On Failure Detectors and Type Boosters *

Rachid Guerraoui and Petr Kouznetsov**

Distributed Programming Laboratory, EPFL

Abstract. The power of a set $\mathcal{S}$ of object types can be measured as the maximum number n of processes that can solve consensus using only types in $\mathcal{S}$ and registers. This number, denoted by $h^r_m(\mathcal{S})$, is called the *consensus power* of $\mathcal{S}$. The use of failure detectors can however "boost" the consensus power of types.
This paper addresses the *weakest failure detector type booster* question, which consists in determining the weakest failure detector $\mathcal{D}$ such that, for any set $\mathcal{S}$ of types with $h^r_m(\mathcal{S}) = n$, $h^r_m(\mathcal{S}; \mathcal{D}) = n + 1$.
We consider the failure detector Ω_n (introduced in [17]) which outputs, at each process, a set of at most n processes so that, eventually, all correct processes detect the same set that includes at least one correct process. We prove that Ω_n is the weakest failure detector type booster for *deterministic one-shot* types.
As an interesting corollary of our result, we show that Ω_n is the weakest failure detector to boost the resilience level of $(n-1)$-resilient objects solving consensus.

1 Introduction

Background. Key agreement problems, such as consensus, are not solvable in an asynchronous system where processes communicate solely through registers (i. e., read-write shared memory), as long as one of these processes can fail by crashing [7, 16]. Circumventing this impossibility has sparked off two research trends:

(1) Augmenting the system model with *synchrony* assumptions about relative process speeds and communication delays [6]. Such assumptions could be encapsulated within a *failure detector* abstraction defined with axiomatic properties [5]. In short, a failure detector uses the underlying synchrony assumptions to provide each process with (possibly unreliable) information about the failure pattern, i. e., about the crashes of other processes. This trend led to the identification of the weakest failure detector to solve consensus [4, 14]. This failure detector, denoted by Ω, outputs one process at every process so that, eventually, all correct processes detect the same correct process. The very fact that Ω is the weakest to solve consensus means

* A short version of the paper can be found in the Proceedings of the 17th International Symposium on Distributed Computing (DISC 2003).

** This work is partially supported by the Swiss National Science Foundation (project number 2100-066768).

that any failure detector that solves consensus can emulate the output of Ω. In a sense, Ω encapsulates the minimum *amount of synchrony* needed to solve consensus among any number of processes communicating through registers.

(2) Augmenting the system model with more powerful communication primitives, typically defined as shared object types with sequential specifications [9, 16]. It has been shown, for instance, that consensus can be solved among any number of processes if objects of the compare&swap type can be used [9]. This trend led to define the power of a set of types $\mathcal{S}$, denoted by $h_m^r(\mathcal{S})$ (we follow the standard notations of [13]), as the maximum number n of processes that can solve consensus using only objects of types in $\mathcal{S}$ and registers. For instance, the power of the register type is simply 1 whereas the compare&swap type has power ∞. An interesting fact here is the existence of types with intermediate power, like test-and-set or FIFO queue, which have power 2 [9, 16].

Motivation. At first glance, the two trends appear to be fundamentally different. Failure detectors encapsulate synchrony assumptions and provide information about failure patterns, but cannot however be used to communicate information between processes. On the other hand, conventional object types with sequential specifications can be used for inter-process communication, but they do not provide any information about failures. It is intriguing to figure out whether these trends can be effectively combined [17]. Indeed, in both cases, the goal is to augment the system model with abstractions that are powerful enough to solve consensus, and it is appealing to determine whether abstractions from different trends add up. For instance, we can question ourselves whether the weakest failure detector to solve consensus using registers and queues is strictly weaker than Ω.

A way to effectively combine the two trends is to determine a failure detector hierarchy, $\mathcal{D}_k$, $k \in \mathbb{N}$, such that $\mathcal{D}_k$ would be the weakest failure detector to solve consensus among $k+1$ processes using any set of types $\mathcal{S}$, such that $h_m^r(\mathcal{S}) = k$. $\mathcal{D}_k$ would thus be the weakest failure detector (in the sense of [4]) to boost (in the sense of [10]) the power of $\mathcal{S}$ to higher levels of the consensus hierarchy.

A reasonable candidate for such a failure detector hierarchy was introduced in [17]. This hierarchy is made of weaker variants of Ω, denoted by Ω_k, $k \in \mathbb{N}$, where Ω_k is a failure detector that outputs, at each process, a set of processes so that all correct processes eventually detect the same set of at most k processes that includes at least one correct process. Clearly, Ω_1 is Ω. It was shown in [17] that Ω_n is sufficient to solve $(n+1)$-process consensus using *any* set of types $\mathcal{S}$, such that $h_m^r(\mathcal{S}) = n$. It was also conjectured in [17] that Ω_n is the weakest failure detector to boost the power of $\mathcal{S}$ to the level $n+1$ of the consensus hierarchy. As pointed out in [17], the proof of this conjecture appears to be challenging. The motivation of this work was to take up that challenge.

Contribution. In this paper, we consider deterministic one-shot types [10]. Although these restrict every process to invoke at most one operation on each ob-

ject, (a) they exhibit complex behavior with respect to the weakest failure detector type booster question (in the parlance of the *robustness* problem [2,10,13,15]) and, as we will explain below, (b) they allow a precise answer to the the weakest failure detector resilience booster question (in the parlance of the *resilience vs. wait-freedom* question [1,3]).

Our result can be viewed as a generalization of the fundamental result of [4], and more precisely its extension to the shared memory model [14]. Indeed, we prove that Ω_n is the weakest failure detector that boosts the power of a collection of deterministic one-shot types from consensus number n to $n+1$.

Proving our result comes down to showing that any algorithm that solves $(n+1)$-process consensus, using any failure detector and any set of deterministic one-shot types $\mathcal{S}$, such that $h_m^r(\mathcal{S}) \leq n$, can be used to emulate the output of Ω_n. A major difficulty in our case, with respect to the proofs of [4,14], is that the consensus algorithm uses not only registers but also other objects. We cannot rely on any information about the operations through which these objects can be accessed. The only information that we can rely on is that these objects instantiate deterministic one-shot types that collectively have consensus power at most n.

