A step of 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 A defines the current state of each process and each object in the system. An *initial configuration* of A specifies the initial state of every $A(p)$ and every object used by the algorithm.[1]

The state of any process p in C determines whether in any step of p applied to C, p queries its failure detector module or accesses a shared object. 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 its failure detector $\mathcal{D}_p$ in its next step in C (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 A is a (finite or infinite) sequence of steps of 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 A using a failure detector* $\mathcal{D}$ is a tuple $R = \langle F, H, I, S, T \rangle$ where F is a failure pattern, $H \in \mathcal{D}(F)$ is a failure detector history, I is an initial configuration of A, S is a *finite* schedule of 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., $d = H(p, T[k])$.

A *run of algorithm A using a failure detector* $\mathcal{D}$ 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 configuration of A, S is an *infinite* schedule of 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.

[1] The consensus power of type T does not depend on whether or not the algorithms are allowed to choose the initial states of objects of type T [5].

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 *A solves a problem M in an environment $\mathcal{E}$ using a failure detector $\mathcal{D}$* if the set of all runs of A in $\mathcal{E}$ using $\mathcal{D}$ satisfies M. We say that a failure detector $\mathcal{D}$ *solves problem M* in $\mathcal{E}$ if there is an algorithm A which solves M in $\mathcal{E}$ using $\mathcal{D}$.

2.7 A weakest failure detector

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

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

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

Algorithm $T_{\mathcal{D}'\to\mathcal{D}}$ that emulates histories of $\mathcal{D}$ using histories of $\mathcal{D}'$ is called a *reduction* algorithm. Note that $T_{\mathcal{D}'\to\mathcal{D}}$ 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 M in an environment* $\mathcal{E}$ if the following conditions are satisfied:

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

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 M in $\mathcal{E}$.

2.8 Consensus

The (binary) *m-process consensus* problem [11] consists for m processes to decide on some final values (0 or 1) based on their initial proposed values in such a way that: (*Agreement*) no two processes decide on 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 a one-shot object type. Formally, the m-process consensus type is specified as a tuple (Q, O, m, R, δ), where $Q = 2^{\mathbb{N}_m} \times \{\lambda, 0, 1\}$, $O = \{propose(v) : v \in \{0, 1\}\}$, $R = \{\perp, 0, 1\}$, and for all $v, v' \in \{0, 1\}$, $S \in 2^{\mathbb{N}_m}$, and $j \in \mathbb{N}_m - S$, : $\delta(j, (S, \lambda), propose(v)) = ((S \cup \{j\}, v), v)$ and $\delta(j, (S, v'), propose(v)) = ((S \cup \{j\}, v'), v')$.

We say that T *solves* m-process consensus if there is an algorithm that solves m-process consensus using registers and objects of type in T.

The *consensus power* [14,17] 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 $cons(T) = \infty$.

To prove our result, we also use a restricted form of consensus, *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 [23,24]:

Lemma 1 *Let T be any type. If T solves team consensus among m processes, then T 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 $m > 2$, T solves team consensus among m processes (for non-empty teams A and B), and (2) for all $2 \le l < m$, T 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 the same team propose the same value, Agreement of m-process consensus is satisfied. □

2.9 Hierarchy of failure detectors Ω_n

The hierarchy of failure detectors Ω_n ($n \in \mathbb{N}$) was introduced in [23]. 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, $\mathcal{R}_{\Omega_n} = \{P \subseteq \Pi : |P| \le n\}$, and for each failure pattern F, $H \in \Omega_n(F) \quad \Leftrightarrow$

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

Clearly, Ω_1 is equivalent to Ω. It was furthermore shown in [23] 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 $cons(T) = n$, Ω_n can be used to solve k-process consensus using registers and objects of type T.

3 Boosting types to level $n+1$

In this section, we assume that $k = n+1$ processes communicate through registers and objects of a one-shot deterministic type T such that $cons(T) \leq n$. We show that Ω_n is necessary to solve consensus in this system. Our proof is a natural generalization of the proof that Ω is necessary to solve consensus in message-passing asynchronous systems [7].

3.1 An overview of the reduction algorithm

Let $\mathsf{Cons}_{\mathcal{D}}$ be any algorithm that solves consensus using registers, objects of a one-shot determinstic type T, and a failure detector $\mathcal{D}$. 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 uses its DAG G_p to periodically 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 $\Upsilon_p^{P,I}$ converge 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 there exists a correct *critical* process whose proposal value in some initial configuration I defines the decision value in all paths in $\Upsilon^{P,I}$, or 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 one of which is correct process. This set of processes is called the *deciding set* of γ. Eventually, the correct processes either detect the same critical process or detect 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 suggested by Lo and Hadzilacos [7] to compute the deciding set. Fortunately, in this case, it is possible to locate a special kind of decision gadget, which we introduce here and which we call a *rake*.

