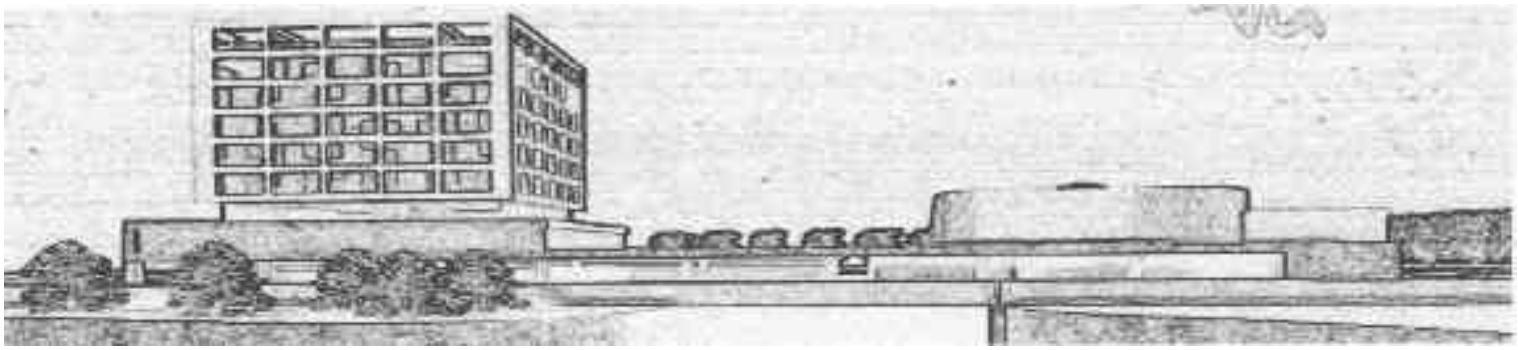
VOLUME IX, NUMBER 2



REPORTS ON SYSTEMS AND COMMUNICATIONS

Weakest failure detectors via an egg-laying simulation (Preliminary Version)

Antonio FERNÁNDEZ ANTA Sergio RAJSBAUM Corentin TRAVERS

Móstoles (Madrid), January 2009
Depósito Legal: M-50653-2004
ISSN: 1698-7489

Table of Contents

Weakest failure detectors via an egg-laying simulation (Preliminary Version)

Antonio FERNÁNDEZ ANTA* Sergio RAJSBAUM** Corentin TRAVERS***

Abstract. In the k-*set agreement task*, n processes propose values, and have to decide on at most k of these values. In particular, consensus is 1-set agreement. In PODC 2008 Zieliński showed that the $\mathrm{anti}{-}\Omega$ failure detector is necessary and sufficient to solve $(n-1)$-set agreement in an asynchronous read/write shared memory system where at most t processes can fail by crashing, $t = n-1$.
In this paper it is shown that $\mathrm{anti}{-}\Omega^t$ is the weakest failure detector to solve t-set agreement in a t-resilient asynchronous distributed system. Each query to $\mathrm{anti}{-}\Omega^t$ returns a set S of process ids, $|S| = n-t$, such that some correct process eventually never appears in such a set S; thus, $\mathrm{anti}{-}\Omega^{n-1} = \mathrm{anti}{-}\Omega$, and $\mathrm{anti}{-}\Omega^1 = \Omega$. Actually, the paper shows a stronger result: Any failure detector that can be used to solve $\mathcal{T}$ is at least as strong as $\mathrm{anti}{-}\Omega^t$, for any agreement task $\mathcal{T}$ that has no t-resilient solution.
The previous results are obtained through a variant of Zieliński's technique, that simplifies some of the arguments and introduces an *egg-laying simulation* of independent interest. The simulation provides the first detailed analysis of immediate snapshot executions in a failure detector enriched environment. It provides a simple proof that Ω is the weakest failure detector to solve consensus when $t = 1$.

1 Introduction

In the k-*set agreement task*, n processes propose values, and have to decide on at most k of these values. Set agreement is not solvable in an asynchronous read/write atomic shared memory system where at most t processes can fail by crashing, when $t = n-1$ (wait-free), even if $k = n-1$ [4,21,25]. More generally, set agreement is not solvable for any k, t such that $1 \leq k \leq t \leq n-1$ [4], and is solvable otherwise. Notice that *consensus* is k-set agreement, when $k = 1$, and the impossibility for $t = 1$ was first proved in [15].

Various partially synchronous models have been considered (e.g., [13]), with upper bounds on process execution speeds and message transmission delays, where set agreement can be solved. A *failure detector* [7] abstracts away such low-level assumptions providing processes with information on process failures. Many failure detectors to solve consensus, set agreement, mutual exclusion, and other problems have been proposed (e.g., [7,10,19,23,24,26]). The problem of finding the weakest failure detector to

* GSyC/LADyR, Universidad Rey Juan Carlos, 28933 Mostoles, Madrid, Spain; anto@gsyc.es.
** Instituto de Matemáticas, Universidad Nacional Autónoma de México, D.F. 04510, México; rajsbaum@math.unam.mx.
*** Department of Computer Science, Technion, Haifa 32000, Israel; corentin@cs.technion.ac.il.

solve a problem has received a lot of attention (e.g., [8,12,10,16]). These results are of theoretical interest because they identify the minimum knowledge about failures that needs to be abstracted to solve a given problem. Furthermore, research on finding a weakest failure detector to solve *any* non-solvable problem for a given model has been done [16].

For consensus, the weakest failure detector, Ω, has been known for a while [8]. For set agreement, the question of a weakest failure detector was settled recently, with $\mathrm{anti}{-}\Omega$ [27], a failure detector that guarantees that there is a correct process whose id will eventually never be output by the detector. Zieliński showed in PODC 2008 [28] that the $\mathrm{anti}{-}\Omega$ failure detector is sufficient to solve $(n-1)$-set agreement in an asynchronous read/write atomic shared memory system where at most $n-1$ processes can crash. He also showed that $\mathrm{anti}{-}\Omega$ is required to solve any non-wait-free solvable task, including $(n-1)$-set agreement. While his proof of this claim is insightful and pioneering, it is complex and subtle.

This paper builds on Zieliński's [28] techniques, providing a somewhat more structured proof that $\mathrm{anti}{-}\Omega$ is required to solve $(n-1)$-set agreement (and is hence the weakest failure detector to solve this task), and generalizes this result to t-set agreement as follows. It is shown that the $\mathrm{anti}{-}\Omega^t$ failure detector is the weakest failure detector to solve t-set agreement in a t-resilient asynchronous distributed system. Each query to $\mathrm{anti}{-}\Omega^t$ returns a set S of processes, $|S| = n-t$, such that some correct process eventually never appears in such a set S; thus, $\mathrm{anti}{-}\Omega^{n-1} = \mathrm{anti}{-}\Omega$, and $\mathrm{anti}{-}\Omega^1 = \Omega$. Actually, the paper shows a stronger result: Any failure detector that can be used to solve $\mathcal{T}$ is at least as strong as $\mathrm{anti}{-}\Omega^t$, for any agreement task $\mathcal{T}$ that has no t-resilient solution. The complementary result is also provided, namely, that $\mathrm{anti}{-}\Omega^t$ is sufficient to solve t-set agreement t-resiliently.

The previous necessity results are obtained through a variant of Zieliński's [28] technique, that simplifies some of the arguments in his paper, and introduces an *egg-laying simulation* of independent interest. In particular, the technique yields a simple proof that Ω is the weakest failure detector to solve consensus when $t = 1$, and the only alternative construction to that of [8] known to us. Essentially, the simulation provides the first detailed analysis of immediate snapshot runs in a failure detector enriched environment. This is explained in more detail next.

Zieliński builds a directed tree representing runs of a protocol together with samples of any failure detector that is able to solve a non-wait-free solvable task, T, in a way similar to the original consensus proof of Chandra et al. [8], except that he considers *immediate snapshot* [4,25] schedules (where a snapshot of a process always occurs immediately after its preceding write). Then, he shows how to emulate runs based on the tree, without using any failure detector, claiming that either T is solved on the set R of runs defined by the tree, or else $\mathrm{anti}{-}\Omega$ is extracted. A crucial step in Zieliński's proof is that a wait-free protocol that solves T on that particular subset of runs implies the existence of a protocol solving T on *all* wait-free runs. We believe that the proof of this claim is easier to understand in our framework. The egg-laying simulation introduced here simplifies the construction of the tree. It constructs a tree of immediate snapshot runs where each local process state includes the result of a failure detector query (valid in the corresponding run). To guarantee the existence of a wait-free protocol, care is

taken to ensure that indistinguishability relations are not disrupted by failure detector values. That is, if two runs are indistinguishable to some process p in the wait-free model, when p queries the failure detector, it gets the same answers from the failure detector in both (hence, for such a subset of runs, the failure detector is useless).

Organization. In Section 2, we describe the model of computation. We describe the usual crash failures, asynchronous R/W model of computation, its extension to failure detectors, and the immediate snapshot model with a failure detector. In Section 3 we present our result, for the wait-free case: $\mathrm{anti}{-}\Omega$ is required to solve any non-wait-free solvable task, including $(n-1)$-set agreement. The main ideas are introduced here, including the egg-laying simulation. In Section 4 the t-resilient extension is described, proving that any failure detector that can be used to solve $\mathcal{T}$ is at least as strong as $\mathrm{anti}{-}\Omega^t$, for any agreement task $\mathcal{T}$ that has no t-resilient solution. The complementary result is presented in Appendix A, namely, that $\mathrm{anti}{-}\Omega^t$ is sufficient to solve t-set agreement t-resiliently.

2 Preliminaries

We consider a standard system consisting of a set $\Pi = \{p_1, p_2, \ldots, p_n\}$ of n asynchronous processes that communicate via an atomic read/write shared memory, and can fail by crashing (i.e., stop taking steps). More details can be found in [2]. We may assume w.l.o.g. the shared memory consists of a single array $R[1..n]$ of Single-Writer/Multi-Reader registers. According to its local, deterministic protocol, process p_i can write a value v to $R[i]$, denoted $R[i] \leftarrow v$, and read every component $R[j]$; the operation $\mathsf{read}(R)$ is shorthand for a sequence of reads to each component of R, in order from 1 to n.

Tasks To model decision tasks, we identify two special components of each process p_i's state: an *input,* $I[i]$, and an *output*, dec_i component. It is assumed that initial states differ only in the value of the input component, which never changes. The output component is initially equal to $\perp$, and once a different value is written, it never changes and we say the process has *decided.*

A *task* $\mathcal{T}$ is specified by a triplet $(\Delta, \mathcal{I}, \mathcal{O})$, where $\mathcal{I}$ and $\mathcal{O}$ are sets of n-entries vectors, containing the valid input and output vectors of $\mathcal{T}$, respectively, and Δ is a relation that specifies for each assignment to input components $I \in \mathcal{I}$ if the processes can decide on the output vector $O \in \mathcal{O}$. A task is *bounded* if both $\mathcal{I}$ and $\mathcal{O}$ are finite. For this paper, we consider only bounded tasks. A protocol *solves* the task if any finite execution where the process start with inputs $I \in \mathcal{I}$, can be extended to an execution in which all processes decide on values $O \in \mathcal{O}$ such that $(I, O) \in \Delta$. A protocol is *wait-free* if in any execution, either has a finite number of events or it decides. We will also consider t-resilient protocols, where in executions with at least $n-t$ processes that have infinite number of events, all processes that have infinitely many events decide.

Without loss of generality (efficiency issues are ignored), we consider *full-information protocols in standard form,* as described in the left part of Figure 1. Each process remembers everything and writes everything it knows. Its protocol consists of repeatedly

init $R[1..n] \leftarrow [\perp, \ldots, \perp]$ array of n SWMR registers;
  $snapshot_i \leftarrow I[i]$;
  $dec_i \leftarrow \perp$;
repeat
  $R[i] \leftarrow \langle snapshot_i \rangle$;
  $snapshot_i \leftarrow \mathsf{read}(R)$;
  $dec_i \leftarrow \mathsf{decide}_i(snapshot_i)$;
end repeat
	init $R[1..n] \leftarrow [\perp, \ldots, \perp]$ array of n SWMR registers;
	  $snapshot_i \leftarrow I[i]$;
	  $dec_i \leftarrow \perp$;
	repeat
	  $d_i \leftarrow \mathsf{fd\_query}()$;
	  $R[i] \leftarrow \langle snapshot_i, d_i \rangle$;
	  $snapshot_i \leftarrow \mathsf{read}(R)$;
	  $dec_i \leftarrow \mathsf{decide}_i(snapshot_i)$;
	end repeat

Fig. 1. IS full information standard form protocol without (left) and with (right) a failure detector.

writing and reading R, and after each $\mathsf{read}(R)$, applying a function decide_i to the vector of values read from R, to possibly decide. We do not require processes to halt; process can continue to participate after deciding.

Two finite executions σ, σ' are *indistinguishable* to process p_i, denoted $\sigma \overset{p_i}{\sim} \sigma'$, if p_i has the same local state at the end of both executions.

Failures and failure detectors A *failure pattern* F is a function from the integers $T = \{0\} \cup N$ to 2^{Π}. Intuitively, $F(\tau)$ denotes the set of processes that have crashed by time τ. We assume that processes do not recover, i.e, $\forall \tau, F(\tau) \subseteq F(\tau + 1)$. $faulty(F) = \bigcup_\tau F(\tau)$ is the set of processes that fail in F. A process is *correct* if it is not faulty, and $Correct(F) = \Pi - faulty(F)$ denotes the set of all correct processes in F. An *environment* is a set of failure patterns. The *wait-free environment* contains all failure patterns F that include at least one correct process, i.e., $|Correct(F)| \geq 1$. Similarly, the t*-resilient environment* is the set of failure patterns in which at least $n - t$ processes are correct.

A *failure detector* is an oracle that, when queried by a process using operation $\mathsf{fd\_query}()$, returns a value in some range $\mathcal{R}$ which conveys information about the current failure pattern. More formally, the behavior of a failure detector is modeled by a set of possible histories. A failure detector history H with range $\mathcal{R}$ is a function from $\Pi \times T$ to $\mathcal{R}$. $H(p_i, \tau)$ represents the value returned by a failure detector query by p_i at time τ. The specification of failure detector D with range $\mathcal{R}_D$ associates each possible failure pattern to an allowable set of histories. $D(F)$ is the non-empty set of failure detector histories that may be output by D when the failure pattern is F. Let $D1$ and $D2$ be two failure detectors. $D1$ is *at least as strong as* $D2$ in environment $\mathcal{E}$ if there is a distributed protocol $\mathcal{A}_{D1 \rightarrow D2}$ that emulates $D2$ in any run whose failure pattern belongs to $\mathcal{E}$.

A failure detector of the class $\mathrm{anti}{-}\Omega^k$ returns, when queried, a set of $n - k$ processes. In every infinite run, the following property is ensured: there exists a correct process that eventually never appears in the sets returned by the queries. More formally, $\forall H \in \mathrm{anti}{-}\Omega^k(F), (\forall p_i \in \Pi, \forall \tau \in T, |H(p_i, \tau)| = n - k) \wedge (\exists p_\ell \in Correct(F), \tau_\ell \in T : \forall p_i \in \Pi, \forall \tau > \tau_\ell, p_\ell \notin H(p_i, \tau))$. When $k = n - 1$, we simply write $\mathrm{anti}{-}\Omega$ instead of $\mathrm{anti}{-}\Omega^{n-1}$.