As an interesting corollary of our result, we show that Ω_n is actually the weakest failure detector to boost the *resilience* of a set of objects solving consensus from level $n-1$ to level n. We show that any algorithm that solves wait-free $(n+1)$-process consensus using a failure detector, registers and any set of $(n-1)$-resilient objects of any (not necessarily one-shot deterministic) type, can be used to emulate the output of Ω_n. On the other hand, there is an algorithm that implements $(n+1)$-process consensus out of registers and $(n-1)$-resilient objects using Ω_n. Thus, Ω_n encapsulates the exact amount of synchrony needed to circumvent the resilience boosting impossibility of [1,3].

Roadmap. Section 2 presents the system model. Section 3 presents the technical details necessary for our result. Section 4 states our main result. Section 5 applies our result to boost the resilience of a set of objects.

2 Model

Our model of processes communicating through shared objects is based on that of [12,13] and our notion of failure detectors follows from [4,14]. Below we recall what is substantial to show our result.

We consider a system Π of $n+1$ asynchronous processes $p_0, \ldots, p_n$ $(n \geq 1)$ that communicate using shared objects. The processes might fail by crashing, i. e. stop executing their steps. A process that never crashes is sait to be *correct.* A process that is not correct is said to be *faulty.*

Objects and types. Let $\mathbb{N}$ denote the set of natural numbers and, for every $k \in \mathbb{N}$, $\mathbb{N}_k = \{0, \ldots, k-1\}$. An *object* is a data structure that can be accessed concurrently by the processes. Every object is an instance of a *type* which is defined

by a tuple (Q, O, n_p, R, δ). Here Q is a set of *states*, O is a set of *operations*, n_p is a positive integer denoting the number of *ports* (used as an interface between processes and objects), R is a set of *responses*, and δ is a relation known as the *sequential specification* of the type that carries each state, operation and port number to a set of response and state pairs. We assume that objects are *deterministic*: the sequential specification is a function $\delta : Q \times O \times \mathbb{N}_{n_p} \to Q \times R$. A type is said to be *k-ported* if $n_p = k$.

We consider here *linearizable* [11] objects: operations on the objects must appear in one-at-a-time order consistent with their real time order. Unless otherwise stated, we assume that the objects are *wait-free* [9]: any process completes any operation in a finite number of steps, regardless of delays or failures of other processes.

A process accesses objects by invoking operations on the ports of the objects. A process can use only one port of each object. Each port of a *one-shot* type can be used only once by a unique process. The *binding scheme* that defines how a process determines the port to access is not important for our result.

Consensus. The (binary) *k-process consensus* problem [7] consists for k processes to decide on some final values (0 or 1) based on their initial proposed values in such a way that: (*agreement*) no two processes decide on different values,[1] (*validity*) every decided value is a proposed value, and (*termination*) every correct process eventually decides.

To prove our result, we also use a restricted form of consensus, *k-process team consensus* [18]. This variant of consensus ensures *agreement* among k processes only if the input values satisfy certain conditions. More precisely, assume that there exists a (known a priori) partition of k processes into two non-empty sets (teams). A k-process team consensus algorithm guarantees agreement if all processes on the same team have the same input value. Obviously, k-process team consensus can be solved whenever k-process consensus can be solved. Surprisingly, the converse is also true [18]:

Lemma 1. *Let $\mathcal{S}$ be any set of types. If $\mathcal{S}$ solves k-process team consensus, then $\mathcal{S}$ also solves k-process consensus.*

Proof. (Sketch) We proceed by induction on k. For any two processes, team consensus is consensus. Assume that, (1) for some $k > 2$, $\mathcal{S}$ can solve k-process team consensus (with teams A and B), and (2) for any set of types $\mathcal{S}'$ and $2 \leq m < k$, if $\mathcal{S}'$ solves m-process team consensus, then $\mathcal{S}'$ also solves m-process consensus. Since only wait-free are considered, (1) implies that, for any $2 \leq m < k$, $\mathcal{S}$ can solve m-process team consensus. Now (2) implies that, for any $2 \leq m < k$, $\mathcal{S}$ also solves m-process consensus. Thus, the two teams A and B ($|A \cup B| = k$, $A \cap B = \emptyset$), can use $|A|$-process consensus and $|B|$-process consensus, respectively to agree on the team input values (A and B are non-empty, thus, $|A| < k$ and $|B| < k$). Once the team input value is known, the

[1] In fact, our weakest failure detection result holds even for the *non-uniform* variant of consensus, where we require only that no two *correct* processes decide differently [8].

processes run k-process team consensus (with teams A and B). Since all processes in the same team propose the same value, the properties of k-process consensus are satisfied. □

The *consensus power* [9,13] of a set of types $\mathcal{S}$, denoted by $h^r_m(\mathcal{S})$, is the largest k, such that k-process consensus can be solved using objects of types in $\mathcal{S}\cup$register, or ∞ if no such largest k exists.

Failure detectors. To simplify the presentation of our model, we assume the existence of a discrete global clock. This is a fictional device: the processes have no direct access to it. We take the range $\mathbb{T}$ of the clock's ticks to be the set of natural numbers and 0 ($\mathbb{T} = \{0\} \cup \mathbb{N}$). A *failure pattern* F is a function from the global time range $\mathbb{T}$ to 2^{Π}, where $F(\tau)$ denotes the set of processes that have crashed by time $\tau \in \mathbb{T}$. Processes do not recover after the crash: $\forall \tau \in \mathbb{T} : F(\tau) \subseteq F(\tau + 1)$. We define $correct(F) = \Pi - \cup_{\tau \in \mathbb{T}} F(\tau)$ to be the set of *correct* (in F) processes. A process that is not correct is said to be *faulty*.

A *failure detector* $\mathcal{D}$ is defined as a map of each failure pattern F (i. e., which processes crash at what times) to a set of *failure detector histories* $\mathcal{D}(F)$ (i. e., what each process *knows* about failures at what time). Any failure detector $\mathcal{D}$ has a range $\mathcal{R}_{\mathcal{D}}$ so that, for any F, any history $H \in \mathcal{D}(F)$ is a function from $\Pi \times \mathbb{T}$ to $\mathcal{R}_{\mathcal{D}}$ ($H(p_i, \tau)$ is the output of the failure detector module of p_i at time τ). Note that the output of a failure detector depends only on the failure pattern: it cannot give any information on the state of processes or shared objects.

