Distrib. Comput. (2007) 19:335–359
DOI 10.1007/s00446-006-0019-4

SPECIAL ISSUE PODC 05

The weakest failure detector to solve nonuniform consensus

Jonathan Eisler · **Vassos Hadzilacos** · **Sam Toueg**

Received: 15 October 2005 / Accepted: 20 February 2006 / Published online: 2 February 2007

Abstract We determine the weakest failure detector to solve nonuniform consensus in any environment, i.e., regardless of the number of faulty processes. Together with previous results, this closes all aspects of the following question: What is the weakest failure detector to solve (uniform or nonuniform) consensus in any environment?

Keywords Distributed algorithms · Fault tolerance · Consensus · Failure detectors

1 Introduction

Consensus is a classical problem that lies at the heart of many important problems in fault-tolerant distributed computing. In consensus each process initially proposes a value, and eventually processes must reach a common decision on one of the proposed values. Two variants of the problem have been studied: in the *uniform* version, no two processes (whether correct or faulty) can reach different decisions. Here a faulty process need not reach a decision at all, but if it does, that decision must be consistent with that of correct processes. In the *nonuniform* version, no two *correct* processes can reach different decisions. In this weaker version of consensus, a faulty process can reach a decision on any proposed value.

It is well-known that consensus is unsolvable in asynchronous systems subject to process crashes (even if communication is reliable) [4]. One way to circumvent this impossibility result is through the use of *unreliable failure detectors* [2]: in this model, each process has access to a failure detector module that provides some (possibly incomplete and inaccurate) information about failures, e.g., a list of processes currently suspected to have crashed. It is natural then to seek the *weakest* failure detector to solve consensus. Informally, $\mathcal{D}^*$ is the weakest failure detector to solve problem P if (a) there is an algorithm that uses $\mathcal{D}^*$ to solve P, and (b) any failure detector $\mathcal{D}$ that can be used to solve P can be transformed to $\mathcal{D}^*$.

Chandra et al. [1] were the first to address the question of the weakest failure detector to solve consensus. They determined the weakest failure detector to solve uniform or nonuniform consensus in systems with a *majority* of correct processes. Delporte et al. [3] determined the weakest failure detector to solve *uniform* consensus in systems with *any* number of failures, i.e., in *any* environment. Informally, an environment describes the number and timing of failures that can occur.

It remained open to identify the weakest failure detector to solve *nonuniform* consensus in *any* environment. In this paper, we resolve this question. Together with the results cited above, this closes all aspects of the following question: what is the weakest failure detector to solve (uniform or nonuniform) consensus in any environment? We summarize the previous results before describing our own contribution in greater detail.

Chandra et al. showed that, in environments where a majority of processes are correct, the weakest failure

Research supported in part by the Natural Sciences and Engineering Council of Canada.

J. Eisler
One Microsoft Way, Microsoft Corporation,
Redmond, WA 98052, USA

V. Hadzilacos (✉) · S. Toueg
Department of Computer Science, University of Toronto,
Toronto, ON M5S 3G4, Canada

detector to solve uniform or nonuniform consensus is Ω, the *leader* failure detector. At each process, Ω outputs (the identity of) a process. Ω guarantees that there is a time after which the same correct process is output at all correct processes.

Delporte et al. proved that the weakest failure detector to solve *uniform* consensus in *any* environment is the pair (Ω, Σ), where Σ is the *quorum* failure detector.[1] At each process, Σ outputs a *set of processes* such that: (1) any two sets, output at any times and at any processes, intersect, and (2) there is a time after which every set output at any correct process consists of only correct processes.

In this paper, we determine that the weakest failure detector to solve *nonuniform* consensus in *any* environment is (Ω, Σ^ν), where Σ^ν is the natural "nonuniform" version of Σ. More precisely, Σ^ν is like Σ, except that the intersection requirement is restricted to quorums output at *correct* processes. In other words, any two quorums output at correct processes intersect. Quorums output at faulty processes, however, may fail to intersect with quorums output at other processes. Clearly, Σ^ν is weaker than Σ.

To prove our result we need to show that the following hold in any environment:

Sufficiency. There is an algorithm that uses (Ω, Σ^ν) to solve nonuniform consensus.

Necessity. Any failure detector $\mathcal{D}$ that can be used to solve nonuniform consensus can be transformed to (Ω, Σ^ν).

To prove *sufficiency*, we proceed in two stages. We first show how to transform Σ^ν into another failure detector, denoted $\Sigma^{\nu+}$. We then give an algorithm that uses $(\Omega, \Sigma^{\nu+})$ to solve nonuniform consensus in any environment.

From Chandra et al. we already know that in any environment, any failure detector $\mathcal{D}$ that can be used to solve nonuniform consensus can be transformed to Ω. Thus, to prove *necessity*, it suffices to show that $\mathcal{D}$ can also be transformed to Σ^ν. We present an algorithm that does so.

To prove that (Ω, Σ^ν) is the weakest failure detector to solve nonuniform consensus we had to use different techniques than those used by Delporte et al. to show that (Ω, Σ) is the weakest failure detector to solve uniform consensus. At the heart of their proof is the fact that uniform consensus can be used to implement registers. Nonuniform consensus, however, is not strong enough to implement registers. As a result, neither their necessity nor their sufficiency techniques can be adopted for our purposes.

Interestingly, our approach also gives an alternative proof that (Ω, Σ) is the weakest failure detector to solve *uniform* consensus: (a) our proof that Σ^ν is necessary to solve nonuniform consensus also shows that Σ is necessary to solve uniform consensus (see Sect. 5.1), and (b) a simple modification of a known algorithm shows that (Ω, Σ) is sufficient to solve uniform consensus (see Sect. 6.3).

From the results of [1,3] and this paper, (Ω, Σ) and (Ω, Σ^ν) are the weakest failure detectors to solve uniform and nonuniform consensus, respectively, in *any* environment. In environments, where half or more of the processes may fail, (Ω, Σ) and (Ω, Σ^ν) are not equivalent. This follows from an observation by Delporte et al. that, in such environments, the weakest failure detector to solve *nonuniform* consensus is not (Ω, Σ) [3]. In this paper, we provide a direct proof that if half or more of the processes may fail, (Ω, Σ) and (Ω, Σ^ν) are not equivalent. On the other hand, if a majority of processes are correct, (Ω, Σ) and (Ω, Σ^ν) are equivalent: in this case, Σ and Σ^ν can be implemented "from scratch", i.e., without using any failure detector.

The rest of the paper is organized as follows: In Sect. 2 we review the model of computation, and in Sect. 3 we define the failure detectors Ω, Σ and Σ^ν. In Sect. 4 we recall a technique introduced by Chandra et al. to prove statements of the form: "any failure detector that can be used to solve a certain problem P can be transformed to failure detector D" [1]. This technique is the starting point for our proof that, in any environment, any failure detector that can be used to solve nonuniform consensus can be transformed to Σ^ν. We give this proof in Sect. 5. In Sect. 6 we prove that, in any environment, nonuniform consensus can be solved using (Ω, Σ^ν). Finally, in Sect. 7, we show that (Ω, Σ^ν) and (Ω, Σ) are not equivalent in environments where half or more of the processes may fail, while they are equivalent in environments where a majority of the processes are correct.

2 The model

Our model of asynchronous computation is the one described in [1], which augments the model of Fischer et al. [4] with failure detectors.

2.1 Systems

We consider distributed message-passing systems with a set of $n \geq 2$ processes $\Pi = \{0, 1, \ldots, n-1\}$. Processes

[1] If $\mathcal{D}$ and $\mathcal{D}'$ are failure detectors $(\mathcal{D}, \mathcal{D}')$, is the failure detector that outputs a vector with two components, the first being the output of $\mathcal{D}$ and the second being the output of $\mathcal{D}'$.

execute steps asynchronously, i.e., there is no bound on the delay between steps. Each pair of processes are connected by a reliable link. The links transmit messages with finite but unbounded delay. They are modeled as a set M, called the *message buffer*, that contains triples of the form $(p, data, q)$ indicating that p has sent the message *data* to q, and q has not yet received it. We assume that each message sent by a process p to a process q is unique; this can be guaranteed by having the sender include a counter with each message.

2.2 Failures, failure patterns and environments

We consider crash failures only: processes fail only by halting prematurely. A *failure pattern* is a function $F: \mathbb{N} \rightarrow 2^{\Pi}$, where $F(t)$ is the set of processes that have crashed through time t. (For presentation simplicity, we assume a discrete global clock to which the processes do not have access. The range of this clock's ticks is $\mathbb{N}$.) Since processes never recover from crashes, $F(t) \subseteq F(t+1)$. Let $\mathit{faulty}(F) = \bigcup_{t \in \mathbb{N}} F(t)$ be the set of faulty processes in a failure pattern F; and let $\mathit{correct}(F) = \Pi - \mathit{faulty}(F)$ be the set of correct processes in F. When the failure pattern F is clear from the context, we say that process p is *correct* if $p \in \mathit{correct}(F)$, and p is *faulty* if $p \in \mathit{faulty}(F)$.

An *environment* is a set of failure patterns. Intuitively, an environment describes the number and timing of failures that can occur in the system. Thus, a result that applies to all environments is one that holds regardless of the number and timing of failures.

2.3 Failure detectors

A *failure detector history* H *with range* $\mathcal{R}$ describes the behavior of a failure detector during an execution. Formally, it is a function $H: \Pi \times \mathbb{N} \rightarrow \mathcal{R}$, where $H(p,t)$ is the value output by the failure detector module of process p at time t.

A *failure detector* $\mathcal{D}$ *with range* $\mathcal{R}$ is a function that maps any failure pattern F to a set of failure detector histories with range $\mathcal{R}$. $\mathcal{D}(F)$ is the set of all possible failure detector histories that may be output by $\mathcal{D}$ in a failure pattern F. Typically we specify a failure detector by stating the properties that its histories satisfy.

Given two failure detectors $\mathcal{D}$ and $\mathcal{D}'$, we denote by $(\mathcal{D}, \mathcal{D}')$ the failure detector whose output is an ordered pair in which the first element corresponds to an output of $\mathcal{D}$, and the second element corresponds to an output of $\mathcal{D}'$. More precisely, if $\mathcal{R}$ and $\mathcal{R}'$ are the ranges of $\mathcal{D}$ and $\mathcal{D}'$, respectively, then the range of $(\mathcal{D}, \mathcal{D}')$ is $\mathcal{R} \times \mathcal{R}'$. For all failure patterns F,

$$
\begin{aligned}
&H'' \in (\mathcal{D}, \mathcal{D}')(F) \iff \\
&\quad \exists H \in \mathcal{D}(F) \wedge \exists H' \in \mathcal{D}'(F) \wedge \\
&\quad \forall p \in \Pi,\ \forall t \in \mathbb{N} : H''(p,t) = (H(p,t), H'(p,t))
\end{aligned}
$$

2.4 Algorithms

An algorithm $\mathcal{A}$ is modeled as a collection of n deterministic automata. There is an automaton $\mathcal{A}(p)$ for each process p. Computation proceeds in steps of these automata. In each step, a process p atomically

- receives a single message m from the message buffer M, or the empty message λ;
- queries its local failure detector module and receives a value d;
- changes its state; and
- sends messages to other processes.

The state transition and the messages that p sends are all uniquely determined by the automaton $\mathcal{A}(p)$, the state of p at the beginning of the step, the received message m, and the failure detector value d. Formally, a step is a tuple $e = (p, m, d, \mathcal{A})$, where p is the process taking step e, m is the message received by p during e, d is the failure detector value seen by p in e, and $\mathcal{A}$ is the algorithm being executed.

The message received in a step is nondeterministically selected from $M \cup \{\lambda\}$. This reflects the asynchrony of the communication channels: a process p may receive the empty message despite the existence of unreceived messages addressed to p.

2.5 Configurations

A *configuration* is a pair (s, M), where s is a function that maps each process to its local state, and M is the message buffer. Recall that M is a set of triples $(p, data, q)$, where p sent *data* to q, which has not yet received it. An *initial* configuration is a pair (s, M), where $M = \emptyset$ and $s(p)$ is an initial state of the automaton $\mathcal{A}(p)$.

A step $(p, m, d, \mathcal{A})$ is *applicable* to a configuration $C = (s, M)$ if and only if $m \in M \cup \{\lambda\}$. If e is a step applicable to configuration C, $e(C)$ denotes the configuration that results when we apply e to C. This is uniquely determined by the automaton $\mathcal{A}(p)$ of the process p that takes step e.

2.6 Schedules and runs

A *schedule* S of an algorithm $\mathcal{A}$ is a finite or infinite sequence of steps of $\mathcal{A}$. The ith step in schedule S is denoted $S[i]$, and the prefix consisting the first j steps of S is denoted $S[1..j]$. A schedule S is *applicable* to a configuration C if S is the empty schedule, or $S[1]$ is applicable to C, $S[2]$ is applicable to $S[1](C)$, etc. If S is finite and is applicable to C, $S(C)$ denotes the configuration that results when we apply schedule S to configuration C. We denote by *participants*(S) the set of processes that take at least one step in schedule S.

Let S be a schedule applicable to an initial configuration I of an algorithm $\mathcal{A}$, and let i, j be positive integers such that $i, j \leq |S|$. We say that *step i causally precedes step j in S with respect to I* if and only if one of the following holds [5]:

- $S[i]$ and $S[j]$ are steps of the same process and $i < j$;
- $S[i]$ and $S[j]$ are the sending and receipt of the same message — i.e., step $S[i]$ applied to configuration $S[1..i-1](I)$ results in the sending of some message m, and $S[j] = (-, m, -, \mathcal{A})$; or
- there is a positive integer $k \leq |S|$ such that step i causally precedes step k, and step k causally precedes step j in S with respect to I.

Note that if $S[i]$ and $S[j]$ are the sending and receipt of the same message m, then $i < j$ (because if $j \leq i$, then $S[j]$ would be receiving m before m is sent in $S[i]$, contradicting the fact that S is applicable to I). This implies:

Observation 2.1 *If step i causally precedes step j in S with respect to I then i < j.*

A *run of algorithm $\mathcal{A}$ using failure detector $\mathcal{D}$ in environment $\mathcal{E}$* is a tuple $R = (F, H, I, S, T)$ where F is a failure pattern in $\mathcal{E}$, H is a failure detector history in $\mathcal{D}(F)$, I is an initial configuration of $\mathcal{A}$, S is a schedule of $\mathcal{A}$, and T is a list of times in $\mathbb{N}$ (informally, $T[i]$ is the time when step $S[i]$ is taken) such that the following hold:

(1) S is applicable to I.
(2) S and T are both finite sequences of the same length, or are both infinite sequences.
(3) For all positive integers $i \leq |S|$, if $S[i] = (p, -, d, \mathcal{A})$,[2] then $p \notin F(T[i])$ and $d = H(p, T[i])$.
(4) For all positive integers $i < j \leq |S|$, $T[i] \leq T[j]$.
(5) For all positive integers $i, j \leq |S|$, if step i causally precedes step j in S with respect to I then $T[i] < T[j]$.

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

A run whose schedule is finite (respectively, infinite) is called a finite (respectively, infinite) run. An *admissible run* of algorithm $\mathcal{A}$ using failure detector $\mathcal{D}$ in environment $\mathcal{E}$ is an infinite run $R = (F, H, I, S, T)$ of $\mathcal{A}$ using $\mathcal{D}$ in $\mathcal{E}$ with two additional properties:

(6) Every correct process takes an infinite number of steps in S.
(7) Each message sent to a correct process is eventually received. More precisely, for every finite prefix S' of S, and every $q \in$ *correct*(F), if the message buffer in configuration $S'(I)$ contains a message $m = (-, -, q)$, then for some $i \in \mathbb{N}$, $S[i] = (q, m, -, \mathcal{A})$.

