Increasing the Power of the Iterated Immediate Snapshot Model with Failure Detectors

Michel Raynal, Julien Stainer

Publications Internes de l'IRISA
ISSN : 2102-6327
PI 1991 – February 2012

Increasing the Power of the Iterated Immediate Snapshot Model with Failure Detectors

Michel Raynal[*] [**] Julien Stainer[**]

Abstract: The base distributed asynchronous read/write computation model is made up of n asynchronous processes which communicate by reading and writing atomic registers only. The distributed asynchronous iterated model is a more constrained model in which the processes execute an infinite number of rounds and communicate at each round with a new object called immediate snapshot object. Moreover, in both models up to $n-1$ processes may crash in an unexpected way. An important computability issue associated with these models concerns their respective computability power. When considering distributed computing problems whose main issue is to cope with asynchrony and failures called decision tasks (such as consensus, set agreement, etc.) two main results are associated with the previous models. The first states that these models are computationally equivalent for decision tasks. The second states that they are no longer equivalent when both are enriched with the same failure detector.

This paper shows how to capture failure detectors in each model in order both models become computationally equivalent. To that end it introduces the notion of a "strongly correct" process which appears particularly well-suited to the iterated model. It also presents simulations that proves the computational equivalence when both models are enriched with a failure detector. Interestingly, these simulations works for a large family of failure detector classes. The paper extends also the simulations to the case where the wait-freedom requirement is replaced by the notion of t-resilience. Last but not least, a noteworthy feature of the proposed approach lies in its simplicity.

Key-words: asynchronous read/write model, decision task, distributed computability, failure detector, immediate snapshot object, iterated model, model equivalence, process crash, simulation, t-resilience, wait-freedom.

Augmenter la puissance du calcul réparti itéré avec des détecteurs de fautes

Résumé : *Les modèles de systèmes distribués asynchrones communicant par mémoire partagée sont équivalents, du point de vue de la calculabilité, au modèle itéré où les processus communiquent via une séquence d'objets* write-snapshot*. Cependant, il a été montré que l'utilisation naïve, dans le modèle itéré, des détecteurs de fautes définis pour la mémoire partagée n'apportait aucune puissance de calcul additionelle.*

Cet article propose une méthode systématique pour porter les détecteurs de fautes du modèle classique communicant par mémoire partagée dans le modèle itéré, ceci en préservant les équivalences de modèles. Dans ce but, il introduit la notion de processus fortement corrects *qui aide à décrire les possibilités de communication dans le modèle itéré.*

L'article fournit des simulations génériques (ne dépendant pas ou peu du détecteur de faute) pour prouver les équivalences de modèles. Les cas de plusieurs détecteurs de fautes connus sont traités et le résultat est étendu au cas de la t-résilience. Enfin, un algorithme de consensus dans le modèle itéré augmenté du détecteur de faute correspondant à Ω illustre une des possibilités offertes par cette étude.

Mots clés : *modèle asynchrone en mémoire partagée, calculabilité distribuée, détecteur de fautes, objet* immediate-snapshot*, modèle itéré, équivalence de modèles, défaillance de processus, simulation, t-résilience, sans attente*

---

[*] Institut Universitaire de France
[**] ASAP : équipe commune avec l'Université de Rennes 1 et Inria

1 Introduction

Base read/write model and tasks The base asynchronous read/write (ARW) computation model consists of n asynchronous sequential processes that communicate only by reading and writing atomic registers. Moreover, any number of processes (but one) are allowed to crash in an unexpected way.

A decision task is the distributed analogous of the notion of a function encountered in sequential computing. Each process starts with its own input value (without knowing the input values of the other processes). The association of an input value with each process define an input vector of the task. Each process has to compute its own output value in such a way that the vector of output values satisfies a predefined input/output relation (this is the relation that defines the task). The most famous distributed task is the consensus task: each process proposes a value and processes have to decide the very same value which has to be one of the proposed values. The progress condition that is usually considered is called *wait-freedom* [12]. It requires that any process that does not crash eventually decides a value. It has been shown that the consensus task cannot be wait-free solved in the ARW model [16]. The tasks that can be wait-free solved in this base model are sometimes called *trivial* tasks.

The iterated immediate snapshot (IIS) model and its power The fact that, in the ARW model, a process can issue a read or write on any atomic register at any time makes difficult to analyze the set of runs that can be generated by the execution of an algorithm that solves a task in this model.

To make such analyses simpler and obtain a deeper understanding of the nature of asynchronous runs, Borowsky and Gafni have introduced the *iterated immediate snapshot* (IIS) model [4]. In this model, each process (until it possibly crashes) executes asynchronously an infinite number of rounds and, in each round, processes communicate through a one-shot *immediate snapshot* object [3] associated with this round. Such an object provides the processes with a single operation denoted write_snapshot(). This operation allows the invoking process to deposit a value into the corresponding object and obtains a snapshot [1] of the values deposited into it. It is important to notice that each immediate snapshot object is accessed at most once by each process but can be simultaneously accessed by any number of processes.

A *colorless* decision task is a task such that any value decided by a process can be decided by any number of processes. The main result associated with the IIS model is the following one: A colorless decision task can be wait-free solved in the ARW model if and only if it can be wait-free solved in the IIS model. This result is due to Borowsky and Gafni [4]. Said another way, the ARW model and the IIS model (which is more constrained as far as runs are concerned) have the same computational power for colorless decision tasks.