Without loss of generality, we extend full information protocols in standard form with failure detector queries, as described in the right part of Figure 1. To that end, each process p_i is provided with an operation fd_query() which returns the current value of the local failure detector module of p_i. More precisely, a fd_query() operation returns one of the values output by the failure detector between the time at which the operation starts and the time at which it ends [22].

IS executions Consider a full-information protocol in standard form; we can restrict our attention to a subset of its executions, called *immediate snapshot* (IS) executions, first used in [4,25]. An IS execution is determined by its input vector I and a sequence of non-empty *concurrency classes* of process ids, $\alpha(I) = s_1, s_2, \ldots$ Every process in s_k performs a write operation (in increasing order of ids), and then every process in s_k reads all the components of R. Intuitively, an IS execution proceeds in rounds, each contains a concurrent write by every active process, followed by a concurrent atomic snapshot [1] by every active process. Sometimes, when I is clear from the context, we use only α to specify an IS execution.

Immediate snapshot executions capture the computational power of the model. If a full-information protocol solves a task, it solves it in the subset of IS executions. Conversely, a protocol for the participating set task [4] can be used to simulate IS executions in a full-information manner, implying the next lemma:

Lemma 1. *There is a wait-free protocol which solves a task if and only if there is a wait-free protocol which solves the task only in IS executions.*

We can consider IS executions of a full-information protocol with a failure detector. An *IS execution with a failure detector* σ is specified by the triplet (I, α, d), where I and α are as before, and $d(i, k)$ is a function that associates to each process p_i the value returned by the kth fd_query() operation of p_i.

Equivalently, an IS execution with failure detector can be defined as a sequence of events satisfying certain conditions. An event is performed by a single process which applies either a write(), read() or fd_query() operation. An event of p_i is denoted by a tuple $\langle id, op, vin, vout\rangle$ where id is p_i; $op \in \{\mathsf{write}(), \mathsf{read}(), \mathsf{fd\_query}()\}$ is the operation applied by p_i; vin and $vout$ are respectively the input and output values of the operation. Given a failure detector D, a (finite or infinite) IS execution with failure detector D has to be D-legal (or simply legal if D is clear from the context), i.e., there must be a timing of the events that respects the IS execution model, the shared memory semantic, and the failure detector specification. Formally:

Definition 1 (D-legality). *Let $\sigma = (I, s_1, s_2, \ldots)$ be an IS execution with failure detector D. If $p_i \in s$, where s is a concurrency class of σ, denote respectively $q_i^{(s)}, w_i^{(s)}$, and $r_i^{(s)}$ the query, write, and read events of p_i associated with s. The execution σ is D-legal if there exists a failure pattern F, a history $H \in D(F)$ and a function T that maps each event ev of σ to an integer $T(ev)$ (which represents the "time" the event was applied), such that*
1. $\forall ev \in \sigma, ev.id \notin F(T(ev))$. (A crashed process cannot apply events.)
2. $\forall ev \in \sigma, ev.op = \mathsf{fd\_query}() \implies H(ev.id, T(ev)) = ev.out$. (The time and value

returned of a failure detector query must be consistent.)
3. $\forall s \in \sigma, \forall p_i \in s, T(q_i^{(s)}) < T(w_i^{(s)}) < T(r_i^{(s)})$. *(Within a concurrency class, processes apply events in the "standard" order.)*
4. $\forall s_k, s_\ell \in \sigma : k < \ell, \forall p_i \in s_k \cap s_\ell, T(r_i^{(s_k)}) < T(q_i^{(s_\ell)})$. *(Between concurrency classes, processes apply events in the "standard" order.)*
5. $\forall s \in \sigma, \forall p_i, p_j \in s, T(w_i^s) < T(r_j^s)$. *(Within a concurrency class, all reads are applied after all writes.)*
6. $\forall s_k, s_{k+1} \in \sigma, \forall p_i \in s_k, \forall p_j \in s_{k+1}, T(r_i^{(s_k)}) < T(w_j^{(s_{k+1})})$. *(Every operation in a class occurs before any write/read operation of the next class.)*

Let σ be a finite IS execution (with or without failure detector). We use $snapshot_i(\sigma)$ to denote the value of $snapshot_i$ at the end of σ. Notice that this value completely defines the local state of p_i. Therefore, an equivalent indistinguishability definition (for both types of IS executions) is:

Definition 2 (Indistinguishability). *Let σ and σ' be two finite IS executions (with or without failure detector). σ and σ' are* indistinguishable *for process p_i, denoted $\sigma \overset{p_i}{\sim} \sigma'$, if the snapshots of p_i after both executions are equal, i.e., $snapshot_i(\sigma) = snapshot_i(\sigma')$.*

3 Wait-free case: Extracting $\mathrm{anti}-\Omega$

Let $\mathcal{T}$ be a task that has no wait-free solution. Assuming that the task can be solved with some failure detector D, this section describes how to extract $\mathrm{anti}-\Omega$ from D. The main result of this section is the following theorem.

Theorem 1. *Let $\mathcal{T}$ be a bounded task that has no wait-free solution. Suppose that $\mathcal{T}$ can be solved with a failure detector D. Then D is at least as strong as* $\mathrm{anti}-\Omega$.

Main idea The idea of the proof is as follows. We assume there exists an algorithm $\mathcal{A}_D$ that solves $\mathcal{T}$ using failure detector D. Without loss of generality, we assume that $\mathcal{A}_D$ is a full information protocol in standard form (Figure 1). Roughly, we describe a protocol that simulates IS executions of $\mathcal{A}_D$. The IS executions of protocol $\mathcal{A}_D$ include failure detector queries, but our simulation shows that there is a subset of IS executions where the failure detector outputs preserve indistinguishability; namely, whenever two IS executions without failure detector are indistinguishable to p_i, they are also indistinguishable to p_i with failure detector values. Thus, there is a subset of the IS executions of $\mathcal{A}_D$ that are exactly isomorphic to the set of all IS executions of a wait-free protocol. But as we are assuming that no algorithm solves $\mathcal{T}$ in the wait-free environment, by Lemma 1, our protocol cannot finish simulating all IS executions of $\mathcal{A}_D$.

In every IS execution of $\mathcal{A}_D$ we simulate, each process stops taking steps immediately after obtaining a decision ; the simulation finishes whenever every process has obtained a decision. As previously noticed in [16,28], two reasons can prevent the simulation of an IS execution α of $\mathcal{A}_D$ from finishing. First, only process p_j can provide the output of the failure detector queries of p_j in α. Thus, if p_j is faulty and does not

provide "enough" failure detector values, the simulation may be blocked forever, waiting for a failure detector value to be announced by p_j. Second, even if we are able to simulate every query of α (each process providing sufficiently enough failure detector values), the simulation may not finish if some process p_j takes infinitely many steps but does not decide. Observe that in every infinite IS execution of $\mathcal{A}_D$ that is legal and fair, every correct process must decide. An infinite execution is *fair* if every correct process takes infinite steps. Every IS execution of $\mathcal{A}_D$ we simulate is legal. Hence, since p_j does not decide, there must exist a correct process that takes only finitely many steps in the simulated execution. The output of $\mathrm{anti}{-}\Omega$ is at each process the process being simulated. In the first case, a faulty process is eventually output forever, whereas in the second case, a correct process is eventually never output, thus extracting $\mathrm{anti}{-}\Omega$.

Trees and egg-trees Our protocol to simulate IS executions of $\mathcal{A}_D$ builds incrementally what we call an *egg-tree*, which represents IS executions with failure detector values. All the IS executions of a protocol $\mathcal{A}$ in standard form without a failure detector (Figure 1) can be represented by an *infinite tree*. Each edge is labeled with a non-empty set of processes, which is a concurrency class. Hence, for a sequence of concurrency classes $s_1, s_2, \ldots, s_k$, there is a path in the tree starting at the root, $root \xrightarrow{s_1} v_1 \xrightarrow{s_2} v_2 \ldots \xrightarrow{s_k} v_k$. The tree is *complete* since, for every node v of the tree, for every $s \subseteq \Pi$, $s \neq \emptyset$, there is an edge labeled s outgoing from v.

Consider an input vector $I \in \mathcal{I}$. Each path $root \xrightarrow{s_1} v_1 \ldots \xrightarrow{s_k} v_k$ in the tree describes a unique IS execution $\alpha(v_k, I)$ of $\mathcal{A}$. Each node v_i represents the state of the shared memory, after every operation of the finite IS execution $s_1, \ldots, s_i$ has been performed with the input vector I; this state is the last snapshot obtained by each process $\in s_i$ in the execution fragment $s_1, \ldots, s_i$. Let $I_g = [\square_1, \square_2, \ldots, \square_n]$ be a generic input vector. For every node v_k in the tree, denote $state(v_k)$ the state of the shared memory after every operation of the finite IS execution $\alpha(v_k, I_g)$ has been performed. Note that, given the node v_k and the input vector I_g, the state $state(v_k)$ is uniquely determined. Let $I \in \mathcal{I}$ be an input vector of task T. The state of v_k can be *instantiated* with input vector I. The instantiated state, denoted $state(v_k, I)$, is obtained by replacing in $state(v_k)$ the generic input value $\square_j$ by the input value $I[j]$, for each $j \in \{1, \ldots, n\}$.

For every $x \geq 0$, we say that p_i *performs* x *steps* in the path $u \rightsquigarrow v$ if p_i is contained in exactly x distinct concurrency classes along the path. I.e., let $u \rightsquigarrow v = u \xrightarrow{s_1} v_1 \ldots \xrightarrow{s_k} v$; then $x = |\{j \in \{1, \ldots, k\} : p_i \in s_j\}|$.

Definition 3 (one step ancestor). *Let v be a node of the tree. The* one step ancestor *of node v for p_i, denoted $osa(v, p_i)$, is the node u such that (1) u is an ancestor of v in the tree, (2) p_i performs exactly one step in the path $u \rightsquigarrow v$ and (3) the label of the incoming edge of u contains p_i. If no such node exists, the one step ancestor of v for p_i is the root of the tree.*

In the sub-tree depicted in Figure 2, u is the one step ancestor of both nodes v and w for p_i. We need one step ancestors to characterize what $state(v)$ is. The idea is to identify what is the last thing written by p_i, when we consider the state of node v. For this, we go up the tree to look for a vertex u, where p_i took a step (p_i got a snapshot $state(u)$)

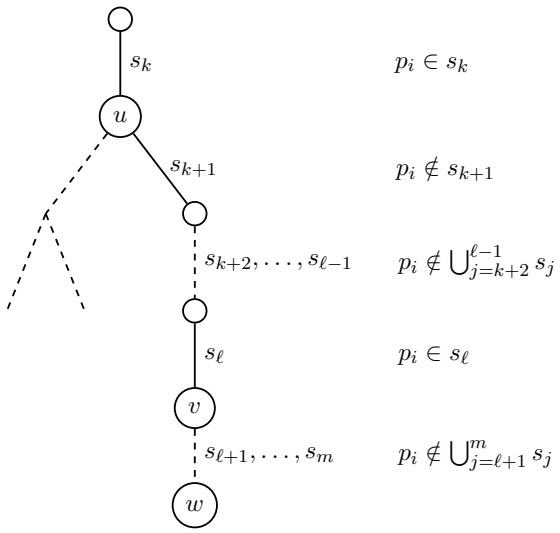


Fig. 2. One step ancestor of node v for p_i.

and such that there is a step of p_i between u and v (in which p_i wrote this snapshot). If this is the only step of p_i between u and v, the register $R[i]$ contains this snapshot in node v. Thus we have:

Definition 4 (node state). *Let v be a node in the tree. Let u be its parent, and s the label of the arc connecting u to v. $state(v)$ is an n-entries vector defined as follows. For every $i \in \{1, \ldots, n\}$:*

- $state(root)[i] = \perp$.
- *If $p_i \notin s$, $state(v)[i] = state(u)[i]$.*
- *If $p_i \in s$, let $u_i = osa(v, p_i)$.*
  - *If $u_i = root$ then $state(v)[i] = \langle \Box_i \rangle$.*
  - *Else, $state(v)[i] = \langle state(u_i) \rangle$.*

An *egg-tree* (or e-tree for short) is a subtree with the same root of the complete infinite tree described, augmented with a failure detector D. In an e-tree, failure detector values are held in "eggs" that are mapped to the nodes of the original subtree as follows. For every node v whose incoming arc has label s, for each $p_i \in s$ an egg $e_i(v) = \langle p_i, d, k \rangle$ is assigned to v by p_i. In the egg $e_i(v)$, the field $e_i(v).d$ is the value returned when p_i queried the failure detector for the $e_i(v).k$th time. A process p_i can assign the same egg to two different nodes v and v', i.e. $e_i(v) = \langle p_i, d, k \rangle = e_i(v')$. The e-tree represents a collection of IS executions with a failure detector (Figure 1). From the tree and the eggs, we define the e-state of a node v.

Definition 5 (node e-state). *Let v be a node in the e-tree. Let u be its parent, and s the label of the arc connecting u to v. $e\text{-}state(v)$ is an n-entries vector defined as follows. For every $i \in \{1, \ldots, n\}$:*

- $e\text{-}state(root)[i] = \perp$.
- *If* $p_i \notin s$, $e\text{-}state(v)[i] = e\text{-}state(u)[i]$.
- *If* $p_i \in s$, *let* $u_i = osa(v, p_i)$.
  - *If* $u_i = root$ *then* $e\text{-}state(v)[i] = \langle \square_i, e_i(v).d \rangle$.
  - *Else,* $e\text{-}state(v)[i] = \langle e\text{-}state(u_i), e_i(v).d \rangle$.

Let the path to a node v_k be $root \xrightarrow{s_1} v_1 \ldots \xrightarrow{s_k} v_k$. Then, $e\text{-}state(v_k)$ is the state of the shared memory after every operation of the finite IS execution $s_1, \ldots, s_k$ of a protocol $\mathcal{A}_D$ has been performed with the generic input vector I_g and failure detector values from D. As in the states of the infinite tree, the e-state of v_k can be *instantiated* with input vector I, denoted $e\text{-}state(v_k, I)$. Note that $e\text{-}state(v_k, I)$ is the state of the shared memory after every operation of the finite IS execution of $\mathcal{A}_D$ with concurrency classes $s_1, \ldots, s_k$ has been performed with input vector I and the failure detector values of the eggs of $\{v_1, \ldots, v_k\}$. We denote such an execution by $e\alpha(v_k, I)$. Given the node v_k, the eggs of $\{v_1, \ldots, v_k\}$, and the input vector I, then $e\text{-}state(v_k, I)$ and $e\alpha(v_k, I)$ are completely determined.

Extraction of $\text{anti}{-}\Omega$ The algorithm of Figure 3 uses algorithm $\mathcal{A}_D$, that solves task $\mathcal{T}$ using failure detector D, to construct, at each process p_i, an egg-tree as described. Each process builds its own tree, but in order to guarantee that all the trees are exactly equal, processes share the eggs assigned to each node, by filling the shared array EGG with the eggs they have assigned to the nodes of the tree.