2.7 Solving problems with failure detectors

A *problem P* is defined by a set of properties that runs must satisfy. We say that *algorithm $\mathcal{A}$ uses failure detector $\mathcal{D}$ to solve problem P in environment $\mathcal{E}$* if and only if all the admissible runs of $\mathcal{A}$ using $\mathcal{D}$ in $\mathcal{E}$ satisfy the properties of P. We say that *failure detector $\mathcal{D}$ can be used to solve problem P in environment $\mathcal{E}$* if and only if there exists some algorithm $\mathcal{A}$ that uses $\mathcal{D}$ to solve P in $\mathcal{E}$.

2.8 Nonuniform consensus

We now define what it means for an algorithm $\mathcal{A}$ to solve nonuniform consensus. Let V be a set of at least two distinct values (when $V = \{0, 1\}$ the problem is called *binary* nonuniform consensus). The automata that define $\mathcal{A}$ must be such that, for each $v \in V$, each process p has a distinct initial state in which p is said to *propose* v. Also, each process has certain states in which it is said to *decide v*. Decisions are irrevocable in the sense that after entering a state in which it decides v, a process remains in such a state forever. We say that a process *proposes v* or *decides v in configuration C* if and only if it does so in its state in C. Let $R = (F, H, I, S, T)$ be a run of $\mathcal{A}$. We say that process *p proposes v in R* if and only if p proposes v in the initial configuration I of R; we say that *p decides v in R* if and only if for some prefix S' of S, p decides v in $S'(I)$.

We say that algorithm $\mathcal{A}$ *uses failure detector $\mathcal{D}$ to solve nonuniform consensus in environment $\mathcal{E}$* if and only if every admissible run R of $\mathcal{A}$ using $\mathcal{D}$ in $\mathcal{E}$ has the following properties:

[2] The symbol "−" in a field of a tuple indicates an arbitrary permissible value for that field of the tuple. We use this convention throughout the paper.

Termination. Every correct process decides a value in R.

Nonuniform agreement. No two correct processes decide different values in R.

Validity. If a process decides v in R, then some process proposes v in R.

2.9 Weakest failure detectors

We now explain how to compare two failure detectors $\mathcal{D}$ and $\mathcal{D}'$ in some environment $\mathcal{E}$. To do so, we first explain what it means for an algorithm to transform $\mathcal{D}$ to $\mathcal{D}'$ in $\mathcal{E}$. Such an algorithm, denoted $\mathcal{T}_{\mathcal{D}\to\mathcal{D}'}$, uses $\mathcal{D}$ to maintain a variable output$_p$ at every process p; output$_p$ functions as the output of the emulated failure detector $\mathcal{D}'$ at p. For each admissible run R of $\mathcal{T}_{\mathcal{D}\to\mathcal{D}'}$, let O_R be the history of all the output variables in R; i.e., $O_R(p,t)$ is the value of output$_p$ at time t in R. Algorithm $\mathcal{T}_{\mathcal{D}\to\mathcal{D}'}$ *transforms* $\mathcal{D}$ *to* $\mathcal{D}'$ *in environment* $\mathcal{E}$ if and only if for every admissible run $R = (F, H, I, S, T)$ of $\mathcal{T}_{\mathcal{D}\to\mathcal{D}'}$ using $\mathcal{D}$ in $\mathcal{E}$, $O_R \in \mathcal{D}'(F)$.

We say that $\mathcal{D}'$ *is weaker than* $\mathcal{D}$ *in* $\mathcal{E}$, denoted $\mathcal{D}' \preceq_{\mathcal{E}} \mathcal{D}$, if there is an algorithm $\mathcal{T}_{\mathcal{D}\to\mathcal{D}'}$ that transforms $\mathcal{D}$ to $\mathcal{D}'$ in $\mathcal{E}$. If $\mathcal{D}' \preceq_{\mathcal{E}} \mathcal{D}$, and $\mathcal{D} \preceq_{\mathcal{E}} \mathcal{D}'$, then $\mathcal{D}'$ is *equivalent to* $\mathcal{D}$ in $\mathcal{E}$, denoted $\mathcal{D}' \equiv_{\mathcal{E}} \mathcal{D}$. The relation *stronger than* ($\succeq_{\mathcal{E}}$) is defined symmetrically.

A failure detector $\mathcal{D}^*$ is the *weakest failure detector to solve problem P in environment* $\mathcal{E}$ if and only if:[3]

Sufficiency. There is an algorithm that uses $\mathcal{D}^*$ to solve P in $\mathcal{E}$.

Necessity. For any failure detector $\mathcal{D}$, if $\mathcal{D}$ can be used to solve P in $\mathcal{E}$ then $\mathcal{D}^* \preceq_{\mathcal{E}} \mathcal{D}$.

2.10 Mergeability

Several proofs in distributed computing employ a technique known as the "partition argument". At the heart of this technique is the ability to combine two different runs R_0 and R_1 of an algorithm $\mathcal{A}$ that involve *disjoint* sets of processes P_0 and P_1, respectively, into a single run R of $\mathcal{A}$ in which the processes in P_0 behave as in R_0 and the processes in P_1 behave as in R_1. We now formalize this, and prove that in our model it is possible to combine such "mergeable" runs in this manner.

Two finite runs $R_0 = (F, H, I_0, S_0, T_0)$ and $R_1 = (F, H, I_1, S_1, T_1)$ of an algorithm $\mathcal{A}$ using failure detector $\mathcal{D}$ in some environment $\mathcal{E}$ are *mergeable* if and only if (a) *participants*$(S_0) \cap$ *participants*$(S_1) = \emptyset$, and (b) $\mathcal{A}$ has an initial configuration I such that the initial state of every process in *participants*(S_0) is the same in I as in I_0, and the initial state of every process in *participants*(S_1) is the same in I as in I_1. A *merging* of two such runs is a tuple $R = (F, H, I, S, T)$ where T is a sequence consisting of the times in T_0 and T_1 in nondecreasing order, and S is the sequence consisting of the steps in S_0 and S_1 merged in the same order as the elements of T_0 and T_1 were merged into T. For example, suppose that $S_0 = a_1, a_2, a_3$, $T_0 = 3, 5, 7$; and $S_1 = b_1, b_2, b_3, b_4$, $T_1 = 2, 4, 5, 6$. (Note that steps a_2 of S_0 and b_3 of S_1 are concurrent.) Then $T = 2, 3, 4, 5, 5, 6, 7$, and the two possibilities for S are $b_1, a_1, b_2, b_3, a_2, b_4, a_3$ or $b_1, a_1, b_2, a_2, b_3, b_4, a_3$. More formally, the requirements on S and T for $R = (F, H, I, S, T)$ to be a merging of R_0 and R_1 are

- $|S| = |S_0| + |S_1|$ and $|T| = |T_0| + |T_1|$;
- T is nondecreasing;
- for each $b \in \{0, 1\}$ and each $i \in \{1, 2, \ldots, |S_b|\}$ there is a $j \in \{1, 2, \ldots, |S|\}$ such that $S_b[i] = S[j]$ and $T_b[i] = T[j]$; and
- for each $j \in \{1, 2, \ldots, |S|\}$ there is a $b \in \{0, 1\}$ and an $i \in \{1, 2, \ldots, |S_b|\}$ such that $S[j] = S_b[i]$ and $T[j] = T_b[j]$.

Lemma 2.2 *Let $R = (F, H, I, S, T)$ be a merging of two mergeable finite runs $R_0 = (F, H, I_0, S_0, T_0)$ and $R_1 = (F, H, I_1, S_1, T_1)$ of an algorithm $\mathcal{A}$ using failure detector $\mathcal{D}$ in some environment $\mathcal{E}$. Then*

(a) *R is also a run of $\mathcal{A}$ using $\mathcal{D}$ in $\mathcal{E}$.*
(b) *For each $b \in \{0, 1\}$ and each process $p \in$ participants (S_b), the state of p is the same in $S(I)$ as in $S_b(I_b)$.*

The proof of Lemma 2.2 is straightforward though somewhat tedious; it is given in Appendix A.

3 Failure detectors used in this paper

In this section, we recall the definitions of the leader failure detector Ω [1], and the quorum failure detector Σ [3]. We then introduce the nonuniform counterpart of Σ, denoted Σ^ν.

3.1 Leader failure detector Ω

The *leader failure detector* Ω outputs a single trusted process at each local module, such that there is a time after which all correct processes trust the same correct process.

[3] There may be several distinct failure detectors that are the weakest to solve a problem P. It is easy to see, however, that they are all equivalent. For this reason we speak of *the* weakest, rather than *a* weakest failure detector to solve P.

Formally, the range of Ω is Π. For all failure patterns F, $H \in \Omega(F)$ if and only if:

$$correct(F) \neq \emptyset \implies \left(\exists p \in correct(F),\ \forall q \in correct(F),\ \exists t \in \mathbb{N},\ \forall t' > t : H(q,t') = p\right)$$

3.2 Quorum failure detector Σ

The *quorum failure detector* Σ outputs a set of processes at each process. Any two quorums, output at any times and at any processes, intersect. Moreover, there is a time after which all quorums output at correct processes include only correct processes.

Formally, the range of Σ is 2^{Π}. For all failure patterns F, $H \in \Sigma(F)$ if and only if:

Intersection. Any two quorums intersect.

$$\forall p,p' \in \Pi,\ \forall t,t' \in \mathbb{N} : H(p,t) \cap H(p',t') \neq \emptyset$$

Completeness. There is a time after which the quorums of correct processes contain only correct processes.

$$\exists t \in \mathbb{N},\ \forall p \in correct(F),\ \forall t' > t : H(p,t') \subseteq correct(F)$$

The definition of Σ does not require that the quorums of correct processes eventually converge to the same set; correct processes are free to keep changing their quorums forever.

3.3 Nonuniform quorum failure detector Σ^{ν}

We now define the nonuniform counterpart of Σ, denoted Σ^{ν}.[4] Σ^{ν} differs from Σ in only one respect: only quorums output by correct processes are required to intersect.

Formally, the range of Σ^{ν} is 2^{Π}. For all failure patterns F, $H \in \Sigma^{\nu}(F)$ if and only if:

Nonuniform intersection. Any two quorums that are output by correct processes intersect.

$$\forall p,p' \in correct(F),\ \forall t,t' \in \mathbb{N} : H(p,t) \cap H(p',t') \neq \emptyset$$

Completeness. There is a time after which the quorums of correct processes contain only correct processes.

$$\exists t \in \mathbb{N},\ \forall p \in correct(F),\ \forall t' > t : H(p,t') \subseteq correct(F)$$

4 DAGs and simulations

Recall that to prove that failure detector $\mathcal{D}^*$ is the weakest failure detector that solves a problem P we must prove that: (a) there is an algorithm that uses $\mathcal{D}^*$ to solve P, and (b) any failure detector that can be used to solve P can be transformed to $\mathcal{D}^*$. In this section, we review a proof technique for (b). It was introduced by Chandra et al., who used it to prove that any failure detector that solves nonuniform consensus can be transformed to Ω [1]. We will use it in Sect. 5 to prove that any failure detector that solves nonuniform consensus can be transformed to Σ^{ν}. We will also use a simpler version of this technique in Sect. 6.2.

Suppose that $\mathcal{D}$ is a failure detector that can be used to solve P in some environment $\mathcal{E}$. In other words, there is an algorithm $\mathcal{A}$ that uses $\mathcal{D}$ to solve P in $\mathcal{E}$. The proof technique shows how to use $\mathcal{D}$ and $\mathcal{A}$ to emulate $\mathcal{D}^*$ in $\mathcal{E}$. The emulation consists of two interacting components: the communication component and the computation component. In the communication component, each process "samples" its local module of $\mathcal{D}$ and exchanges messages with other processes to construct a directed acyclic graph (DAG) of failure detector samples of $\mathcal{D}$. In the computation component, p uses this DAG to simulate schedules of the algorithm $\mathcal{A}$ (which uses $\mathcal{D}$ to solve P). Based on these simulated schedules, p simulates the output of the failure detector $\mathcal{D}^*$ that we want to emulate. We now explain in more detail how each process builds its DAG of failure detector samples of $\mathcal{D}$ and how it uses this DAG to simulate schedules of $\mathcal{A}$.

4.1 Building DAGs of failure detector samples

The DAG-building algorithm, denoted $\mathcal{A}_{\text{DAG}}$, is shown in Fig. 1. In our algorithm descriptions, which we give in pseudocode, we use the following conventions. Variables of process p are subscripted with p. If $\mathcal{D}$ is a failure detector, then $\mathcal{D}_p$ denotes the function call by which p can access its local module of $\mathcal{D}$; this call returns the current value of p's local module of $\mathcal{D}$. The pseudocode of each process begins with an **initialize** clause, which defines the process' state in the initial configuration. (Variables whose values are not explicitly set in this clause, can be assigned arbitrary values in the initial configuration.)

In $\mathcal{A}_{\text{DAG}}$, each process p maintains a DAG of failure detector samples of $\mathcal{D}$ in the variable G_p. Each node of this DAG is of the form (q,d,k); such a triple indicates that process q obtained value d when it queried its failure detector module $\mathcal{D}_q$ for the kth time. (The third component is included to ensure that distinct samplings of the failure detector result in distinct nodes.) We call such triples *samples*; a sample $(q,-,-)$ is said to be *of* or *taken by* process q. We use the terms "node (of the DAG)" and "sample" interchangeably.

[4] The superscript reflects the fact that the Greek letter ν is rendered in English as "nu", which is also a suitable abbreviation for the word "nonuniform".

```
CODE FOR EACH PROCESS p:
1  initialize
2    k_p ← 0
3    G_p ← empty graph

4  loop
5    receive a message m
6    d_p ← D_p
7    if m ≠ λ then G_p ← G_p ∪ m
8    k_p ← k_p + 1
9    v_p ← (p, d_p, k_p)
10   add node v_p to G_p and an edge from every other node in G_p to v_p
11   send G_p to every process
```

Fig. 1 Algorithm $\mathcal{A}_{\mathrm{DAG}}$ builds an ever-increasing DAG of failure detector samples of $\mathcal{D}$

Process p periodically performs the following actions:

(a) it receives a message, which is either a DAG previously sent to p by another process, or the empty message (line 5);
(b) it queries its local failure detector module $\mathcal{D}_p$, receiving a value that it stores in variable d_p (line 6);
(c) it updates its DAG G_p by first adding to it the DAG that it received in (a), and then adding to it a new node with the failure detector value it got in (b), as well as edges from all other nodes to the new node (lines 7–10); and
(d) it sends the updated G_p to all processes(line 11).

Note that this sequence of actions (receiving a message, querying the local failure detector module, changing local state, and sending messages to other processes) corresponds exactly to the sequence of actions taken in a single step in our model. Thus, each iteration of the loop in Fig. 1 is executed as a single step.

We now present some properties of the DAGs of samples computed by algorithm $\mathcal{A}_{\mathrm{DAG}}$. In the following, we consider an arbitrary admissible run $R = (F, H, I, S, T)$ of $\mathcal{A}_{\mathrm{DAG}}$ using failure detector $\mathcal{D}$ in some arbitrary environment $\mathcal{E}$. We use the following notation throughout the paper: in the context of a given run of an algorithm, the value of variable x_p at time t is denoted x_p^t; if p takes a step at time t, then x_p^t is the value of x_p *after* that step.