Enriching a model with a failure detector One way to enrich the base read/write model in order to obtain a stronger model consists in providing the processes with operations whose computational power in presence of asynchrony and process crashes is stronger than the one of the base read or write operations [12]. An example of such an operation is the compare&swap operation [12]. Another way to enrich the base read/write model consists in adding to it a failure detector [6, 24].

A failure detector is a device that provides each process with a read-only variable that gives it information on failures. According to the type and the quality of this information, several classes of failure detectors can be defined. As an example, a failure detector of the class Ω [7] provides each process p_i with a read-only local variable denoted $\texttt{leader}_i$ that contains always a process identity. The property associated with these read-only local variables is the following: there is an unknown but finite time after which all the variables $\texttt{leader}_i$ contain forever the same identity and this identity is the one of a non-faulty process. A failure detector is non-trivial if it cannot be built in the base read/write model (i.e., if it enriches the system with additional power).

A natural question is then the following: Are the ARW model and the IIS model still equivalent for wait-free task solvability when they are enriched with the same non-trivial failure detector? It has been shown by Rajsbaum, Raynal and Travers that the answer to this question is "no" [23]. It follows that, from a computability point of view, the ARW model enriched with a non-trivial failure detector is more powerful than the IIS model enriched with the same failure detector.

An approach aiming at introducing the power of failure detectors into the IIS model has been investigated in [9, 22]. This approach consists in requiring some property P to be satisfied by the successive invocations of write_snapshot() issued on the sequence of immediate snapshot objects. Hence the name *iterated restricted immediate snapshot* (IRIS) given to this model. For each failure detector class taken separately, this approach requires (a) to associate a specific property P with the considered failure detector class, (b) design an ad hoc simulation of the write_snapshot() operation suited to this failure detector class in order to simulate IIS in ARW and (c) design a specific simulation of the output of the failure detector to simulate ARW in IIS. Interestingly, this approach was the first to show that failure detectors are related to fairness and can be considered as schedulers (see also [20]).

Content of the paper Let C be a failure detector class defined in the context of the base ARW model. This paper is motivated by the following question: Is it possible to associate with C (in a systematic way) a failure detector class C^* such that the ARW model enriched with C and the IIS model enriched with C^* are equivalent for wait-free task solvability? To answer that question, the paper considers failure detector classes whose output eventually involves (a) at least one non-faulty process and possibly faulty processes (such as Ω_k) or (b) only non-faulty processes (such as P, Σ, $\diamond P$ or $\diamond S_x$). The contributions of the paper are the following.

- The answer to the previous question is based on a simple modification of the definition of what is a *correct* process (i.e., a process that does not crash in a run of the base read/write model). The notion of a correct process is replaced in the IIS model by what we call a *strongly correct* process. Such a process is a process that does not crash and whose all invocations of write_snapshot() are seen (directly or indirectly) by all other non-crashed processes[1].

  Given this definition, and a failure detector class C designed for the ARW model, its IIS counterpart C^* is obtained by a simple and systematic replacement of the words "correct process(es)" by "strongly correct process(es)" in the definition of C.
- An immediate benefit of the previous definition is the fact that, when we want to simulate the ARW model in the IIS model, we can directly benefit from the simulation of the read and write operations defined in [4] by Borowsky and Gafni. The only addition to that simulation that has to be done concerns the local outputs of the corresponding failure detector C.
- Given the ARW model enriched with a failure detector class C, the paper presents a generic simulation of the IIS model enriched with C^*. This simulation is generic in the sense that it works for any of the previously cited failure detectors. The simulation algorithm has only to be instantiated with a predicate associated with the corresponding failure detector C.
- An interesting consequence of the fact that, given a failure detector class C, we have "for free" a corresponding failure detector for the IIS model, not only makes simpler the understanding of IIS enriched with a failure detector but allows for relatively easy proofs.
- The paper also generalizes the previous wait-free simulations to t-resilient simulations (let us remind that, in a system of n processes, wait-freedom is $(n-1)$-resilience).

Among other benefits, an important corollary result of the paper is the fact that if C is the weakest failure detector to solve a given task T in the base read/write model [10], it follows from the previous simulations that (when considering all failure detector classes D^* obtained from a detector class D designed for the read/write model) C^* is the weakest failure detector to solve T in the IIS model and vice-versa.

Let $\mathcal{ARW}_n[C]$ denote the base asynchronous read/write model with n processes enriched with a failure detector of the class C and $\mathcal{IIS}_n[C^*]$ denote the IIS model enriched with the corresponding failure detector C^*. The wait-free simulations presented in the paper are summarized in Figure 1.

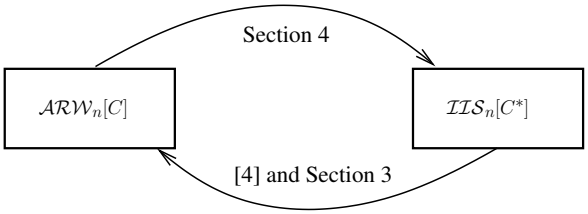

Figure 1: From $\mathcal{ARW}_n[C]$ to $\mathcal{IIS}_n[C^*]$ and vice-versa

Roadmap The paper is made up of 6 sections. Section 2 introduces the IIS model and failure detectors. Section 3 provides simulations from the iterated immediate snapshot model enriched with a failure detector to the read/write model while Section 4 provides simulations in the other direction. Section 5 considers the t-resilience case (instead of wait-freedom). Finally, Section 6 concludes the paper. (For completeness, an Ω-based consensus algorithm suited to the IIS model is described in an appendix).