Processes traverse the IS executions tree, starting from the root, and all in the same order, asynchronously as done in [28]. When a process p_i gets to a node v_k, through the path $root \xrightarrow{s_1} \ldots \xrightarrow{s_k} v_k$, it first attempts to determine $e\text{-}state(v_k)$, corresponding to a D-legal IS execution with concurrency classes $s_1, \ldots, s_k$ of $\mathcal{A}_D$. For that, the process needs to know failure detector outputs, for all processes p_j in s_k. Thus, process p_i waits until a failure detector value is ready for each such process, in increasing ids order. A failure detector value for p_j is ready when an egg appears in the shared array $EGG(v_k)[j]$. While it is waiting for an egg of $p_j \in s_k$, p_i produces as simulated output fd_i for $\text{anti}{-}\Omega$, precisely p_j.

Only p_j is in charge of laying an egg in $EGG(v_k)[j]$. To do this, it queries the failure detector, unless p_j can find somewhere else in the e-tree a node v which shares the same one step ancestor u with v_k (In short, $u = osa(v_k, p_j) = osa(v, p_j)$). If an egg for p_j has already been laid in v, p_j copies that egg instead of querying the failure detector. As can be observed, a process p_j assigns the same egg to all the nodes v with the same $state(osa(v, p_j))$ by means of the map m_j. This implies that, if $state(v) = state(v_k)$, processes assign the same eggs to nodes v and v_k, therefore preserving indistinguishability since it follows that $e\text{-}state(v) = e\text{-}state(v_k)$. Moreover, we show that copying eggs does not break legality (Lemma 3); $e\text{-}state(v_k)$ is the state of the shared memory as can be observed in a real IS execution of $\mathcal{A}_D$.

If no egg is laid in v_k by process $p_j \in s_k$, this implies that p_j is faulty. Hence p_j is a valid output for $\text{anti}{-}\Omega$. Otherwise, p_i is eventually able to compute $state(v_k)$. In order to select which node to explore next, p_i checks if the protocol $\mathcal{A}_D$ decides for each possible input I, and when it finishes simulating the node. It traverses an edge

labeled with a concurrency class s', provided for some I all processes in s' have not yet decided. If no such I can be found, the exploration backtracks.

The e-tree obtained is thus *complete* in the following sense. For any node v of the e-tree and any $I \in \mathcal{I}$, let $undecided_i(v, I)$ be the set of processes p_i that have not decided after execution $e\alpha(v, I)$. Then, for every non-empty $s \subseteq undecided_i(v, I)$, there is an edge labeled s outgoing from v. The construction of the e-tree may generate an infinite branch b. By the observation above, for some input vector I, at least one process never decides in this branch. Hence, the corresponding IS execution of $\mathcal{A}_D$ must be unfair: a correct process p_ℓ does not belong to the set S of processes that appear infinitely often in b. Moreover, due to the particular way the tree is explored, eventually only arcs labeled $s \subseteq S$ are traversed. Therefore, the output of $\mathrm{anti}{-}\Omega$ produced at each process eventually excludes the correct process p_ℓ.

The rest of this section is devoted to prove that the algorithm of Figure 3 in fact implements $\mathrm{anti}{-}\Omega$, hence proving Theorem 1.

Indistinguishability and legality To any IS execution $e\alpha(v_k, I)$ of $\mathcal{A}_D$ with failure detector D derived from the path $root \xrightarrow{s_1} v_1 \ldots \xrightarrow{s_k} v_k$ in the e-tree, it can be associated an IS execution $\alpha(v_k, I)$ without a failure detector, by removing the failure detector values. More precisely, $\alpha(v_k, I)$ is defined by the sequence of concurrency classes $s_1, \ldots, s_k$, and is associated to the path $root \xrightarrow{s_1} v_1 \ldots \xrightarrow{s_k} v_k$ in the infinite tree. Reciprocally, to an IS execution $\alpha(v_k, I)$ of a standard form full-information protocol without failure detector, with concurrency classes $s_1, \ldots, s_k$ and input vector $I \in \mathcal{I}$, it can be associated IS execution $e\alpha(v_k, I)$ defined in the e-tree (assuming the path $root \xrightarrow{s_1} v_1 \ldots \xrightarrow{s_k} v_k$ exists in the e-tree). We establish that the e-tree construction of the algorithm in Figure 3 preserves indistinguishability between both executions.

Lemma 2 (indistinguishability). *Let $\alpha(v_k, I)$ and $\alpha'(v'_{k'}, I')$ be two IS executions without failure detector, with concurrency classes $\beta = s_1, \ldots, s_k$ and $\beta' = s'_1, \ldots, s'_{k'}$, respectively. Assume that there exists two paths $root \rightsquigarrow v_k$ and $root \rightsquigarrow v'_{k'}$ in the e-tree built with the algorithm of Figure 3 whose arcs are labeled $s_1, \ldots, s_k$ and $s'_1, \ldots, s'_{k'}$. For every process p_i, $\alpha(v_k, I) \overset{p_i}{\sim} \alpha'(v'_{k'}, I') \Rightarrow e\alpha(v_k, I) \overset{p_i}{\sim} e\alpha'(v'_{k'}, I')$.*

Proof From Definition 2 we have to prove that $snapshot_i(e\alpha(v_k, I)) = snapshot_i(e\alpha'(v'_{k'}, I'))$. Let s_ℓ and $s'_{\ell'}$ be the last concurrency classes in which p_i appear in β and β', respectively. If there are not such s_ℓ and $s'_{\ell'}$, the claim follows, since then $snapshot_i(e\alpha(v_k, I)) = snapshot_i(\alpha(v_k, I)) = I[i]$ and $snapshot_i(e\alpha'(v'_{k'}, I')) = snapshot_i(\alpha'(v'_{k'}, I')) = I'[i]$; and $I[i] = I'[i]$ from $\alpha(v_k, I) \overset{p_i}{\sim} \alpha'(v'_{k'}, I')$. Otherwise, let v_ℓ and $v'_{\ell'}$ be the nodes whose incoming edges are labelled s_ℓ and $s'_{\ell'}$ in paths $root \rightsquigarrow v_k$ and $root \rightsquigarrow v'_{k'}$, respectively. Since $\alpha(v_k, I) \overset{p_i}{\sim} \alpha'(v'_{k'}, I')$, from Definition 2, $snapshot_i(\alpha(v_k, I)) = snapshot_i(\alpha'(v'_{k'}, I'))$, and from Figure 1, $snapshot_i(\alpha(v_\ell, I)) = snapshot_i(\alpha'(v'_{\ell'}, I'))$; i.e., $state(v_\ell, I) = state(v'_{\ell'}, I')$. Then, we have to prove that $snapshot_i(e\alpha(v_\ell, I)) = snapshot_i(e\alpha'(v'_{\ell'}, I'))$, i.e., that $e\text{-}state(v_\ell, I) = e\text{-}state(v'_{\ell'}, I')$.

Let p_j be any process that appears in $\beta_\ell = s_1, \ldots, s_\ell$. Assume p_j appears exactly t times. Then, there are exactly t concurrency classes $s_{r_1}, \ldots, s_{r_t}$, with $1 \leq r_1 < \ldots < r_t \leq \ell$, that contain p_j. Since $state(v_\ell, I) = state(v'_{\ell'}, I')$, there are also exactly t concurrency classes $s'_{r'_1}, \ldots, s'_{r'_t}$, with $1 \leq r'_1 < \ldots < r'_t \leq \ell'$, that contain p_j in

$\beta'_{\ell'} = s'_1, \ldots, s'_{\ell'}$. Furthermore, for each $x \in \{1, \ldots, t\}$ it holds that $state(v_{r_x}, I)[j] = state(v'_{r'_x}, I')[j]$. Let $u_{r_x} = osa(v_{r_x}, p_j)$ and $u'_{r'_x} = osa(v'_{r'_x}, p_j)$. Then, from Definition 4, $state(u_{r_x}, I) = state(u'_{r'_x}, I')$, which implies $state(u_{r_x}) = state(u'_{r'_x})$. (Note that in the special case of $x = 1$, $u_{r_1} = u'_{r'_1} = root$ and it holds trivially.) It follows from the text of the egg_laying() procedure that the eggs $e_j(v_{r_x})$ and $e_j(v'_{r'_x})$, laid by p_j at v_{r_x} and $v'_{r'_x}$, respectively, are equal, since $m_j(state(u_{r_x})) = m_j(state(u'_{r'_x}))$.

Since this holds for every process that appears in β_ℓ, it follows from Definition 5 that in $e\alpha(v_\ell, I)$ and $e\alpha'(v'_{\ell'}, I')$, for each $x \in \{1, \ldots, t\}$ it holds that $e\text{-}state(v_{r_x}, I)[j] = e\text{-}state(v'_{r'_x}, I')[j]$. Also from Definition 5 we have that $e\text{-}state(v_\ell, I)[j] = e\text{-}state(v_{r_t}, I)[j]$ and $e\text{-}state(v'_{\ell'}, I')[j] = e\text{-}state(v'_{r'_t}, I')[j]$. Finally, for any process p_j that does not appear in β_ℓ, Definition 5 sets $e\text{-}state(v_\ell, I)[j] = e\text{-}state(v'_{\ell'} I')[j] = \perp$. Hence, $e\text{-}state(v_\ell, I) = e\text{-}state(v'_{\ell'}, I')$. $\square_{Lemma\ 2}$

Legality We show now that any execution described by the e-tree is D-legal. The goal is to prove the following main claim, whose proof follows from the subsequent Lemmas 8, 9, 10.(a), 11, 10.(b), and 10.(c).

Lemma 3 (D-legality). *Let b be a finite or infinite path of the e-tree. The execution $e\alpha$ associated with b is D-legal.*

Let $b = root \xrightarrow{s_1} v_1 \xrightarrow{s_2} v_2, \ldots$ a path of the e-tree. Assume that for each node v_k in b, all eggs have been laid, i.e., for every $p_j \in s_k$, $EGG(v_k)[j] \neq \perp$. Recall that the execution $e\alpha$ described by b is an IS execution whose successive concurrency classes are $s_1, s_2, \ldots$ If p_i belongs to s_k, it performs first a failure detector query, then a write, followed by a read of the whole shared memory. Each write/read operation of s_k must be scheduled after the last read operation of s_{k-1} and before the first write operation of s_{k+1}.

The real execution is D-legal. So, there exists a failure pattern F, a failure detector history $H \in D(F)$ and a "timing" RT that are consistent in the sense of Definition 1. We first define a timing T, based on RT and then we show that T, together with H and F satisfies the conditions of Definition 1 for the execution $e\alpha$. For technical reasons, if the last event in $e\alpha$ of a process $p_i \in faulty(F)$ is a read event, it will be removed from execution $e\alpha$. Note that this event has no impact in the shared memory.

Recall that an event ev is represented by a tuple $\langle id, op, vin, vout \rangle$ whose meaning is process id performs operation op with input value vin and gets back $vout$. We sometimes write $ev \in s_k$ if the event ev is generated at process p_i when p_i is scheduled in the concurrency class s_k.

Let s be a concurrency class of $e\alpha$, and v the associated node in b, i.e., the edge pointing to v is labeled s. Let $p_i \in s$. $q_i^{(s)}, w_i^{(s)}$, and $r_i^{(s)}$ denote respectively the query, write, and read events that occur at p_i when it is scheduled in the concurrency class s.

Let us observe that the result of the query $q_i^{(s)}$ is the failure detector value in egg e_i stored in $EGG(v)[i]$, where v is the node whose incoming edge is labeled s. By definition, this egg e has been generated in a (real) fd_query() event $q(e_i)$ performed by p_i in the underlying real run. We are now ready to define the timing T associated with the simulated execution.

- $T(q_i^{(s)}) = RT(q(e_i))$, the timing associated with the real query that gives rise to e_i.
- $T(w_i^{(s)}) = \max(RT(q(e_i)), T(r^\rho)) + 1$, where ρ is the concurrency class that immediately precedes s. $T(r^\rho) = 0$ if there is no such concurrency class.
- $T(r^{(s)}) = \max(T(w_i^{(s)}) : p_i \in s) + 1$. All read events $r_i^{(s)}$ of class s are assigned the same time.

To prove the lemma, we show that F, H, T and $e\alpha$ verify the conditions of Definition 1.

Notations Given a concurrency class s in $e\alpha$, the last query of s denoted $\ell q(s)$ is the query operation of s that occurs last according to T: $\ell q(s) = q_i^{(s)}$ such that $p_i \in s$ and $\forall p_j \in s, p_j \neq p_i, T(q_j^{(s)}) < T(q_i^{(s)})$. Thus, $T(\ell q(s)) = \max(T(q_j^{(s)}) : p_j \in s)$. $\ell q(s)$ is well defined since at any given time, at most one process is querying its local failure detector. This follows from the **wait** statement of Line 28, and the fact that processes explore the tree in the same order. Hence, no two query events are mapped to the same integer by RT. We first state technical lemmas that follows from the definition of T and the egg_laying procedure.

Lemma 4. *Let s_k and $s_{k'}$ be two concurrency classes in $e\alpha$ such that $k < k'$, $p_i \in s_k \cap s_{k'}$. Then $T(q_i^{(s_k)}) < T(q_i^{(s_{k'})})$.*

Proof We first assume that there is no step by p_i between s_k and $s_{k'}$. More precisely, $\forall j \in \{k+1, \ldots, k'-1\}, p_i \notin s_j$. Let v_k and $v_{k'}$ be the nodes whose incoming arcs are labeled s_k and $s_{k'}$ respectively. We denote e_i and e'_i the two eggs written by p_i in $EGG(v_k)[i]$ and $EGG(v_{k'})[i]$, respectively. In the path $v_k \rightsquigarrow v_{k'}$, p_i appears only in the last concurrency class $s_{k'}$. Moreover, the label of the incoming arc of v_k contains p_i. Thus, $v_k = osa(v_{k'}, p_i)$. Whether p_i generates or not a new egg while it is exploring node $v_{k'}$ depends on the value of $m_i(state(v_k))$ (Line 23). We consider both cases.

- $m_i(state(v_k))$ is undefined, i.e., no egg is associated with the state of node v_k when p_i explores node $v_{k'}$. In that case, p_i queries its failure detector module to generate a new egg e'_i. This occurs at ("real") time $RT(q(e'_i))$, which by definition of T is equal to $T(q_i^{(s_{k'})})$. Observe that v_k is explored before $v_{k'}$ by p_i. p_i does not leave node v_k before all the eggs of the node (an egg e_j for each $p_j \in s_k$) have been written in the array $EGG(v_k)$. In particular, the egg e_i has been written in $EGG(v_k)[i]$ when p_i leaves node v_k. Since the write of an egg always follows its generation, we obtain $T(q_i^{(s_k)}) = RT(q(e_i)) < RT(q(e'_i)) = T(q_i^{(s_{k'})})$.
- $m_i(state(v_k)) = e'_i$. In this case, the egg e'_i written by p_i in $v_{k'}$ had been generated previously. Let u' be the node in which e'_i is generated. This means that, while exploring node u', p_i queries its failure detector module and creates the egg e'_i. Let $u = osa(u', p_i)$. We then have $m_i(state(u)) = e'_i$ (Line 25). As e'_i is written in $v_{k'}$ and $v_k = osa(v_{k'}, p_i)$, $state(u) = state(v_k)$.
  Let $u_0 = osa(u, p_i)$ and $v_{k0} = osa(v_k, p_i)$. Observe that, if $u_0 \neq root$ then $state(u)[i] = \langle state(u_0) \rangle$ and $state(v_k)[i] = \langle state(v_{k0}) \rangle$ and consequently, $state(u_0) = state(v_{k0})$. Otherwise, $u_0 = v_{k0} = root$, and trivially $state(u_0) = state(v_{k0})$. Then, as e_i is written by p_i in $EGG(v_k)[i]$, we must have $m_i(state(v_{k0})) = e_i$, from which it follows that e_i is also written by p_i in u.