If an asynchronous system with registers is augmented with the failure detector Ω, which eventually permanently outputs the identifier of the same correct process at all correct processes, then consensus is solvable for any number of processes [14]. Moreover, it was shown that the output of Ω can be emulated by any consensus algorithm using a failure detector and registers [14]. In a sense we recall below, Ω is said to be the *weakest* failure detector to solve consensus with registers.

Algorithms. An *algorithm A using a failure detector $\mathcal{D}$* is a collection of $n+1$ deterministic automata, one for each process in the system. Computation proceeds in atomic *steps* of A. In each step, a process (1) invokes an operation on a shared object and receives a response,[2] *or* queries its failure detector module of $\mathcal{D}$, and (2) updates its local state according to the current state, the response from the shared object *or* the value output by the failure detector. A step s is defined by the triple (p_i, o, v) where p_i is the identity of the process that takes the step, o is either an operation (on a register or an object of a one-shot deterministic type) invoked by p_i during the step or *query*, and v is the response of the invoked operation or, if $o = query$, the failure detector value seen by p_i during the step. If o is an operation on a shared object X, we say that the step s *acesses* X. Otherwise, if $o = query$, we say that the steps s is a *query step*.

[2] Our objects are linearizable [11], so any execution can be viewed as a sequence of atomic invocation-response pairs.

A *configuration* defines the current state of each process and each object in the system. A step $s = (p_i, o, v)$ of an algorithm A is *applicable to a configuration* C if and only if o is the next operation of p_i defined by A for C. An *execution* e of an algorithm A is a (finite or infinite) sequence of steps of A. ($e_\perp$ denotes an empty execution.) An execution $e = s_1, s_2, \ldots$ is *applicable to a configuration* C if and only if (a) $e = e_\perp$, or (b) s_1 is applicable to C, s_2 is applicable to $s_1(C)$, etc. Given an execution e applicable to a configuration C, $e(C)$ denotes the configuration resulting from applying e to C.

Reducibility. We say that a failure detector $\mathcal{D}$ is *weaker* than a failure detector $\mathcal{D}'$, we write also $\mathcal{D} \preceq \mathcal{D}'$, if there exists an algorithm $T_{\mathcal{D}' \to \mathcal{D}}$ (it is called a *reduction algorithm*) that, for any failure pattern, can emulate the output of $\mathcal{D}$ using *only* $\mathcal{D}'$ and registers. We say that $\mathcal{D}$ is *strictly weaker* than $\mathcal{D}'$, we write also $\mathcal{D} \prec \mathcal{D}'$, if $\mathcal{D} \preceq \mathcal{D}'$, but $\mathcal{D}' \not\preceq \mathcal{D}$.

Hierarchy of Ω_k. In this paper, we focus on the hierarchy of failure detectors Ω_k introduced in [17]. For any $k \in \mathbb{N}$, the output of Ω_k is a set of *at most k* processes so that, eventually, the same set is output at all correct processes and this set includes at least one correct process. One can easily see that Ω_1 is Ω [4]. It was furthermore shown in [17] that, for any $1 \leq k \leq n$, (a) $\Omega_{k+1} \prec \Omega_k$ and, (b) for any set of types $\mathcal{S}$, such that $h_m^r(\mathcal{S}) = k$, $\mathcal{S}$ and Ω_k (shared by n processes) can solve n-process consensus.

3 The Proof Technique

In this section, we introduce some necessary technical details of our proof that Ω_n is necessary to boost the power of any set $\mathcal{S}$ of deterministic one-shot types from level n to level $n+1$. In particular, we recall and generalize the notions of DAG, decision gadget and deciding process [4, 14].

An outline of the proof. Let $\mathsf{Cons}_{\mathcal{D}}$ be any consensus algorithm that solves $(n+1)$-process consensus using registers, a failure detector $\mathcal{D}$, and objects of types in $\mathcal{S}$, for any set $\mathcal{S}$ of deterministic one-shot types, such that $h_m^r(\mathcal{S}) \leq n$. Our goal is to define a reduction algorithm $T_{\mathcal{D} \to \Omega_n}$ that emulates the output of Ω_n out of $\mathcal{D}$. $T_{\mathcal{D} \to \Omega_n}$ should have all correct processes agree eventually on the same set of at most n processes that includes at least one correct process.

The principle of the reduction algorithm $T_{\mathcal{D} \to \Omega_n}$ is the following. Processes periodically query their failure detector modules of $\mathcal{D}$ and exchange the values returned using read-write memory. As a result, each process p_i maintains an ever growing *directed acyclic graph (DAG)*, denoted by G_i, that captures a sample of the failure detector history output by $\mathcal{D}$. This information allows p_i to simulate *locally*, for any initial configuration I, a number of finite executions of the $\mathsf{Cons}_{\mathcal{D}}$ algorithm and build an ever growing *simulation tree*, denoted by Υ_i^I. Since registers provide reliable (though asynchronous) communication, all such Υ_i^I converge to the same infinite simulation tree Υ^I. It turns out, that, for some

initial configuration I, Υ^I has a finite subtree γ, called a *decision gadget*, that provides sufficient information to detect a set of at most n processes, called the *deciding set* of γ, that includes at least one correct process. Thus, eventually, the correct processes detect the gadget and agree on its deciding set which is sufficient to emulate Ω_n.