Leaves of the rake are configurations that result after each process applies one operation on the same object of type T to a given configuration of $\mathsf{Cons}_{\mathcal{D}}$. Moreover, every leave x of the rake is *univalent*, i.e., there is exactly one value that can be decided in any run of $\mathsf{Cons}_{\mathcal{D}}$ extending x, and there is at least one 0-valent and at least one 1-valent leave of the rake. Using the assumptions that T is a one-shot deterministic type and $cons(T) \leq n$, we derive that there must be at least one "confused" process p_i that is not able to distinguish, in any solo execution, two univalent configurations x_0 and x_1 of opposite valence. Thus, p_i will never be able to decide on its own starting from x_0 or x_1. This implies that the set of n other processes (the deciding set of the rake) must include at least one correct process that would provide p_i with the decision value.

3.2 The communication task and DAGs

The communication task of algorithm $T_{\mathcal{D}\to\Omega}$ is presented in Fig. 1. 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 can be updated by p and read by all processes.)

DAG G_p has some special properties which follow from its construction [7]. Let F be the current failure pattern in $\mathcal{E}$ and H be the current failure detector history in $\mathcal{D}(F)$. Then for any correct process p and any time t ($x(t)$ denotes the value of variable x at time t):

(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 τ:

```
Initially:
  G_p ← empty graph
  k_p ← 0

while true
  for all q ∈ Π do  G_p ← G_p ∪ G_q
  d_p ← query failure detector D
  k_p ← k_p + 1
  add [p, d_p, k_p] and edges from all vertices of G_p to [p, d_p, k_p] to G_p
```

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

vertices of$G_p(t) \mapsto \mathbb{T}$, associate a time with every vertex of $G_p(t)$, 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'), $\tau(v) < \tau(v')$. 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 $G_p(t)$ and $k < k'$ then (v, v') is an edge of $G_p(t)$.
(3) $G_p(t)$ is transitively closed: if (v, v') and (v', v'') are edges of $G_p(t)$, then (v, v'') is also an edge of $G_p(t)$.
(4) For all correct processes q, 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 $G_p(t')$.

Note that properties (1)–(4) imply that for any set of vertices V of $G_p(t)$, there is a time 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 $G_p(t')$.

3.3 Simulation trees

Let I^l ($l = 0, \ldots, n+1$) denote an initial configuration of $\mathsf{Cons}_{\mathcal{D}}$ in which processes $p_1, \ldots, p_l$ propose 1 and processes $p_{l+1}, \ldots, p_{n+1}$ 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 every $P \subseteq \Pi$, 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 schedule $S = e_1 e_2, \ldots, e_s$.

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 $\{u\}$ ($u \in \{0, 1\}$), 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. The tree $\Upsilon_p^{P,l}$ is called u-valent (resp., bivalent) if $S_\perp$ is u-valent (resp., bivalent) in $\Upsilon_p^{P,l}$.

Thanks to reliable communication provided by the read/write shared memory, 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 2 *Let $correct(F) \subseteq P \subseteq \Pi$, $m \geq 1$, 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 3 *Let $correct(F) \subseteq P \subseteq \Pi$. Let S_0 and S_1 be two univalent vertexes 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 2, 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. □

3.4 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

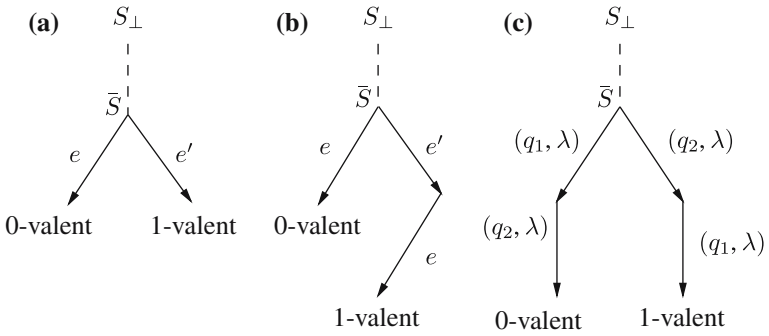

Fig. 2 A fork, a hook, and a rake

opposite valence. Then $\bar{S} \cdot e$ and $\bar{S} \cdot e'$ constitute the set of leaves of γ.

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 $\bar{S}(I^l)$.

Then $\bar{S} \cdot e$ and $\bar{S} \cdot e' \cdot e$ constitute the set of leaves of γ. Note (ii) implies that either at least one step in $\{e, e'\}$ is a query step, or q and q' access different objects in $\bar{S}(I^l)$, or q and q' access the same register in $\bar{S}(I^l)$.

If for every $x \in \mathcal{R}_\mathcal{D} \cup \{\lambda\}$, $\bar{S} \cdot e \cdot (q', x)$ 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, for any $q \in U$, the next step of q 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. Note that every such S is applicable to $\bar{S}(I^l)$. $\bar{S}$, 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 $q_1, q_2, \ldots, q_{|U|}$ 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.