Consider the path $u \rightsquigarrow u'$. $u = osa(u', p_i)$, the eggs written by p_i in u and u' are respectively e_i and e'_i, and e'_i is generated while p_i is exploring node u'. Hence, the previous case applies, from which we obtain $RT(q(e_i)) < RT(q(e'_i))$, which implies by definition of T that $T(q_i^{(s_k)}) < T(q_i^{(s_{k'})})$.

To complete the proof, suppose that p_i appears in the concurrency classes $s_k = s_{k_1}$, $s_{k_2}, \ldots, s_{k_x} = s_{k'}$. We have shown that, for every $j \in \{1, \ldots, x-1\}$, $T(q_i^{(s_{k_j})}) < T(q_i^{(s_{k_{j+1}})})$, from which we conclude that $T(q_i^{(s_k)}) < T(q_i^{(s_{k'})})$. $\square_{Lemma\ 4}$

Lemma 5. *Let s_k and $s_{k'}$ be two concurrency classes in $e\alpha$ such that (1) $k < k'$, (2) $p_i \in s_k \cap s_{k'}$ and (3) $p_i \notin s_y$, $\forall y : k < y < k'$. Then, for every $x \in \{1, \ldots, k\}$, $T(\ell q(s_x)) < T(q_i^{(s_{k'})})$.*

Proof Consider any $x' \in \{1, \ldots, k\}$ and any $p_j \in s_{x'}$, and let e_j be the egg written in $EGG(v_{x'})[j]$ by p_j. We first assume that there is no step of p_j in the path $v_{x'} \rightsquigarrow v_k$, i.e., p_j does not appear in the concurrency classes $s_{x'+1}, \ldots, s_k$. Let $v_x = osa(v_{x'}, p_j)$. When e_j is written in $EGG(v_{x'})[j]$ by p_j, it has $m_j(state(v_x)) = e_j$.

In the IS execution $s_1, \ldots, s_k$, the last time p_j updates its cell of the shared memory is when it is scheduled in the concurrency class $s_{x'}$. The value written is the latest state of the shared memory, $state(v_x)$, observed by p_j so far. Hence, $state(v_k)[j] = state(v_{x'})[j] = \langle state(v_x) \rangle$ (or $\langle \Box_j \rangle$, if $v_x = root$).

Let e_i and e'_i be the two eggs written by p_i in v_k and $v_{k'}$, respectively. We consider two cases according to when e'_i is generated. By the assumption of the claim, in the path $v_k \rightsquigarrow v_{k'}$, p_i appears only in the last concurrency class $s_{k'}$. Moreover, the label of the incoming arc of v_k contains p_i. Thus, $v_k = osa(v_{k'}, p_i)$. Whether p_i generates or not a new egg while it is exploring node $v_{k'}$ depends on the value of $m_i(state(v_k))$ (Line 23).

- $m_i(state(v_k))$ is undefined, i.e., no egg is associated with the state of node v_k when p_i explores node $v_{k'}$. In that case, p_i queries its failure detector module to generate a new egg e'_i. This occurs at ("real") time $RT(q(e'_i))$, which by definition of T is equal to $T(q_i^{(s_{k'})})$. $v_{x'}$ is explored by p_i before $v_{k'}$. Specifically, p_i does not leave node $v_{x'}$ before all the eggs of node $v_{x'}$ have been written in the array $EGG(v_{x'})$. In particular, the egg e_j has been written in $EGG(v_{x'})[j]$ when p_i leaves node $v_{x'}$. Since the write of an egg always follows its creation, we obtain $T(q_j^{(s_{x'})}) = RT(q(e_j)) < RT(q(e'_i)) = T(q_i^{(s_{k'})})$.
- $m_i(state(v_k)) = e'_i$. In this case, the egg e'_i written by p_i in $v_{k'}$ has been generated before node $v_{k'}$ is explored. Let u' be the node in which e'_i is generated. Let $u = osa(u', p_i)$. We then have $m_i(state(u)) = e'_i$ (Line 25). As e'_i is written in $v_{k'}$ and $v_k = osa(v_{k'}, p_i)$, $state(u) = state(v_k)$. In particular, $state(v_k)[j] = state(v_{x'})[j] = state(u)[j] = \langle state(v_x) \rangle$ if $v_x \neq root$, or $state(v_k)[j] = state(v_{x'})[j] = state(u)[j] = \langle \Box_j \rangle$, if $v_x = root$. In either case, there is a predecessor w of u, whose incoming arc is labeled p_j such that (1) e_j is written in $EGG(w)[j]$ and (2) there is no step by p_j in the path $w \rightsquigarrow u$. Therefore, the previous case applies (by identifying w with $v_{x'}$, u with v_k, and u' with $v_{k'}$). Hence $RT(q(e_j)) < RT(q(e'_i))$, from which we conclude that $T(q_j^{(s_{x'})}) < T(q_i^{(s_{k'})})$.

To complete the proof, suppose that p_j appears in the concurrency classes $s_{x_1}, s_{x_2}, \ldots, s_{x_\ell}$ in the path $v_{x'} \rightsquigarrow v_k$. Per Lemma 4, $T(q_j^{(s_{x'})}) < T(q_j^{(s_{x_1})}) < T(q_j^{(s_{x_2})}) < \ldots < T(q_j^{(s_{x_\ell})})$. Moreover, we have established that $T(q_j^{(s_{x_\ell})}) < T(q_i^{(s_{k'})})$. Hence $T(q_j^{(s_{x'})}) < T(q_i^{(s_{k'})})$. $\square_{Lemma\ 5}$

Lemma 6. *Let e and e' be two eggs, $q(e)$ and $q(e')$ the (real) failure detector queries that give rise to them. $|RT(q(e)) - RT(q(e'))| > \delta$.*

Proof All processes explore the nodes of the tree in the same order and, for a given node v with incoming arc labeled s, the eggs in $EGG(v)$ are read in increasing order of the IDs in s. Let e be created by p_i when exploring node v, and e' be created by p_j when exploring v'. W.l.o.g., assume that, in the exploration order, v precedes v', or $i < j$ if $v = v'$. Then, p_j must wait to read $EGG(v)[i] \neq \perp$ before generating egg e'. Since e is created, at time $RT(q(e))$ (Line 25), until p_i writes it in $EGG(v)[i]$ (Line 27), there are δ dummy read events (Line 26). Hence, the time at which p_j reads $EGG(v)[i] \neq \perp$ (and hence the time $RT(q(e'))$ when it generates e') cannot be less than $RT(q(e)) + \delta + 1$. $\square_{Lemma\ 6}$

Lemma 7. *Let s_k be the kth concurrency class in $e\alpha$. Then, $T(r^{(s_k)}) = T(\ell q(s_\ell)) + 2(k - \ell + 1)$ for some $\ell \in \{k - n + 1, \ldots, k\}$.*

Proof Consider any $x \leq k$. By definition of T, we have

$$\begin{aligned} T(r^{(s_x)}) &= \max(T(w_j^{(s_x)}) : p_j \in s_x) + 1 \\ &= \max(\{T(q_j^{(s_x)}) : p_j \in s_x\} \cup \{T(r^{(s_{x-1})})\}) + 2 \\ &= \max(T(\ell q(s_x)), T(r^{(s_{x-1})}) + 2. \end{aligned}$$

Let ℓ be the largest integer such that $\ell \leq k$ and $T(\ell q(s_\ell)) \geq T(r^{(s_{\ell-1})})$. ($\ell$ is well defined, as by convention we assume $T(r^{(s_0)}) = 0$.) Hence, for every $x \in \{\ell + 1, \ldots, k\}$, $T(r^{(s_x)}) = T(r^{(s_{x-1})}) + 2$. Consequently,

$$T(r^{(s_x)}) = T(r^{(s_\ell)}) + 2(x - \ell) = T(\ell q(s_\ell)) + 2(x - \ell + 1), \tag{1}$$

and, in particular, $T(r^{(s_k)}) = T(\ell q(s_\ell)) + 2(k - \ell + 1)$.

It remains to show that $k - \ell \leq n - 1$. Assume for contradiction that $k - \ell > n - 1$. Then the sequence $s_\ell, \ldots, s_k$ consists of at least $n + 1$ concurrency classes. By the pigeonhole principle, there exists a process p_i and two concurrency classes s_x and $s_{x'}$, $\ell \leq x < x' \leq k$, such that $p_i \in s_x \cap s_{x'}$, $p_i \notin s_y, \forall y : x < y < x'$, and $x' - \ell \leq n$. Per Lemma 5, it follows that $\forall y \leq x, T(\ell q(s_y)) < T(q_i^{(s_{x'})})$. In particular, $T(\ell q(s_\ell)) < T(\ell q(s_{x'}))$. Observe that, by definition of T and $\ell q()$, there exists two distinct eggs e and e' such that $T(\ell q(s_\ell)) = RT(e)$ and $T(\ell q(s_{x'})) = RT(e')$. It thus follows from Lemma 6 that $T(\ell q(s_{x'})) > T(\ell q(s_\ell)) + \delta$.

On the other hand, it follows from Equation (1) that $T(r^{(s_{x'})}) = T(\ell q(s_\ell)) + 2(x' - \ell + 1)$. However, according to the definition of T, $T(r^{(s_{x'})}) \geq T(\ell q(s_{x'})) + 2$. Hence,

we must have $T(\ell q(s_\ell)) + 2(x' - \ell + 1) \geq T(\ell q(s_{x'})) + 2 > T(\ell q(s_\ell)) + \delta + 2$. Then, as $x' - \ell \leq n$, the inequality becomes $\delta < 2(x' - \ell) \leq 2n$. However, $\delta = 2n$, which leads to a contradiction. $\square_{Lemma\ 7}$

We now show that, when a (simulated) event ev is performed by some process p_i, this process is still alive according to T and F.

Lemma 8. *Let ev be an event in $e\alpha$. Then, $ev.id \notin F(T(ev))$.*

Proof Suppose that ev is performed by p_i while it is scheduled in the concurrency class s. Assume that ev is the event $q_i^{(s)}$ or $w_i^{(s)}$ [1]. The key to the proof is to establish that $T(ev) \leq RT(rev)$ where rev is a real event performed by p_i in the underlying real execution. Since the real execution is D-legal, and $F(\tau) \subseteq F(RT(rev))$ for every $\tau \leq RT(rev)$, it follows that $p_i \notin F(T(ev))$.

Assume that $s = s_k$, and the path in the e-tree to be $root \xrightarrow{s_1} v_1 \xrightarrow{s_2} v_2 \ldots \xrightarrow{s_k} v_k$. Let e_i be the egg written by p_i in $EGG(v_k)[i]$. In the real execution, the event in which this occurs is denoted $w(e_i)$. Let $RT(w(e_i))$ be the (real) timing associated with this event. By definition of T, $T(q_i^{(s_k)}) < T(w_i^{(s_k)})$. Thus, it is enough to show that $T(w_i^{(s_k)}) \leq RT(w(e_i))$.

We have $T(w_i^{(s_k)}) = \max(RT(q(e_i)), T(r^{(s_{k-1})})) + 1$. First assume that $RT(q(e_i)) \geq T(r^{(s_{k-1})})$ (this is the case if $k = 1$, since $T(r^{(s_0)}) = 0$, by convention), which implies that $T(w_i^{(s_k)}) = RT(q(e_i)) + 1$. As e_i is generated before being written in $EGG(v_k)[i]$, $RT(q(e_i)) < RT(w(e_i))$, from which we obtain $T(w_i^{(s_k)}) \leq RT(w(e_i))$. Assume now that $T(w_i^{(s_k)}) = T(r^{(s_{k-1})}) + 1$. Per Lemma 7, there exists ℓ such that $T(r^{(s_{k-1})}) = T(\ell q(s_\ell)) + 2(k - \ell)$ where $k - \ell \leq n$. Hence, from the definition of T,

$$T(w_i^{(s_k)}) = T(\ell q(s_\ell)) + 2(k - \ell) + 1. \tag{2}$$

We now compute the earliest time mapped by RT to the real event $w(e_i)$. The exploration of each node v_x, $x \leq k$, generates at least $\delta + 1$ consecutive events at each process $p_j \in s_x$: δ dummy reads and a write of the shared variable $EGG(v_x)[j]$ (lines 26-27). In addition, each process waits until an egg for each process in s_x has been written in $EGG(v_x)$. Moreover, RT defines a linearization of the real execution. The timing of the events in RT respects the real-time order of those events. Therefore the linearization time $RT(w(e_i))$ for the event $w(e_i)$ is at least

$$RT(w(e_i)) \geq T(\ell q(s_\ell)) + 1 + (\delta+1)(k-1-\ell) + \delta + 1 = T(\ell q(s_\ell)) + (k-\ell)(\delta+1) + 1 \tag{3}$$

Where $T(\ell q(s_\ell)) + 1$ is the earliest time at which the last egg is written in $EGG(v_\ell)$, there are at least $\delta + 1$ events in the exploration of each node $v_{\ell+1}, \ldots, v_{k-1}$, there are δ events in the linearization of the dummy reads performed by p_i while exploring v_k, and $w(e_i)$ is an additional event. As $\delta + 1 > 2$, it follows from equations 2 and 3 that $RT(w(e_i)) > T(w_i^{(s_k)})$. $\square_{Lemma\ 8}$

[1] Recall that we have removed from execution $e\alpha$ the last event of a faulty process p_i if that event was a read $r_i^{(s)}$. Hence, we only need to consider query and write events.

Lemma 9. *Let $ev = q_i^{(s)}$ be a failure detector query in $e\alpha$, and d the value returned by that query. Then, $d = H(p_i, T(q_i^{(s)}))$.*

Proof Let v be the node whose incoming arc is labeled s. Let $e_i(v) = \langle p_i, d, k\rangle$ be the egg written by p_i in $EGG(v)[i]$. d is the failure detector value returned by the simulated query $q_i^{(s)}$. By definition of T, $T(q_i^{(s)}) = RT(q(e_i(v)))$, the time at which the real query that gives rise to d occurs. By the D-legality of the real execution, $d = H(p_i, RT(q(e_i))) = H(p_i, T(q_i^{(s)}))$. $\Box_{Lemma\ 9}$

The following properties directly follow from the definition of T.

Lemma 10. *Let s and s' be two consecutive concurrency classes in $e\alpha$.*
(a) For every $p_i \in s$, $T(q_i^{(s)}) < T(w_i^{(s)}) < T(r_i^{(s)})$,
(b) For every $p_i, p_j \in s$, $T(w_i^{(s)}) < T(r_j^{(s)})$ and,
(c) For every $p_i \in s, p_j \in s'$, $T(r^{(s)}) < T(w_j^{(s')})$.