2 Base definitions

Notation Shared objects are denoted with uppercase letters. Local variables are denoted with lowercase letters (the index of the corresponding process is sometimes used as a subscript of a local variable).

2.1 Process model

In the rest of the paper, the n asynchronous and sequential processes are denoted p_1, ..., p_n; i is called the index of p_i. Moreover, any number of processes may crash (stop executing). A process that crashes in a run is said to be *faulty* in that run, otherwise it is *correct* in the corresponding run. A correct process executes an infinite number of steps. Given an execution, let $\mathcal{C}$ denote the set of processes that are correct in that execution and $\mathcal{F}$ the set of the faulty ones.

[1] Although it is not explicitly defined in [22], the notion of a *strongly correct* process is implicitly used in the proof of a theorem in that paper. It is not used to define failure detectors suited to the IIS model.

2.2 Decision tasks

A *decision task* is a one-shot decision problem specified in terms of an input/output relation Δ. Each process starts with a private input value and must eventually decide on a private output value. From a global observer point of view, an input vector $I[1..n]$ specifies the input $I[i] = v_i$ of each process p_i. Similarly, an output vector $O[1..n]$ specifies a decision value $O[j]$ for each process p_j.

A task is defined by a set of input vectors and a relation Δ that describes which output vectors are correct for each input vector I. More precisely, for each valid input vector I, the values decided by the processes are such that there is an output vector $O \in \Delta(I)$ such that, for each j, $O[j]$ is the value decided by p_j and, if no value is decided by p_j, it is because p_j has crashed during the computation.

2.3 Base read/write model

This model, denoted $\mathcal{ARW}_n[\emptyset]$ in the following, has been presented in the introduction. Instead of atomic read/write registers, we consider here that the processes communicate through *snapshot* objects [1]. This is at no additional computability cost as the operations on a snapshot object can be wait-free implemented from single-writer/multi-reader atomic read/write registers. Given a snapshot object S, these operations are denoted S.write() and S.snapshot().

S is initially empty. When p_i invokes S.write(v), it adds the pair $\langle i, v\rangle$ to S and suppresses the previous pair $\langle i, -\rangle$ if any. When it invokes S.snapshot(), p_i obtains a set containing all the pairs $\langle k, v_k\rangle$ currently contained in S. Such a set is called a *view* that we denote arw_view_i.

A snapshot object S is *atomic* (we also say said that it is *linearizable* [14]). This means that each operation invocation appears as if it has been executed at a single point of the time line between its start event and its end event in such a way that no two invocations are associated with the same point of the time line. The corresponding total order is called a *linearization*.

2.4 The iterated immediate snapshot model

One-shot immediate snapshot object Basically, such an object is similar to a snapshot object where the write() and S.snapshot() operations are encapsulated into a single operation denoted write_snapshot() (where the write is executed just before the snapshot). Moreover, *one-shot* means that, given an object, each process invokes write_snapshot() at most once.

If several processes invoke concurrently IS.write_snapshot(), their write operations are executed concurrently followed by a concurrent execution of their snapshot operations. The invocations of IS.write_snapshot() are not linearizable but *set-linearizable* [18]. This is similar to atomicity except that the same point of the time line can be associated with several concurrent invocations of IS.write_snapshot().

Let us consider a process p_i that invokes IS.write_snapshot(v_i) where v_i is the value it wants to write into IS. When it returns from its invocation, p_i obtains a *view* of the object that we denote iis_view_i, Moreover, let us define $iis\_view_i = \emptyset$ if p_i never invokes IS.write_snapshot(). A one-shot immediate snapshot object is defined by the following properties associated with the views obtained by the processes.

- Self-inclusion. $\forall\, i :\ \langle i, v_i\rangle \in iis\_view_i$.
- Containment. $\forall\, i, j :\ (iis\_view_i \subseteq iis\_view_j)\ \vee\ (iis\_view_j \subseteq iis\_view_i)$.
- Immediacy. $\forall\, i, j :\ (\langle i, v_i\rangle \in iis\_view_j) \Rightarrow\ (iis\_view_i \subseteq iis\_view_j)$.

The first property states that a process sees its write. The second property states that the views are totally ordered by containment. The third property states that, when a process invokes IS.write_snapshot(), its snapshot is scheduled just after its write. This last property can be re-written as follows: $\forall\, i, j :\ \big((\langle i, v_i\rangle \in iis\_view_j) \wedge (\langle j, v_j\rangle \in iis\_view_i)\big) \Rightarrow (iis\_view_j = iis\_view_i)$ (which means that concurrent invocations of IS.write_snapshot() obtain the same view).

The operation write_snapshot() can be wait-free implemented from atomic read/write registers (i.e., in the base read/write model (e.g., [3, 5]). Hence, it does not add computational power to the read/write model.

The iterated immediate snapshot model In the base IIS model (denoted $\mathcal{IIS}_n[\emptyset]$), the shared memory is made up of an infinite number of immediate snapshot objects $IS[1]$, $IS[2]$, ... These objects are accessed sequentially and asynchronously by the processes according to the following round-based pattern executed by each process p_i. The variable r_i is local to p_i and denotes its the current round number.