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

Intuitively, the rake handles the case when the decision value is encoded in the responses returned by an object X of type T: no process p_i can decide in any execution extending $\bar{S}(I^l)$ unless p_i previously accessed X or heard from another process that accessed X. Furthermore, we show in this section that in case the rake is complete and $U = \Pi$ (all $n+1$ processes access X in $\bar{S}(I^l)$), there must be at least one "confused" process p_i that cannot distinguish two univalent configurations of opposite valence. Since T is one-shot, this confused process p_i can only learn the decision value from another process and thus p_i cannot be the only correct process.

Examples of decision gadgets are depicted in Fig. 2: (a) a fork with $e = (q, d)$ 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 $\bar{S}(I^l)$.

Lemma 4 *Let correct*$(F) \subseteq P \subseteq \Pi$ *and* $l \in \{1, 2 \ldots, n\}$. *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 [7], 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.

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 $S = (q_1, \lambda), (q_2, \lambda), \ldots, (q_{|U|}, \lambda)$ and $q_1, q_2, \ldots, q_{|U|}$ is a permutation of processes in U
14 $\quad$ **else exit**

Fig. 3 Locating a rake in $\Upsilon^{P,l}$

(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 Fig. 3 locates a rake in $\Upsilon^{P,l}$.

We show first that the algorithm terminates. Indeed, eventually either $U = \Pi$ and there is trivially no $(\Pi - U)$-schedules applicable to all $S \in \{\bar{S}\} \cup E$ in $\Upsilon^{P,l}$, or the algorithm terminates earlier in line 14.

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 $\bar{S}(I^l)$. Then:

(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 Fig. 3, $S = S^* \cdot S'$ where every process in U takes exactly one step in S', and, 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 is 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. □

3.5 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 5 *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 4 and 5 imply the following:

Corollary 6 *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.*

3.6 Confused processes

Lemma 7 *Let 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 q and q' access different objects in $\bar{S}(I^l)$, or q' is not a query step in $\bar{S}(I^l)$. Thus, $e' = (q', \lambda)$, and $\bar{S} \cdot e \cdot e'$ 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 r 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 r 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 r 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 assumption that type T is *deterministic*.

Lemma 8 *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 $|U| = n + 1$. Let E be the set of leaves of γ. There exist a process $p \in U$ and two univalent vertices $\bar{S} \cdot S_0$ and $\bar{S} \cdot S_1$ in E such that*

(a) *val$(\bar{S} \cdot S_0) \neq$ val$(\bar{S} \cdot S_1)$, and*
(b) *p has the same state in $\bar{S} \cdot S_0(I^l)$ and $\bar{S} \cdot S_1(I^l)$.*

Proof Assume that there are two vertices $\bar{S} \cdot S$ and $\bar{S} \cdot S'$ in E, such that S and S' begin with a step of the same process p, and val$(\bar{S} \cdot S) \neq$ val$(\bar{S} \cdot S')$. Since p takes exactly one step in both S and S', and this step of p is the first step in both S and S', the states of p in $\bar{S} \cdot S(I^l)$ and $\bar{S} \cdot S'(I^l)$ are identical, and we have the lemma.

Assume now that the valence of every vertex $\bar{S} \cdot S$ in E is defined by the id of a process that takes the first step in S. 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 one process p has the same state 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$(\bar{S} \cdot S)$. Since every vertex of E is univalent, the vertices of $\mathcal{K}$ have are colored 0 or 1.

Claim 9 $\mathcal{K}$ has at least one vertex of color 0 and at least one vertex of color 1.

Proof of Claim 9 Immediate from the definition of $\mathcal{K}$. □

{Initially:}
  X is initialized to its state in $\bar{S}(I^l)$

Procedure TCPROPOSE(v): { *let $p \in \Pi_i$, i is 1 or 2* }
1 $R_i \leftarrow v$ { write the proposal in the team's register }
2 let p be initialized to its state in $\bar{S}(I^l)$
3 take one step of $\mathsf{Cons}_{\mathcal{D}}$ { *invoke an operation on X* }
4 **if** ⟨ the state of p corresponds to a vertex in $\mathcal{K}_1$ ⟩
5 **then** { *Π_1 is the winner* }
6 **return** R_1
7 **else** { *Π_2 is the winner* }
8 **return** R_2

Fig. 4 A team consensus algorithm using $\bar{S}(I^l)$ and $\mathsf{Cons}_{\mathcal{D}}$

Claim 10 $\mathcal{K}$ is connected.

Proof of Claim 10 We assume the opposite, and we show that type T then solves consensus among $n + 1$ processes.

Indeed, assume that $\mathcal{K}$ is not connected, i.e., $\mathcal{K}$ consists of two or more connected components. Clearly, any two vertices $\bar{S} \cdot S$ and $\bar{S} \cdot S'$ of $\mathcal{K}$, such that S and S' begin with a step of the same process, belong to the same connected component of $\mathcal{K}$.