Proof These properties directly follow from the definition of T. $\Box_{Lemma\ 10}$

Lemma 11. *Let s and s' be two concurrency classes in $e\alpha$ such that s appears before s'. For every $p_i \in s \cap s'$, $T(r_i^{(s)}) < T(q_i^{(s')})$.*

Proof Suppose that $s = s_k$ and $s' = s_{k'}$, the kth and k'th concurrency classes in $e\alpha$ respectively. Let us first assume that $\forall x, k < x < k', p_i \notin s_x$. Per Lemma 7, there exists ℓ, such that (1) $\ell \leq k$, (2) $k - \ell \leq n-1$, and (3) $T(r^{(s_k)}) = T(\ell q(s_\ell)) + 2(k-\ell+1)$. It follows from (1) and Lemma 5 that $T(\ell q(s_\ell)) < T(q_i^{(s_{k'})})$. Therefore, by Lemma 6, we derive that (4) $T(\ell q(s_\ell)) + \delta < T(q_i^{(s_{k'})})$. On the other hand, from (2) and the value of δ we have that $2(k-\ell+1) \leq 2n = \delta$. This and (3) imply that $T(r^{(s_k)}) \leq T(\ell q(s_\ell)) + \delta$. Combining this with (4), we conclude that $T(r^{(s_k)}) < T(q_i^{(s_{k'})})$.

Suppose that p_i appears in the concurrency classes $s = t_1, t_2, \ldots, t_x = s'$. By Lemma 10.(b) and (c), $T(r^{(s)}) < T(r^{(t_{x-1})})$. Moreover, we have established that $T(r^{(t_{x-1})}) < T(q_i^{(t_x)}) = T(q_i^{(s)})$. Hence $T(r^{(s)}) < T(q_i^{(s')})$. $\Box_{Lemma\ 11}$

Finally, we prove Lemma 3.

Proof (Lemma 3) We check that the failure pattern F, the failure detector history H, and the timing T are consistent with $e\alpha$ in the sense of Definition 1. An execution σ is D-legal if there exists a failure pattern F, an history $H \in D(F)$ and a function T that maps each event of σ to an integer such that ($T(e)$ is the "time" at which the event e occurs). We show that $e\alpha$ is D-legal.
1. $\forall ev \in \sigma, ev.id \notin F(T(ev))$. This property follows from Lemma 8.
2. $\forall ev \in \sigma, ev.op = \mathsf{fd\_query}() \implies H(ev.id, T(ev)) = ev.out$. This property is proved in Lemma 9.
3. $\forall s \in \sigma, \forall p_i \in s, T(q_i^{(s)}) < T(w_i^{(s)}) < T(r_i^{(s)})$. Proved as Lemma 10.(a).
4. $\forall s_k, s_\ell \in \sigma : k < \ell, \forall p_i \in s_k \cap s_\ell, T(r_i^{(s_k)}) < T(q_i^{(s_\ell)})$. Proved as Lemma 11.
5. $\forall s \in \sigma, \forall p_i, p_j \in s, T(w_i^s) < T(r_j^s)$. Proved as Lemma 10.(b).

6. $\forall s_k, s_{k+1} \in \sigma, \forall p_i \in s_k, \forall p_j \in s_{k+1}, T(r_i^{(s_k)}) < T(w_j^{(s_{k+1})})$. Proved as Lemma 10.(c). $\square_{Lemma\ 3}$

Lemma 12. *Let us assume the algorithm of Figure 3 explores an infinite branch* $b = root \xrightarrow{s_1} v_1 \xrightarrow{s_2} v_2 \ldots$ *Let* $\mathcal{C}$ *be the set of processes that appear infinite number of times in* b*. There is a* k' *such that* $\forall k > k'$*, (1) the algorithm executes* $\mathsf{explore}(v_k, s_k, \mathcal{C})$ *and (2) this call never terminates.*

Proof Let $s_{k'}$ be the last concurrency class in b with processes not in $\mathcal{C}$. Then, for all $k > k', s_k \subseteq \mathcal{C}$. We first show that the algorithm executes $\mathsf{explore}(v_k, s_k, \mathcal{C})$. Assume this call never happens. Since the algorithm explores the branch b, it explores node v_k, and hence it must execute calls of the form $\mathsf{explore}(v_k, s_k, \mathcal{C}')$ with $\mathcal{C}' \neq \mathcal{C}$. Observe that $\mathcal{C} \not\subset \mathcal{C}'$, from the order used in the algorithm (Line 14). Then, there is some process $p_i \in \mathcal{C}$ that does not appear in $\mathcal{C}'$. Since the subtree rooted at v_k explored by call $\mathsf{explore}(v_k, s_k, \mathcal{C}')$ only contain nodes that can be reached from v_k with concurrency classes $s \subseteq \mathcal{C}'$, branch b has not been completely explored in any of these calls. Since the collection of possible sets $\mathcal{C}'$ is finite, the branch has not been fully explored, and hence call $\mathsf{explore}(v_k, s_k, \mathcal{C})$ is executed. Now, the subtree explored by call $\mathsf{explore}(v_k, s_k, \mathcal{C})$ contains the subtree rooted at v_k that can be reached with paths of concurrency classes $s \subseteq \mathcal{C}$. Then, the infinite branch of that tree $v_k \xrightarrow{s_{k+1}} v_{k+1} \xrightarrow{s_{k+2}} v_{k+2} \ldots$ is explored. As part of the exploration, calls $\mathsf{explore}(v_x, s_x, \mathcal{C})$ are executed, $\forall x > k$. Since there are infinite such calls, the exploration never finishes and the call never terminates. $\square_{Lemma\ 12}$

Extraction of $\mathrm{anti}{-}\Omega$ Finally, we show that the algorithm described in Figure 3 implements an $\mathrm{anti}{-}\Omega$ failure detector.

Lemma 13. *Let* T *a task with finite inputs that has no wait-free solution. Let* D *a failure detector and* $\mathcal{A}_D$ *a protocol in standard form using* D *to solve* T*. The algorithm described in Figure 3 implements* $\mathrm{anti}{-}\Omega$ *using* D *in a wait-free environment.*

Proof Consider an execution of the algorithm described in Figure 3. Let $\mathcal{C}$ be the set of correct processes in that execution. We have to show that there exists a correct process p_c such that eventually, for every $p_i \in \mathcal{C}$, $\text{FD}_i \neq p_c$ permanently. Let $p_i \in \mathcal{C}$. There are two main cases.

- Some invocation of $\mathsf{explore}()$ issued by p_i at Line 08 never terminates. This can only happen if (1) p_i never exits the **wait until** loop at Line 28 while exploring some node v, or (2) p_i explores an infinite branch.
  (1) There exists v such that p_i waits forever for an egg from p_ℓ while exploring node v. Observe that every correct process p_j eventually reaches node v, and then eventually waits for an egg from p_ℓ. This follows from the fact that processes explore the tree in the same order starting from the root. Moreover, when a process arrives at a node w along an arc labeled s, it waits until an egg is laid by each process $p_x \in s$, one by one in the order of their ids in s.

Therefore, p_ℓ cannot be a correct process. Otherwise, it would eventually explore node v and assign an egg to v, contradicting the fact that p_i waits forever for the egg of p_ℓ to be assigned to v. Finally, while waiting for an egg from p_ℓ, the $\mathrm{anti}{-}\Omega$ output FD_i at p_i is $\{p_\ell\}$. Hence, eventually, for every correct process, the output of $\mathrm{anti}{-}\Omega$ is forever $\{p_\ell\}$ which is a valid output as $p_\ell \notin \mathcal{C}$.

(2) For every node v, no process is blocked in the **wait until** loop while exploring node v. Since the invocation of $\mathsf{explore}()$ does not terminate, this implies that an infinite branch b is produced in the e-tree. Let $\mathcal{C}'$ be the set of processes that appear infinitely often in the labels of the branch b. Clearly, $\mathcal{C}' \subseteq \mathcal{C}$, since otherwise at some point all the eggs assigned by a faulty process $p_\ell \in \mathcal{C}'$ to nodes in b would be used, and case (1) would occur. Suppose for contradiction that $\mathcal{C} = \mathcal{C}'$. Let $p_i \in \mathcal{C}'$. There exists an input vector $I \in \mathcal{I}$ such that p_i does not decide in the execution $e\alpha$ that corresponds to b when the input vector is I. From $\mathcal{C} = \mathcal{C}'$ it follows that $e\alpha$ is a fair execution. Moreover, by Lemma 3, $e\alpha$ is D-legal. Thus, $e\alpha$ is a possible legal and fair execution of $\mathcal{A}_D$ using D when the input vector is I. Then, every process in $\mathcal{C}' = \mathcal{C}$ should eventually decide in $e\alpha$, a contradiction. Hence, $\mathcal{C}' \subset \mathcal{C}$.
Moreover, it follows from Lemma 12 that there exists a call of the form $\mathsf{explore}(v, s, \mathcal{C}')$, where v is a node in the infinite branch, that does not terminate. Every call to $\mathsf{explore}(-, -, S)$ that follows it are such that $S \subseteq \mathcal{C}'$. So after this call, only arcs with label $s \subseteq \mathcal{C}'$ are traversed. Since in a node v the output of FD_i can only be processes of s, the label of the incoming arc of v, FD_i at each process p_i is permanently contained in $\mathcal{C}'$.
Therefore each correct process in $\mathcal{C} - \mathcal{C}'$ is eventually never output by the emulated failure detector, satisfying the specification of $\mathrm{anti}{-}\Omega$.

– All invocations of $\mathsf{explore}()$ issued by p_i at Line 08 eventually terminate. In that case, the e-tree is finite, and all eggs used in the e-tree (the array EGG) are also finite.
  - Consider a path in the e-tree and an input vector I. The corresponding execution is D-legal (Lemma 3), thus all decisions made along this path respect the specification of the task T.
  - Let I be an input vectors, and b and b' be two branches in the e-tree. Denote $e\alpha$ and $e\alpha'$ the corresponding IS execution of $\mathcal{A}$ using D, and α and α' the two corresponding IS executions (removing failure detector values). Suppose that p_i decides in $e\alpha$ and $\alpha \overset{p_i}{\sim} \alpha'$. Per Lemma 2 $e\alpha \overset{p_i}{\sim} e\alpha'$. So p_i decides also in $e\alpha'$ and decides the same value as in $e\alpha$.
  - As the exploration procedure backtracks only when it reaches a node in which every process decides for every input vector, for every branch in the e-tree, for every process p_i, for every input vector I, there exists a node v in the branch in which p_i decides; i.e., $p_i \notin \mathit{undecided}_i(v, I)$

The three properties above imply the existence of a wait-free protocol $\mathcal{A}$ without a failure detector, that instead uses the e-tree and the eggs as part of its initial state and solves T: a contradiction.
The protocol works as follows. Each process has the e-tree built by the algorithm of Figure 3 and the associated array EGG as local initial data. Then a process p_i executing $\mathcal{A}$ simply executes the protocol $\mathcal{A}_D$ that solves T with a failure detector

D. p_i follows the appropriate path in the tree, using the eggs taken from the array instead of querying the failure detector.
Each time p_i reads a snapshot from the shared memory, it can identify which path in the tree it is following. Unfortunately, this snapshot may correspond to several paths. As we have shown, the egg laying mechanism of Figure 3 guarantees that along each possible path the same egg is used, and hence p_i can continue emulating $\mathcal{A}$.
When p_i decides on input I, it stops emulating $\mathcal{A}$ and modifying the shared memory. This assumption allows the processes that have not decided yet, to follow a path in the subtree whose root is the node of the e-tree at which p_i decided.

$\square_{Lemma\ 13}$

4 t-resilient Case

This section establishes that, in the t-resilient environment, $\mathrm{anti}{-}\Omega^t$ is necessary to solve any unsolvable *agreement* task $\mathcal{T}$. More precisely, given an agreement task $\mathcal{T}$, a failure detector D, and an algorithm $\mathcal{A}_D$ that solves $\mathcal{T}$ using D in the t-resilient environment, we show that, if no read/write protocol solves $\mathcal{T}$ in the t-resilient environment, then D is at least as strong as $\mathrm{anti}{-}\Omega^t$.

Recall that a task $\mathcal{T}$ is specified by an input domain $\mathcal{I}$, an output domain $\mathcal{O}$, and a relation Δ that specifies, for each assignment of inputs to the processes, on which outputs processes can decide. For this section, we consider only agreement tasks: for any input $I \in \mathcal{I}$, if a value can be decided by p_i, it can also be decided by any other process p_j. Formally,

Definition 6 (agreement task). *A task $(\Delta, \mathcal{I}, \mathcal{O})$ is an agreement task if $\forall I \in \mathcal{I}$, $\forall O \in \mathcal{O}$, $(I, O) \in \Delta \implies (I, O') \in \Delta, \forall O' = [O[i_1], \ldots, O[i_n]], \forall i_1, \ldots, i_n \in \{1, \ldots, n\}$.*

The k-set agreement task is an example of an agreement task, and there are many other examples, such as loop agreement and convergence tasks [20,6]. The following is the main result of this section.

Theorem 2. *Let $\mathcal{T}$ be an agreement task that cannot be solved in the t-resilient environment. Let D be a failure detector and $\mathcal{A}_D$ a protocol that solves $\mathcal{T}$ using D in the t-resilient environment. Then, D is at least as strong as $\mathrm{anti}{-}\Omega^t$ in the t-resilient environment.*

In [28], Zieliński describes an algorithm based on $\mathrm{anti}{-}\Omega$ that solves $(n-1)$-set-agreement in the wait-free environment. It is easy to generalize it, and derive from his construction an algorithm, based on $\mathrm{anti}{-}\Omega^k$, that solves k-set-agreement (see Section A). As t-set-agreement is an agreement task with no read/write protocol in the t-resilient environment, Theorem 2 implies the following corollary.

Corollary 1. *In the t-resilient environment, $\mathrm{anti}{-}\Omega^t$ is the weakest failure detector to solve t-set-agreement.*

Overview of the proof As first proved by Borowsky and Gafni [5], task solvability in read/write executions in which at most t processes fail can be captured by a subset of all possible IS executions. We define the set of t-resilient IS executions (abbreviated in t-IS) as a particular subset of all IS executions and establish that, if there exists a protocol that solves $\mathcal{T}$ in every t-IS execution, a protocol that solves $\mathcal{T}$ in every execution in which at most t processes fail can be inferred. Our definition of the set of t-IS executions differs from the one in [5], but we believe it is simpler to analyze.

As in the wait-free case, the extraction protocol $\mathcal{A}_{D\rightarrow \mathrm{anti}-\Omega^t}$ simulates possible t-IS executions of $\mathcal{A}_D$ using D. We reuse the extraction protocol of the wait-free case (Figure 3), with small modifications. As before, simulated executions are organized in an e-tree, built by the processes while executing the extraction protocol. Failure detector queries are simulated by invoking the egg_laying() procedure. If the modified extraction protocol finishes, it yields a finite subtree of the wait-free e-tree. As shown for the wait-free case, each path in the e-tree describes a legal execution of $\mathcal{A}_D$. Also, the procedure preserves indistinguishability with respect to executions without failure detector. Both properties allow to infer that, if the extraction protocol finishes, the obtained e-tree can be used to create a protocol $\mathcal{A}$, without failure detector, such that, in each IS execution that corresponds to a path from the root in the e-tree, some set of processes decide respecting the specification of $\mathcal{T}$. The extraction algorithm for the wait-free case (Figure 3) is modified as follows:

1. The output of the simulated failure detector at process p_i is now a set of $n-t$ process ids. FD_i contains the set of the latest $n-t$ processes simulated by p_i, including the process p_i is currently simulating.
2. Instead of simulating an e-tree that includes all possible IS executions, processes simulate a sub-tree that is *complete for the set of all possible t-IS executions*. In this sub-tree, every path corresponds to a t-IS execution and, reciprocally, every t-IS execution can be associated with a unique path in the sub-tree. As mentioned, the two main properties of the construction in Section 3, namely indistinguishability and D-legality are preserved. (Indeed, the correctness of Lemmas 2 and 3 rely on the egg_laying procedure and the fact that processes traverse the tree in the same order. They do not depend on which particular sub-tree is explored.)