DAGs. Let F be any failure pattern, H be any failure detector history in $\mathcal{D}(F)$ and I be any initial configuration of $\mathsf{Cons}_{\mathcal{D}}$. Let G be an infinite *directed acyclic graph (DAG)* defined by the set of vertexes $\mathcal{V}(G)$ and a set of directed edges $\mathcal{E}(G)$ of the form $v \rightarrow v'$, where $v \in \mathcal{V}(G)$ and $v' \in \mathcal{V}(G)$, with the following properties:

(1) The vertexes of G are of the form $[p_i, d, k]$ where $p_i \in \Pi$, $d \in \mathcal{R}_{\mathcal{D}}$ and $k \in \mathbb{N}$. There is a mapping $f : \mathcal{V}(G) \rightarrow \mathbb{T}$ that associates a time with each vertex of G, such that:
   (a) For any $v = [p_i, d, k] \in \mathcal{V}(G)$, $p_i \notin F(f(v))$ and $d = H(p_i, f(v))$.
   (b) For any edge $v \rightarrow v' \in \mathcal{E}(G)$, $f(v) < f(v')$.
(2) If $[p_i, d, k] \in \mathcal{V}(G)$,$[p_i, d', k'] \in \mathcal{V}(G)$ and $k < k'$ then $[p_i, d, k] \rightarrow [p_i, d', k'] \in \mathcal{E}(G)$.
(3) G is transitively closed.
(4) Let $U \subseteq \mathcal{V}(G)$ be a finite set of vertexes and p_i be any correct process in F. There is $d \in \mathcal{R}_{\mathcal{D}}$ and $k \in \mathbb{N}$, such that for every vertex $v \in \mathcal{V}(G)$, $v \rightarrow [p_i, d, k]$ is an edge of G.

Informally, G stores a sample of $\mathcal{D}$'s output at different processes and some temporal relationships between them: an edge $[p_i, d, k] \rightarrow [p_j, d', k'] \in \mathcal{E}(G)$ can be interpreted as "p_i saw failure detector value d (in its k-th query) *before* p_j saw failure detector value d' (in its k'-th query)".

Simulation trees. A path $g = [q_1, d_1, k_1], [q_2, d_2, k_2], \ldots$ in G and an initial configuration I of $\mathsf{Cons}_{\mathcal{D}}$ *induce* a unique execution $e = (q_1, o_1, u_1), (q_2, o_2, u_2), \ldots$ of $\mathsf{Cons}_{\mathcal{D}}$ applicable to I, such that $u_i = d_i$ whenever $o_i = \mathit{query}$. The set of all executions of $\mathsf{Cons}_{\mathcal{D}}$ induced by G and I implies a tree Υ^I, called the *simulation tree induced by G and I*, defined as follows. The set of vertexes of Υ^I is the set of *finite* executions e that are induced by G and I. The root of Υ^I is an empty execution $e_\perp$. There is an edge from a vertex e to a vertex e' if and only if $e' = e \cdot s$ for a step s. Thus, for each (finite or infinite) path of Υ^I, there is a unique execution $e = s_1, s_2, \ldots$.

Any path in the tree in which every correct process takes an infinite number of steps is a run of $\mathsf{Cons}_{\mathcal{D}}$. Thus, eventually every correct process decides in the path. As a result, every vertex of Υ^I has a descendant in the tree in which every correct process decides (Lemma 6.2.6 of [4]). We assign a set of tags to each vertex of Υ^I. Vertex C of Υ^I gets tag u if and only if it has a descendant C' such that some correct process has decided u in $C'(I)$. according to the decisions taken by their descendants. If the only decision taken by descendants of a vertex is $u \in \{0, 1\}$, the vertex is called *u-valent.* A 0-valent or 1-valent vertex is called *univalent.* A vertex is called *bivalent* if it has both tags.

A tree Υ^I is called u-valent (bivalent) if $e_\perp$ is u-valent (bivalent) in Υ^I. For a univalent vertex C of Υ^I, $val(C)$ denotes the valence of C.

Decision gadgets. From now on, we assume that processes communicate using shared objects of types in $\{\mathsf{register}\} \cup \mathcal{S}$ where $\mathcal{S}$ contains only deterministic *one-shot* types and $h_m^r(\mathcal{S}) \leq n$. Following [4], we introduce the notion of a *decision gadget.*[3] A decision gadget γ is a finite subtree of Υ^I rooted at $e_\perp$ that includes a vertex C (called the *pivot* of the gadget), such that one of the following conditions is satisfied:

(*fork*) There are two steps s_i and s_i' of a process p_i, such that $s_i(C)$ and $s_i'(C)$ are two leaves of γ of opposite valence.

(*hook*) There is a step s_i of a process p_i and step s_j of a process p_j $(i \neq j)$, such that $s_i(s_j(C))$ and $s_i(C)$ are two leaves of γ of opposite valence, and s_i and s_j do not access the same object of a type in $\mathcal{S}$.

(*rake*) There is a set $U \subseteq \Pi$, $|U| \geq 2$, an object X of a type in $\mathcal{S}$, such that, for any $p_i \in U$, any step s_i of p_i applicable to C (w.r.t. $\mathsf{Cons}_\mathcal{D}$) accesses X (U is called the *participating set of* γ). Let E be the set of all sequences of steps in $\{s_i | p_i \in U\}$ in which every process in U takes *at most* one step, such that $\forall e \in E$, $e(C) \in \Upsilon^I$ and, for any e and e' in E, e is not a prefix of e' (E is called the *execution set* of γ). Then γ, E, C and U satisfy the following conditions:
   (i) C' is a leaf of γ if and only if $\exists e \in E$: $C' = e(C)$.
   (ii) No process $p_j \in \Pi - U$ ever accesses X in any descendant of C in Υ^I.
   (iii) If E includes all sequences of steps in $\{s_i | p_i \in U\}$ in which every process takes *exactly* one step, then every leaf of γ is univalent, and γ has at least one 0-valent leaf and at least one 1-valent leaf.

Let γ be a rake. If the condition of item (iii) above is satisfied, we say that γ is *complete*. Otherwise, γ is said to be *incomplete*.

Examples of decision gadgets are depicted in Figure 1.

The following lemma is the key result of our proof. Note that the lemma uses our assumption that types in $\mathcal{S}$ are deterministic.

Lemma 2. *Let γ be a complete rake with a pivot C, a participating set U and an execution set E, such that $|U| = n + 1$ and, for any executions e and e' in E that begin with the same step s_j, $val(e(C)) = val(e'(C))$. There exist a process p_j and two executions e_0 and e_1 in E, such that (a) $val(e_0(C)) \neq val(e_1(C))$, and (b) p_j has the same state in $e_0(C)$ and $e_1(C)$.*

Proof. For each execution $e \in E$, we associate the configuration $e(C)$ with a vertex of a graph, denoted by $\mathcal{K}$. Two vertexes of $\mathcal{K}$ corresponding to the configurations $e(C)$ and $e'(C)$ are connected with an edge if and only if at least one process p_j has the same state in $e(C)$ and $e'(C)$.
Claim 1: $\mathcal{K}$ is connected.