Let $\mathcal{K}_1$ be one of the connected components of $\mathcal{K}$. We partition the system into two teams Π_1 and Π_2. Team Π_1 consists of all processes p, such that all $\bar{S} \cdot S$ where S begins with a step of p are in $\mathcal{K}_1$. Team Π_2 consists of all other processes. Since $\mathcal{K}$ consists of at least two components, Π_1 and Π_2 are non-empty.

The algorithm in Fig. 4 solves team consensus among $n + 1$ processes for teams Π_1 and Π_2, using one object X of type T and two registers. Let X be initialized to its state in $\bar{S}(I^l)$. Every process $p \in U$ writes its input value into its team's register and then executes one step of $\mathsf{Cons}_{\mathcal{D}}$ according to p's state in $\bar{S}(I^l)$ (by the definition of γ, in this step, p accesses X). The resulting state of p corresponds to a vertex of exactly one component of $\mathcal{K}$. If the state of p corresponds to a vertex in $\mathcal{K}_1$, then p outputs the value of Π_1's register, otherwise, p outputs the value of Π_2's register.

That is, the processes agree on the component to which the resulting state of the system belongs. If the resulting state belongs to $\mathcal{K}_1$, then a process in Π_1 was the first to access X in the corresponding execution (team Π_1 is the winner). Otherwise, if the resulting state does not belong to $\mathcal{K}_1$, then a process in Π_2 was the first to access X (team Π_2 is the winner).

Consider any execution of the algorithm. Clearly, every correct process decides. The first step accessing X in the execution is of a process q in the winner team. By the algorithm, q has previously written its proposal value in its team's register. Since every process first accesses X and then decides

a value in the winner team's register, any decided value is necessarily a proposed value of some process.

Now assume that all processes on a team (Π_1 or Π_2) propose the same value. Since the processes return values previously written in the winner team's register, and, by the assumption, no two different values can be written in a team's register, no two processes decide differently.

Thus, T solves team consensus among $n+1$ processes when object X is initialized to its state in $\bar{S}(I^l)$. By Lemma 1, T solves consensus among $n+1$ processes—a contradiction with the assumption that $cons(T) \leq n$.

Thus, $\mathcal{K}$ is connected. □

By Claims 9 and 10, 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 process p that has the same state in $\bar{S} \cdot S(I^l)$ and $\bar{S} \cdot S'(I^l)$, and we have the lemma. □

3.7 Critical index

We say that index $l \in \{1, 2, \ldots, n+1\}$ 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 11 *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 $\Upsilon^{P,n+1}$ is 1-valent. Hence, there exists $l \in \{1, \ldots, n+1\}$ 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 4, $\Upsilon^{P,l}$ contains a decision gadget. Thus l is critical. □

3.8 Deciding sets

Instead of the notion of a *deciding process* used in [7], we introduce the notion of a *deciding set* $V \subset \Pi$. The deciding set V of a *complete* decision gadget γ is computed 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 7, 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 $V = \{p\}$.

(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 $|U| = n+1$, then by Lemma 8 there is a "confused" process $p \in U$ such that, for some $\bar{S} \cdot S$ and $\bar{S} \cdot S'$ in E, p has the same state 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 $V = U - \{p\}$ where p is the smallest confused process.

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 12 *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 3, $V = \{p\}$ includes exactly one correct process.

(2) Let γ be a hook with a deciding set $V = \{p\}$. By Lemma 3, 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 step $p \in U$ in $\bar{S}(I^l)$ accesses X, 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 2, 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 $\Upsilon^{P,l}$.
By the definition of a rake, S applied to $\bar{S}(I^l)$ does not access X, 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 $\Pi - V$ can distinguish $\bar{S} \cdot S(I^l)$, $\bar{S} \cdot S_0 \cdot S(I^l)$ and $\bar{S} \cdot S_1 \cdot S(I^l)$, the correct processes have decided the same values in these configurations—a contradiction.

(3b) $|U| = n+1$, i.e., $U = \Pi$. Let $V = U - \{p\}$ be the deciding set of γ, i.e., for some $\bar{S} \cdot S_0$ and $\bar{S} \cdot S_1$,

the vertices of $\Upsilon^{P,l}$ of opposite valence, $\bar{S} \cdot S_0(I^l)$ and $\bar{S} \cdot S_1(I^l)$ differ only in the states of processes in V and object X. Assume, by contradiction, that all processes in V are faulty (i.e., since $k = n + 1$, the only correct process is p).

By Lemma 2, there is a schedule S containing only steps of correct processes (i.e., only steps of p) such that all correct processes have decided in $\bar{S} \cdot S_0 \cdot S(I^l)$, and if S is applicable to $\bar{S} \cdot S_1(I^l)$, then $\bar{S} \cdot S_1 \cdot S$ is a vertex of $\Upsilon^{P,l}$.