We start with some simple observations, in each of which p is an arbitrary process. Since p never removes any nodes or edges from G_p, the DAG contained in this variable is nondecreasing. That is,

Observation 4.1 *For all $t, t' \in \mathbb{N}$, if $t \leq t'$ then G_p^t is a subgraph of $G_p^{t'}$.*

We define the *limit DAG of a process p* to be $G_p^\infty = \cup_{t\in\mathbb{N}} G_p^t$. Since k_p is incremented in each iteration of p's loop, when p takes sample $(p, -, k)$, it has already taken samples $(p, -, k')$ for all $k' < k$; and, at that time, it adds edges from all such nodes to $(p, -, k)$. Thus,

Observation 4.2 *If $v = (p, -, k)$ and $v' = (p, -, k')$ are nodes of G_p^∞ and $k \geq k'$, then v is a descendant of v' in G_p^∞.*

Let $v = (q, d, k)$ be any node of G_p^∞. It is obvious from the code of $\mathcal{A}_{\mathrm{DAG}}$ that process q received d from its failure detector module in its kth step. Let $\tau(v)$ to be the time when q takes this step. More precisely, if $S[i]$ is the kth step of q in S, then $\tau(v) = T[i]$. (Recall that S is the schedule and T is the sequence of times of the run R of $\mathcal{A}_{\mathrm{DAG}}$ that we are considering.) From property (3) of runs, we have:

Observation 4.3 *If $v = (q, d, k)$ is a node of G_p^∞, then $q \notin F(\tau(v))$ and $d = H(q, \tau(v))$.*

From the algorithm $\mathcal{A}_{\mathrm{DAG}}$, it is clear that if (u, v) is an edge of the limit DAG G_p^∞, then the step in which sample u was taken causally precedes the step in which sample v was taken in schedule S with respect to I. (Recall that I is the initial configuration of the run R of $\mathcal{A}_{\mathrm{DAG}}$ that we are considering.) From property (5) of the runs of $\mathcal{A}_{\mathrm{DAG}}$, it follows that $\tau(u) < \tau(v)$. By induction we can generalize this observation from single edges to finite or infinite paths of G_p^∞:

Observation 4.4 *If $g = v_0, v_1, \ldots$ is a finite or infinite path of G_p^∞, then the sequence of times $\tau(v_0), \tau(v_1), \ldots$ is strictly increasing.*

Let G be any DAG; if v is a node of G, then $G|v$ is the subgraph of G induced by the descendants of v in G; otherwise, $G|v$ is the empty graph. Informally, the next lemma states that any finite path in process p's limit DAG eventually appears permanently in p's DAG.

Lemma 4.5 *Let p be a correct process and v be a node of G_p^∞. For each finite path g in $G_p^\infty|v$, there is a time t such that, for all $t' \geq t$, $g \in G_p^{t'}|v$.*

Proof Let g be any finite path in $G_p^\infty|v$. Since $G_p^\infty = \cup_{t \in \mathbb{N}} G_p^t$, it is clear that for each edge e of g there is a time $t(e)$ such that e is in $G_p^{t(e)}$. Let $t = \max\{t(e) : e \text{ is an edge of } g\}$. By Observation 4.1, every edge e of g (and hence the entire path g) is in $G_p^{t'}$ for all $t' \geq t$. Since g is in $G_p^\infty|v$, every node in g is a descendant of v. Thus, g is in $G_p^{t'}|v$, for all $t' \geq t$. □

Since faulty processes eventually crash and cease to take steps, from a certain point on only correct processes take samples. This is the basic intuition underlying the next lemma.

Lemma 4.6 *For every correct process p, there is a sample v^* of p in G_p^∞ such that $G_p^\infty|v^*$ contains only samples of correct processes. Furthermore,*

(a) *There is a time after which any node v in variable v_p (line 9) is a descendant of v^* in G_p^∞.*
(b) *For any descendant v of v^* in G_p^∞ and any $t \in \mathbb{N}$, $G_p^t|v$ contains only samples of correct processes.*

Proof Since p is correct, it takes infinitely many steps, Let t^* be the first time that p takes a step after all faulty processes have crashed, and let v^* be the sample that p takes in that step. Consider any node v of $G_p^\infty|v^*$. Since v is a descendant of v^* in G_p^∞, by Observation 4.4, $\tau(v) \geq \tau(v^*) = t^*$. Since all faulty processes have crashed by time t^*, the process that takes sample v (at time $\tau(v) \geq t^*$) must be correct. So, $G_p^\infty|v^*$ contains only samples of correct processes.

(a) Let $v^* = (p, -, k^*)$. Since k_p increases in each iteration of p's loop, eventually k_p has values that are more than k^*. Therefore, eventually only nodes whose third entry is more than k^* are assigned to v_p. By Observation 4.2 all these nodes are descendants of v^* in G_p^∞.

(b) Consider any descendant v of v^* in G_p^∞ and any time $t \in \mathbb{N}$. Clearly, $G_p^t|v$ is a subgraph of $G_p^\infty|v$, and $G_p^\infty|v$ is a subgraph of $G_p^\infty|v^*$. Since $G_p^\infty|v^*$ contains only samples of correct processes, so does its subgraph $G_p^t|v$. □

Since correct processes keep taking samples and exchanging their DAGs forever, every correct process' limit DAG has an infinite path with infinitely many samples of each correct process. This observation is formalized by Lemma 4.8. To prove it, it is convenient to prove the following lemma first.

Lemma 4.7 *Let p be a correct process, t be a time, and G be a subgraph of G_p^t. For any correct process q, there is a time t' such that $G_p^{t'}$ contains a sample u of q and an edge from every node of G to u.*

Proof Let s be the first step that p takes after time t. By Observation 4.1, G is still in p's DAG just before this step. There are two cases:

$p = q$. In step s, p adds to its DAG a new sample $u = (p, -, -)$, and edges from every other node in its DAG (in particular, from every node in G) to u. Thus, when this step is completed, say at time t', $G_p^{t'}$ has the desired properties.

$p \neq q$. In step s, p sends to all processes a DAG that contains G. Now consider the step in which q receives that DAG. In that step, q first incorporates the DAG it receives, which contains G, into its own DAG. Then q adds to its DAG a new sample $u = (q, -, -)$, and edges from every other node in its DAG (in particular, from every node in G) to u. Finally, q sends the resulting DAG to all processes. Consider the step in which p receives that DAG. When it does so, p incorporates the DAG it receives into its own DAG. Thus, when this step is completed, say at time t', $G_p^{t'}$ has the desired properties. □

Lemma 4.8 *Let p be a correct process and v be a node of G_p^∞. $G_p^\infty|v$ has an infinite path g^∞ that starts with v and contains infinitely many samples of each correct process.*

Proof Since v is a node of G_p^∞, there is a time t_0 such that v is in $G_p^{t_0}$. By repeated application of Lemma 4.7, there is an infinite sequence of times $t_0, t_1, \ldots$ and an infinite sequence of paths $g^0, g^1, \ldots$ such that for all $i \in \mathbb{N}$, (a) g^i is in $G_p^{t_i}$ and starts with v, (b) g^i is a prefix of g^{i+1}, and (c) each correct process has at least i steps in g^i.

Let g^∞ be the "limit" of sequence $g^0, g^1, \ldots$ That is, g^∞ is the infinite path which, up to length $|g^i|$, is identical to g^i. (This is well-defined because of (b).) It is now easy to see that g^∞ is a path in $G_p^\infty|v$ that starts with v and contains infinitely many samples of each correct process. □

4.2 Simulating schedules of an algorithm $\mathcal{A}$

In the previous section, we saw how each process p can execute algorithm $\mathcal{A}_{\text{DAG}}$ using a failure detector $\mathcal{D}$ to build an ever-increasing DAG of samples of $\mathcal{D}$ (under the "current" failure pattern F and failure detector history $H \in \mathcal{D}(F)$). We now explain how each process p can use its DAG of samples of $\mathcal{D}$ to simulate schedules of runs of *any* algorithm $\mathcal{A}$ using $\mathcal{D}$ (with failure pattern F and failure detector history $H \in \mathcal{D}(F)$). These are called *simulated schedules* of $\mathcal{A}$. Another way of thinking about these simulated schedules is that they are schedules of runs that could have occurred if processes were running algorithm $\mathcal{A}$ using $\mathcal{D}$, instead of running $\mathcal{A}_{\text{DAG}}$ using $\mathcal{D}$.

Fix an initial configuration I of algorithm $\mathcal{A}$, and a path $g = (p_1, d_1, k_1), (p_2, d_2, k_2), \ldots$ of the DAG contained in G_p at some time t, or of the limit DAG G_p^∞. Our goal is to define the set of simulated schedules determined by path g and initial configuration I. Path g tells us that the following could have happened in an execution of algorithm $\mathcal{A}$ under the current failure pattern F and failure detector history $H \in \mathcal{D}(F)$: process p_1 takes the first step and gets value d_1 from its failure detector module; then process p_2 takes the second step and gets value d_2 from its failure detector module; and so on. This sequence of process ids and failure detector values, along with the initial configuration I, define a *set* of schedules of $\mathcal{A}$, each schedule in this set corresponding to different delays that the messages sent might experience.

More precisely, we say that a schedule S is *compatible* with the path $g = (p_1, d_1, k_1), (p_2, d_2, k_2), \ldots$ if and only if it has the same length as g, and $S = (p_1, m_1, d_1, \mathcal{A}), (p_2, m_2, d_2, \mathcal{A}), \ldots$ for some (possibly null) messages $m_1, m_2, \ldots$ The set of simulated schedules determined by g and initial configuration I is the set of all schedules that are compatible with g and applicable to I.

Let G be any DAG of samples and I be any initial configuration of $\mathcal{A}$. $\mathbf{Sch}(G, I)$ denotes the set of schedules of $\mathcal{A}$ that are compatible with some path in G and are applicable to I. Note that if G is finite then $\mathbf{Sch}(G, I)$ contains a finite number of finite schedules.

We now present some properties of simulated schedules. In the following, we consider an arbitrary admissible run R of $\mathcal{A}_{\text{DAG}}$ using failure detector $\mathcal{D}$ in some arbitrary environment $\mathcal{E}$. Let $F \in \mathcal{E}$ be the failure pattern of this run and $H \in \mathcal{D}(F)$ its failure detector history.

The first lemma justifies the name "simulated schedules"; it states that these schedules really are schedules of runs of algorithm $\mathcal{A}$ using $\mathcal{D}$, with failure pattern F and failure detector history H.

Lemma 4.9 *Let p be a process, $t \in \mathbb{N} \cup \{\infty\}$, G be a subgraph of G_p^t, and I be an initial configuration of algorithm $\mathcal{A}$. For each schedule $S \in \mathbf{Sch}(G_p^t|u, I)$, there is a list of times T such that $R_\mathcal{A} = (F, H, I, S, T)$ is a run of $\mathcal{A}$ using $\mathcal{D}$ in $\mathcal{E}$.*

Proof Let S be any schedule in $\mathbf{Sch}(G, I)$. Thus, S is applicable to I and compatible with some path $g = v_1, v_2, \ldots$ in G. Let $T = \tau(v_1), \tau(v_2), \ldots$ We claim that $R_\mathcal{A} = (F, H, I, S, T)$ is a run of $\mathcal{A}$ using $\mathcal{D}$ in $\mathcal{E}$. Since $F \in \mathcal{E}$, $H \in \mathcal{D}(F)$ and I is an initial configuration of $\mathcal{A}$, it suffices to verify that $R_\mathcal{A}$ satisfies properties (1)–(5) of runs. S is applicable to I (property (1)) by definition of $\mathbf{Sch}(G, I)$. S and T have the same length (property (2)) because each of them has the same length as g. The fact that in R no process takes a step after it has crashed, and that the failure detector value in each step is consistent with the history H (property (3)) follows from Observation 4.3, since S is compatible with path $g = v_1, v_2, \ldots$ and $T = \tau(v_1), \tau(v_2), \ldots$ Observation 4.4 implies that T is strictly increasing, and so property (4) is also satisfied. To show property (5), we must prove that if step i causally precedes step j in S with respect to I then $T[i] < T[j]$. This follows from Observation 2.1 and the fact that T is strictly increasing. □

Since $G_p^\infty|u$ is a subgraph of G_p^∞, by Lemma 4.9 every infinite schedule S^∞ in $\mathbf{Sch}(G_p^\infty|u, I)$ is a schedule of an infinite run of $\mathcal{A}$ using $\mathcal{D}$ in $\mathcal{E}$. However, S^∞ is not necessarily a schedule of an *admissible* run, i.e., a run where each correct process takes an infinite number of steps (property (6)) and eventually receives every message sent to it (property (7)). The next lemma, however, states that $\mathbf{Sch}(G_p^\infty|u, I)$ contains at least one infinite schedule of an *admissible* run of $\mathcal{A}$.

Lemma 4.10 *Let p be a correct process, u be a node of G_p^∞, and I be an initial configuration of $\mathcal{A}$. There is a schedule $S^\infty \in \mathbf{Sch}(G_p^\infty|u, I)$ and a list of times T^∞ such that $R_\mathcal{A} = (F, H, I, S^\infty, T^\infty)$ is an admissible run of $\mathcal{A}$ using $\mathcal{D}$ in $\mathcal{E}$.*

Proof By Lemma 4.8, $G_p^\infty|u$ has an infinite path g^∞ that contains infinitely many samples of each correct process. We define an infinite sequence of schedules $S^0, S^1, \ldots$ such that for each $i \in \mathbb{N}$, (a) S^i has length i, (b) S^i is compatible with the path consisting of the first i nodes of g^∞, (c) S^i is applicable to I, and (d) if $i > 0$, S^{i-1} is a prefix of S^i. The definition is by induction:

Basis S^0 is the empty schedule. It is obvious that this has the required properties.

Induction step Let i be an arbitrary positive integer, and assume that S^{i-1} with the required properties has been defined. Let the ith node of g^∞ be $(p, d, -)$. Then S^i is obtained from S^{i-1} by appending to it the step $(p, m, d, \mathcal{A})$, where m is the message defined as follows: if the message buffer of configuration $S^{i-1}(I)$ has no message to p (i.e., no message of the form $(-,-,p)$), then $m=\lambda$; otherwise, m is the *oldest* message to p in the message buffer of $S^{i-1}(I)$ (i.e., there is no message m' to p in the message buffer of $S^{i-1}(I)$ and a $j<i-1$ such that the message buffer of $S^j(I)$ contains m' but not m). It is obvious that S^i has the required properties: length i, compatible with the first i nodes of g^∞, applicable to I, and an extension of S^{i-1}.