The proof of correctness closely follows the proof of the wait-free case. There are three cases. (a) Processes may be blocked forever in the egg_laying() procedure waiting for an egg from a faulty process p_j. In that case, the simulated output of $\mathrm{anti}-\Omega^t$ eventually stabilizes on a set of $n-t$ processes that includes faulty processes. This is a valid output, since there are at least $n-t$ correct processes. Alternatively, (b) processes may simulate an infinite t-IS execution $e\alpha$ of $\mathcal{A}_D$ in which some process takes infinitely many steps, but never decides. Then, $e\alpha$ must be unfair, i.e., the set S of processes that take infinitely many steps in $e\alpha$ excludes a correct process. By construction, every infinite t-IS execution has at least $n-t$ processes that take infinitely many steps. When a new step of $e\alpha$ is simulated, the output of $\mathrm{anti}-\Omega^t$ is the set of the $n-t$ processes whose steps were last simulated. Therefore, the output of $\mathrm{anti}-\Omega^t$ is eventually always included in S, which is valid since S excludes a correct process. Finally, if none of the previous cases occur, then (c) a finite e-tree with all possible t-IS executions is generated, in which at least $t+1$ processes decide in every leaf, for every input vector

allowed by $\mathcal{T}$. Since $\mathcal{T}$ is an agreement task, and t-IS executions can be simulated in the read/write model, we derive in this case a t-resilient protocol $\mathcal{A}$ that solves $\mathcal{T}$ without using failure detectors: a contradiction.

Capturing t-resilience into IS executions Given a sequence of concurrency classes $s_1, \ldots, s_k$, let $\#_{p_i}(s_1, \ldots, s_k)$ denote the number of concurrency classes in which p_i appears: $\#_{p_i}(s_1, \ldots, s_k) = |\{x \in \{1, \ldots, k\} : p_i \in s_x\}|$. An IS execution α is a *t-resilient IS execution* (t-IS for short) if it satisfies the following property.

Definition 7 (t-joint progress). *Let $\alpha = s_1, s_2, \ldots$ be a finite or infinite sequence of concurrency classes. α satisfies the* t-joint progress property *if* $\forall s_k \in \alpha$, $\forall p_i \in \Pi$, $\forall r \geq 0$, $((p_i \in s_k) \wedge (\#_{p_i}(s_1, \ldots, s_k) = r) \wedge ((r \mod n) + 1 = i)) \implies (\exists S \subseteq \Pi : (|S| \geq n - t - 1) \wedge (\forall p_j \in S, \#_{p_j}(s_1, \ldots, s_{k-1}) \geq r))$.

Let p_i be a process and r such that $(r \mod n) + 1 = i$. Assume that the rth class in which p_i appears is s_k. In other words, $p_i \in s_k$ and p_i appears exactly r times in the sets $s_1, \ldots, s_k$. The t-joint progress property specifies that there exists a set S of at least $n - t - 1$ processes such that each process $p_j \in S$ appears at least r times in the prefix $s_1, \ldots, s_{k-1}$. (observe that $p_i \notin S$.) Notice that this implies that, in every infinite t-IS execution, at least $n - t$ processes appear infinitely often. The next lemma shows that, for every choice of $S \subseteq \Pi$ of size $n - t$, and every finite t-IS execution α, there exists an extension of α in which all concurrency clases are contained in S (only processes in S take steps). Given a sequence $\alpha = s_1, \ldots, s_k$, let $\alpha.s$ denote the sequence $s_1, \ldots, s_k, s$.

Lemma 14. *Let $\alpha = s_1, \ldots, s_k$ be a finite t-IS execution. There exists $S \subseteq \Pi$ such that (1) $|S| \geq t + 1$ and (2) $\forall s \subseteq S, \alpha.s$ is a t-IS execution.*

Proof Let $E = \{p_i : i \neq (\#_{p_i}(\alpha)+1) \mod n+1\}$ and $E' = \{p_i : (i = (\#_{p_i}(\alpha)+1) \mod n+1) \wedge (\exists\, P_i \in \Pi : (|P_i| = n-t-1) \wedge (\forall p_j \in P_i, \#_{p_j}(\alpha) \geq \#_{p_i}(\alpha)+1))\}$. Let us denote $G = E \cup E'$. Observe that α can be extended with any subset $s \subseteq G$ without violating the t-joint progress property. In fact, the property imposes no constraints on the $(\#_{p_i}(\alpha) + 1)$th occurrence of p_i, for every $p_i \in E$. Moreover, for every $p_i \in E'$, one can find the set P_i of $n - t - 1$ processes, where each $p_j \in P_i$ appears at least $(\#_{p_i}(\alpha) + 1)$ times in α. (Observe that $p_i \notin P_i$.) We prove the lemma by establishing that $|G| \geq t + 1$.

Assume for contradiction that $|G| = x < t + 1$. Let $p_i \notin G$. Notice that, as $p_i \notin E$, $(\#_{p_i}(\alpha)+1) \mod n+1 = i$ and thus no other process $p_j \in \Pi \backslash G$ has the same number of occurrences $\#_{p_j}(\alpha)$ in α. Let L denote the $t + 1 - x$ processes in $\Pi \backslash G$ with the smallest number of occurrences in α, and $H = \Pi \backslash (G \cup L)$. That is, $\forall p_i \in L, p_j \in H : \#_{p_i}(\alpha) < \#_{p_j}(\alpha)$. By definition of H, we have $|H| = n-(x+(t+1-x)) = n-(t+1)$. Therefore, for every process $p_i \in L$, there exists a set $P_i = H$ of $n - t - 1$ processes, each of them appears at least $(\#_{p_i}(\alpha) + 1)$ times in α. Consequently, $L \subseteq E' \subseteq G$, contradicting the fact that $p_i \notin G$. $\square_{Lemma\ 14}$

The next theorem shows that the set of all t-IS executions captures the read/write model in the t-resilient environment, in the following sense: if a protocol $\mathcal{B}$ solves some task $\mathcal{T}$ within the set of all possible t-IS executions, then there exists a read/write protocol that solves $\mathcal{T}$ in the t-resilient environment.

Theorem 3. *Let $\mathcal{B}$ a t-resilient protocol in standard form that solves $\mathcal{T}$ in every infinite t-IS execution. There exists a protocol that solves $\mathcal{T}$ in every executions allowed by the t-resilient environment.*

To prove the theorem, we present a t-resilient simulation described in Figure 4. Each run of the simulation, in which at least $n - t$ processes take steps forever produces an infinite t-IS executions α. The sets of processes that appear infinitely often in α is the set of processes that take infinitely many steps in the simulation protocol.

The simulated Protocol $\mathcal{B}$ is in standard form. To simulate each step of the **repeat** loop, which consists of a write followed by a snapshot of the memory (see Figure 1), process p_i invokes the write_snapshot() operation provided by the simulation, with as parameter the value v it has to write according to $\mathcal{B}$. This operation returns an immediate snapshot [5] (a snapshot that includes v), with the illusion that the write operation has been immediately followed by a snapshot operation. The simulation relies on an underlying immediate snapshot object OIS. Such an object can be implemented wait-free from atomic registers [3,5]. It exports one operation denoted WRITE_SNAPSHOT() that ensures the following properties. W.l.o.g., let us assume that the parameter of the rth invocation of WRITE_SNAPSHOT() performed by p_j is a tuple $\langle j, r, data\rangle$ for every p_j and every r. The invocation returns a set σ_i^r of tuples $\langle j, r_j, data_j\rangle$ such that

1. (self inclusion) $\langle i, r, \_\rangle \in \sigma_i^r$
2. (containment) $\forall p_i, p_j, r, r' : (\sigma_i^r \subseteq \sigma_j^{r'}) \vee (\sigma_j^{r'} \subseteq \sigma_i^r)$
3. (immediacy) $\forall p_i, p_j, r, r' : \langle j, r', \_\rangle \in \sigma_i^r \Rightarrow \sigma_j^{r'} \subseteq \sigma_i^r$

Observe that value returned by an invocation of write_snapshot is provided by an invocation of WRITE_SNAPSHOT(). It thus follows from these properties that the processes simulate an IS execution α of $\mathcal{B}$. More precisely, by the containment property, all sets σ_i^r are related by containment. Let $\sigma = \sigma(1), \sigma(2), \ldots$ be the ordered sequence that contains all these sets (i.e., for every i, $\sigma(i) \subset \sigma(i+1)$). The concurrency classes of α are defined as follows: $s_1 = \{p_i : \langle i, \_, \_\rangle \in \sigma(1)\}$, and for every $k > 1, s_k = \{p_i : \langle i, \_, \_\rangle \in \sigma(k) \backslash \sigma(k-1)\}$. Per the immediacy and self inclusion properties, each process active in s_k obtains the same snapshot σ_k, which contains the value written by every $p_i \in s_{k'}$ if p_i is active in $s_{k'}$, for every $k' \leq k$.

To ensure the t-joint progress property, and thus to simulate t-IS executions, a process is allowed to access the underlying OIS only after "enough" processes have performed "enough" WRITE_SNAPSHOT() operations. Consider the rth operation performed by p_i and assume that $(r \mod n)+1 = i$. Before invoking WRITE_SNAPSHOT(), p_i waits for a set S of $n-t-1$ processes to complete their rth invocation. This ensures that the snapshot eventually obtained by p_i contains the first r values written by each process in S. Before returning from an invocation of write_snapshot(), process p_j increments a counter $C[j]$ in shared memory. Hence, by repeatedly reading the array C, p_i eventually discovers that $n-t-1$ processes have completed their rth invocation. To prevent an invocation from blocking forever, we assume that (1) at least $n-t$ processes do not fail and (2) every process keeps invoking write_snapshot until it possibly fails.

Proof As discussed previously, a run of the algorithm simulates an IS execution. It remains to show that every simulated IS execution is indeed a t-IS execution. We first

prove that every invocation of write_snapshot() by every correct process terminates. Then we establish that the t-joint progress property is satisfied.

Termination. Every invocation of write_snapshot() by every correct process terminates. The proof is by induction on operation occurences r. Let $1 \leq r$, and let us assume that for every $r' < r$, and for every correct process p_c the r'th invocation of write_snapshot() by p_c terminates. By definition, every "0th" invocation terminates. Let us the consider the rth invocation of write_snapshot() by a correct process p_i.

- $i \neq (r \mod n) + 1$. In that case, p_i directly invokes WRITE_SNAPSHOT() on the underlying immediate snapshot object OIS. This object is wait-free: as p_i does not fail, it eventually gets back a response and then returns.
- $i = (r \mod n) + 1$. Assume for contradiction that p_i is blocked forever in the **wait until** statement. Let us first observe that there are at least $n - t - 1$ correct processes other than p_i. Let p_j be one of them. Since we assume that processes keep invoking write_snapshot, p_j invokes write_snapshot() at least r times. By the induction hypothesis, the first $r - 1$ invocations of p_j terminates. As $j \neq (r \mod n) + 1$, it follows from the first case above that the rth invocation made by p_j terminates as well. Observe that when this occurs, $C[j] = r$. Therefore, eventually at least $n - t - 1$ processes complete each r write_snapshot() operations. Hence, eventually at least $n-t-1$ entries in the shared counter C are greater than or equal to r, from which we conclude that p_i eventually exits the **wait until** statement and then completes its rth write_snapshot() operation.

t-joint progress. Let σ_i^r denote the immediate snapshot returned by the r-th invocation of write_snapshot() by p_i. Let us assume that $(r \mod n) + 1 = i$. Before invoking WRITE_SNAPSHOT() on the underlying immediate snapshot object, p_i has observed $n - t - 1$ values in the shared array C greater than or equal to r. Each process p_j increments its entry only after it has completed an invocation of WRITE_SNAPSHOT(). Hence, the rth invocation of WRITE_SNAPSHOT() issued by p_i is preceded by at least r invocations by p_j, for every $p_j \neq p_i$ contains in a set P_i of at least $n - t - 1$ processes. Observe that $\forall p_j \in P_i, \forall r' \leq r$, $\sigma_j^{r'} \subset \sigma_i^r$. This is because immediate snapshots are related by containment and the first r invocations of write_snapshot() executed by p_j terminate before the rth invocation by p_i starts. Hence, in the simulated IS execution, there are r concurrency classes in which p_j appears that precede the concurrency class containing the rth occurrence of p_i. $\square_{Theorem\ 3}$

Extracting $\mathrm{anti}{-}\Omega^t$ As in the wait-free case we start with a task $\mathcal{T}$. We assume that there is no t-resilient protocol that solves $\mathcal{T}$. However, we assume that there exists a t-resilient protocol $\mathcal{A}_D$ and a failure detector D such that (1) $\mathcal{A}_D$ uses D, (2) $\mathcal{A}_D$ is a t-resilient protocol that solves $\mathcal{T}$ and (3) $\mathcal{A}_D$ is a standard form full information protocol.

The extraction algorithm The algorithm is the same as in the wait free case with the following modifications. Instead of the full tree that represents all possible IS executions, the algorithm explores the subtree that represents all possible t-res executions.

To reflect this change, we modify lines 16 to 18 of Figure 3 (see Figure 5). Processes simulate $\mathcal{A}_D$ along each branch, creating and copying eggs as needed in order.
Moreover, as the goal is to simulate a failure detector $\text{anti}-\Omega^t$, the variable FD_i should now contain a set of $n-t$ processes ids. In the wait-free case FD_i is the process currently simulated by p_i. Here, the output of $\text{anti}-\Omega^t$ at process p_i is the ids of the last $n-t$ distinct processes simulated by p_i. Line 20 in the eggs_laying procedure is modified accordingly (see Figure 6).

Proof of Theorem 2 The following lemma shows that the exploration of the tree cannot terminate, since otherwise an algorithm $\mathcal{A}$ that solves $\mathcal{T}$ without failure detector can be obtained.

Lemma 15. *In every infinite execution of the extraction algorithm described in Figure 3, with the modifications described in Figures 5 and 6, at least one invocation of* explore() *issued by a correct process does not terminate.*

Proof Consider an infinite execution of the extraction algorithm. Assume for contradiction that every invocation of explore() issued by process p_i terminates. As in the proof of Lemma 13, we construct a t-resilient protocol $\mathcal{A}$ that solves $\mathcal{T}$ without using failure detectors, which contradicts the fact that no t-resilient read/write protocol solves $\mathcal{T}$.