Note that, since X is an object of a one-shot type, and p has already accessed X at least once in $\bar{S} \cdot S_0(I^l)$, every subsequent operation of p on object X returns $\perp$. Since the states of p and all objects except of X are the same in $\bar{S} \cdot S_0(I^l)$ and $\bar{S} \cdot S_1(I^l)$, and p has already accessed X at least once in $\bar{S} \cdot S_1(I^l)$, S is also applicable to $\bar{S} \cdot S_1(I^l)$ and p cannot distinguish $\bar{S} \cdot S_0 \cdot S(I^l)$ and $\bar{S} \cdot S_1 \cdot S(I^l)$. Thus, $\bar{S} \cdot S_1 \cdot S$ is a vertex of $\Upsilon^{P,l}$, and p has decided the same value in $\bar{S} \cdot S_0 \cdot S(I^l)$ and $\bar{S} \cdot S_1 \cdot S(I^l)$—a contradiction.

In each case, the deciding set V contains at least one correct process. □

3.9 The reduction algorithm

Theorem 13 *Let T be any one-shot deterministic type, such that $cons(T) \leq n$. If a failure detector $\mathcal{D}$ solves consensus in a system of $n + 1$ processes using only registers and objects of type T, then $\Omega_n \preceq \mathcal{D}$.*

Proof The communication task presented in Fig. 1 and the computation task presented in Fig. 5 constitute the reduction algorithm $T_{\mathcal{D} \to \Omega_n}$. The current estimate of Ω_n at process p is stored in a variable $\Omega_n\text{-}output_p$.

In the communication task (Fig. 1), every process p maintains an ever-growing DAG G_p. In the computation task (Fig. 5), for each $P \subseteq \Pi$ and each $l \in \{0, \ldots, n+1\}$, 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 $\Upsilon_p^{P,l}$ according to the decision taken in the descendants of S (if any).

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.

First we observe that the "repeat-until" cycle in lines 6–20 is non-blocking. Indeed, each process p eventually sets V to a non-empty value or reaches $P = \emptyset$. In both cases, p exits the "repeat-until" cycle.

Claim 14 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.

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

```
1  while true
2    for all P ⊆ Π and l ∈ {0, 1, ..., n + 1}
3      Υ_p^{P,l} ← simulation tree induced by P, I^l and G_p
4    V ← ∅
5    P ← Π
6    repeat
7      if P has no critical index then
8        V ← {p}
9      else
10       let l be the smallest critical index of P
11       if l is univalent critical then
12         V ← {p_l}
13       else
14         γ ← the smallest decision gadget in Υ_p^{P,l}
15         if γ is complete then
16           V ← the deciding set of γ
17         else
18           let W be the set of missing processes in γ
19           P ← P − W
20   until V ≠ ∅ or P = ∅
21   if P = ∅ then V ← {p} then V ← {p}
22   Ω_n-output_p ← V
```

Fig. 5 Extracting Ω_n: process p

Proof of Claim 14 By Lemma 11, every P 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 5, 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 either

(a) p always exits the "repeat-until" cycle in line 12 after locating a univalent critical index in some P such that $correct(F) \subseteq P$, or
(b) p always reaches line 14 with $P = correct(F)$.

In case (b), by Corollary 6, 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 both cases, there exists P^* 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 reach line 21 with $P = P^*$ and $V = V^*$. Let l be the smallest critical index in P^*. According to 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 2, 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 12, 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 includes at least one correct process, i.e., the output of Ω_n is emulated. □

Theorem 13 and the algorithm of [23] imply the following result:

Theorem 15 *Let T be any one-shot deterministic type such that $cons(T) = n$. Then Ω_n is the weakest failure detector to solve consensus in a system of $n+1$ processes using registers and objects of type T.*

4 Boosting types to any level

Consider now a set Π of k processes ($k > n$) that communicate through registers and objects of an *m-ported* one-shot deterministic type T such that $cons(T) \leq n$ and $m \leq n+1$.

Theorem 16 *Let T be any m-ported one-shot deterministic type, such that $cons(T) \leq n$ and $m \leq n+1$. If a failure detector $\mathcal{D}$ solves consensus in a system of k ($k > n$) processes using only registers and objects of type T, then $\Omega_n \preceq \mathcal{D}$.*

Proof Let F be any failure pattern and $\mathsf{Cons}_{\mathcal{D}}$ be any algorithm that solves consensus using $\mathcal{D}$. The reduction algorithm $T_{\mathcal{D}\rightarrow\Omega_n}$ is exactly the same as the algorithm described in Fig. 5, except that now we have $k \geq n+1$ processes, and variable l thus takes values in $\{0, 1, \ldots, k\}$. The decision gadget and deciding sets are defined in the same way as in Sect. 3. The deciding sets of forks and hooks do not depend on the system size. Consider a rake γ with a participating set U. Since the objects of type T are at most $(n+1)$-ported, and processes in U access the same object of type T, U can include at most $n+1$ processes. If $|U| \leq n$, then the deciding set of γ is U. If $|U| = n+1$, then, by Lemma 8, there is at least one "confused" process p, and the deciding set V is defined as $U - \{p\}$. In both cases, V is of size at most n. By Lemma 12, V includes at least one correct process. □

Theorem 16 and the algorithm of [23] imply the following result:

Theorem 17 *Let T be any $(n+1)$-ported one-shot deterministic type such that $cons(T) = n$. Then Ω_n is the weakest failure detector to solve consensus in a system of k ($k \geq n+1$) processes using registers and objects of type T.*

As a corollary of Theorem 17, assuming that only registers are available, we obtain the following result, outlined in [19].