$r_i \leftarrow 0$; $\ell s_i \leftarrow$ initial local state of p_i (including its input, if any);
repeat forever (asynchronous IIS rounds)
  $r_i \leftarrow r_i + 1$;
  $iis\_view_i \leftarrow IS[r_i]$.write_snapshot($\ell s_i$);
  computation of a new local state ℓs_i (which contains iis_view_i)
end repeat.

It is easy to see that the runs of an algorithm in $\mathcal{IIS}_n[\emptyset]$ are more structured than runs in $\mathcal{ARW}_n[\emptyset]$. As indicated in the Introduction, the most important result associated with this model, which is due to Borowsky and Gafni, states that the $\mathcal{ARW}_n[\emptyset]$ and $\mathcal{IIS}_n[\emptyset]$ models have the same computability power for colorless wait-free decision tasks [4].

Full information algorithm As the aim of the IIS model is to address computability issues (and not efficiency), we consider *full information* algorithms for this computation model.

This means that, at each round r, a process p_i writes its current local state ℓs_i into $IS[r]$. Consequently, the view it obtains from its invocation $IS[r]$.write_snapshot(ℓs_i) contains all its causal past (hence the name *full information*). More precisely, for any k and any x, let $\ell s_k[x]$ be the local state of process p_k at beginning of the round x. "Full information" means that, if $\langle j, \ell s_j[r]\rangle$ belongs to the view obtained by p_i during round r, then $\ell s_j[r]$ is part of $\ell s_i[r+1]$ (let us observe that this definition is recursive).

2.5 Failure detectors

A failure detector is a device that provides processes with information on failures [6]. As already said in the introduction, several classes of failure detectors can be defined according to the type and the quality of the information given to the processes. We consider here that the information given to each process is a set of process indexes. This information is given by a failure detector to a process through a local read-only variable.

Classes of failure detectors that eventually output only correct processes We consider three classes of such failure detectors: the classes P of perfect failure detectors and $\diamond P$ of eventually perfect failure detectors (defined in [6]) and the class Σ of quorum failure detectors (defined in [10]).

- A failure detector of the class P provides each process p_i with a set $\texttt{trusted}_i$ that, at any time τ, contains all the processes that have not crashed by time τ and eventually contains only correct processes.[2]
- The class $\diamond P$ is weaker than the class P. Namely, there is an arbitrary long finite period during which the sets $\texttt{trusted}_i$ can contain arbitrary values and when this period terminates a failure detector of $\diamond P$ behaves as a failure detector of P.
- A failure detector of the class Σ provides each process p_i with a set $\texttt{qr}_i$ that eventually contains only correct processes and is such that the value of $\texttt{qr}_i$ at any time τ and the value of any $\texttt{qr}_j$ at any time τ' have a non-empty intersection.

Classes of failure detectors that eventually output correct and possibly faulty processes We consider here the class of eventual leaders failure detectors denoted Ω_k which has been proposed in [19]. This class class is a straightforward generalization of the failure detector class Ω introduced in [7]. Actually, Ω_1 is Ω which has been shown to be the weakest failure detector class for solving the consensus task in shared memory systems [7, 15].

A failure detector of the class Ω_k provides each process p_i with a read-only local variable $\texttt{leaders}_i$ that always contains k process indexes and satisfies the following property.

- The local variables $\texttt{leaders}_i$ offered by a failure detector of the class Ω_k are such that, after some unknown but finite time τ, they all contain forever the same set of k process indexes and at at least one of these indexes is the one of a correct process.

Let us notice that, before time τ, the sets $\texttt{leaders}_i$ can contain arbitrarily changing sets of k process indexes.

2.6 Strongly correct processes (wrt the IIS model)

Motivation When considering the base read/write model, if a process issues a write into a snapshot object S, the value it has written can be read by any process that invokes S.snapshot().

This is no longer the case in the IIS model. As an example, let us consider an IIS execution in which, at any round r, the very same process p_x is always the first to invoke $IS[r]$.write_snapshot(ℓs_x) and this invocation is not concurrent with any other invocation of $IS[r]$.write_snapshot(). If follows that, at each round r, $view_x$ contains only the pair $\langle x, \ell s_x\rangle$. Hence, p_ℓ never sees values written by other processes.

The previous observation motivates the definition of a *strongly correct* process. Such a process is a process whose writes into the immediate snapshot objects are seen (directly or indirectly) infinitely often by the all correct processes. A process that is not strongly correct is consequently a process such that only a finite number of its writes into immediate snapshot objects are eventually propagated to all the correct processes.

[2] The traditional definition of P provides each process p_i with a set $\texttt{faulty}_i$ that does not contain a process before it crashes and eventually contains all faulty processes. It is easy to see that $\texttt{trusted}_i = \{1 \ldots, n\} \setminus \texttt{faulty}_i$.

Formal definition Let $iis\_view_j[r]$ be the view obtained by p_j at round r. Let $\mathcal{SC}_0$ be the set defined as follows (let us remember that $\mathcal{C}$ denotes the set of correct processes):

$$\mathcal{SC}_0 \stackrel{def}{=} \{\ i \text{ such that } |\{r \text{ such that } \forall\ j \in \mathcal{C} :\ \exists\ \langle i, -\rangle \in iis\_view_j[r]\}| = \infty\ \},$$

i.e., $\mathcal{SC}_0$ is the set of processes that have issued an infinite sequence of (not necessarily consecutive) invocations of write_snapshot() and these invocations have been seen by each correct process (this is because these invocations are set-linearized in the first position when considering the corresponding one-shot immediate snapshot objects).