Now define S^∞ to be the infinite schedule whose prefix of length i is S^i. (This is well-defined because, for all $i \in \mathbb{N}$, S^i has length i and is a prefix of S^{i+1}.) Clearly S^∞ is compatible with g^∞ and applicable to I. Since g^∞ is a path in $G_p^\infty|u$, we have that $S^\infty \in \mathbf{Sch}(G_p^\infty|u,I)$. By Lemma 4.9, there is a time list T^∞ such that $R_\mathcal{A}=(F,H,I,S^\infty,T^\infty)$ is a run of $\mathcal{A}$ using $\mathcal{D}$ in $\mathcal{E}$. It remains to prove that $R_\mathcal{A}$ is admissible. We first note that each correct process takes infinitely many steps in $R_\mathcal{A}$; this is because S^∞ is compatible with g^∞ and g^∞ contains infinitely many samples of each correct process. Furthermore, from the way we choose the message received in each step of S^∞, every message sent to a correct process is eventually received in $R_\mathcal{A}$. So, $R_\mathcal{A}$ has properties (6) and (7) of admissible runs. □

The following lemma is an immediate consequence of Lemma 4.5 and the definition of $\mathbf{Sch}(G|u,I)$:

Lemma 4.11 *Let p be a correct process, u be a node of G_p^∞, and I be an initial configuration of $\mathcal{A}$. For each finite schedule $S \in \mathbf{Sch}(G_p^\infty|u,I)$, there is a time t such that, for all $t' \geq t$, $S \in \mathbf{Sch}(G_p^{t'}|u,I)$.*

5 (Ω, Σ^ν) is necessary for solving nonuniform consensus

In this section, we prove that in any environment $\mathcal{E}$, any failure detector $\mathcal{D}$ that can be used to solve binary nonuniform consensus can be transformed to Σ^ν. Intuitively, this says that, in any environment, Σ^ν is necessary to solve nonuniform consensus. Previously, Chandra et al. had shown that, in any environment, Ω is also necessary to solve nonuniform consensus [1]. By combining these two results, we get that, in any environment, (Ω, Σ^ν) is necessary to solve nonuniform consensus.

Let $\mathcal{A}$ be any algorithm that uses $\mathcal{D}$ to solve binary nonuniform consensus in $\mathcal{E}$. In Fig. 2, we present an algorithm $\mathcal{T}_{\mathcal{D}\to\Sigma^\nu}$ that uses $\mathcal{A}$ to transform $\mathcal{D}$ to Σ^ν. This algorithm incorporates verbatim the DAG-building algorithm $\mathcal{A}_{\text{DAG}}$ (Fig. 1) on lines 3–12. In the rest of the algorithm, each process p uses a "recent" subgraph of its current DAG G_p to simulate schedules of runs of

```
CODE FOR EACH PROCESS p:
1  initialize
2  Σ^ν-output_p ← Π
3  k_p ← 0
4  G_p ← empty graph

5  loop
6    receive a message m
7    d_p ← D_p
8    if m ≠ λ then G_p ← G_p ∪ m
9    k_p ← k_p + 1
10   v_p ← (p, d_p, k_p)
11   add node v_p to G_p and an edge from every other node in G_p to v_p
12   send G_p to every process
13   if k_p = 1 then u_p ← v_p
14   let G_p|u_p be the subgraph induced by the descendants of u_p in the DAG G_p
15   let I_0 and I_1 be the initial configurations of A where all processes propose 0 and 1, respectively
16   let Sch(G_p|u_p, I_0) and Sch(G_p|u_p, I_1) be the sets of schedules of A that are compatible with some path
       of G_p|u_p and applicable to I_0 and I_1, respectively
17   if ∃ S_0 ∈ Sch(G_p|u_p, I_0) and S_1 ∈ Sch(G_p|u_p, I_1) such that p decides in both S_0(I_0) and S_1(I_1) then
18     Σ^ν-output_p ← participants(S_0) ∪ participants(S_1)
19     u_p ← v_p
```

Fig. 2 Algorithm $\mathcal{T}_{\mathcal{D}\to\Sigma^\nu}$

$\mathcal{A}$ using $\mathcal{D}$. Using these schedules p periodically determines a new quorum of Σ^ν. We now explain this in more detail.

Process p maintains a "recent" sample of its own in variable u_p. This is initialized to p's first sample (line 13), and is updated to p's most recent sample each time p outputs a new quorum (lines 18–19). The sample stored in u_p acts as a "freshness barrier": p's new quorum contains only processes that have taken samples at least as recent as u_p. This ensures the completeness property of Σ^ν.

Process p maintains two sets of simulated schedules, $\mathbf{Sch}(G_p|u_p, I_0)$ and $\mathbf{Sch}(G_p|u_p, I_1)$, where I_0 and I_1 are the initial configurations of $\mathcal{A}$ in which all processes propose 0 and 1, respectively (lines 14–16). Process p checks whether these two sets contain simulated schedules S_0 and S_1, respectively, such that p decides in both $S_0(I_0)$ and $S_1(I_1)$; if p finds such schedules, p updates its quorum by assigning to Σ^ν*-output*$_p$ the set of processes that take steps in either of these two schedules (lines 17–18). As we will see, this way of choosing quorums ensures the nonuniform intersection property of Σ^ν.

Note that in each iteration of the loop, p performs the actions that correspond to a step in our model: receive a message, query the failure detector, change state and send messages. Thus, in our model, a process executes an iteration of the loop in one atomic step.

In what follows, we fix an arbitrary admissible run of algorithm $\mathcal{T}_{\mathcal{D}\rightarrow\Sigma^\nu}$ using failure detector $\mathcal{D}$ in some arbitrary environment $\mathcal{E}$. Let $F \in \mathcal{E}$ be the failure pattern of this run and $H \in \mathcal{D}(F)$ be its failure detector history. We will prove that the quorums assigned to the variables Σ^ν-output$_\mathrm{p}$ in this run satisfy the properties of Σ^ν.

Lemma 5.1 *Every correct process p assigns a quorum to* Σ^ν*-output*$_p$ *and a node to* u_p *infinitely often.*

Proof Let p be any correct process. From the algorithm, it is clear that Σ^ν-output$_\mathrm{p}$ and u_p are either both assigned infinitely often or both assigned a finite number of times. For contradiction, suppose that they are both assigned a finite number of times. By line 13, p assigns a node to u_p at least once. Let t_0 be the time when the final assignment of u_p occurs and let u be the final value of u_p.

By line 11, u is a node of G_p^∞. By Lemma 4.10, there is a schedule S_0^∞ in $\mathbf{Sch}(G_p^\infty|u, I_0)$ such that there is an admissible run $R_0 = (F, H, I_0, S_0^\infty, -)$ of $\mathcal{A}$ using $\mathcal{D}$. Recall that $\mathcal{A}$ uses $\mathcal{D}$ to solve nonuniform consensus in environment $\mathcal{E}$, and $F \in \mathcal{E}$. Since all processes propose 0 in I_0, by the termination and validity properties, every correct process—and in particular p—decides 0 in R_0. Thus, there is a finite prefix S_0 of S_0^∞ such that p decides 0 in $S_0(I_0)$. By similar reasoning, there is a finite prefix S_1 of some schedule S_1^∞ in $\mathbf{Sch}(G_p^\infty|u, I_1)$ such that p decides 1 in $S_1(I_1)$.

Since S_0 and S_1 are *finite* schedules in $\mathbf{Sch}(G_p^\infty|u, I_0)$ and $\mathbf{Sch}(G_p^\infty|u, I_1)$, respectively, by Lemma 4.11, there is a time t_1 such that for all $t \geq t_1$, $S_0 \in \mathbf{Sch}(G_p^t|u, I_0)$ and $S_1 \in \mathbf{Sch}(G_p^t|u, I_1)$.

Let $t^* = \max(t_0, t_1)$. Thus, after time t^*, $u_p = u$ and the condition of the if-statement on line 17 is true. So, the first time after t^* that p executes line 17, it finds that the condition of that if-statement is satisfied, and assigns a node to u_p in line 19. This occurs after time t_0, contradicting the definition of t_0. □

Lemma 5.2 (Completeness) *For every correct process p, there is a time after which the quorums assigned to* Σ^ν*-*output$_\mathrm{p}$ *contain only correct processes.*

Proof Let p be a correct process. By Lemma 4.6, there is a sample v^* of p in G_p^∞ such that $G_p^\infty|v^*$ contains only samples of correct processes. By Lemma 4.6(a), there is a time after which any node v contained in v_p is a descendant of v^* in G_p^∞. By Lemma 5.1, there are infinitely many assignments to u_p; in each such assignment, u_p is assigned the node that is currently in v_p (see lines 13 and 19). Thus, there is a time t^* such that for all $t \geq t^*$, u_p^t is a descendant of v^* in G_p^∞. By Lemma 4.6(b), for all $t \geq t^*$, $G_p^t|u_p^t$ contains only samples of correct processes.

By Lemma 5.1, p assigns a quorum to Σ^ν-output$_\mathrm{p}$ infinitely often after time t^*. Since every assignment to Σ^ν-output$_\mathrm{p}$ (other than the initialization) occurs on line 18, it suffices to prove that any quorum assigned to Σ^ν-output$_\mathrm{p}$ after time t^* on line 18 contains only correct processes. Consider any such assignment, say at time $t \geq t^*$ (see lines 17–18). The quorum assigned to Σ^ν-output$_\mathrm{p}$ at time t is *participants*$(S_0) \cup$*participants*(S_1), where $S_0 \in \mathbf{Sch}(G_p^t|u_p^t, I_0)$ and $S_1 \in \mathbf{Sch}(G_p^t|u_p^t, I_1)$. Since $t \geq t^*$, $G_p^t|u_p^t$ contains only samples of correct processes. This implies that all processes in *participants*$(S_0) \cup$ *participants*(S_1) are correct; and so the quorum assigned to Σ^ν-output$_\mathrm{p}$ at time t contains only correct processes. □

Lemma 5.3 (Nonuniform intersection) *For all correct processes p and q, any two quorums assigned to* Σ^ν-output$_\mathrm{p}$ *and* Σ^ν-output$_\mathrm{q}$ *intersect.*

Proof Suppose, by way of contradiction, that there are correct processes p and q such that at some time t Σ^ν-output$_\mathrm{p}$ = P, at some time t' Σ^ν-output$_\mathrm{q}$ = Q, and $P \cap Q = \emptyset$. By the algorithm (see lines 17–18), there is a schedule $S_0 \in \mathbf{Sch}(G_p^t|u_p^t, I_0)$, such that p decides 0 in $S_0(I_0)$ and *participants*$(S_0) \subseteq P$; and a schedule $S_1 \in \mathbf{Sch}(G_q^{t'}|u_q^{t'}, I_1)$, such that q decides 1 in $S_1(I_1)$ and *participants*$(S_1) \subseteq Q$.

By Lemma 4.9, for some time lists T_0 and T_1, $R_0 = (F, H, I_0, S_0, T_0)$ and $R_1 = (F, H, I_1, S_1, T_1)$ are runs of $\mathcal{A}$ using $\mathcal{D}$. Since P and Q are disjoint, so are their subsets, $participants(S_0)$ and $participants(S_1)$. Moreover, $\mathcal{A}$ has an initial configuration I in which every process in $participants(S_0)$ proposes 0 (as in I_0), and every process in $participants(S_1)$ proposes 1 (as in I_1). Thus, R_0 and R_1 are mergeable. Let $R = (F, H, I, S, T)$ be a merging of R_0 and R_1. By Lemma 2.2, R is a run of $\mathcal{A}$ using $\mathcal{D}$, the state of p is the same in $S(I)$ as in $S_0(I_0)$, and the state of q is the same in $S(I)$ as in $S_1(I_1)$. Thus, R is a run of $\mathcal{A}$ using $\mathcal{D}$ in which p decides 0 and q decides 1. This contradicts the agreement property of nonuniform consensus. □

By Lemmata 5.2 and 5.3, the values of Σ^ν-output$_p$ satisfy the properties of Σ^ν. Therefore,

Theorem 5.4 *For all environments $\mathcal{E}$, if a failure detector $\mathcal{D}$ can be used to solve nonuniform consensus in $\mathcal{E}$, then algorithm $\mathcal{T}_{\mathcal{D}\to\Sigma^\nu}$ transforms $\mathcal{D}$ to Σ^ν in $\mathcal{E}$.*

Corollary 5.5 *For all environments $\mathcal{E}$, if a failure detector $\mathcal{D}$ can be used to solve nonuniform consensus in $\mathcal{E}$, then $\Sigma^\nu \preceq_{\mathcal{E}} \mathcal{D}$.*

Informally, this corollary says that Σ^ν is necessary to solve nonuniform consensus. Chandra et al. proved that Ω is also necessary to solve nonuniform consensus and, *a fortiori*, uniform consensus [1]:

Theorem 5.6 *For all environments $\mathcal{E}$, if a failure detector $\mathcal{D}$ can be used to solve nonuniform or uniform consensus in $\mathcal{E}$, then $\Omega \preceq_{\mathcal{E}} \mathcal{D}$.*

From Corollary 5.5 and Theorem 5.6, the pair (Ω, Σ^ν) is necessary to solve nonuniform consensus:

Theorem 5.7 *For all environments $\mathcal{E}$, if a failure detector $\mathcal{D}$ can be used to solve nonuniform consensus in $\mathcal{E}$, then $(\Omega, \Sigma^\nu) \preceq_{\mathcal{E}} \mathcal{D}$.*

5.1 Remark on uniform consensus

It turns out that the transformation algorithm $\mathcal{T}_{\mathcal{D}\to\Sigma^\nu}$, given in Fig. 2, also shows that Σ is necessary to solve *uniform* consensus: if $\mathcal{D}$ is a failure detector that can be used to solve uniform consensus then algorithm $\mathcal{T}_{\mathcal{D}\to\Sigma^\nu}$ transforms $\mathcal{D}$ to Σ. This provides an alternative proof of a result previously shown by Delporte et al. [3].

Theorem 5.8 *For all environments $\mathcal{E}$, if a failure detector $\mathcal{D}$ can be used to solve uniform consensus in $\mathcal{E}$, then algorithm $\mathcal{T}_{\mathcal{D}\to\Sigma^\nu}$ transforms $\mathcal{D}$ to Σ in $\mathcal{E}$.*

The proof of Theorem 5.8 is identical to the proof of Theorem 5.4, except that, in Lemma 5.3, we remove each occurrence of the word "correct" and replace every occurrence of the word "nonuniform" by "uniform".

Corollary 5.9 (Delporte et al. [3]) *For all environments $\mathcal{E}$, if a failure detector $\mathcal{D}$ can be used to solve uniform consensus in $\mathcal{E}$, then $\Sigma \preceq_{\mathcal{E}} \mathcal{D}$.*

From Corollary 5.9 and Theorem 5.6, the pair (Ω, Σ) is necessary to solve uniform consensus:

Theorem 5.10 (Delporte et al. [3]) *For all environments $\mathcal{E}$, if a failure detector $\mathcal{D}$ can be used to solve uniform consensus in $\mathcal{E}$, then $(\Omega, \Sigma) \preceq_{\mathcal{E}} \mathcal{D}$.*

6 (Ω, Σ^ν) is sufficient for solving nonuniform consensus

We now prove that (Ω, Σ^ν) can be used to solve nonuniform consensus in any environment. To do so, we first introduce $\Sigma^{\nu+}$, a failure detector that appears to be stronger than Σ^ν (Sect. 6.1). We then prove that $\Sigma^{\nu+}$ is actually equivalent to Σ^ν, by giving an algorithm that transforms Σ^ν to $\Sigma^{\nu+}$ in any environment (Sect. 6.2). Finally, we present an algorithm that uses $(\Omega, \Sigma^{\nu+})$ to solve nonuniform consensus in any environment (Sect. 6.3).