[3] We slightly modify here the definition of a hook given in [4] and introduce a new notion of a rake.

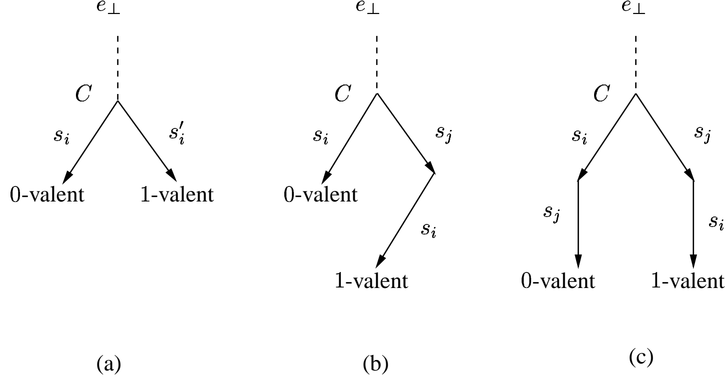

Fig. 1. Examples of decision gadgets: (a) a fork with $s_i = (p_i, query, d)$ and $s'_i = (p_i, query, d')$, (b) a hook where s_i and s_j do not access the same object of a type in $\mathcal{S}$, (c) a complete rake with the participating set $U = \{p_i, p_j\}$ and the execution set $E = \{s_i \cdot s_j, s_j \cdot s_i\}$, where s_i and s_j access the same object X of a type in $\mathcal{S}$.

Proof of Claim 1: By contradiction, assume that $\mathcal{K}$ is not connected, i. e., consists of two or more disconnected components.

Let $e(C)$ and $e'(C)$ be any two vertexes of $\mathcal{K}$, such that the corresponding executions e and e' begin with the step s_i of the same process p_i. Since p_i takes exactly one step in both e and e' and all types considered are deterministic, p_i has the same state in $e(C)$ and $e'(C)$. Thus, the two vertexes belong to the same component of $\mathcal{K}$.

Let $\mathcal{K}_1$ be one of the components of $\mathcal{K}$. We divide the system into two teams Π_1 and Π_2. Team Π_1 consists of all processes p_i, such that any execution that begins with the step s_i of p_i is in $\mathcal{K}_1$. Team Π_2 consists of all other processes. Since $\mathcal{K}$ consists of at least two disconnected components, Π_1 and Π_2 are non-empty. We show now that $\mathcal{S}$ and two registers can solve $(n+1)$-process team consensus for teams Π_1 and Π_2.

Let X be the object of a type in $\mathcal{S}$ accesses by each step in $\{s_j | p_j \in U\}$. We initialize X to its state in C.[4] Every process p_i writes its input value into its team's register and then executes the step s_i of $\mathsf{Cons}_{\mathcal{D}}$ defined for C. By construction of $\mathcal{K}$, the resulting state of p_i can belong to a vertex of exactly one component of $\mathcal{K}$. If the state of p_i corresponds to a vertex in $\mathcal{K}_1$, then p_i outputs the value of Π_1's register, otherwise, p_i outputs the value of Π_2's register. As a result, processes agree on the component to which the resulting state of the system belongs.

Clearly, every correct process eventually decides on some proposed value. Assume now that all processes on the same team (Π_1 or Π_2) propose the same value. Since the processes always agree on the component in $\mathcal{K}$, and no two

[4] The possibility to initialize objects cannot increase their consensus power [2].

different values are ever written into the teams' registers, no two processes decide on different values.

Thus, $\mathcal{S} \cup \{\mathsf{register}\}$ solve $(n+1)$-process team consensus. By Lemma 1, $\mathcal{S} \cup \{\mathsf{register}\}$ solve $(n+1)$-process consensus - a contradiction with the assumption that $h_m^r(\mathcal{S}) \leq n$. Thus, $\mathcal{K}$ is connected. *(End of proof of Claim 1)*

Now we color each vertex $e(C)$ of $\mathcal{K}$ with $val(e(C))$. Since γ is complete, there are two executions e and e' in E of opposite valence, and for every e in E, $e(C)$ is univalent. That is, each vertex of $\mathcal{K}$ has exactly one color (0 or 1) and, for each $u \in \{0,1\}$, there is at least vertex in $\mathcal{K}$ colored by u. Since $\mathcal{K}$ is connected, $\mathcal{K}$ includes at least two vertexes of different colors, $e(C)$ and $e'(C)$, connected with an edge. By construction of $\mathcal{K}$, there is a process p_j that has the same state in $e(C)$ and $e'(C)$ and $val(e(C)) \neq val(e'(C))$. □

The following result is a generalization of Lemma 6.4.1 of [4]:

Lemma 3. *Any bivalent simulation tree Υ^I has a decision gadget.*

Proof. (by construction) By Lemma 6.4.1 (more precisely, Claim 6.4.2) of [4], for any bivalent tree Υ^I, there exists an algorithm to identify a bivalent configuration C in Υ^I and a correct process p_i, such that, for any descendant C' of p_i and any step s_i of p_i applicable to C', $s_i(C')$ is monovalent. Moreover, C satisfies one of the following conditions:

(1) There is a step s_i' of p_i, such that $s_i(C)$ and $s_i'(C')$ are vertexes of Υ^I of opposite valence. That is, a fork is identified.
(2) There is a step s_j of a process p_j $(i \neq j)$, such that $s_i(C)$ and $s_i(s_j(C))$ are vertexes of Υ^I of opposite valence, ands_i and s_j do not access the same object of a type in $\mathcal{S}$. That is, a hook is identified.
(3) There are two steps, s_i of a process p_i and s_j of a process p_j, $(i \neq j)$, such that $s_i(C)$ and $s_i(s_j(C))$ are vertexes of Υ^I of opposite valence, and s_i and s_j access the same object X of a type in $\mathcal{S}$.
Assume that there exist a process p_k and a finite execution e_k in Υ^I, applicable C, that consists of p_k's steps only, such that no step of e_k accesses X and any step of p_k applicable to $e_k(C)$ accesses X. Let U be the set of all such processes p_k and e_U be the concatenation of all such executions e_k, such that $e_U(s_i(C))$ and $e_U(s_i(s_j(C))$ belong to Υ^I. Clearly, $p_i \in U$ and $p_j \in U$ with $e_i = e_j = e_\perp$. Consider the set E of all sequences of steps of processes in U, such that in any $e \in E$ every process in U takes *at most* one step, $\forall e \in E$, $e(C) \in \Upsilon^I$, and, for any e and e' in E, e is not a prefix of e'. By construction, no step of a process in $\Pi - U$ accesses X in any descendant of $e_U(C)$. That is, properties (i) and (ii) of a rake with a participating set U, a set of executions E and a pivot $e_U(C)$ are satisfied
Assume that E includes all sequence of steps in which every process in U takes exactly one step. The configurations $e_U(s_i(C))$ and $s_i(e_U(C)$ are identical, as well as $e_U(s_i(s_j(C)))$ and $s_i(s_j((e_U(C))$, thus the set of vertexes $E(e_U(C))$ includes both 0-valent and 1-valent vertexes. That is property (iii) of a rake is also satisfied.

That is, we identified a rake with a participating set U, a set of executions E and a pivot $e_U(C)$.

As a result, we obtained an algorithm to identify a decision gadget in any bivalent simulation tree Υ^I. □

Deciding sets. Instead of the notion of a *deciding process* given in [4], we introduce the notion of a *deciding set* $V \subset \Pi$. The deciding set V of a decision gadget γ is computed as follows:

(1) Let γ be a fork defined by steps s_i and s'_i. Then $V = \{p_i\}$.
(2) Let γ be a hook defined by steps s_i and s_j. Since, by definition, s_i and s_j do not access the same object of a type in $\mathcal{S}$, there are the following cases to consider:
  (2a) s_j is a query step or s_j reads a register. Then $V = \{p_j\}$.
  (2b) s_j writes into a register, or s_j accesses an object X of a type in $\mathcal{S}$ and s_i does not access X. Then $V = \Pi - \{p_j\}$.
(3) Let γ be a rake defined by a pivot C, a participating set U and an executions set E and a pivot C. The following cases are possible:
  (3a) γ is incomplete, i. e., there is an execution $e \in E$ and a process $p_j \in U$, such that p_j takes no steps in e. Then $V = \Pi - \{p_j\}$.
  (3b) γ is complete and $|U| \leq n$. Then $V = U$.
  (3c) γ is complete, $|U| = n + 1$, and there is a process $p_j \in U$ such that, for some e and e' in E that begin with s_j, $e(C)$ and $e'(C)$ have different valences. Then $V = \Pi - \{p_j\}$.
  (3d) γ is complete, $|U| = n+1$, and for any e and e' in E that begin with the same step s_j, $e(C)$ and $e'(C)$ have the same valence. Lemma 2 guarantees that some process p_j "mixes" two configurations of opposite valence, i. e., there exist two executions e and e' in E, such that p_j has the same state in $e(C)$ and $e'(C)$, and $val(e(C)) \neq val(e'(C))$. Then $V = \Pi - \{p_j\}$.

By construction, in each case, V is a set of at most n processes.

Lemma 4. *The deciding set of a decision gadget contains at least one correct process.*

Proof. The following cases are possible:

(1) Let γ be a fork defined by steps s_i and s'_i. Then $V = \{p_i\}$. Since we consider here deterministic objects and deterministic algorithms, s_i and s'_i can only be query steps (otherwise, $s_i(C) = s'_i(C)$). Thus, the only difference between $s_i(C)$ and $s'_i(C)$ consists in the local state of p_i, more precisely, the failure detector values seen by p_i in s_i and s'_i. Thus, $V = \{p_i\}$ includes exactly one correct process (Lemma 6.5.3 of [4]).
(2) Let γ be a hook defined by steps s_i and s_j. Since, by definition, s_j and s_j do not access the same object of a type in $\mathcal{S}$, there are two following cases to consider:

(2a) s_j is a query step or s_j reads a register. As in case (1), $s_i(C)$ and $s_i(s_j(C))$ differ in the local state of p_j only. Then $V = \{p_j\}$ includes exactly one correct process (Lemma 6.5.3 of [4]).

(2b) s_j writes into a register, or s_j accesses an object X of a type in $\mathcal{S}$ and s_i does not access X. Then $V = \Pi - \{p_j\}$. If Υ^I includes the vertex $s_j(s_i(C))$, then the configurations $s_i(s_j(C))$ and $s_j(s_i(C))$ are identical and cannot have opposite valence. Hence p_j does not take any step applicable to $s_i(C)$. Thus, p_j is faulty, and $V = \Pi - \{p_j\}$ contains at least one correct process.

(3) Let γ be a rake defined by a participating set U, a set of executions E and a pivot C. The following cases are possible:

(3a) γ is incomplete, i.e., there is a process $p_j \in U$ that does not take a step applicable to some descendant of C'. Thus, p_j is faulty, and $V = \Pi - \{p_j\}$ contains at least one correct process.

(3b) γ is complete and $|U| \leq n$. Then U includes at least one correct process to help processes in $\Pi - U$ distinguish any two vertexes of opposite valence, $e_0(C)$ and $e_1(C)$, such that $e_0 \in E$ and $e_1 \in E$. Thus, $V = U$ contains at least one correct process.

(3c) γ is complete, $|U| = n + 1$ and there is a process $p_j \in U$ such that, for some e and e' in E that begin with s_j, $e(C)$ and $e'(C)$ have different valences. Since X is a one-shot object and p_j has taken one step on X in each of $e(C)$ and $e'(C)$, p_j is not able to distinguish $e_0(C)$ and $e_1(C)$ in any solo execution. Thus, p_j is not the only correct process, and $V = \Pi - \{p_j\}$ contains at least one correct process.