Let us observe that, as soon as we assume that there is at least one correct process, it follows from the fact that the number of processes is bounded that $|\mathcal{SC}_0| \neq 0$. Given $k > 0$ let us recursively define $\mathcal{SC}_k$ as follows:

$$\mathcal{SC}_k \stackrel{def}{=} \{\ i \text{ such that } |\{r \text{ such that } \exists\ j \in \mathcal{SC}_{k-1} :\ \exists\ \langle i, -\rangle \in iis\_view_j[r]\}| = \infty\ \}.$$

Hence, $\mathcal{SC}_k$ contains all the correct processes that have issued an infinite sequence of (not necessarily consecutive) invocations of write_snapshot() which have been seen by at least one process of $\mathcal{SC}_{k-1}$. It follows from the self-inclusion property of the views and the definition of $\mathcal{SC}_k$ that $\mathcal{SC}_0 \subseteq \mathcal{SC}_1 \subseteq \cdots$. Moreover, as all the sets are included in $\{1, \ldots, n\}$, there is some K such that $\mathcal{SC}_0 \subseteq \mathcal{SC}_1 \subseteq \cdots \subseteq \mathcal{SC}_K = \mathcal{SC}_{K+1} = \mathcal{SC}_{K+2} = \cdots$.

$\mathcal{SC}_K$ defines the set of of strongly correct processes which is denoted $\mathcal{SC}$. This is the set of processes that have issued an infinite sequence of (not necessarily consecutive) invocations of write_snapshot() which have been propagated to all the correct processes.

IIS enriched with a failure detector Let C be a failure detector class. C^* denotes the same failure detector class where the word "correct" is replaced by the word "strongly correct". Moreover, $\mathcal{IIS}_n[C^*]$ denotes the IIS model enriched with a failure detector of the class C^* where, during each round r, a process p_i reads its failure detector variable at the beginning of round r and saves its value fd_i in its local state ℓs_i before writing it into $IS[r]$.

3 From $\mathcal{IIS}_n[C^*]$ to $\mathcal{ARW}_n[C]$

This section describes a simulation in $\mathcal{IIS}_n[C^*]$ of a run of an algorithm designed for $\mathcal{ARW}_n[C]$. Except for the simulation of the detector output, this simulation is from [4]. In order not to confuse a simulated process in $\mathcal{ARW}_n[C]$ and its simulator in $\mathcal{IIS}_n[C^*]$, the first one is denoted p_i while the second one is denoted q_i.

3.1 Description of the simulation

It is assumed, without loss of generality, that the simulated processes communicate through a single snapshot object S. A simulator q_i is associated with each simulated process p_i. It locally executes the code of p_i and uses the algorithms described in Figure 2 when p_i invokes S.write($-$), S.snapshot() or queries the failure detector.

Immediate snapshot objects of $\mathcal{IIS}_n[C^*]$ These objects are denoted $IS[1], IS[2], ...$ Each object $IS[r]$ stores a set of triples (this set is denoted ops_i in Figure 2). If the triple $(j, sn, x) \in ops_i$, then the simulator q_i knows that the process p_j (simulated by q_j) has issued its sn-th invocation of an operation on the simulated snapshot object S; $x \neq \perp$ means that this invocation is S.write(x) while $x = \perp$ means that it is S.snapshot().

Local variables of a simulator q_i The variable r_i contains the current round number of the simulator q_i. It is increased before each invocation of $IS_n[r_i]$.write_snapshot(ops_i) (line 3). As this is the only place where, during a round, a simulator invokes the operation write_snapshot(), the simulators obey the IIS model.

The local variable sn_i is a sequence number that measures the progress of the simulation by q_i of the process p_i. It is increased at line 1 when p_i starts simulating a new invocation of S.write() or S.snapshot() on behalf on p_i.

As already indicated, the local variable ops_i contains the triples associated with all the invocations of S.write() and S.snapshot() that have been issued by the processes and are currently known by the simulator q_i. This local variable (which can only grow) is updated at line 1 when q_i starts simulating the next operation S.write() or S.snapshot() issued by p_i or at line 4 when q_i learns operations on the snapshot object S issued by other processes.

The local variable iis_view_i stores the value returned by the last invocation of $IS[r_i]$.write_snapshot() issued by the simulator q_i (line 3). When simulating an invocation of S.snapshot() issued by p_i, q_i computes for each simulated process p_j the sequence number max_snj_i (line 12) of the last value it knows (saved in v_snj_i at line 13) that has been written by p_j in the snapshot object S. This allows q_i to compute the view arw_view_i (line 14) that it returns (line 17) as the result of the invocation of S.snapshot() issued by p_i.

The local variable fd_i is used to store the last value obtained by the simulator q_i from its read-only local failure detector variable denoted C^*.read().