6.1 Failure detector $\Sigma^{\nu+}$

Failure detector $\Sigma^{\nu+}$ is obtained by adding two properties to Σ^ν, as explained below. The range of $\Sigma^{\nu+}$ is 2^Π. For all failure patterns F, $H \in \Sigma^{\nu+}(F)$ if and only if H satisfies the properties of Σ^ν (completeness and nonuniform intersection), as well as the following:

Conditional nonintersection. Any quorum that does not intersect with some quorum of a correct process contains only faulty processes.

$$\forall p \in correct(F),\ \forall p' \in \Pi,\ \forall\ t, t' \in \mathbb{N}:$$
$$H(p,t) \cap H(p',t') = \emptyset \implies H(p',t') \subseteq faulty(F)$$

Self-inclusion. Each process is contained in all its quorums.

$$\forall p \in \Pi,\ \forall t \in \mathbb{N} : p \in H(p,t)$$

It is easy to see that the above two properties imply the nonuniform intersection property of Σ^ν. It is convenient, however, to keep nonuniform intersection as an explicit property of $\Sigma^{\nu+}$.

CODE FOR EACH PROCESS p:

1 **initialize**
2   $\Sigma^{\nu+}\text{-}output_p \leftarrow \Pi$
3   $k_p \leftarrow 0$
4   $G_p \leftarrow$ empty graph

5 **loop**
6   receive a message m
7   $d_p \leftarrow \Sigma^{\nu}_p$
8   **if** $m \neq \lambda$ **then** $G_p \leftarrow G_p \cup m$
9   $k_p \leftarrow k_p + 1$
10   $v_p \leftarrow (p, d_p, k_p)$
11   add node v_p to G_p and an edge from every other node in G_p to v_p
12   send G_p to every process
13   **if** $k_p = 1$ **then** $u_p \leftarrow v_p$
14   let $G_p|u_p$ be the subgraph induced by the descendants of u_p in the DAG G_p
15   **if** $\exists$ path g in $G_p|u_p$ such that $trusted(g) \subseteq participants(g)$ and $p \in participants(g)$ **then**
16     $\Sigma^{\nu+}\text{-}output_p \leftarrow participants(g)$
17     $u_p \leftarrow v_p$

18 **function** $trusted(g)$
19   **return** $\bigcup\{d \mid \exists i : g[i] = (-, d, -)\}$

20 **function** $participants(g)$
21   **return** $\{q \mid \exists i : g[i] = (q, -, -)\}$

Fig. 3 Algorithm $\mathcal{T}_{\Sigma^{\nu} \to \Sigma^{\nu+}}$ transforms Σ^{ν} to $\Sigma^{\nu+}$

6.2 Equivalence of Σ^{ν} and $\Sigma^{\nu+}$

We now describe an algorithm, denoted $\mathcal{T}_{\Sigma^{\nu} \to \Sigma^{\nu+}}$, that transforms Σ^{ν} to $\Sigma^{\nu+}$ in any environment $\mathcal{E}$. This algorithm, shown in Fig. 3, is explained below.

Algorithm $\mathcal{T}_{\Sigma^{\nu} \to \Sigma^{\nu+}}$ incorporates the DAG-building algorithm $\mathcal{A}_{\text{DAG}}$ (Fig. 1) verbatim on lines 13–12. Here, the failure detector that is getting sampled is Σ^{ν} (line 7). In lines 13–17, each process p uses a "recent" subgraph of its DAG of samples G_p to determine the next $\Sigma^{\nu+}$-quorum to output. The "freshness" of the subgraph used is achieved by the same technique employed in algorithm $\mathcal{T}_{\mathcal{D} \to \Sigma^{\nu}}$ (Fig. 2). That is, p keeps a "recent" sample of its own in variable u_p. This variable is initialized to p's first sample (line 13), and updated to p's most recent sample each time p outputs a new quorum (lines 16–17). Process p's new quorum (line 16) includes only processes that have taken samples in $G_p|u_p$ — i.e., samples at least as recent as u_p.

We now explain how this quorum is chosen. Given a path $g = (p_1, d_1, k_1), (p_2, d_2, k_2), \ldots$ in $G_p|u_p$, let *participants*(g) be the set $\{p_1, p_2, \ldots\}$ of processes that appear in the first components of the nodes of g, and *trusted*(g) be the union of the Σ^{ν}-quorums $d_1, d_2, \ldots$ in the second components of the nodes of g. Initially, each process p outputs Π (line 2). To determine its next $\Sigma^{\nu+}$-quorum, p looks at paths of $G_p|u_p$. If it finds one such path g with the property that *trusted*$(g) \subseteq$ *participants*(g) and $p \in$ *participants*(g), then its next $\Sigma^{\nu+}$-quorum is *participants*(g) (lines 14–16). As we will see, the "freshness" of the samples considered ensures completeness, while the rule for choosing new quorums ensures the remaining properties of $\Sigma^{\nu+}$.

Note that in our model, process p executes an iteration of the loop in one atomic step. In what follows, we consider an arbitrary admissible run of algorithm $\mathcal{T}_{\Sigma^{\nu} \to \Sigma^{\nu+}}$ using Σ^{ν} in some arbitrary environment $\mathcal{E}$. Let $F \in \mathcal{E}$ be the failure pattern of this run and $H \in \mathcal{D}(F)$ be its failure detector history. We will prove that the quorums assigned to the variables $\Sigma^{\nu+}$-output$_p$ in this run satisfy the properties of $\Sigma^{\nu+}$.

Lemma 6.1 *Every correct process p assigns a quorum to* $\Sigma^{\nu+}$*-output*$_p$ *and a node to* u_p *infinitely often.*

Proof Let p be a correct process. From the algorithm, it is clear that Σ^{ν}-output$_{\mathrm{p}}$ and u_p are either both assigned infinitely often or both assigned a finite number of times. For contradiction, suppose that they are both assigned a finite number of times. By line 13, p assigns a node to u_p at least once. Let u be the final value of u_p. By line 11, u is a node of G_p^{∞}. By Lemma 4.8, $G_p^{\infty}|u$ has an infinite path $v_1, v_2, \ldots$ that contains infinitely many samples of each correct process.

By the definition of faulty processes and the completeness property of Σ^{ν}, there is a time t after which (a) only correct processes take steps, and (b) the Σ^{ν}-quorums of correct processes contain only correct processes. By Observation 4.4, the sequence $\tau(v_1), \tau(v_2), \ldots$ (the sequence of times at which samples $v_1, v_2, \ldots$ were taken) is strictly increasing. Thus, there is some k such that for all $j \geq k$, $\tau(v_k) \geq t$. So by the definition of t, for each $j \geq k$, the process associated with v_j must be correct and the Σ^{ν}-quorum associated with v_j contains only correct processes. Let $g = v_k, v_{k+1}, \ldots, v_{\ell}$ be a finite subpath of $v_1, v_2, \ldots$ so that every correct process has at least one sample in g. (Such a path exists because every correct process has infinitely many samples in $v_1, v_2, \ldots$, and therefore also in $v_k, v_{k+1}, \ldots$.) By definition of g, $participants(g) = correct(F)$; and, since p is correct, $p \in participants(g)$. As argued above, the Σ^{ν}-quorum associated with each v_j, $k \leq j \leq \ell$, contains only correct processes. Thus, $trusted(g) \subseteq correct(F)$. So, $trusted(g) \subseteq participants(g)$ and $p \in participants(g)$.

Since g is a finite path in $G_p^{\infty}|u$, by Lemma 4.5, there is a time t_1 such that for all $t \geq t_1$, $g \in G_p^t|u$. Let t_2 be a time after the final assignment to u_p occurs, and let $t^* = \max(t_1, t_2)$. Thus, after t^*, $u_p = u$, and the condition of the if-statement on line 15 is true. So, the first time after t^* that p executes line 15, it finds the condition of that if-statement to be true, and assigns a node to u_p in line 17. This occurs after time t_1, contradicting the definition of t_1. □

Lemma 6.2 (Completeness) *There is a time after which, for every correct process p, the quorums assigned to $\Sigma^{\nu+}$-output$_p$ contain only correct processes.*

Proof Let p be a correct process. By Lemma 4.6, there is a sample v^* of p in G_p^{∞} such that $G_p^{\infty}|v^*$ contains only samples of correct processes. By Lemma 4.6(a), there is a time after which any node v contained in v_p is a descendant of v^* in G_p^{∞}. By Lemma 6.1, there are infinitely many assignments to u_p; in all of these u_p is assigned the node in v_p (see lines 13 and 17). Thus, there is a time t^* such that for all $t \geq t^*$, u_p^t is a descendant of v^* in G_p^{∞}. By Lemma 4.6(b), for all $t \geq t^*$, $G_p^t|u_p^t$ contains only samples of correct processes.

By Lemma 6.1, p assigns a quorum to $\Sigma^{\nu+}$-output$_p$ infinitely often after time t^*. Since every assignment to $\Sigma^{\nu+}$-output$_p$ (other than the initialization) occurs on line 16, it suffices to prove that any quorum assigned to $\Sigma^{\nu+}$-output$_p$ after time t^* on line 16 contains only correct processes. Consider any such assignment, say at time $t \geq t^*$ (see lines 15–16). The quorum assigned to Σ^{ν}-output$_p$ at time t is $participants(g)$, where g is a path in $G_p^t|u_p^t$. Since $t \geq t^*$, $G_p^t|u_p^t$ contains only samples of correct processes. Thus all processes in $participants(g)$ are correct; and so the quorum assigned to Σ^{ν}-output$_p$ at time t contains only correct processes. □

Lemma 6.3 (Self-inclusion) *For each process p, all quorums assigned to $\Sigma^{\nu+}$-output$_p$ contain p.*

Proof Initially, $\Sigma^{\nu+}$-output$_{\mathrm{p}} = \Pi$ (line 15) and so the initial quorum in $\Sigma^{\nu+}$-output$_{\mathrm{p}}$ includes p. Any subsequent assignment of a quorum to $\Sigma^{\nu+}$-output$_{\mathrm{p}}$, occurs on line 16; by the condition on line 15, such quorum includes p. □

Lemma 6.4 *Let p, q be any two processes, and P, Q be any quorums assigned to $\Sigma^{\nu+}$-output$_p$ and $\Sigma^{\nu+}$-output$_q$ respectively. If P contains a correct process and Q contains a correct process, then $P \cap Q \neq \emptyset$.*

Proof We first show that, for any processes p, q, if q belongs to a quorum P assigned to $\Sigma^{\nu+}$-output$_p$, then P is a superset of some Σ^{ν}-quorum output at q. This is obvious if P is the initial value Π of $\Sigma^{\nu+}$-output$_p$, since Π is a superset of every quorum output at any process. Otherwise, P is the quorum assigned to $\Sigma^{\nu+}$-output$_p$ on line 16 at some time t. By the algorithm, we have that $P = participants(g)$, where g is a path in $G_p^t|u_p^t$, and $trusted(g) \subseteq participants(g)$. For any process $q \in participants(g)$, there is a node $(q, Q', -)$ in g, where Q' is a Σ^{ν}-quorum output at q. By definition of $trusted(g)$, $Q' \subseteq trusted(g) \subseteq participants(g) = P$. Thus, P is a superset of a Σ^{ν}-quorum Q' output at q.

Now, suppose that for some processes p and q, P and Q are quorums assigned to variables $\Sigma^{\nu+}$-output$_p$ and $\Sigma^{\nu+}$-output$_q$, respectively, and there are correct processes $p' \in P$ and $q' \in Q$. By the previous paragraph, $p'$and q' output Σ^{ν}-quorums P' and Q', respectively, such that $P' \subseteq P$ and $Q' \subseteq Q$. Since p' and q' are correct, by nonuniform intersection of Σ^{ν}, $P' \cap Q' \neq \emptyset$. Thus, $P \cap Q \neq \emptyset$. □

Lemma 6.5 (Nonuniform intersection) *For all correct processes p and q, any two quorums assigned to $\Sigma^{\nu+}$-output$_p$ and $\Sigma^{\nu+}$-output$_q$ intersect.*

Proof Let P and Q by any quorums assigned to $\Sigma^{\nu+}$-output$_{\mathrm{p}}$ and $\Sigma^{\nu+}$-output$_{\mathrm{q}}$, respectively. By Lemma

6.3, $p \in P$ and $q \in Q$. Since p and q are correct, by Lemma 6.4, $P \cap Q \neq \emptyset$. □

Lemma 6.6 (Conditional nonintersection) *Let p be a correct process and q be any process. Any quorum assigned to $\Sigma^{\nu+}$-output$_q$ that does not intersect a quorum assigned to $\Sigma^{\nu+}$-output$_p$ contains only faulty processes.*

Proof Let P and Q by any quorums assigned to $\Sigma^{\nu+}$-output$_\mathrm{p}$ and $\Sigma^{\nu+}$-output$_\mathrm{q}$, respectively. Suppose, by way of contradiction, that $P \cap Q = \emptyset$ and Q contains a correct process. By Lemma 6.3, $p \in P$, and so P also contains a correct process. By Lemma 6.4, $P \cap Q \neq \emptyset$ — a contradiction. □

By Lemmata 6.2, 6.3, 6.5 and 6.6, in any run of algorithm $\mathcal{T}_{\Sigma^\nu \to \Sigma^{\nu+}}$, the values of the variables $\Sigma^{\nu+}$-output$_\mathrm{p}$ satisfy the properties of $\Sigma^{\nu+}$. Therefore,

Theorem 6.7 *For all environments $\mathcal{E}$, algorithm $\mathcal{T}_{\Sigma^\nu \to \Sigma^{\nu+}}$ transforms Σ^ν to $\Sigma^{\nu+}$ in $\mathcal{E}$. Thus, for all environments $\mathcal{E}$, $\Sigma^{\nu+} \preceq_{\mathcal{E}} \Sigma^\nu$.*

Since $\Sigma^{\nu+}$ satisfies the properties of Σ^ν, it is clear that $\Sigma^\nu \preceq_{\mathcal{E}} \Sigma^{\nu+}$ for all environments $\mathcal{E}$. Therefore,

Corollary 6.8 *For all environments $\mathcal{E}$, $\Sigma^\nu \equiv_{\mathcal{E}} \Sigma^{\nu+}$.*

6.3 Using $(\Omega, \Sigma^{\nu+})$ to solve nonuniform consensus

We now describe an algorithm that uses failure detector $(\Omega, \Sigma^{\nu+})$ to solve nonuniform consensus in any environment. We start with a high-level description of the algorithm, which motivates the need for, and explains, the mechanisms it uses. We then give a detailed description of the algorithm and prove its correctness.

High-level description

The starting point for our algorithm is an algorithm due to Mostéfaoui and Raynal. That algorithm uses Ω to solve *uniform* consensus in environments where *a majority of processes are correct* [6]. Roughly speaking, the Mostéfaoui–Raynal algorithm works as follows.

Each process maintains an "estimate" (of what will become its decision), which is initially set to the value the process wants to propose. Processes proceed in asynchronous rounds (in each round k, processes send and receive messages tagged with round number k). Each asynchronous round is divided into three phases.

In the first phase, each process p sends a *leader message*, containing p's current estimate, to all processes. Then p waits to receive a leader message from its current leader c, i.e., from the process c currently output by the failure detector Ω at p. Finally, p adopts the estimate contained in c's leader message as its own estimate.

In the second phase, each process p sends a *report message*, containing p's current estimate, to all processes. Then p waits to receive reports from a majority of processes. Based on the reports it receives, p prepares a *proposal message* that it will send in the third and final phase: If p receives reports with the same estimate v from a majority of processes, it will send a proposal for v; otherwise it will send a proposal for the special value "?".