Corollary 18 *Ω is the weakest failure detector to solve consensus using only registers.*

Also, given that n-process consensus is a one-shot deterministic n-ported type of consensus power n [18], we immediately obtain:

Corollary 19 *Ω_n is the weakest failure detector to solve consensus among $k > n$ processes using registers and n-process consensus objects.*

5 Boosting resilience

So far we considered systems in which processes communicate through *wait-free* linearizable implementations of deterministic one-shot object types. Every operation invoked by a correct process on a wait-free object returns, regardless of the behavior of other processes.

In contrast, in this section we assume that processes communicate through wait-free registers and *t-resilient* implementations of object types (not necessarily one-shot and deterministic), where $0 \leq t < n$. We will simply call these t-resilient objects. Informally, a t-resilient object guarantees that a correct process completes its operation on the object, 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. This corresponds to the *weakly* t-resilient implementations of [6]. We refer to [3] for a formal definition of t-resilient objects based on I/O automata [22, Chap. 8].

It is shown in [3] that no composition of t-resilient objects can be used to solve consensus among $n > t-1$ processes. In this section we show that Ω_{t+1} captures the exact amount of information about failures sufficient to circumvent this impossibility. But first we recall a few earlier results that are instrumental for our proof.

The following two lemmas are restatements in our terminology of the "necessity" part and the "sufficiency" part of Theorem 6.1 in [6], respectively.

Lemma 1 *Let t and n be integers, $0 \leq t$, $1 \leq n$. Then there exists an t-resilient n-process implementation of consensus from wait-free $(t+1)$-process consensus objects and wait-free registers.*[2]

Lemma 2 *Let f and n be integers, $2 \leq t < n$. Then there exists a wait-free $(f+1)$-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 [14]:

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

Finally, we are ready to demonstrate how our result on boosting the power of deterministic one-shot deterministic types can be used to derive the following:

Theorem 20 *Let t be any integer, $2 \leq t < n-1$. 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-resilient objects of type T implement wait-free $(t+1)$-process consensus. The algorithm of [23] 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.

Thus, $\mathcal{D}$ solves consensus using registers and objects of $(t+1)$-process consensus objects. By Corollary 19, $\Omega_{t+1} \preceq \mathcal{D}$. This gives the necessary part of the theorem. □

6 Concluding remarks

The conjecture that Ω_n is the weakest failure detector to boost the power of T to the level $n+1$ of the consensus hierarchy was given in [23]. As pointed out in [23], the proof of this conjecture appeared to be challenging and was indeed left open. However, Neiger also gave in [23] an outline of some preliminary elements that could be used to construct the proof. In this section, we give an overview of major features that distinguish our proof from the outline sketched by Neiger [23, Sect. 5]. We also point out some potential problems that arise in Neiger's outline an the specific assumptions made in that outline. Since the outline is given in a quite informal manner, we would like to emphasize that the discussion below is subject to our interpretation of the missing details.

6.1 Restrictions on failure detectors

Neiger's outline [23] is constructed as follows. Consider any algorithm that solves $n+1$ consensus using some failure detector $\mathcal{D}$, read-write registers, and objects of *deterministic* (but necessarily one-shot) type T. The aim is to use the algorithm for extracting the output of Ω_n.

Following the arguments of [7], we can identify a decision gadget of the *hook* type. Recall that a hook has a bivalent *pivot* $\bar{S}$ such that for some processes p and q, such that $\bar{S}$ extended with a step of p results in a 0-valent vertex, $\bar{S}$ extended with a step of q followed by a step p results in a 1-valent vertex, and any vertex $\bar{S} \cdot S$ where S includes a step of p is univalent. Then it is argued that for each process r, (1) there exists a bound b_r on the number of steps of r such that whenever r takes b_r steps the system ends up in a univalent configuration, or (2) r is not the only correct process. Indeed, suppose that r is the only correct process. Since all other processes take only a bounded number of steps in extensions of $\bar{S}$ in Υ^I, there is a bound b_r on the number r needs to take to decide, and thus bring the system to a univalent state.

Suppose property (1) above holds for every process r, and consider all possible schedules S in which every process r takes up to b_r steps extending $\bar{S}$. If one of the schedules does not belong to Υ^I, then we can identify a faulty process q (whose step is missing), and thus conclude that $\Pi - \{q\}$ contains at least one correct process. Now assume that all these schedules are in Υ^I.