Init: $ops_i \leftarrow \emptyset$; $r_i \leftarrow 0$; $sn_i \leftarrow 0$; $iis\_view_i \leftarrow \emptyset$; $fd_i \leftarrow C^*$.read().

internal operation publicize&progress (x) **is**
(1) $sn_i \leftarrow sn_i + 1$; $ops_i \leftarrow ops_i \cup \{(i, sn_i, x)\}$;
(2) **repeat** $r_i \leftarrow r_i + 1$; $fd_i \leftarrow C^*$.read();
(3) $iis\_view_i \leftarrow IS[r_i]$.write_snapshot$(ops_i)$;
(4) $ops_i \leftarrow \bigcup_{\langle k, ops_k\rangle \in iis\_view_i} ops_k$
(5) **until** $\big((i, sn_i, x) \in \bigcap_{\langle k, ops_k\rangle \in iis\_view_i} ops_k\big)$ **end repeat**;
(6) return().

operation S.write(v) **is**
(7) publicize&progress (v); return().

operation S.snapshot() **is**
(8) publicize&progress $(\perp)$;
(9) $arw\_view_i \leftarrow \emptyset$;
(10) **for each** j **in** $\{1, \ldots, n\}$ **do**
(11) **if** $\big(\exists v \mid (j, -, v) \bigcap_{\langle k, ops_k\rangle \in iis\_view_i} ops_k \wedge v \neq \perp\big)$
(12) **then** $max\_snj_i \leftarrow \max\{sn \mid (j, sn, v) \in \bigcap_{\langle k, ops_k\rangle \in iis\_view_i} ops_k \wedge v \neq \perp\}$;
(13) $vj_i \leftarrow v$ such that $(j, max\_snj_i, v) \in ops_i$;
(14) $arw\_view_i \leftarrow arw\_view_i \cup \{\langle j, vj_i\rangle\}$
(15) **end if**
(16) **end for**;
(17) return (arw_view_i).

operation C.read() **is** return (fd_i).

Figure 2: Simulation of $\mathcal{ARW}_n[C]$ in $\mathcal{IIS}_n[C^*]$: code for a simulator q_i (extended from [4])

Simulation of S.write(v) To simulate the invocation of S.write(v) issued by p_i, the simulator q_i invokes the internal operation publicize&progress(v). It first increments sn_i and adds the triple (i, sn_i, v) to ops_i (line 1). Then, it repeatedly invokes write_snapshot(ops_i) on successive immediate snapshot objects and enriches its set of triples ops_i (lines 2-4) until it obtains a view iis_view_i in which all the simulators it sees in that view are aware of the invocation of the operation S.write(v) that it is simulating (line 6).

Simulation of S.snapshot() To simulate an invocation of S.snapshot() issued by p_i, the simulator q_i first invokes publicize&progress$(\perp)$. When this invocation terminates, q_i knows that all the simulators it sees in the last view iis_view_i it has obtained are aware of its invocation of S.snapshot(). Moreover, as we have seen, the execution of publicize&progress$(\perp)$ makes q_i aware of operations simulated by other simulators.

Then the simulator q_i browses all the operations it is aware of in order to extract, for each simulated process p_j, the last value effectively written by p_j (lines 10-16). This (non-$\perp$) value is extracted from the triple with the largest sequence number among all those that appear in all the sets ops_k that belong to the view iis_view_i returned to q_i by its last invocation of write_snapshot().

Simulation of C.read() When a process p_i reads its local failure detector output, the simulator q_i simply returns it the current value of fd_i.

3.2 From strongly correct simulators to correct simulated processes

Strongly correct vs weakly correct simulators Let $\mathcal{WC} = \mathcal{C} \setminus \mathcal{SC}$ (the set of weakly correct simulators). It follows from the definition of the strongly correct simulators that, for any simulated process p_i whose simulator q_i is such that $i \in \mathcal{WC}$, there is a round $rmin_i$ such that, $\forall j \in \mathcal{SC}, \forall r \geq rmin_i : \langle i, -\rangle \notin iis\_view_j[r]$, which means that, for $r \geq rmin_i$, no invocation $IS[r]$.write_snapshot() issued by the simulator q_i is seen by a strongly correct simulator.

This means that, after $rmax = \max\{rmin_i\}_{i \in \mathcal{WC}}$ and after all simulator crashes have occurred, the invocations of write_snapshot() by the simulators of $\mathcal{SC}$ are always set-linearized strictly before the ones of the simulators of $\mathcal{WC}$. Said differently, there is a round after which no strongly correct simulator ever receives information from a weakly correct simulator. From the point of view of a strongly correct simulator, any weakly correct simulator appears as a crashed simulator. Differently, any weakly correct simulator receives forever information from all the strongly correct simulators[3].

[3]This situation is similar to the case where, in the base read/write model, after some finite time, some subset of processes (the ones corresponding here to the set of weakly correct simulators) commit forever send omission failures with respect to the correct processes (the ones corresponding here to the set of strongly correct simulators).

Crashed and slow IIS simulators simulate crashed ARW processes An important feature of the simulation described in Figure 2 is that, not only the crash of a simulator q_i gives rise to the crash of the associated simulated process p_i, but a slow simulator q_j entails the crash of its simulated process p_j.

As an example, let us consider a correct simulator q_j which, at any round r, is always strictly the last simulator which invokes $IS[r]$.write_snapshot(). It follows that no other simulator is ever informed of the operations S.write() and S.snapshot() issued by the process p_j simulated by q_j. When the simulator q_j executes line 3, it is informed of the operations issued by the strongly correct simulators q_i on the behalf of the processes they simulate but, as the operation of p_j it is simulating (which is encoded in the triple (j, sn_j, x)) is never known by the other simulators, it follows that q_j loops forever in the **repeat** loop (lines 2-5). Hence, the simulation of p_j does not longer progress and p_j is considered as a crashed process in the simulated ARW model. This means that the simulation described in Figure 2 guarantees wait-freedom for the processes simulated by the strongly correct simulators only.