In the third phase, each process p sends its proposal as described above to all processes. Then p waits to receive proposals from a majority of processes. If p receives at least one proposal for a value $v \neq ?$, then p adopts v as its new estimate. If p receives a majority of proposals for $v \neq ?$, then it decides v. Process p then proceeds to the next round.

The following two properties are key for the correctness of this algorithm:

(A) *In each round, no process receives proposals for different values $v \neq ?$ and $v' \neq ?$.* This is because a process receives a proposal for $v \neq ?$ from a process q only if q previously received reports for v from a majority of processes, and any two majorities must intersect.

(B) *If a process decides v in some round, then all processes that complete that round do so with estimate v.* Again, this is because of the intersection property of majorities: A process p decides v in some round only if it receives proposals for v from a majority of processes. This implies that any other process that completes that round receives a proposal from at least one of the processes from which p received the proposals for v, and will therefore adopt v as its estimate.

The fact that any two majorities intersect is crucial to ensure properties (A) and (B), and these properties in turn ensure *uniform agreement*. Since any two Σ-quorums also intersect, it is not hard to see that we can use them to the same effect. That is, instead of waiting for messages from a majority of processes, each process p can wait for messages from all processes in the quorum presently output by Σ at p. It is easy to see that the resulting algorithm, which uses (Ω, Σ), also solves *uniform* consensus, and it does so in *all* environments.[5]

Recall that nonuniform consensus differs from uniform consensus in that only *correct* processes need to

[5] Together with Theorem 5.10, this shows that in all environments $\mathcal{E}$, (Ω, Σ) is the weakest failure detector to solve uniform consensus in $\mathcal{E}$.

agree on the decision value. Similarly, Σ^ν differs from Σ in that only quorums output at *correct* processes are required to intersect. So it may appear that by replacing majorities by Σ^ν-quorums (and, *a fortiori*, by $\Sigma^{\nu+}$-quorums) in the Mostéfaoui–Raynal algorithm, we would get an algorithm that solves *nonuniform* consensus—which is our goal here. Unfortunately, this is not so, as the following scenario shows.

Suppose that a correct process p receives unanimous round k proposals for v from all the processes in its $\Sigma^{\nu+}$-quorum P, and so it decides v, in round k. Since $\Sigma^{\nu+}$-quorums at correct processes intersect, all correct processes receive at least one proposal for v and adopt estimate v at the end of round k. Some *faulty* process q, however, does not receive any proposals for v (because the quorum Q that q uses to collect proposals in round k does not intersect with p's quorum P), and q's estimate at the end of round k is some $v' \neq v$. In phase one of round $k+1$, q sends a leader message containing estimate v', and the failure detector Ω outputs q at all processes. So, every process that completes this phase adopts v' from q as its estimate. It is now easy to extend this scenario so that eventually some correct process decides v', violating nonuniform agreement.

The above scenario shows an example of *contamination*: informally, contamination occurs when a correct process adopts an estimate v' from a faulty process in some round even though some correct process decided $v \neq v'$ in an earlier round. In the above algorithm, a correct process can be contaminated in the two places where it can adopt a new estimate: when it receives a leader message (in the first phase), and when it receives proposals (in the third phase). To prevent such contamination, and ensure nonuniform agreement, our algorithm makes processes more circumspect both about changing their estimates and about deciding.

Changing estimate. In our algorithm, a process p changes its estimate upon receiving a leader message from its current leader q *only* if p does not "distrust" q. We now explain what we mean by "p distrusts q". To do so, we also explain what we mean by "p considers q faulty".

Process p maintains a *quorum history* variable H_p in which it stores all its past quorums, as well as all other processes' quorums of which it is aware. More precisely, H_p is an array indexed by the set of processes, and $H_p[r]$ is a set that contains all the quorums of r that p knows about. Processes learn of the quorums of other processes by including their quorum history variables in messages they exchange.

Suppose p finds a quorum P in $H_p[p]$ and a quorum Q in $H_p[q]$ that do not intersect. By the nonuniform intersection property of $\Sigma^{\nu+}$, p knows that either it is faulty or q is faulty, and so p *considers q to be faulty*. (This is because in nonuniform consensus, it is safe for a process to always consider itself correct.) Note that a correct process never considers another correct process to be faulty, since their $\Sigma^{\nu+}$-quorums always intersect.

Now suppose p finds a quorum R in $H_p[r]$ and a quorum Q in $H_p[q]$ that do not intersect. Process p knows that either r or q is faulty. If p does not consider r to be faulty (by the above definition), then *p distrusts q*. Symmetrically, if p does not consider q to be faulty, then p distrusts r.

Deciding. In our algorithm, a process p that receives unanimous proposals for v from a quorum P in round k decides v *only* if the following two conditions hold: (a) p does not distrust any process in the quorum P; and, (b) p knows that, if it is correct, then by the end of round k, all the correct processes are aware that p has seen P as one of its $\Sigma^{\nu+}$-quorums.

We call (b) the "quorum awareness property". We now describe the mechanism that ensures this property. The first time that a process p uses a quorum P for collecting proposals, it sends to all processes in P a message that it "saw" quorum P. A process q receiving that message inserts P into $H_q[p]$ and sends back to p an acknowledgment that includes q's current round number j. This signifies that by round j, q is aware that p saw quorum P. Process p is allowed to decide v in round k only if (a) it receives unanimous proposals for v from a quorum P none of whose members it distrusts, and (b) every process q in P has acknowledged having inserted P into $H_q[p]$ in a round *strictly less* than p's current round k. The latter condition ensures that every round k proposal message sent by any process in P contains a quorum history H such that $P \in H[p]$. By nonuniform intersection, the quorums of correct processes intersect. So, if p is correct, then the quorums of every correct process c intersect P, and c collects a round k proposal from a process in P. As a result, after c incorporates into its own quorum history all the quorum histories included in the round k proposals it receives, c is aware that p has seen the quorum P, i.e., by the end of round k, $P \in H_c[p]$.

Let us now revisit the contamination scenario described previously, and see how the above rules (on changing estimate and deciding) prevent this contamination. Recall that, in that scenario, a correct process p decides v in round k after receiving unanimous proposals for v from a quorum P, a faulty process q retains estimate $v' \neq v$ in round k after collecting proposals from a quorum Q that does not intersect with P, q sends a leader message containing estimate v' in round $k+1$, at every process Ω outputs q throughout round $k+1$, and so each correct process c adopts estimate v' from q in round $k+1$, i.e., c gets contaminated. With the above two rules about changing estimates and deciding, this contamination does not occur, as we now explain.

Since p decides in round k using quorum P, the quorum awareness property ensures that, by the end of round k, every correct process c is aware that p has seen P as one of his quorums, i.e., $P \in H_c[p]$. Since q saw the quorum Q in round k, it has $Q \in H_q[q]$ by the end of that round. So when c receives a leader message with estimate v' from q in the first phase of the round $k+1$, and c incorporates q's quorum history H_q into its own, it has $Q \in H_c[q]$. So at this point, $P \in H_c[p]$ and $Q \in H_c[q]$, i.e., c is aware that p and q saw two non-intersecting quorums P and Q, respectively. Since c does not consider p to be faulty (because both c and p are correct), by definition, c distrusts q. Thus c does *not* adopts q's estimate v', and it avoids contamination.

In the above, we have focused on a scenario where contamination occurs only one round after a correct process decides. In general, however, contamination can occur several rounds after a correct process decides. The rules that we described above prevent contamination in all cases, and so they ensure (nonuniform) agreement in all the runs of the algorithm.

The reader may have noticed that in our discussion so far we have used only the properties of Σ^{ν}. As will become clear in the proof, the two additional properties of $\Sigma^{\nu+}$ are also necessary for the correctness of $\mathcal{A}_{\text{nuc}}$. At a high level, we can now say that the self-inclusion property (every process is included in all its quorums) implies that a process p never considers itself faulty, and so p distrusts every process that it considers faulty. This, in turn, ensures that in every round p receives no conflicting proposals. The conditional nonintersection property (a quorum that fails to intersect a correct process's quorum contains only faulty processes) implies that every correct process eventually ceases to distrust correct processes—a fact that is important for liveness.

Detailed description and correctness proof

We now describe our algorithm in more detail and then prove its correctness. The algorithm, denoted $\mathcal{A}_{\text{nuc}}$, is shown in Figs. 4 and 5. In the first phase of each asynchronous round of $\mathcal{A}$nuc, each process p sends to all processes its leader message. This message, which is tagged with LEAD and the current round number k_p, contains the estimate x_p and quorum history H_p (line 15). Process p then waits for a (round k_p) leader message from the process output by Ω at p (line 16). Upon receiving it, p incorporates into H_p the quorum history contained in this message (17—see also lines 44–46). If p does not distrust the sender of this leader message, it adopts the estimate that the message contains (line 18). To determine if p distrusts a process q (lines 51–53), p first determines the set F_p of processes that it considers faulty because some of their quorums do not intersect some of its own (line 52). Process p then distrusts q if there is some $r \notin F_p$ such that $H_p[q]$ and $H_p[r]$ contain nonintersecting quorums (line 53).

In the second phase, p sends to all processes its report message. This message, which is tagged with REP and the current round number k_p, contains the current estimate x_p (line 19). Process p then waits for (round k_p) reports from the quorum currently output by $\Sigma^{\nu+}$ (line 20—see also lines 47–50).

In the third phase, p sends to all processes its proposal. This message, which is tagged with PROP and the current round k_p, contains a value and the current quorum history H_p. The proposal's value is v if all the reports that p received in round k_p were for v (line 22); otherwise, it is the special value ? (line 24) Process p then waits for proposals from the quorum Q_p presently output by $\Sigma^{\nu+}$ (line 26), incorporates into its quorum history H_p the quorum histories contained in the proposals that p received from processes in Q_p (line 27), and repeats this until none of the processes in Q_p is distrusted (lines 25–28). If p receives a proposal for a value $v \neq ?$ from *some* process in Q_p, then p adopts v as its estimate (line 29). If p receives unanimous proposals for a value $v \neq ?$ from *all* processes in Q_p, and p is sure that every process q in Q_p has inserted Q_p into $H_q[p]$ in a previous round (in which case $seen_p[Q_p] < k_p$ as we will see below),[6] then p decides v (line 30).

Finally, if this is the first time that p has used the quorum Q_p to collect proposals, then p sends the message (SAW, p, Q_p) to every process q in Q_p (line 32), so that process q may insert Q_p into $H_q[p]$. Process p receives acknowledgments of the form (ACK, q, Q_p, k) (line 39), indicating that q inserted Q_p into $H_q[p]$ by round k (line 36). While p receives such acknowledgments, p keeps track of the maximum round in which they were sent (lines 39–41). When p has received such acknowledgments from *every* process in Q_p, it records the overall maximum in $seen_p[Q_p]$ (line 42).

We now prove the correctness of $\mathcal{A}_{\text{nuc}}$. We assume that function calls of $\mathcal{A}_{\text{nuc}}$ are uninterruptible, i.e., after any process invokes any function in Fig. 5, it does not execute any line *outside* that function's definition until the call terminates. In what follows, we consider an arbitrary admissible run of $\mathcal{A}_{\text{nuc}}$ using $(\Omega, \Sigma^{\nu+})$ in an arbitrary environment $\mathcal{E}$. We begin with a lemma and two

[6] This condition ensures that every correct process c has inserted Q_p into $H_c[p]$ by the end of round k_p, i.e., it ensures the quorum awareness property that we described earlier.

observations concerning H_p (which contains the quorum history of p) and F_p (which contains the set of processes that p considers to be faulty).

Lemma 6.9 *For all processes p and q and any set Q, if at some time $Q \in H_p[q]$, then q previously received quorum Q from its failure detector $\Sigma^{\nu+}$.*

```
CODE FOR EACH PROCESS p:
initialize
  x_p ← value that p proposes
  k_p ← 0
  F_p ← ∅
  for all q ∈ Π do
    H_p[q] ← ∅   /* Variable H_p is shared by the procedure below and those on Fig. 5 */
  for all Q such that Q ⊆ Π do
    sent_p[Q] ← false
    Acks_p[Q] ← ∅
    round_p[Q] ← 0
    seen_p[Q] ← ∞

cobegin
  loop
    k_p ← k_p + 1
    send (LEAD, k_p, x_p, H_p) to all
    repeat q ← Ω_p until received (LEAD, k_p, v, Hist_q) from q
    import_history(Hist_q)
    if ¬ distrusts(q) then  x_p ← v
    send (REP, k_p, x_p) to all
    repeat Q_p ← get_quorum() until received (REP, k_p, −) from all q ∈ Q_p
    if ∃v such that p received (REP, k_p, v) from every q ∈ Q_p then
      send (PROP, k_p, v, H_p) to all
    else
      send (PROP, k_p, ?, H_p) to all
    repeat
      repeat Q_p ← get_quorum() until received (PROP, k_p, −, Hist_q) from all q ∈ Q_p
      for all q ∈ Q_p do import_history(Hist_q)
    until ∀q ∈ Q_p ¬ distrusts(q)
    if ∃v ≠ ? such that p received (PROP, k_p, v, −) from some q ∈ Q_p then  x_p ← v
    if ∃v ≠ ? such that p received (PROP, k_p, v, −) from every q ∈ Q_p and seen_p[Q_p] < k_p then decide(x_p)
    if ¬ sent_p[Q_p] then
      send (SAW, p, Q_p) to all q ∈ Q_p
      sent_p[Q_p] ← true
||
  upon receipt of (SAW, q, Q)
    H_p[q] ← H_p[q] ∪ {Q}
    send (ACK, p, Q, k_p) to q
||
  upon receipt of (ACK, q, Q, k)
    Acks_p[Q] ← Acks_p[Q] ∪ {q}
    round_p[Q] ← max{round_p[Q], k}
    if Acks_p[Q] = Q then seen_p[Q] ← round_p[Q]
coend
```

Fig. 4 Algorithm $\mathcal{A}_{nuc}$ uses $(\Omega, \Sigma^{\nu+})$ to solve nonuniform consensus

```
CODE FOR EACH PROCESS p:
44 procedure import_history(H)
45   for all r ∈ Π do
46     H_p[r] ← H_p[r] ∪ H[r]

47 function get_quorum()
48   Q_p ← Σ_p^{ν+}
49   H_p[p] ← H_p[p] ∪ {Q_p}
50   return Q_p

51 function distrusts(q)
52   F_p ← {q' ∈ Π | ∃Q' ∈ H_p[q'], ∃P ∈ H_p[p] : Q' ∩ P = ∅}
53   return ∃r ∈ Π − F_p, ∃Q ∈ H_p[q], ∃R ∈ H_p[r] : Q ∩ R = ∅
```

Fig. 5 Functions used by $\mathcal{A}_{nuc}$

Proof The proof is by a simple induction that uses the following observations. Initially, $H_p[q]$ is the empty set. In the algorithm, there are three locations where p may insert Q into $H_p[q]$: (1) on line 49, where $p = q$, and p inserts Q into $H_p[q]$ after receiving Q from its failure detector $\Sigma^{\nu+}$, (2) on line 36, after p receives a message (SAW, q, Q) from some process q that previously received Q from $\Sigma^{\nu+}$, and (3) on line 46, where p incorporates into $H_p[q]$ a quorum history $H[r]$ that contains Q. □

Since a process never removes quorums from its quorum history variable,

Observation 6.10 *For all processes p and q and any quorum Q, if $Q \in H_p[q]$ at some time t, then $Q \in H_p[q]$ at all times $t' \geq t$.*

From the previous observation, and the statement that computes the variable F_p (on line 52),

Observation 6.11 *For all processes p and q, if $q \in F_p$ at some time t, then $q \in F_p$ at all times $t' \geq t$.*

Next, we turn our attention to termination. The following lemma implies that no correct process is stuck forever in the loop of lines 25–28.