(3d) E is complete, $|U| = n + 1$ and for any two executions e and e' in E that begin with a step of the same process p_j, $e(C)$ and $e'(C)$ have the same valence. By Lemma 2, there is a "confused" process p_j that has the same state in some two execution e_0 and e_1, such that $e_0(C)$ and $e_1(C)$ are two vertexes of γ of opposite valence. Since X is a one-shot object and p_j has already taken one step on X in $e_0(C)$ and $e_1(C)$, p_j is not able to distinguish $e_0(C)$ and $e_1(C)$ in any solo execution. Thus, p_j is not the only correct process, and $V = \Pi - \{p_j\}$ contains at least one correct process.

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

4 The Main Result

In this section, we present our reduction algorithm $T_{\mathcal{D}\rightarrow\Omega_n}$ (Figure 2). As in [4,14], each process p_i maintains a shared variable G_i that contains a finite ever growing DAG representing a sample of failure detector output values at different processes as well as causal relationships between them. Let $G_i(\tau)$ denote the value of G_i at the end of the last step of p_i before time $\tau \in \mathbb{T}$. There is an infinite DAG G (defined in Section 3) such that, for any correct process p_i, $\cup_{\tau\in\mathbb{T}} G_i(\tau) = G$ (Lemma 6.6.1.2 and Lemma 6.6.1.4 of [4]).

Every process p_i periodically scans the registers G_j $(j = 1, .., n)$ and adds the new vertexes and edges from each G_j $(j = 1, .., m)$ to G_i. Then p_i queries its failure detector module, adds a new vertex corresponding to the output value to G_i and an edge to the vertex from each old vertex of G_i.

```
1:  output_i ← {p_i}
2:  G_i ← ∅
3:  k ← 1
4:  while true do
5:    for all j ∈ {0,...,n}, i ≠ j do
6:      G_i ← G_i ∪ G_j
7:    d ← D_i
8:    k ← k + 1
9:    G_i ← G_i ∪ G_j
10:   add [p_i, d, k] to G_i and edges from all other vertexes of G_i to [p_i, d, k]
11:   for all j ∈ {0,...,n+1} do
12:     Υ_i^j ← simulation tree induced by G_i and I^j
13:   if there is no critical Υ_i^k in {Υ_i^j}_{j=0,...,n+1} then
14:     output_i ← {p_i}
15:   else
16:     if every Υ_i^j is univalent then
17:       k ← the smallest j, such that Υ_i^j is 1-valent
18:       output_i ← {p_k}
19:     else
20:       output_i ← the deciding set of the smallest decision gadget in {Υ_i^j}_{j=0,...,n+1}
```

Fig. 2. Reduction algorithm $T_{\mathcal{D}\rightarrow\Omega_n}$ for process p_i.

Let I^j $(j = 0, .., n + 1)$ denote an initial configuration of $\mathsf{Cons}_{\mathcal{D}}$ in which processes $p_0, .., p_{j-1}$ propose 1 and processes $p_j, .., p_n$ propose 0. Process p_i constructs then a *simulation forest* — the set of simulation trees $\{\Upsilon_i^j\}_{j=1,\ldots,n+1}$, where Υ_i^j denotes the simulation tree induced by G_i and I^j. Let $\Upsilon_i^j(\tau)$ $(\tau \in \mathbb{T})$ denote the simulation tree induced by $G_i(\tau)$ and I^j. Similarly, $\cup_{\tau\in\mathbb{T}} \Upsilon_i^j(\tau) = \Upsilon^j$, where Υ^j is induced by G and I^j.

Process p_i tags each vertex $C \in \Upsilon_i^j$ according to the decision taken in C's descendants. We say that Υ_i^j is *critical* if and only if Υ_i^{j-1} is 0-valent and Υ_i^j is 1-valent or bivalent. A finite tree Υ_i^j might have no tags though, hence there might be no critical simulation tree in $\{\Upsilon_i^j\}_{j=0,\ldots n+1}$.

As a result of the reduction algorithm, every process p_i maintains a variable $output_i$, the value of which is returned each time the failure detector module of Ω_n at p_i is queried.

Theorem 1. *Let $\mathcal{S}$ be any set of one-shot deterministic types, such that $h_m^r(\mathcal{S}) \leq n$. If a failure detector $\mathcal{D}$ implements $(n+1)$-process consensus in a system of processes, using only registers and objects of types in $\mathcal{S}$, then $\Omega_n \preceq \mathcal{D}$.*

Proof. (Sketch) By the validity property of consensus, Υ^0 is 0-valent and Υ^{n+1} is 1-valent. Hence, there exists a critical tree in $\{\Upsilon^j\}_{j=0,\dots n+1}$.

In the algorithm of Figure 2, every correct process eventually identifies the critical tree Υ^k. The following cases are possible:

(1) Υ^k is univalent. In this case, every correct process p_i eventually permanently outputs p_k. By Lemma 6.5.1 of [4], p_k is correct.

(2) Υ^k is bivalent. By Lemma 3, there exists a decision gadget in Υ^k. In this case, eventually, every correct process p_i eventually permanently outputs the deciding set V of the smallest decision gadget of Υ^k. By Lemma 4, the deciding set (of size at most n) of any decision gadget includes at least one correct process.

In both cases, the output of Ω_n is emulated. □

Theorem 1 and [17] imply the following result:

Theorem 2. *Let $\mathcal{S}$ be any set of one-shot deterministic types, such that $h_m^r(\mathcal{S}) = n$. Ω_n is the weakest failure detector $\mathcal{D}$, such that $h_m^r(\mathcal{S};\mathcal{D}) = n+1$.*

5 Boosting Resilience with Ω_n

So far we considered systems in which processes communicate through *wait-free* linearizable implementations of shared objects. Every process can complete every operation on a wait-free object in a finite number of its own steps, regardless of the behavior of other processes.

In contrast, in this section we consider *t-resilient* implementations. They guarantee that a process completes its operation, as long as no more than t process crash, where t is a specified parameter. If more than t processes crash, no operation on a t-resilient implementation is obliged to return.

Assume that k processes communicate through registers and t-resilient linearizable implementations of shared object [3]. We will simply call these *t-resilient* objects (these are called *t-resilient services* in [1]).