To summarize When simulating $\mathcal{ARW}_n[C]$ on top of $\mathcal{IIS}_n[C^*]$, we have the following: (a) a faulty or weakly correct simulator q_i gives rise to a faulty simulated process p_i and (b) a strongly correct process gives rise to a correct simulated process p_i in $\mathcal{ARW}_n[C]$.

The next theorem captures the previous discussion. As it is a simple formalization of this discussion, its proof is given only in Appendix A.

Theorem 1 *Let A be an algorithm solving a colorless task in the $\mathcal{ARW}_n[C]$ model. Let us consider an execution of A simulated in the $\mathcal{IIS}_n[C^*]$ model by the algorithms S.write(), S.snapshot() and C.read() described in Figure 2. A process p_i is correct in the simulated execution if and only if its simulator q_i is strongly correct in the simulation.*

4 From $\mathcal{ARW}_n[C]$ to $\mathcal{IIS}_n[C^*]$

This section presents a generic simulation of $\mathcal{IIS}_n[C^*]$ in $\mathcal{ARW}_n[C]$. Its generic dimension lies in the fact that C can be any failure detector class cited in the Introduction and defined in Section 2.5 (namely, P, Σ, $\diamond P$, Ω, Ω_k, $\overline{\Omega}_k$ and others such as S, $\diamond S$ [6] and $\diamond S_x$ [2]). As far terminology is concerned, q_i is used to denote a simulated IIS process while p_i is used to denote the corresponding ARW simulator process. The simulation is described in Figure 3. Differently from the simulation described in Figure 2, the algorithms of Figure 3 are not required to be full-information algorithms.

4.1 Description of the simulation

The simulated model $IS[1]$, $IS[2]$, ... denote the infinite sequence of one-shot immediate snapshot objects of the simulated IIS model. Hence, a simulated process q_i invokes $IS[r]$.write_snapshot() and C^*.read().

Shared objects of the simulation The simulation uses an infinite sequence of objects $S[1]$, $S[2]$, ... The object $S[r]$ is used to implement the corresponding one-shot immediate snapshot object $IS[r]$.

Each object $S[r]$ can be accessed by two operations denoted collect() and arw_write_snapshot(). The later is nothing else than the operation write_snapshot() (which satisfies the self-inclusion, containment and immediacy properties defined in Section 2.4). It is prefixed by "arw" in order not to be confused with the operation of the IIS model that it helps simulate. The operation collect() is similar to the operation snapshot(), except that it is not required to be atomic. It consists in an asynchronous scan of the corresponding $S[r]$ object which returns the set of pairs it has seen in $S[r]$. Both collect() and arw_write_snapshot() can be wait-free implemented in $\mathcal{ARW}_n[\emptyset]$ (an implementation of arw_write_snapshot() is described in Appendix B).

FD_VAL is an array of single-writer/multi-reader atomic registers. The simulator p_i stores in the register $FD\_VAL[i]$ the last value it has read from its local failure detector variable which is denoted C.read().

```
operation IS[r].write_snapshot(v) is
(1)  if ((r mod n) + 1 = i)
(2)    then repeat arw_view_i ← S[r].collect(); FD_VAL[i] ← C.read()
(3)         until (PROP_C(arw_view_i)) end repeat
(4)    else FD_VAL[i] ← C.read()
(5)  end if;
(6)  iis_view ← S[r].arw_write_snapshot(v);
(7)  return (iis_view).

operation C*.read() is return (FD_VAL[i]).
```

Figure 3: A generic simulation of $\mathcal{IIS}_n[C^*]$ in $\mathcal{ARW}_n[C]$: code for a simulator p_i

Where is the problem to solve If the underlying model was $\mathcal{ARW}_n[\emptyset]$ (no failure detector), the simulation of the operation $IS[r]$.write_snap would boil down to a simple call to $S[r]$.arw_write_snapshot() (lines 6-7). Hence, the main difficulty to simulate $IS[r]$.write_snapshot(v) comes from the presence of the failure detector C.

This comes from the fact that, in all executions, we need to guarantee a correct association between the schedule of the (simulated) invocations of $IS[r]$.write_snapshot() and the outputs of the simulated failure detector C^*. This, which depends on the output of the underlying failure detector C, requires to appropriately synchronize, at every round r, the simulation of the invocations of $IS[r]$.write_snapshot(). Once, this is done, the set-linearization of the simulated invocations of $IS[r]$.write_snapshot() follows from the set-linearization of these invocations in the $\mathcal{ARW}_n[C]$ model.

Associate each round with a simulator The simulation associates each round r with a simulator (we say that the corresponding simulator "owns" round r) in such a way that each correct simulator owns an infinite number of rounds. This is implemented with a simple round-robin technique (line 1).

Simulation of $IS[r]$.write_snapshot(v) To simulate an invocation $IS[r]$.write_snapshot(v) issued by the simulated process q_i, the simulator p_i first checks if it is the owner of the corresponding round r. If it is not, it refreshes the value of $FD\_VAL[i]$ (line 4) and executes the "common part", namely, it invokes $S[r]$.arw_write_snapshot(v) (line 6) which returns it a set iis_view that constitutes the result of the invocation of $IS[r]$.write_snapshot(v).