As every invocation of explore() terminates, the extraction builds a finite e-tree, in which each node is tagged with eggs. More precisely, for every node v in the tree, $EGG(v)[j] \neq \perp$ for every $p_j \in s$, where s is the label of the incoming arc of v. A path α in the e-tree starting from the root, together with an input vector $I \in \mathcal{I}$, describes a possible t-IS execution $e\alpha$ of $\mathcal{A}_D$, in which initial values are assigned to processes according to I. In this execution, the successive states of the shared memory, each observed by at least one process, are $e\text{-}state(v_1, I), \ldots, e\text{-}state(v_x, I)$, where $v_1, \ldots, v_x$ are the successive nodes in the path. Specifically, as $\mathcal{A}_D$ is a standard form full information protocol, $e\text{-}state(v_k, I)$ is a snapshot obtained by p_i in $e\alpha$, if the label of the incoming arc of v_k contains p_i. To obtain a decision, process p_i applies $\text{decide}_i()$ (see Figure 1) to each snapshot it gets.

Therefore, every node v in the e-tree can be tagged by decision vectors O^I_v by applying $\text{decide}_1(), \ldots, \text{decide}_n()$ to $e\text{-}state(v, I)$, for every $I \in \mathcal{I}$. The sequence of decision vectors $O^I_{v_1}, O^I_{v_2}, \ldots, O^I_{v_x}$, where $v_1, \ldots, v_x$ are the successive nodes in some path of the e-tree can be observed in an execution of $\mathcal{A}_D$ since per Lemma 3, there exists a D-legal execution $e\alpha$ of $\mathcal{A}_D$ with input vector I in which the shared memory states $e\text{-}state(v_1, I)$, $e\text{-}state(v_2, I), \ldots, e\text{-}state(v_x, I)$ are successively observed by some processes. In particular, for every $p_j \in \Pi$ and $\ell, 1 \leq \ell < x$, $O_{v_\ell}[j] \neq \perp \Rightarrow O_{v_{\ell+1}}[j] = O_{v_\ell}[j]$ and $O^I_{v_\ell}$ is a valid output for input I (*Property* $P1$). I.e., $O^I_{v_\ell}$ can be extended into a vector $O \in \Delta(I)$ by replacing each $\perp$ entry by a decision value.

Recall that each node v in the e-tree is associated with $state(v, I)$, for every $I \in \mathcal{I}$. $state(v, I)$ represents a state of the shared memory, as observed by some processes in an IS execution with input vector I of a full-information protocol in standard form not using failure detector. Notice that for every two nodes v, v' in the e-tree whose incoming arcs are labeled s and s' respectively, and for every $I, I' \in \mathcal{I}$, $((s \cap s') \neq \emptyset) \wedge (state(v, I) = state(v', I')) \Rightarrow O^I_v = O^{I'}_{v'}$ (*Property* $P2$). This property stems

from the indistinguishability lemma (Lemma 2). Let $s_1, \ldots, s_k$ and $s'_1, \ldots, s'_{k'}$ be the sequences of labels in the paths $root \rightsquigarrow v$ and $root \rightsquigarrow v'$ respectively. Let α and α' be two IS executions with input I and I' respectively of a standard form full information protocol without failure detector whose concurrency classes are $s_1, \ldots, s_k$ and $s'_1, \ldots, s'_{k'}$. Since $s_k \cap s_{k'} \neq \emptyset$ and $state(v, I) = state(v', I')$, there exists a process $p_j \in s_k \cap s'_{k'}$ such that $\alpha \overset{p_j}{\sim} \alpha'$. Similarly, let $e\alpha$ and $e\alpha'$ be two executions of $\mathcal{A}_D$ with input I and I' described by the path $root \rightsquigarrow v$ and $root \rightsquigarrow v'$, respectively. Per Lemma 2, $e\alpha \overset{p_j}{\sim} e\alpha'$. As $p_j \in s_k \cap s'_{k'}$, we have $e\text{-}state(v, I) = e\text{-}state(v', I')$ because $e\text{-}state(v, I)$ and $e\text{-}state(v', I')$ are the last snapshots observed by p_j in $e\alpha$ and $e\alpha'$ respectively. Hence, $O^I_v = O^{I'}_{v'}$.

A t-resilient protocol $\mathcal{A}$ solving $\mathcal{T}$ without failure detector is described in Figure 7. It consists of two concurrent tasks T1 and T2. Each process p_i is initially provided with a local copy of the (finite) e-tree denoted $e\text{-tree}_i$. In the first task, each process p_i simulates a t-IS execution of a full-information protocol in standard form. To that end, p_i invokes operation write_snapshot() described in Figure 4 to simulate each write/read step of the standard form protocol. Each time p_i gets a new snapshot $snap_i$, it checks whether there exits a node v in the e-tree whose associated $state(v, I)$ is equal to $snap_i$, for some $I \in \mathcal{I}$. If $O^I_v[i] = d_i \neq \perp$, p_i writes d_i in a shared array D and exits the simulation. In the second task, p_i repeatedly read D until a non-$\perp$ value d is found. When this happens, p_i decides d and exits.

Consider a execution of $\mathcal{A}$ in which at most t processes fail. $\mathcal{A}$ solves $\mathcal{T}$ if every correct process decides and the set of decided values is valid for input vector $I, \forall I \in \mathcal{I}$ according to the specification of $\mathcal{T}$. The proof is divided in four Claims. Termination is established by Claim *C3*, and Claim *C4* shows that the decision vector formed by the decided values is a valid output for I.

We first claim that in every leaf of the e-tree, at least $t + 1$ processes decide.
Claim C1 Let v be a leaf of the e-tree. $\forall I \in \mathcal{I} : |\{p_j : O^I_v[j] = \perp\}| \leq n - t - 1$.
Proof of the Claim C1 Let v be a leaf of the tree and denote $\alpha = s_1, \ldots, s_\ell$ the sequence of the labels in the path $root \rightsquigarrow v$. Denote $undecided(v, I) = \{p_j : O^I_v[j] = \perp\}$. Assume for contradiction that $|undecide(v, I)| > n - t - 1$, for some $I \in \mathcal{I}$. By the definition of O^I_v, $undecided(v, I)$ is the set computed by each process in its invocation of the explore procedure on node v.

Per Lemma 14, there exists a set $S \subseteq \Pi$, $|S| \geq t + 1$ such that $\alpha.s$ is a t-IS execution, for every $s \subseteq S$. We assumed that $|undecided(v, I)| \geq n - t$. Hence, there is a non empty set $s \subseteq \Pi$ such that $\alpha.s$ is a t-IS execution and $s \subseteq undecided(v, I)$: a contradiction with the fact that v is a leaf of the e-tree. In facts, after every egg has been laid in v, processes look for a set s' with the following properties (1) $\alpha.s'$ is a t-IS execution and (2) $s' \subseteq undecided(v, I')$ for some input vector I' (Figure 5). If such a set is found, the node v' that is reached from v through the arc labeled s' is eventually explored. As no descendant of v is ever explored, no such sets exist. *End of the proof of the Claim C1*

Next, we claim that the simulated t-IS execution corresponds to a path in the e-tree. In particular, the claim implies that processes simulate a finite t-IS execution.
Claim C2 Let $\alpha = s_1, s2, \ldots$ be the concurrency classes of a t-IS execution simulated

by task $T1$. There exists a path in the e-tree starting from the root whose sequence of labels is α.
Proof of the Claim C2 Let $\beta = s_1, \ldots, s_x$ denote the largest prefix of α for which there exists a path in the e-tree starting from the root (whose sequence of labels is β). Denote v_x the last node in this path. In the e-tree, node v_x has no outgoing arc labeled s_{x+1}. Task $T1$ simulates a t-IS execution. Therefore, $\beta.s_{x+1}$ satisfies the t-joint progress property. Then, it follows from the exploration procedure (Figure 5) that $s_{x+1} \not\subseteq undecided(v_x, I')$, for every $I' \in \mathcal{I}$ $(\star)$.

Let $p_j \in s_{x+1}$ and $I = [input_1, \ldots, input_n]$ be the input vector in the execution of $\mathcal{A}$. Let k be the largest integer $\leq x$ such that $p_j \in s_k$ and v_k the node with incoming arc labeled s_k in the path β. Then, in the simulated execution, $state(v_k, I)$ is the last snapshot $snap_j$ observed by p_j before being active in s_{x+1}. Since p_j does not exit the simulation after obtaining $snap_j = state(v_k, I)$, $O^I_{v_k}[j] = \perp$. Thus $p_j \in undecided(v_k, I)$, and $p_j \in undecided(v_x, I)$ since p_j is not active $s_{k+1}, \ldots, s_x$. Therefore, there exists I such that $s_{x+1} \subseteq undecided(v_x, I)$ contradicting $(\star)$. *End of the proof of the Claim C2.*

Claim C3 (Termination) Every process that does not fail decides.
Proof of the Claim C3 Let us first observe that, if a value is written in D, every correct process decides. Assume for contradiction that no value is written in D. Since at least $n - t$ processes do not fail, and each process keeps invoking write_snapshot() until it possibly fails, any invocation of write_snapshot() issued by a correct process terminates (see the proof of Theorem 3). Therefore, an infinite t-IS execution $\alpha = s_1, s_2, \ldots$ is simulated by $T1$. For every $\beta = s_1, \ldots, s_x$ prefix of α, there exists per Claim *C2* a path in the e-tree starting from the root whose sequence of labels is β. Denote v_β the last node in this path. Since the e-tree is finite, there exists a prefix γ of α such that v_γ is a leaf. We know by Claim *C1* that $|\{p_j : O^I_{v_\gamma}[j] = \perp\}| = |undecided(v_\gamma, I)| \leq n-t-1$, for every $I \in \mathcal{I}$. Consequently, there is a process p_c, which does not fail, such that $p_c \notin undecided(v_\gamma, I_e)$, where $I_e = [input_1, \ldots, input_n]$ is the input vector in the execution of $\mathcal{A}$. Let s_c be the last concurrency class in the sequence γ in which p_c is active, and v_c the node whose incoming arc is labeled s_c. In the simulation performed in Task $T1$, one of the snapshot obtained by p_c is $state(v_c, I_e)$. As $p_c \notin undecided(v_\gamma, I_e)$, s_c is the last concurrency class in which p_c is active in γ, p_c must have decided in node v_c. Hence, $O^{I_e}_{v_c}[c] \neq \perp$ from which we conclude that the exit condition of the **repeat** loop is eventually satisfied at process p_c. As p_c does not fail, it eventually writes a non-$\perp$ value in D: a contradiction. *End of the proof of Claim C3*

Claim C4 (Decision validity) Let DEC be the decision vector ($DEC[j] = d_j \neq \perp$ if p_j executes *return*(d_j)) in the execution of $\mathcal{A}$ and $I_e = [input_1, \ldots, input_n]$. DEC is a valid decision vector for input I_e according to the specification of $\mathcal{T}$.
Proof of Claim C4 Let O be the decision vector equal to the value of the shared array D at some point after which no process decides. As $\mathcal{T}$ is an agreement task, and every decided value is some non$-\perp$ entry of O, it is sufficient to establish that O is a legal output for input vector I. We prove that all values $O[i] \neq \perp$ are decided in some execution of $\mathcal{A}_D$.

Let $d_j = O[j] \neq \perp$ be the value written by process p_j in D. It follows from the pseudo-code that $d_j = O_v^I[j]$, where $I \in \mathcal{I}$ and v is a node of the e-tree whose incoming arc is labeled s, $p_j \in s$. Moreover, we have that $state(v, I) = snap_j$ where $snap_j$ is a snapshot observed by p_j in the simulated t-IS execution. Claim *C2* tells us that there exists a path $\alpha = root \xrightarrow{s_1} v_1 \ldots \xrightarrow{s_k} v_k$ in the e-tree whose sequence of labels is the sequence of concurrency classes of the simulated t-IS execution. Hence, there must exist a node $v_\ell \in v_1 \ldots v_k$ such that $snap_j = state(v_\ell, I_e)$ and $p_j \in s_\ell$. To summarize, $state(v, I) = state(v_\ell, I_e)$ and $p_j \in s \cap s_\ell$. According to Property *P3*, $O_v^I = O_{v_\ell}^{I_e}$. Consequently, for every p_j such that $d_j = O[j] \neq \perp$, there exists $v \in \{v_1, \ldots, v_k\}$ such that $d_j = O_v^{I_e}[j]$.

Consider a t-IS execution $e\alpha$ of $\mathcal{A}_D$ with input I_e described by the path α. Per property $P1$, each value $d_j \neq \perp$ is decided in this execution, from which we conclude that they are valid outputs for input vector I_e according to the specification of $\mathcal{T}$.

$\square_{Lemma\ 15}$

Finally, the following lemma immediately implies Theorem 2.

Lemma 16. *Let $\mathcal{T}$ a bounded task that has no solution in the t-resilient environment. Let D a failure detector and $\mathcal{A}_D$ a t-resilient protocol in standard form using D to solve $\mathcal{T}$. The algorithm described in Figure 3 with the modifications described in Figures 5 and 6 simulates a failure detector $\mathrm{anti}{-}\Omega^t$ using D in the t-resilient environment.*

Proof Consider an infinite run of the algorithm described in Figure 3, with the modifications described in Figures 5 and 6. Let $\mathcal{C}$ be the set of correct processes in that run, with $|\mathcal{C}| \geq n - t$. We have to show that there exists a correct process p_c such that, eventually, for every $p_i \in \mathcal{C}$, $p_c \notin \text{FD}_i$ permanently. Let $p_i \in \mathcal{C}$.

- Some invocation of explore() issued by p_i at Line 08 never terminates. This can only happen if (1) p_i never exits the **wait until** loop at line 28 while exploring some node v, or (2) p_i explores an infinite branch.
  (1) There exists v such that p_i waits forever for an egg from p_ℓ while exploring node v. As explained in the proof of Lemma 13, (i) every correct process p_j eventually reaches node v, and then eventually waits for an egg from p_ℓ and, (ii) p_ℓ cannot be a correct process.
  It then follows from the way FD_j is updated (Figure 6) that after some time, FD_j always contains the same set S for every correct process p_j. This set is such that $|S| = n - t$ and $p_\ell \in S$. Since $|\mathcal{C}| \geq n - t$ and $p_\ell \notin \mathcal{C}$, there exists a correct process $p_c \notin S$, from which we conclude that S is a valid output according to the specification of $\mathrm{anti}{-}\Omega^t$.
  (2) For every node v, no process is blocked in the **wait until** loop while exploring node v. As observed in the proof of Lemma 13, this implies that an infinite branch $b = root \xrightarrow{s_1} v_1 \xrightarrow{s_2} v_2 \ldots$ is produced in the e-tree, and the set $\mathcal{C}'$ of processes that appear infinitely often in the labels of this branch is a subset of $\mathcal{C}$.
  The sequence of labels $s_1, s_2, \ldots$ in b satisfy the t-joint progress property (Definition 7). Therefore, at least $n - t$ processes appear infinitely often in b, i.e., $|\mathcal{C}'| \geq n - t$. It follows from the way FD_i is updated (Figure 6) that eventually

$\text{FD}_j \subseteq \mathcal{C}'$, for every correct process p_j. Therefore, if $\mathcal{C}' \subsetneq \mathcal{C}$, FD_j eventually always contains a valid output for $\text{anti}{-}\Omega^t$ for every correct process p_j.
Assume for contradiction that $\mathcal{C} = \mathcal{C}'$. Per Lemma 3, each execution $e\alpha$ associated with b is D-legal. Moreover, $e\alpha$ is fair: every correct process takes infinitely many steps. So, as $\mathcal{A}_D$ is a t-resilient protocol that solves $\mathcal{T}$, for every choice of input vector I, every correct process should decide in $e\alpha(I)$ in which each process p_i is initially assigned the ith entry of I.
Let $p_\ell \in \mathcal{C}'$. For each $s_k : p_\ell \in s_k$, there exits an input vector $I(v_{k-1})$ such that $p_\ell \in undecided(v_{k-1}, I(v_{k-1}))$. As $\mathcal{I}$ is finite, it follows that there exits an input vector I_b such that p_ℓ never decide in $e\alpha(I)$: a contradiction.