At this point, the Neiger's outline [23] seems to require that for every process p, the sequences of failure detector values seen by p in each of this schedules are identical, i.e., the failure detector output does not depend on the order in which processes query their failure detector modules. To make this argument work we need to impose certain restrictions on the class of failure detectors we consider. For instance, we can suppose that for every failure detector history there is a time after which some infinite sequences of failure detector values seen by the processes do not depend on the interleaving of their steps. This can be obtained, e.g., if the domain of the failure detector is *finite* [23], or if in every run, the output of the failure detector *eventually stabilizes* at every correct process. For simplicity, assume the latter and consider the simulated executions in which every process always sees exactly one "stable" failure detector value.

[2] Theorem 6.1 in [6] assumes $2 \leq t$. However, the necessity part of the theorem holds for $0 \leq t$.

6.2 Atomically readable objects

Now the outline claims that there must exist two univalent descendants of $\bar{S}$, S_0 and S_1, and a process q such that the state of q and the states of all shared objects are identical in $S_0(I)$ and $S_1(I)$. Thus, q cannot be the only correct process: q is not able to decide in any solo extension of S_0 or S_1. The claim is proved by contradiction, presenting an algorithm that solves $n+1$ consensus using objects of type T and read-write registers.

The contradiction is established on the assumption that type T is *readable*—every object of type T exports a read operation that returns the current state of the object. If the claim does not hold, then $n+1$ processes can solve team consensus as follows: each process r runs b_r steps, reads the states of all shared objects, and decide on the valence of any compatible configuration. The conclusion is that, since process r took b_r steps, the system reached a univalent configurations and all univalent configuration that are compatible with the object states and the state of r are of the same valence.

This conclusion seems to depend on the assumption that all objects can be read *atomically*. Otherwise, the states of objects might not correspond to any state reachable by extensions of $\bar{S}$ (e.g., r reads object A, then q modifies A, then q modifies object B, and then r reads B). It may even happen that the system state, as observed by r, is compatible with a state of arbitrary valence, causing the processes to disagree. Even though atomic accesses can be emulated in the read-write shared memory model [1], it is difficult to say whether this emulation can be generalized to larger classes of object types. Besides, just assuming readable types considerably simplifies reasoning about the power of types [24].

6.3 Reduction algorithm

Now assume that an atomic read is available and, thus, the algorithm above establishes that there is a "confused" process that can never decide in a solo run. In this case, a reduction algorithm is suggested that, starting from a hook in a simulation tree, makes sure that all correct processes eventually agree on the same process that is not the only correct process in the system. In the reduction algorithm, every process p periodically looks at its finite simulation tree Υ_p^I and computes b_r for each process r such that condition (1) above is satisfied, or, if there is a process r for which no such b_r exists, outputs $\Pi - \{r\}$.

In the first case, we use the reasoning above to identify a faulty or "confused" process q, and output $\Pi - \{q\}$. If the situation stabilizes, i.e., the valences of the selected set of extensions of the pivot of the hook do not change, all correct processes output set $\Pi - \{q\}$ that obviously contains at least one correct process.

Unfortunately, as Neiger observes [23], it can happen that in a given finite tree Υ_p^I, first, some process r does not satisfy (1) (e.g., because some steps of r are still missing), in which case the algorithm outputs $\Pi - \{r\}$, and then, in the next iteration of the reduction algorithm, r satisfies (1) (e.g., because more steps of r came out), in which case the algorithm outputs $\Pi - \{q\}$ where $q \neq r$, and so on, i.e., the output of the algorithm never stabilizes. There does not seem to be an obvious way to handle this "stabilization" issue.[3]

6.4 One-shot types

To conclude, we give an intuition of how our assumption of objects of type T being one-shot makes life easier. As we have shown in Sect. 3.4, each bivalent infinite simulation tree Υ^I either contains a fork, or a hook that allows us to factor out a single correct process using a simple case analysis, or an *incomplete rake* which allows us to identify a set of processes that does *not* include all correct processes, or a *complete rake*.

The latter is of particular interest for us, because, when the participating set of the complete rake is Π, it is a special case of the extended hook of the Neiger's outline discussed above. Indeed, then the rake has a pivot $\bar{S}$ such that every process is about to access the same object X of type T in $\bar{S}(I)$. Moreover, Υ^I contains all vertices of the form $\bar{S} \cdot S$ where S is a schedule in which every process takes exactly one step ($\bar{S} \cdot S$ is called a leave of the rake). Further, all the rake's leaves are univalent, and there are leaves of opposite valence. Note that since every process is poised on accessing object X in $\bar{S}(I)$, no such S contains a query step. Thus, we do not have to restrict the properties of the failure detector at this point—simply because the decision values do not depend on the failure detector output.