Lemma 6.12 *There is a time after which, for all correct processes q, every call to distrusts(q) returns false.*

Proof Suppose, by way of contradiction, that there is a correct process q and a process p such that infinitely many calls of p to *distrusts*(q) return *true*. Clearly, p is correct.

Since p makes an infinite number of calls to the function *distrusts*(q), either p executes infinitely many rounds, or p blocks in the repeat loop of lines 25–28. Either way, p calls *get_quorum*() infinitely often. Thus, by the completeness property of $\Sigma^{\nu+}$ and by Observation 6.10, there is a time t after which $H_p[p]$ contains a quorum P consisting of only correct processes.

Consider any call of p to *distrusts*(q) that returns *true* and is made after time t. Let T be the closed time interval whose endpoints are the invocation (line 51) and termination (line 53) of this call. Since the call returns *true*, it must be that at some time $t'' \in T$, $\exists r \notin F_p$, $\exists Q \in H_p[q]$, $\exists R \in H_p[r] : Q \cap R = \emptyset$ (line 53). By our choice of t, at time $t'' > t$, $H_p[p]$ contains the quorum P which consists of only correct processes. Since function calls are uninterruptible and the function *distrusts* does not modify H_p, H_p does not change during T. Therefore, during this entire interval, $R \in H_p[r]$ and $P \in H_p[p]$. The quorum R and the quorum Q output by the correct process q do not intersect, so by the conditional nonintersection property of $\Sigma^{\nu+}$, R contains only faulty processes. Since P contains only correct processes, $R \cap P = \emptyset$. So when p evaluates F_p on line 52, it finds $R \in H_p[r]$, $P \in H_p[p]$, and $R \cap P = \emptyset$, and p inserts r in F_p at some time $t' \in T$.

Clearly, $t' \leq t''$, since p executes line 52 before line 53. So, $r \in F_p$ at time t', and $r \notin F_p$ at time $t'' \geq t'$ — a contradiction to Observation 6.11. □

Lemma 6.13 *Every correct process executes infinitely many rounds.*

Proof Suppose, by way of contradiction, that some correct process executes only a finite number of rounds. Consider the earliest line of the earliest round in which a correct process blocks; let k be that round, and let p be such a process. There are exactly four cases, depending on the place in the algorithm where p blocks.

(1) Line 16: In this case, p waits forever for a message $(\text{LEAD}, k, -, -)$ in round k. Since there is a time after which Ω forever outputs a correct process c at p, there is a time after which process p waits forever for a message $(\text{LEAD}, k, -, -)$ from c in round k. By our definition of p, all correct processes, including c, execute up to line 15 in round k. Thus, c sends a message of the form $(\text{LEAD}, k, -, -)$ to p in round k, which p eventually receives. This contradicts that p is blocked on line 16.

(2) Line 20: In this case, p calls *get_quorum*() infinitely often. Since there are finitely many different quorums, infinitely many of p's calls to *get_quorum*() return Q, for some quorum Q. By completeness of $\Sigma^{\nu+}$, all processes in Q are correct. By definition of p, all correct processes execute line 19 in round k. Therefore, every correct process sends a message $(\text{REP}, k, -)$ to p, and p eventually receives such a message from every process in Q. This contradicts that p is blocked on line 20.

(3) Line 26: An argument similar to that used in case (2) shows that this case cannot occur.

(4) Loop on lines 25–28: In this case, p calls the function *get_quorum*() infinitely often on line 26. Since there are finitely many different quorums, infinitely many of p's calls to *get_quorum*() return Q, for some quorum Q. By completeness of $\Sigma^{\nu+}$, all processes in Q are correct. By Lemma 6.12, for all $q \in Q$, there is a time after which every call of p to *distrusts*(q) returns *false*. Thus, there is a time after which the condition on line 28 is always true. This contradicts that p is blocked in the loop on lines 25–28.

Thus, no correct process blocks forever. □

Lemma 6.14 *There is a round and a value $v \neq$? such that all the processes that start this round do so with the same estimate v.*

Proof From the definition of Ω and Lemma 6.12, there is a time t after which

(1) all faulty processes have crashed;
(2) Ω forever outputs the same correct process c at all correct processes; and,
(3) for all correct processes q, every call to *distrusts*(q) returns *false*.

Let k be the maximum round number of all correct processes at time t, and consider round $k+1$ of correct processes (which exists by the previous lemma). In the first phase of round $k+1$, c sends its estimate x_c, which is some value $v \neq$?, to all processes. By (2) and (3), every correct process waits for and receives the message $(\text{LEAD}, k+1, v, -)$ from c on line 16, gets *false* when it calls *distrusts*(c), and updates its estimate to v on line 18. So all the correct processes have estimate v just before sending their round $k+1$ reports.

From the above and (1), in round $k+1$, only reports for v are sent and received, and only proposals for v are sent and received. Thus, no process changes its estimate to a value other than v in that round. Hence, at the beginning of round $k+2$, the estimate of all correct processes is v. By (1) faulty processes do not begin round $k+2$. Thus all the processes that start round $k+2$ do so with the same estimate v. □

Lemma 6.15 *If all the processes that start some round k do so with the same estimate v, then no process changes its estimate to $v' \neq v$ in any round $k' \geq k$.*

Proof Suppose all the processes that start round k do so with the same estimate v. In round k, only leader messages, reports, and proposals for v are sent and received. Thus, no process changes its estimate to $v' \neq v$ in round k, and so all the processes that start round $k+1$ do so with the same estimate v. The lemma follows by a straightforward induction. □

From the previous two lemmata, there is a round k and a value $v \neq$? such that all processes that start round $k' \geq k$, do so with the same estimate v. This implies:

Corollary 6.16 *There is a round k and a value $v \neq$? such that for every $k' \geq k$, all round k' proposals are for v.*

The following lemma describes the key properties of the quorum awareness mechanism (lines 31–42) mentioned in our informal algorithm description. Part (a) is used for termination and ensures that the condition $seen_p[Q_p] < k_p$ on line 30 is eventually satisfied. Part (b) will be used later for nonuniform agreement.

Lemma 6.17 (a) *Let p be any correct process and P be any quorum of correct processes. If p sends the message (SAW, p, P) to all processes in P, then there is an $\ell \neq \infty$ and a time after which $seen_p[P] = \ell$ forever.*

(b) *For any process p and any quorum P, if at some time $seen_p[P] = \ell$ with $\ell \neq \infty$, then every process $q \in P$ inserted P into $H_q[p]$ in some round $j_q \leq \ell$.*

Proof (a) Suppose that some correct process p sends the message (SAW, p, P) to every process in some quorum $P \subseteq correct(F)$ on line 32. Let q be any process in P.

Since q is correct, it eventually receives (SAW, p, P) on line 35 in some round j_q. So q inserts P into $H_q[p]$ and sends (ACK, q, P, j_q) to p on lines 36 and 37, respectively, in round j_q. Since p is correct, it eventually receives (ACK, q, P, j_q) from q.

Note that p sends (SAW, p, P) to each $q \in P$ only once (see lines 31 and 33), so p receives exactly one $(\text{ACK}, q, P, -)$ back from each $q \in P$. Moreover, since p sends (SAW, p, P) messages only to processes in P, it receives $(\text{ACK}, -, P, -)$ messages only from processes in P.

By inspection of the algorithm (see lines 39–41), it is clear that if P' is the subset of P from which p has received $(\text{ACK}, -, P, -)$ messages, then $Acks_p[P] = P'$ and $round_p[P] = \max\{j_q : q \in P'\}$ (where we adopt the convention that $\max\{\} = 0$). As we showed above, p eventually receives exactly one $(\text{ACK}, -, P, -)$ from each process in P and from no other process. Thus, there is a time after which $Acks_p[P] = P$ and $round_p[P] = \max\{j_q : q \in P\}$ forever. Let $\ell = \max\{j_q : q \in P\}$. From line 42 we see that as soon as $Acks_p[P]$ is assigned its final value P, $seen_p[P]$ is assigned its final value ℓ. This proves part (a) of the lemma.

(b) Let p be any process and P be any quorum. Suppose that at some time $seen_p[P] = \ell \neq \infty$. Since p initializes $seen_p[P]$ to ∞, this means that p assigns ℓ to $seen_p[P]$ on line 42. From our description in part (a), it is clear that this implies that every process $q \in P$ inserted P into $H_q[p]$ in some round $j_q \leq \ell = \max\{j_q : q \in P\}$. This proves part (b) of the lemma. □

Lemma 6.18 (Termination) *Every correct process eventually decides.*

Proof Let p be any correct process. Suppose, by way of contradiction, that p never decides. By Lemma 6.13, p executes infinitely many rounds. Since there are only finitely many different quorums, there is some P such that p executes line 30 infinitely often with $Q_p = P$. From the algorithm, it is clear that p gets the quorum P from $\Sigma_p^{\nu+}$ infinitely often (via *get_quorum*(), on line 26). By the completeness property of $\Sigma^{\nu+}$, P contains only correct processes.

Since p never decides, p finds that the condition of line 30 with $Q_p = P$ is *false* infinitely often. This implies that either

(a) infinitely often, the proposals that p receives from processes in P are not all for the same $v \neq ?$, or
(b) infinitely often, p finds $seen_p[P] < k_p$ is *false*.

Corollary 6.16 contradicts case (a). Since all the processes in P are correct, by Lemma 6.17(a), there is an $\ell \neq \infty$ and a time after which $seen_p[P] = \ell$ (forever). Since p executes infinitely many rounds, there is a time after which $k_p > \ell$. This contradicts case (b). □

Lemma 6.19 (Validity) *If a process decides v, then some process proposes v.*

Proof From line 30 we see that a process p decides its current estimate x_p. Initially, x_p is the value that p proposes. By inspection of the algorithm and a simple induction, it is easy to show that the value of x_p remains a value proposed by one of the processes. □

We now turn our attention to nonuniform agreement.

Lemma 6.20 *For all processes p and at all times, $p \notin F_p$.*

Proof Suppose, by way of contradiction, that at some time $p \in F_p$. At the time when p first adds p to F_p on line 52, there must be two quorums $Q \in H_p[p]$ and $P \in H_p[p]$ such that $Q \cap P = \emptyset$. By the self-inclusion property of $\Sigma^{\nu+}$, every quorum output at p contains p. Thus, $p \in Q \cap P$—a contradiction. □

Lemma 6.21 *For all correct processes p and q, at all times, $q \notin F_p$.*

Proof Let p and q be correct processes. Suppose, by way of contradiction, that at some time $q \in F_p$. When p first adds q to F_p on line 52, it must be that $\exists Q \in H_p[q]$ and $\exists P \in H_p[p]$ such that $Q \cap P = \emptyset$. By Lemma 6.9, Q and P are quorums of $\Sigma^{\nu+}$ output at q and p, respectively. Since q and p are correct processes, by nonuniform intersection of $\Sigma^{\nu+}$, $Q \cap P \neq \emptyset$—a contradiction. □

We say that *process p distrusts q at time t* if and only if, at time t, there is a process r that is not in F_p, such that $H_p[q]$ and $H_p[r]$ contain nonintersecting quorums.

Lemma 6.22 *For all processes p and q, and at all times, if $q \in F_p$ then p distrusts q.*

Proof Suppose that $q \in F_p$ at time t. At the time $t' \leq t$ when p first adds q to F_p (on line 52), there must be two quorums $Q \in H_p[q]$ and $P \in H_p[p]$ such that $Q \cap P = \emptyset$. Since p does not remove quorums from H_p (Observation 6.10), and $p \notin F_p$ at all times (by Lemma 6.20), then at time $t \geq t'$, we have: $p \notin F_p$, $Q \in H_p[q]$, $P \in H_p[p]$, and $Q \cap P = \emptyset$. By definition, p distrusts q at time t. □

We say that P is the quorum that process p *uses to collect round k reports* if and only if P is the value of Q_p when p executes line 21 in round k. Similarly, P is the quorum that process p *uses to collect round k proposals*, if P is the value of Q_p when p exits the repeat-until loop of lines 25–28 in round k (note that P is also the value of Q_p when p executes lines 29 and 30 in round k).

The next lemma shows that $\mathcal{A}_{\text{nuc}}$ has a property similar to property (A) of the Mostéfaoui–Raynal algorithm discussed on page 349.

Lemma 6.23 *Let P be the quorum that some process p uses to collect round k proposals. If p receives round k proposals for $v \neq ?$ and $v' \neq ?$ from processes in P, then $v = v'$.*

Proof Let P be the quorum that p uses to collect round k proposals. Suppose, by way of contradiction, that p receives a round k proposal for $v \neq ?$ from process $q \in P$, and a round k proposal for $v' \neq ?$ such that $v \neq v'$ from process $q' \in P$. Let Q and Q' be the quorums that q and q', respectively, used to collect round k reports. From the algorithm, it is clear that q and q' received unanimous reports for v and v' from all the processes in Q and Q', respectively, on line 20 in round k. Since $v \neq v'$, $Q \cap Q' = \emptyset$.

When q first obtained Q from Σ_q^{v+}, it added Q into $H_q[q]$ (line 49) and never subsequently removed Q from $H_q[q]$. Thus, the proposal sent by q to p in round k contains a quorum history H such that $Q \in H[q]$. Similarly, the proposal sent by q' to p in round k contains a quorum history H' such that $Q' \in H'[q']$. When p receives these proposals, it incorporates the corresponding quorum histories in H_p (line 27). So, before p executes line 28 for the last time in round k, $Q \in H_p[q]$ and $Q' \in H_p[q']$.

Let T (respectively, T') be the closed interval between the time p makes the last call to *distrusts*(q) (respectively *distrusts*(q')) on line 28 of round k and the time that call returns. It is clear from the algorithm that both these calls return *false*. Without loss of generality, assume that T precedes T'. Since the call to *distrusts*(q') during T' returns false, by Lemma 6.22, $q' \notin F_p$ throughout T'. Thus, by Observation 6.11 and the fact that T precedes T', $q' \notin F_p$ throughout T. Furthermore, since $Q \in H_p[q]$ and $Q' \in H_p[q']$ before T, by Observation 6.10, $Q \in H_p[q]$ and $Q' \in H_p[q']$ throughout T. Recalling that $Q \cap Q' = \emptyset$, we conclude that when p executes line 53 during T, it returned *true*. This contradicts the fact that this call to *distrusts*(q) returns *false*. □

In the following, we say that *process p decides v in round k using quorum P* if and only if p decides v on line 30 in round k and P is the quorum that p uses to collect proposals in round k. The next lemma shows that $\mathcal{A}_{nuc}$ has the quorum awareness property discussed during the informal presentation of the algorithm.

Lemma 6.24 *If some process p decides v in round k using quorum P, then every process $q \in P$ that starts round k has $P \in H_q[p]$ at the beginning of round k.*

Proof Assume p decides v in round k using quorum P. Suppose $seen_p[P] = \ell$ when this occurs. By the condition on line 30, $\ell < k$. So, by Lemma 6.17(b), every $q \in P$ inserted P into $H_q[p]$ in some round $j_q \leq \ell$. Since processes do not remove quorums from their quorum history variables (Observation 6.10), every process $q \in P$ that starts round $k > \ell$ has $P \in H_q[p]$ at the beginning of round k. □

We say that process q *intersects* (*quorum*) P *in round* k if and only if the quorum that q uses to collect proposals in round k intersects P. The next lemma shows that $\mathcal{A}_{\text{nuc}}$ has a property analogous to property (B) of the Mostéfaoui–Raynal algorithm discussed on page 349.