– All invocations of explore() issued by p_i at Line 08 eventually terminate. By Lemma 15, this is not possible.

$\square_{Lemma\ 16}$

Acknowledgments

The authors would like to thank Michel Raynal and Piotr Zieliński for useful discussions on the topics of this paper.

References

1. Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, Nir Shavit: Atomic Snapshots of Shared Memory. J. ACM 40(4): 873-890 (1993)
2. Hagit Attiya, Sergio Rajsbaum. The Combinatorial Structure of Wait-Free Solvable Tasks. *SIAM J. Comput.* 31(4): 1286-1313 (2002).
3. Yehuda Afek, Gideon Stupp, Dan Touitou: Long-lived and adaptive atomic snapshot and immediate snapshot (extended abstract). PODC 2000: 71-80.
4. Elizabeth Borowsky, Eli Gafni. Generalized FLP impossibility result for t-resilient asynchronous computations. ACM STOC 1993: 91-100.
5. Elizabeth Borowsky, Eli Gafni. Immediate Atomic Snapshots and Fast Renaming (Extended Abstract). ACM PODC 1993: 41-51.
6. Elizabeth Borowsky, Eli Gafni, Nancy A. Lynch, Sergio Rajsbaum: The BG distributed simulation algorithm. Distributed Computing 14(3): 127-146 (2001)
7. Tushar Deepak Chandra, Sam Toueg: Unreliable Failure Detectors for Reliable Distributed Systems. *J. ACM* 43(2): 225-267 (1996).
8. Tushar Deepak Chandra, Vassos Hadzilacos, Sam Toueg: The Weakest Failure Detector for Solving Consensus. *J. ACM* 43(4): 685-722 (1996).
9. Wei Chen, Jialin Zhang, Yu Chen, Xuezheng Liu: Weakening Failure Detectors for k -Set Agreement Via the Partition Approach. DISC 2007:123-138.
10. Carole Delporte-Gallet, Hugues Fauconnier, Rachid Guerraoui, Vassos Hadzilacos, Petr Kouznetsov, Sam Toueg: The weakest failure detectors to solve certain fundamental problems in distributed computing. ACM PODC 2004: 338-346.
11. Carole Delporte-Gallet, Hugues Fauconnier, Rachid Guerraoui: Sharing is harder than agreeing. ACM PODC 2008: 85-94.
12. Carole Delporte-Gallet, Hugues Fauconnier, Rachid Guerraoui, Andreas Tielmann: The Weakest Failure Detector for Message Passing Set-Agreement. DISC 2008: 109-120.

13. Cynthia Dwork, Nancy Lynch and Larry Stockmeyer: Consensus in the Presence of Partial Synchrony. *J. of the ACM*, 35(2): 288-323 (1988).
14. Jonathan Eisler, Vassos Hadzilacos, Sam Toueg: The weakest failure detector to solve nonuniform consensus. *Distributed Computing* 19(4): 335-359 (2007).
15. Michael Fischer, Nancy Lynch and Mike Paterson: Impossibility of distributed commit with one faulty process. *J. of the ACM*, 32(2): 374-382 (1985).
16. Rachid Guerraoui, Maurice Herlihy, Petr Kouznetsov, Nancy A. Lynch, Calvin C. Newport: On the weakest failure detector ever. ACM PODC 2007: 235-243.
17. Rachid Guerraoui, Petr Kouznetsov: Failure detectors as type boosters. *Distributed Computing* 20(5): 343-358 (2008).
18. Rachid Guerraoui, Michal Kapalka, Petr Kouznetsov: The weakest failure detectors to boost obstruction-freedom. *Distributed Computing* 20(6): 415-433 (2008).
19. Maurice Herlihy, Lucia Penso: Tight Bounds for k-Set Agreement with Limited Scope Accuracy Failure Detectors. *Distributed Computing*, 18(2):157-166, (2005).
20. Maurice Herlihy, Sergio Rajsbaum: A classification of wait-free loop agreement tasks. *Theor. Comput. Sci.* 291(1): 55-77 (2003).
21. Maurice Herlihy, Nir Shavit. The asynchronous computability theorem for t-resilient tasks. ACM STOC 1993: 111-120.
22. Prasad Jayanti, Sam Toueg: Every problem has a weakest failure detector. ACM PODC 2008: 75-84.
23. Achour Mostefaoui, Michel Raynal: k-Set Agreement with Limited Accuracy Failure Detectors. ACM PODC 2000: 143-152.
24. Achour Mostefaoui, Sergio Rajsbaum, Michel Raynal, Corentin Travers: The Combined Power of Conditions and Information on Failures to Solve Asynchronous Set Agreement. *SIAM J. Comput.* 38(4): 1574-1601 (2008).
25. Michael Saks, Fotios Zaharoglou. Wait-free k-set agreement is impossible: The topology of public knowledge. ACM STOC 1993: 101110.
26. Jiong Yang, Gil Neiger, Eli Gafni: Structured Derivations of Consensus Algorithms for Failure Detectors. ACM PODC 1998: 297-308.
27. Piotr Zieliński. Automatic Classification of Eventual Failure Detectors. DISC 2007:465-479.
28. Piotr Zielinski. Anti-Ω: the weakest failure detector for set agreement. ACM PODC 2008:55-64.

A Solving k-set agreement with $\mathrm{anti}{-}\Omega^k$

A failure detector of the class $\mathrm{vector}{-}\Omega^k$ is a vector of k sub-detectors, $\Omega_1, \ldots, \Omega_k$, such that at least one Ω_i is a failure detector of the class Ω. When $k = n - 1$, the $\mathrm{vector}{-}\Omega$ failure detector proposed in [28] is obtained. It was shown there how $\mathrm{vector}{-}\Omega$ can be implemented from $\mathrm{anti}{-}\Omega$, and how it can be used to solve $n - 1$-set agreement.

The failure detectors $\mathrm{anti}{-}\Omega^k$ and $\mathrm{vector}{-}\Omega^k$ were also presented in [28] as $k - \mathrm{anti}{-}\Omega$ and $k - \mathrm{vector}{-}\Omega$, respectively. It is claimed there that the algorithm to transform $\mathrm{anti}{-}\Omega$ into $\mathrm{vector}{-}\Omega$ (Figure 1 in [28]), can be generalized to transform $\mathrm{anti}{-}\Omega^k$ into $\mathrm{vector}{-}\Omega^k$. We present here the generalized algorithm for completeness, as Figure 8. The following theorem claims that this algorithm implements $\mathrm{vector}{-}\Omega^k$. Its proof is almost verbatim to that of Theorem 3.1 in [28], and is omitted.

Theorem 4. *The algorithm of Figure 8 implements a failure detector* $\mathrm{vector}{-}\Omega^k$ *using any failure detector of the class* $\mathrm{anti}{-}\Omega^k$.

Now, solving k-set agreement with the help of a failure detector $\mathrm{vector}{-}\Omega^k$ is trivial, as was done in [28] with the special case $\mathrm{vector}{-}\Omega$. The algorithm simply starts k concurrent instances of an Ω-Consensus algorithm, where instance i uses in all processes the sub-detector Ω_i of the $\mathrm{vector}{-}\Omega^k$ failure detector. As soon as any Consensus instance decides at any process p_j, the decided value is used to decide at p_j for k-set agreement. Since at least one sub-detector behaves is in the class Ω, there is eventually a decision. Since k instances are used, no more than k different values can be decided. Hence the following theorem.

Theorem 5. *A failure detector* $\mathrm{anti}{-}\Omega^k$ *can be used to solve k-set agreement in an asynchronous atomic shared-memory distributed system.*

init for every node v, define $EGG(v)$ as an array of n SWMR registers;
$\forall v, \forall p_j \in \Pi, EGG(v)[j] \leftarrow \perp$;

task % anti$-\Omega$ extraction
(01) $m_i \leftarrow$ empty map; % maps states of p_i to eggs
(02) $undecided_i \leftarrow$ empty map; % maps pairs $(v, I \in \mathcal{I})$ to subsets of Π
(03) $\forall I \in \mathcal{I}$, define $undecided_i(root, I) \leftarrow \Pi$;
(04) $count_i \leftarrow 0$;
(05) $\text{FD}_i \leftarrow \{p_1\}$;
(06) **foreach** non-empty $P' \subseteq \Pi$ in deterministic order consistent with $\subset$ **do**
(07) **foreach** non-empty $s \subseteq P'$ in deterministic order consistent with $\subset$ **do**
(08) explore(v, s, P') where v is the end point of the arc labeled s;

procedure explore(v, s, P): % the incoming edge of v is labeled s
(09) **if** v has not been visited **then** egg_laying(v, s);
(10) **foreach** $I \in \mathcal{I}$ **do**
(11) define $undecided_i(v, I) \leftarrow undecided_i(v', I)$; % v' is the father of v
(12) **foreach** $p_j \in s$: decide$_j(e\text{-}state(v, I)) \neq \perp$ **do**
(13) $undecided_i(v, I) \leftarrow undecided_i(v, I) \setminus \{p_j\}$;
(14) **foreach** non-empty $P' \subseteq P$ in deterministic order consistent with $\subset$ **do**
(15) **foreach** non-empty $s' \subseteq P'$ in deterministic order consistent with $\subset$ **do**
(16) **let** v' be the node reached from v through the edge labeled s' ;
(17) **if** $\exists I \in \mathcal{I} : P' \cap undecided_i(v, I) \neq \emptyset$ **do**
(18) explore(v', s', P');

procedure egg_laying(v, s): % the incoming edge of v is labeled s
(19) **foreach** $p_j \in s$ **do** % in increasing ids order
(20) $\text{FD}_i \leftarrow \{p_j\}$; % update anti$-\Omega$ output
(21) **if** $(p_i = p_j)$ **then**
(22) **let** $u_i = osa(v, p_i)$;
(23) **if** $m_i(state(u_i))$ is undefined **then** %generate a new egg
(24) $count_i \leftarrow count_i + 1$;
(25) define $m_i(state(u_i)) \leftarrow \langle p_i, \text{fd\_query}(), count_i \rangle$;
(26) **repeat** $\delta = 2n$ times read$(EGG(v)[i])$; % dummy reads, see the proof of Lemma 7
(27) $EGG(v)[i] \leftarrow m_i(state(u_i))$; % write the egg in the shared memory
(28) **wait until** $EGG(v)[j] \neq \perp$;

Fig. 3. Extracting anti$-\Omega$, code for p_i

init *in shared memory* OIS % a shared immediate snapshot object, initially empty;
$C[1..n] \leftarrow [0, \ldots, 0]$ % an array of n counters
local variable: $r_i \leftarrow 0$ % local counter

operation write_snapshot(v_i):
$r_i \leftarrow r_i + 1$ % r_ith operation by p_i
if $(r_i \mod n) + 1 = i$ **then**
wait until $|\{j : C[j] \geq r_i\}| \geq n - t - 1$ **endif**
$snapshot_i \leftarrow OIS$.WRITE_SNAPSHOT$(v_i)$;
$C[i] \leftarrow C[i] + 1$;
return$(snapshot_i)$

Fig. 4. Simulating write-read cycle of an IS execution (code for p_i)

(16)' **if** $(\exists I \in \mathcal{I}$ s.t. $undecided_i(v, I) \neq \emptyset)$ **then**
(17)' let $s_1, \ldots, s_x = s$ be the sequence of labels in the path $root \rightsquigarrow v$;
(18)' **foreach** $s' \subseteq \Pi$ such that $(s_1, \ldots, s_x, s'$ satisfies t-joint progress) and $(\exists I \in \mathcal{I} : s' \subseteq undecided_i(v, I))$
% the sets s' are deterministically ordered
(18') explore(v', s', P'), where v' is the end point of the edge labeled s'

Fig. 5. extracting $\mathrm{anti}{-}\Omega^t$, modification of explore()

init $seq_i \leftarrow p_1.p_2 \ldots p_{n-t}$; $\mathrm{FD}_i \leftarrow \{p_1, \ldots, p_{n-t}\}$

(20a) $seq_i \leftarrow seq_i.p_j$ % append p_j to the sequence
(20b) $\mathrm{FD}_i \leftarrow$ the last $n - t$ distinct processes ids in seq_i

Fig. 6. extracting $\mathrm{anti}{-}\Omega^t$, modification of the update of FD_i

init $D[1..n] \leftarrow [\perp, \ldots, \perp]$
$d_i \leftarrow \perp$; $snap_i \leftarrow input_i$; e-tree$_i \leftarrow$ local copy of the e-tree

T1: **repeat** % simulate a t-IS execution of a normal form, full information protocol
$snap_i \leftarrow$ write_snapshot$(snap_i)$;
if $(\exists I \in \mathcal{I}, \exists v$ node in e-tree$_i$ with incoming arc labeled $s)$ such that $(state(v, I) = snap_i) \wedge (O_v^I[i] \neq \perp) \wedge (p_i \in s)$
then $d_i \leftarrow O_v^I[i]$
until $d_i \neq \perp$
$D[i] \leftarrow d_i$

T2: **repeat** $dec_i \leftarrow$ read(D) **until** $(\exists j : dec_i[j] \neq \perp)$
stop T1; *return*$(dec_i[j])$;

Fig. 7. t-resilient protocol $\mathcal{A}$ for $\mathcal{T}$ without failure detector, code for p_i

(01) **init** for every $p_i \in \Pi$ define $COUNT(i)[1..n] \leftarrow [0, \ldots, 0]$ array of n SWMR registers;

(02) **repeat**
(03) $d_i \leftarrow$ fd_query();
(04) **for each** $j \in d_i$ **do** $COUNT(i)[j] \leftarrow COUNT(i)[j] + 1$
(05) **end repeat**

(06) **when** vector$-\Omega^k$ is queried **do**
(07) **foreach** $j \in \{1, \ldots, n\}$ **do** $total_i[j] \leftarrow COUNT(1)[j] + \cdots + COUNT(n)[j]$;
(08) **let** $r_1, \ldots, r_n$ be such that $\{p_{r_1}, \ldots, p_{r_n}\} = \Pi$ and $\forall j \in \{1, \ldots, n-1\}, (total_i[r_j], r_j) < (total_i[r_{j+1}], r_{j+1})$;
(09) **return** $[p_{r_1}, \ldots, p_{r_k}]$

Fig. 8. Algorithm to extract vector$-\Omega^k$ from anti$-\Omega^k$. Code for process p_i. In Line 09 lexicographic ordering is used in $(total_i[r_j], r_j) < (total_i[r_{j+1}], r_{j+1})$.