The classical results on (a) consensus universality [9] and (b) t-resiliency [3] imply the following impossibility result[5]:

Theorem 3. *Let k and t be any integers, such that $k > t \geq 1$. There is no t-resilient implementation of k-process consensus from registers and $(t-1)$-resilient objects.*

Before presenting the proof of Theorem 3, we recall the following two lemmas from [3] (Theorem 4.2 and a slightly generalized variant of Theorem 4.1) based on Herlihy's universal construction [9]:

[5] An alternative self-contained proof for message passing systems is given in [1].

Lemma 5. *Let t and k be any integers, such that $k > t \geq 0$. Let $\mathcal{S}$ be any set of types that includes* register. *If $(t+1)$-process consensus has a wait-free implementation from $\mathcal{S}$, then k-process consensus has a t-resilient implementation from $\mathcal{S}$.*

Lemma 6. *Let t and k be any integers, such that $k > t \geq 0$. Let T be any type. There is a t-resilient implementation of T for k processes from registers and any t-resilient k-process consensus implementation.*

Thus, we can implement any $(t-1)$-resilient object shared by k processes out of wait-free t-process consensus objects and registers. The following key result (Theorem 4.1 of [3]) relates the existence of a t-resilient implementation of k-process consensus with the existence of a wait-free implementation of $(t+1)$-process consensus.

Lemma 7. *Let k and t be any integers, such that $k > t \geq 2$. Any t-resilient implementation of k-process consensus can be transformed into a wait-free implementation of $(t+1)$-process consensus.*

Note that the lemma above is stated only for $t \geq 2$. In fact, its proof [3] presents a simulation algorithm using objects of type test&set that can be implemented from 2-process consensus objects and registers (but not from registers only).

Now we are ready to proof Theorem 3.

Proof. (of Theorem 3) By contradiction, assume that such an implementation exists. By Lemma 6, any $(t-1)$-resilient object can be implemented out of $(t-1)$-resilient consensus.

Let $t = 1$. Since 0-resilient consensus can be implemented from registers, by assumption, there is an implementation of 1-resilient k-process consensus ($k \geq 2$) from registers, contradicting [16].

Let $t \geq 2$. By Lemmata 5, 6 and 7, t-process consensus can be wait-free implemented from $(t-1)$-process consensus objects, contradicting the robustness of Herlihy's hierarchy with respect to consensus objects [9]. □

Thus, it is not possible to obtain a more fault-tolerant system solving consensus by combining less fault-tolerant components. Not surprisingly, this impossibility can be circumvented by augmenting the system with a failure detector abstraction. Interestingly, our result on boosting the power of deterministic one-shot types implies the following theorem:

Theorem 4. *Let $n > 1$ be any integer. Let $\mathcal{S}$ be any set of types (not necessarily deterministic one-shot), such that registers and $(n-1)$-resilient objects of types in $\mathcal{S}$ implement $(n-1)$-resilient $(n+1)$-process consensus. Ω_n is the weakest failure detector to implement wait-free $(n+1)$-process consensus using registers and $(n-1)$-resilient objects of types in $\mathcal{S}$.*

Proof. By Lemma 7, $(n-1)$-resilient objects of types in $\mathcal{S}$ implement wait-free n-process consensus. The algorithm of [17] implements wait-free $(n+1)$-process consensus using objects of types in $\{\textsf{register}\} \cup \{n\textsf{-process consensus}\}$ and Ω_n. This gives the sufficient part of the theorem.

Assume that a failure detector $\mathcal{D}$ implements wait-free $(n+1)$-process consensus using $(n-1)$-resilient objects in $\mathcal{S}$ and registers. By Lemmata 5 and 6, any $(n-1)$-resilient object can be implemented out of registers and wait-free n-process consensus objects. Clearly, $h^r_m(\{n\textsf{-process consensus}\}) = n$ [9] and n-process consensus is a one-shot deterministic type. By Theorem 1, $\Omega_n \preceq \mathcal{D}$. This gives the necessary part of the theorem. □

References

1. P. Attie, N. Lynch, and S. Rajsbaum. Boosting fault-tolerance in asynchronous message passing systems is impossible. Technical report, MIT Laboratory for Computer Science, MIT-LCS-TR-877, 2002.
2. E. Borowsky, E. Gafni, and Y. Afek. Consensus power makes (some) sense! In *Proceedings of the 13th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 363–372, August 1994.
3. T. Chandra, V. Hadzilacos, P. Jayanti, and S. Toueg. Wait-freedom vs. t-resiliency and the robustness of wait-free hierarchies. In *Proceedings of the 13th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 334–343. ACM Press, 1994.
4. T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *Journal of the ACM (JACM)*, 43(4):685–722, July 1996.
5. T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM (JACM)*, 43(2):225–267, March 1996.
6. C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 35(2):288 – 323, 1988.
7. M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(3):374–382, April 1985.
8. V. Hadzilacos. On the relationship between the atomic commitment and consensus problems. In *Proceedings of the Workshop on Fault-Tolerant Distributed Computing*, volume 448 of *LNCS*, pages 201–208. Springer-Verlag, 1986.
9. M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1):124–149, January 1991.
10. M. Herlihy and E. Ruppert. On the existence of booster types. In *Proceedings of the 41st IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 653–663, 2000.
11. M. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463 – 492, June 1990.
12. P. Jayanti. Wait-free computing. In *Proceedings of the 9th International Workshop on Distributed Algorithms (WDAG)*, volume 972 of *LNCS*, pages 19–50. Springer Verlag, 1995.
13. P. Jayanti. Robust wait-free hierarchies. *Journal of the ACM (JACM)*, 44(4):592–614, 1997.

14. W.-K. Lo and V. Hadzilacos. Using failure detectors to solve consensus in asynchronous shared-memory systems. In *Proceedings of the 8th International Workshop on Distributed Algorithms (WDAG)*, volume 857 of *LNCS*, pages 280–295. Springer Verlag, 1994.
15. W.-K. Lo and V. Hadzilacos. All of us are smarter than any of us: Nondeterministic wait-free hierarchies are not robust. *SIAM Journal of Computing*, 30(3):689–728, 2000.
16. M. C. Loui and H. H. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. *Advances in Computing Research*, pages 163–183, 1987.
17. G. Neiger. Failure detectors and the wait-free hierarchy. In *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 100–109, August 1995.
18. E. Ruppert. Determining consensus numbers. *SIAM Journal of Computing*, 30(4):1156–1168, 2000.