Lemma 6.25 *Suppose that some process p decides v in round k using quorum P. For every process q that intersects P in round k,*

(a) *when q completes round k, $x_q = v$ and $P \in H_q[p]$; and,*
(b) *at any time after q completes round k, either $x_q = v$ or $p \in F_q$.*

Proof Suppose a process p decides v in round k using quorum P.

(a) Let q be a process that intersects P in round k; i.e., the quorum Q that q uses to collect its round k proposals intersects P. Consider any process $r \in P \cap Q$. Since p decides v in round k using quorum P, and $r \in P$, we have the following:

- The proposal that r sent to p in round k is for v. So, the proposal that q receives from r in round k is also for v. Thus, by Lemma 6.23, q receives only proposals for v or ? in round k. Therefore, q sets x_q to v on line 29 in round k, and $x_q = v$ at the end of round k.
- By Lemma 6.24, when r starts round k, it has $P \in H_r[p]$. So the proposal that q receives from r on line 26 in round k contains a quorum history H such that $P \in H[p]$. Thus, q adds P into $H_q[p]$ on line 27 in round k, and $P \in H_q[p]$ at the end of round k.

(b) Suppose, by way of contradiction, that there is a process q that intersects P in round k, such that at some time t after q completes round k, $x_q \neq v$ and $p \notin F_q$.

Without loss of generality, let q be the *first* process for which the above hold, i.e., t is as small as possible. By Observation 6.11, a change from $p \in F_q$ to $p \notin F_q$ cannot occur, and so it must be the case that at time t, while q is in some round $k' > k$, q changes its estimate x_q from v to some value $v' \neq v$ on lines 18 or 29. This implies that by time t in round k', either:

- q received a message $m = (\text{LEAD}, k', v', H)$ from some process c on line 16, and the subsequent call to *distrusts*(c) on line 18 returned *false*, or

- q received a message $m = (\textsc{prop}, k', v', H)$ from some process c on line 26, c belongs to the quorum Q that q uses to collect round k' proposals, and when q executed line 28 for the last time in round k' the call to *distrusts*(c) returned *false*.

In either case, let t_c be the time when q invokes the above call to *distrusts*(c) that returns *false*. Note that $t_c \leq t$. There are exactly two cases regarding process c:

(1) *c does not intersect P in round k.* In this case, the message m that c sends in round $k' > k$ carries a quorum history H such that $H[c]$ contains a quorum that does not intersect P. Thus, after q receives m on line 16 or 26 in round k', and then executes line 17 or 27, $H_q[c]$ contains a quorum that does not intersect P. By part (a) of this lemma and the fact that q does not remove quorums from H_q (Observation 6.10), by the time q starts round $k' > k$, $H_q[p]$ contains P. Thus, by time t_c, $H_q[c]$ and $H_q[p]$ contain nonintersecting quorums.

(2) *c intersects P in round k.* Let t' be the time when c sent message m. Clearly, $t' < t$ and at time t', c is in round $k' > k$ and $x_c = v' \neq v$. Thus, by the minimality of t, $p \in F_c$ at time t'. So m carries a quorum history H such that $H[c]$ and $H[p]$ contain nonintersecting quorums (to see this, note the condition under which c puts p in F_c on line 52). Thus, after q receives m on line 16 or 26 in round k', and then executes line 17 or 27, i.e., by time t_c, $H_q[c]$ and $H_q[p]$ contain nonintersecting quorums.

In either case, $H_q[c]$ and $H_q[p]$ contain nonintersecting quorums at time t_c. Furthermore, $p \notin F_q$ at time t_c (this is because $p \notin F_q$ at time $t \geq t_c$). Therefore q's call to *distrusts*(c) at time t_c returns *true*—a contradiction. □

Lemma 6.26 (Nonuniform agreement) *No two correct processes decide differently.*

Proof Let p and q be any two correct processes that decide in some rounds k and k', respectively. Assume, without loss of generality, that $k' \geq k$. Suppose that p decides some value v in round k using quorum P. We now show that the estimate of process q at the end of round k, and at any time thereafter, is v. This implies that when q decides in round $k' \geq k$, it also decides v.

Since p and q are correct, by the nonuniform intersection property of $\Sigma^{\nu+}$, q intersects P in round k. By Lemma 6.25(a), q has $x_q = v$ at the end of round k. Furthermore, by Lemma 6.21, $p \notin F_q$ (always). So, by Lemma 6.25(b), q also has $x_q = v$ at any time after round k. □

By Lemmata 6.18, 6.19 and 6.26, we have:

Theorem 6.27 *For all environments $\mathcal{E}$, algorithm $\mathcal{A}$nuc uses $(\Omega, \Sigma^{\nu+})$ to solve nonuniform consensus in $\mathcal{E}$.*

Theorem 6.28 *For all environments $\mathcal{E}$, there is an algorithm that uses (Ω, Σ^{ν}) to solve nonuniform consensus in $\mathcal{E}$.*

Proof Given failure detectors Ω and Σ^{ν}, we can solve nonuniform consensus as follows. We use $\mathcal{T}_{\Sigma^{\nu} \to \Sigma^{\nu+}}$ (Fig. 3), to transform the given failure detector Σ^{ν} to $\Sigma^{\nu+}$. Concurrently, we run $\mathcal{A}$nuc (Figs. 4 and 5), which solves nonuniform consensus using the failure detectors Ω (provided directly) and $\Sigma^{\nu+}$ (obtained through the variables $\Sigma^{\nu+}$-output of $\mathcal{T}_{\Sigma^{\nu} \to \Sigma^{\nu+}}$). □

By Theorems 5.7 and 6.28, we have:

Theorem 6.29 *For all environments $\mathcal{E}$, (Ω, Σ^{ν}) is the weakest failure detector to solve nonuniform consensus in $\mathcal{E}$.*

7 Comparison of (Ω, Σ^{ν}) and (Ω, Σ)

Let $\mathcal{E}_t$ be the environment that includes all failure patterns in which any set of up to t processes can crash. Formally, $\mathcal{E}_t = \{F \colon |\mathit{faulty}(F)| \leq t\}$.

Note that, in any environment, (Ω, Σ^{ν}) is weaker than (Ω, Σ), since the outputs of (Ω, Σ) immediately satisfy the properties of (Ω, Σ^{ν}). Whether (Ω, Σ^{ν}) is *strictly* weaker than—i.e., weaker than, and not equivalent to—(Ω, Σ) depends on the environment. In environments where at least half of the processes can fail, (Ω, Σ^{ν}) is strictly weaker than (Ω, Σ); in environments where a majority of the processes are correct, the two failure detectors are equivalent. These facts are observed by Delporte et al. [3]; for completeness, we provide direct proofs below.

Theorem 7.1 *For all $t \leq n$, $(\Omega, \Sigma^{\nu}) \equiv_{\mathcal{E}_t} (\Omega, \Sigma)$ if and only if $t < n/2$.*

Proof Clearly, for every environment $\mathcal{E}$, $(\Omega, \Sigma^{\nu}) \preceq_{\mathcal{E}} (\Omega, \Sigma)$. Thus, it suffices to show that $(\Omega, \Sigma^{\nu}) \succeq_{\mathcal{E}_t} (\Omega, \Sigma)$ if and only if $t < n/2$.

[IF] Suppose $t < n/2$. We must prove that $(\Omega, \Sigma^{\nu}) \succeq_{\mathcal{E}_t} (\Omega, \Sigma)$. To do so, it suffices to show that in environment $\mathcal{E}_t$ where $t < n/2$, there is an algorithm that implements Σ "from scratch" —i.e., without using any failure detector. The algorithm proceeds in asynchronous rounds. Initially, each process p outputs Π as its quorum. At the beginning of each round k, p sends a message (k, p) to each process. Process p waits to receive $n - t$ messages of the form $(k, -)$ in round k. It then outputs as its new quorum the set of $n-t$ processes from which it received a message in round k.

Since at least $n - t$ processes are correct, every correct process keeps outputting quorums forever. We now prove that the quorums output satisfy the completeness and intersection properties of Σ. Eventually, all faulty processes crash, and only correct processes exchange messages; therefore, eventually, the quorums of correct processes include only correct processes. Since $t < n/2$, any quorum output by a process contains a majority of processes, and so any two quorums intersect.

[ONLY IF] Suppose $t \geq n/2$. We show that there is no algorithm that transforms (Ω, Σ^ν) to (Ω, Σ) in $\mathcal{E}_t$. In particular, there is no algorithm $\mathcal{T}$ that transforms (Ω, Σ^ν) to Σ in $\mathcal{E}_t$. Suppose, by way of contradiction, that such an algorithm $\mathcal{T}$ exists. Since $t \geq n/2$, we can partition the set of processes Π into two sets A and B, where $|A| \leq t$ and $|B| \leq t$. Consider the following two runs of $\mathcal{T}$.

In the first run R, all processes in B crash before taking a step, and all processes in A are correct. At each process $p \in A$, the output of (Ω, Σ^ν) is always $(\min(A), A)$; at each process $p \in B$, the output of (Ω, Σ^ν) is always $(\min(B), B)$. Note that these outputs satisfy the requirements of (Ω, Σ^ν) in the current failure pattern. Since $\mathcal{T}$ transforms (Ω, Σ^ν) to Σ, at each process $p \in A$, $\mathcal{T}$ eventually outputs some set that consists entirely of correct processes. So, at some time τ and at some process $a \in A$, $\mathcal{T}$ outputs a set $A' \subseteq A$.

In the second run R', (i) all processes in B are correct, but their messages to processes in A are delayed up to time $\tau + 1$, and (ii) all processes in A are faulty, and they crash at time $\tau + 1$. As in run R, at each process $p \in A$, the output of (Ω, Σ^ν) is always $(\min(A), A)$; at each process $p \in B$, the output of (Ω, Σ^ν) is always $(\min(B), B)$. These outputs also satisfy the requirements of (Ω, Σ^ν) in the current failure pattern. Note that up to time $\tau + 1$, processes in A cannot distinguish between runs R and R'. So, at time τ, $\mathcal{T}$ outputs $A' \subseteq A$ at process $a \in A$ exactly as in run R. Now consider any process $b \in B$. Since only processes in B are correct, the completeness property of Σ requires that eventually the transformation algorithm $\mathcal{T}$ outputs some set $B' \subseteq B$ at b.

So, in run R', $\mathcal{T}$ outputs $A' \subseteq A$ at a and $B' \subseteq B$ at b. Since A' and B' are disjoint, this violates the intersection property of Σ — a contradiction of the claim that $\mathcal{T}$ transforms (Ω, Σ^ν) to Σ in $\mathcal{E}_t$. □

Acknowledgements We thank the anonymous referees for suggestions that improved this paper.

Appendix A: Proof of Lemma 2.2

Lemma 2.2 *Let $R = (F, H, I, S, T)$ be a merging of two mergeable finite runs $R_0 = (F, H, I_0, S_0, T_0)$ and $R_1 = (F, H, I_1, S_1, T_1)$ of an algorithm $\mathcal{A}$ using failure detector $\mathcal{D}$ in some environment $\mathcal{E}$. Then*

(a) *R is also a run of $\mathcal{A}$ using $\mathcal{D}$ in $\mathcal{E}$.*
(b) *For each $b \in \{0, 1\}$ and each process $p \in$ participants(S_b), the state of p is the same in $S(I)$ as in $S_b(I_b)$.*

Proof To prove that $R = (F, H, I, S, T)$ is a run of $\mathcal{A}$ using $\mathcal{D}$ in $\mathcal{E}$, we first note that $F \in \mathcal{E}$, $H \in \mathcal{D}(F)$, and I is indeed an initial configuration of $\mathcal{A}$. It now suffices to show that R satisfies properties (1)–(5) of runs. The fact that S and T have the same length (property (2)) is obvious from the definition of R. The fact that in R no process takes a step after it has crashed, and that the failure detector value in each step is consistent with the history H (property (3)) follows from the way R is constructed from R_0 and R_1, and the fact that R_0 and R_1 have this property. The fact that T is nondecreasing (property (4)) is clear from the way T is formed from two nondecreasing sequences T_0 and T_1. To show property (5), we must prove that the times of the steps in the merged run R respect the causal precedence relation. This is true because each of R_0 and R_1 has this property, and no process takes a step in both R_0 and R_1.

It remains to prove that run R satisfies property (1), namely that S is applicable to I. To show this, we use the following notation: for any schedule $\hat{S}$ and $i \in \{0, 1, \ldots, |\hat{S}|\}$, $\hat{S}^i$ is the prefix of $\hat{S}$ that has length i ($\hat{S}^0$ is the empty schedule). Also, for the schedule S of the merged run R, and $b \in \{0, 1\}$, let $f_b(i)$ be the number of steps of S^i that come from S_b. Using a straightforward induction, we can show that for all $i \in \{0, 1, \ldots, |S|\}$:

(i) For all $b \in \{0, 1\}$, the set of messages between processes in *participants*(S_b) (i.e., messages of the form $(p, -, q)$ where $p, q \in$ *participants*(S_b)) in the message buffer of configuration $S^i(I)$ is equal to the set of messages between processes in *participants*(S_b) in the message buffer of configuration $S_b^{f_b(i)}(I_b)$.
(ii) For all $b \in \{0, 1\}$, the state of every process $p \in$ *participants*(S_b) is the same in $S^i(I)$ as in $S_b^{f_b(i)}(I_b)$.

Below we use (i) to show that, for each $i \in \{1, 2, \ldots, |S|\}$, $S[i]$ is applicable to $S^{i-1}(I)$. This proves that S is applicable to I.

Let $S[i] = (p, m, d, \mathcal{A})$. Let $b \in \{0,1\}$ be such that $p \in$ *participants*(S_b) (such a b exists because every step of S is in either S_0 or S_1). Thus, $(p, m, d, \mathcal{A})$ is step $f_b(i)$ of S_b. Therefore, m was sent to p by some process in *participants*(S_b). Since R_b is a run, S_b is applicable to I_b. In particular, step $(p, m, d, \mathcal{A})$ of S_b is applicable to $S_b^{f_b(i)-1}(I_b)$. Note that $f_b(i-1) = f_b(i) - 1$. So, $(p, m, d, \mathcal{A})$ is applicable to $S_b^{f_b(i-1)}(I_b)$. Thus, m is in the message buffer of $S_b^{f_b(i-1)}(I_b)$. By (i), m is in the message buffer of $S^{i-1}(I)$. So, $(p, m, d, \mathcal{A})$ is applicable to $S^{i-1}(I)$, as wanted.

Part (b) of the lemma follows directly from (ii), taking $i = |S|$. □

References

1. Chandra, T.D., Hadzilacos, V., Toueg, S.: The weakest failure detector for solving consensus. JACM **43**(4), 685–722 (1996)
2. Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. JACM **43**(2), 225–267 (1996)
3. Delporte-Gallet, C., Fauconnier, H., Guerraoui, R.: Shared memory vs. message passing. Technical Report IC/2003/77, EPFL (2003)
4. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. JACM **32**(2), 374–382 (1985)
5. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. CACM **21**(7), 558–565 (1978)
6. Mostéfaoui, A., Raynal, M.: Leader-based consensus. Parall Process. Lett. **11**(1), 95–107 (2001)