Using the fact that $cons(T) \leq n$, we conclude that at least one process r cannot distinguish two leaves of the rake of the opposite valence, S_0 and S_1. Thus, in any solo extension of S_0 or S_1, r can never decide. This is because r can get a non-$\perp$ response from object X at most once (type T is one-shot) and all other objects have the same state in $S_0(I)$ and $S_1(I)$. Thus, we do not have to rely upon the objects being atomically readable or simply readable.

We suspect that relaxing the one-shot requirement will not be straightforward and we leave it for future research. Getting rid of the assumption that objects of type T are $(n+1)$-ported when boosting their power to levels higher than $n+1$ (Sect. 4) is another direction for future work.

[3] Interestingly, however, a similar approach can be used for extracting a failure detector that is strictly weaker than Ω_n but still provides enough information to circumvent some asynchronous impossibility—to solve n-set agreement [12].

References

1. Afek, Y., Attiya, H., Dolev, D., Gafni, E., Merrit, M., Shavit, N.: Atomic snapshots of shared memory. J. ACM **40**(4), 873–890 (1993)
2. Attie, P., Lynch, N.A., Rajsbaum, S.: Boosting fault-tolerance in asynchronous message passing systems is impossible. Technical report. MIT Laboratory for Computer Science, MIT-LCS-TR-877, (2002)
3. Attie, P.C., Guerraoui, R., Kouznetsov, P., Lynch, N.A., Rajsbaum, S.: The impossibility of boosting distributed service resilience. In: Proceedings of the 25th IEEE International Conference on Distributed Computing Systems (ICDCS'05), June (2005)
4. Attiya, H., Welch, J.L.: Distributed Computing: Fundamentals, Simulations and Advanced Topics, 2nd edn. Wiley, New York (2004)
5. Borowsky, E., Gafni, E., Afek, Y.: Consensus power makes (some) sense! In: Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing (PODC), pp. 363–372, August (1994)
6. Chandra, T.D., Hadzilacos, V., Jayanti, P., Toueg, S.: Generalized irreducibility of consensus and the equivalence of t-resilient and wait-free implementations of consensus. SIAM J. Comput. **34**(2), 333–357 (2004)
7. Chandra, T.D., Hadzilacos, V., Toueg, S.: The weakest failure detector for solving consensus. J. ACM **43**(4), 685–722 (1996)
8. Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. J. ACM **43**(2), 225–267 (1996)
9. Dolev, D., Dwork, C., Stockmeyer, L.J.: On the minimal synchronism needed for distributed consensus. J. ACM **34**(1), 77–97 (1987)
10. Dwork, C., Lynch, N.A., Stockmeyer, L.J.: Consensus in the presence of partial synchrony. J. ACM **35**(2), 288–323 (1988)
11. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. J. ACM **32**(3), 374–382 (1985)
12. Guerraoui, R., Herlihy, M., Kouznetsov, P., Lynch, N., Newport, C.: On the weakest failure detector ever. Technical report, Max Planck Institute for Software Systems
13. Guerraoui, R., Kouznetsov, P.: On failure detectors and type boosters. In: Proceedings of the 17th International Symposium on Distributed Computing (DISC'03), October (2003)
14. Herlihy, M.: Wait-free synchronization. ACM Trans. Program. Lang. Syst. **13**(1), 124–149 (1991)
15. Herlihy, M., Ruppert, E.: On the existence of booster types. In: Proceedings of the 41st IEEE Symposium on Foundations of Computer Science (FOCS), pp 653–663 (2000)
16. Herlihy, M., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. **12**(3), 463–492 (1990)
17. Jayanti, P.: Robust wait-free hierarchies. J. ACM **44**(4), 592–614 (1997)
18. Jayanti, P., Toueg, S.: Some results on the impossibility, universality and decidability of consensus. In: Proceedings of the 6th International Workshop on Distributed Algorithms (WDAG'92). LNCS, vol 647. Springer, Heidelberg (1992)
19. Lo, W.-K., Hadzilacos, V.: Using failure detectors to solve consensus in asynchronous shared-memory systems. In: Proceedings of the 8th International Workshop on Distributed Algorithms (WDAG'94). LNCS, vol. 857, pp. 280–295. Springer, Heidelberg (1994)
20. Lo, W.-K., Hadzilacos, V.: All of us are smarter than any of us: Nondeterministic wait-free hierarchies are not robust. SIAM J. Comput. **30**(3), 689–728 (2000)
21. Loui, M.C., Abu-Amara, H.H.: Memory requirements for agreement among unreliable asynchronous processes. Adv. Comput. Res., pp. 163–183 (1987)
22. Lynch, N.A.: Distributed Algorithms. Morgan Kaufmann Publishers,San Francisco (1996)
23. Neiger, G.: Failure detectors and the wait-free hierarchy. In: Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing (PODC), pp. 100–109, August (1995)
24. Ruppert, E.: Determining consensus numbers. SIAM J. Comput. **30**(4), 1156–1168 (2000)