If the simulator p_i is the owner of the round, it repeatedly reads asynchronously the current value of the implementation object $S[r]$ (that it stores in arw_view_i) and refreshes the value of $FD\_VAL[i]$ (line 2). This **repeat** statement terminates when the values of arw_view_i it has obtained satisfy some predicate (line 3). This predicate, denoted $\text{PROP}_C()$, which depends on the failure detector class C, encapsulates the generic dimension of the simulation. Then, after it has exited the loop, the simulator p_i executes the "common" part, i.e., lines 6-7. It invokes $S[r]$.arw_write_snapshot(v) which provides it with a view iis_view which is returned as the result of the invocation of $IS[r]$.write_snapshot(v).

The fact that, during each round, (a) some code is executed only by the simulator that owns r, (b) some code is executed only by the other simulators and (c) some code is executed by all simulators, realizes the synchronization discussed above that allows for a correct set-linearization of the invocations of $IS[r]$.write_snapshot() in $\mathcal{IIS}_n[C^*]$.

Simulation of C^*.read() When a simulated process q_i wants to read its local failure detector output, its simulator p_i returns it the last value it has read from its local failure detector variable.

To summarize When simulating $\mathcal{IIS}_n[C^*]$ on top of $\mathcal{ARW}_n[C]$, we have the following: (a) a faulty simulator p_i gives rise to a faulty simulated process q_i and (b) a correct simulator p_i gives rise either to a strongly correct, a weakly correct or a faulty simulated process q_i in $\mathcal{IIS}_n[C^*]$ (this can depend on $\text{PROP}_C()$).

Moreover, whatever C, we have to show that there is at least one correct process in $\mathcal{IIS}_n[C^*]$. This amounts to show that there is a simulator p_i that executes the infinite sequence $\{IS[r]\text{.write\_snapshot}()\}_{r\geq 1}$. To that end, we have to show that each object $IS[r]$ is non-blocking [12] (i.e., whatever the round r and the concurrency pattern, at least one invocation of $IS[r]$.write_snapshot() terminates). The corresponding proof is given when we consider specific failure detector classes (see below). Then, due to the structure of the IIS model, the very existence of at least one correct process in $\mathcal{IIS}_n[C^*]$ entails the existence of at least one strongly correct process in this model (see the definition of the set $\mathcal{SC}$ in Section 2.6).

4.2 Instantiating the simulation with $C = \Omega_k$

When $C = \Omega_k$, the property $\text{PROP}_C(arw\_view)$ can be instantiated at each simulator p_i as follows:

$$\text{PROP}_{\Omega_k}(arw\_view_i) \;=\; \big(\exists\, \ell \in FD\_VAL[i] :\; (\ell = i \;\vee\; \exists \langle \ell, - \rangle \in arw\_view_i)\big).$$

Let $leaders_i = FD\_VAL[i]$ (the last value of Ω_k read by the simulator p_i). The previous predicate directs the simulator p_i, at each round r it owns, to wait until $i \in leaders_i$ or until it has seen the simulation of $IS[r]$.write_snasphot() issued by a simulator q_j such that $j \in leaders_i$.

Theorem 2 *Let A be an algorithm solving a colorless task in the $\mathcal{IIS}_n[\Omega_k^*]$ model. The simulation of A on top of $\mathcal{ARW}_n[\Omega_k]$ where the invocations of $IS[r]$.write_snapshot() and C^*.read() are implemented by the algorithms described in Figure 3 and the predicate PROP_C is instantiated by PROP_{Ω_k}, produces an execution of A that could have been obtained in $\mathcal{IIS}_n[\Omega_k^*]$. Moreover, there is a one-to-one correspondence between the correct (simulated) processes in $\mathcal{IIS}_n[\Omega_k^*]$ and the correct simulators in $\mathcal{ARW}_n[\Omega_k]$.*

Proof The proof has to show that: (a) there is at least one correct process in $\mathcal{IIS}_n[\Omega_k^*]$ (consequently, as we have seen previously, there is a strongly correct process in $\mathcal{IIS}_n[\Omega_k^*]$); (b) there is a one-to-one correspondence between correct simulators (p_i) and correct simulated processes (q_i); (c) the behavior of the local failure detector variables of the processes in $\mathcal{IIS}_n[\Omega_k^*]$ is the one defined by Ω_k^*;

and, for any round r, (d) the invocations of $IS[r]$.write_snapshot() satisfy the self-inclusion, containment, and immediacy properties (defined in Section 2.4).

Proof of (a). As (by definition) one of the leaders eventually output by Ω_k is a correct simulator p_i, there is a finite time τ after which the predicate PROP_{Ω_k} returns always $true$ when evaluated by the simulator p_i. Hence, this simulator can never be blocked forever at line 3 when it executes (on behalf of the simulated process q_i) the algorithm implementing the operation $IS[r]$.write_snapshot(). It follows from this observation that there is at least one correct simulated process q_i.

Proof of (b). The correct simulator p_i which eventually belongs forever to the output of Ω_k invokes arw_write_snapshot(v) at each simulated round r (line 6). Let τ_r be the finite time instant at which this invocation terminates. As there is a finite time $\tau' \geq \tau$ after which i belongs to all the failure detector outputs, it follows that, at some finite time $\tau'' \geq \max(\tau', \tau_r)$, the evaluation of PROP_{Ω_k} by any correct simulator returns $true$ at round r. Hence that any correct simulator terminates its simulation of $IS[r]$.write_snapshot(). As this is true for any round r, any correct simulator simulates an infinite number of rounds of the IIS model. Consequently, if a simulator p_j is correct in $\mathcal{ARW}_n[\Omega_k^*]$ the associated simulated process q_j is correct in $\mathcal{IIS}_n[\Omega_k^*]$ (which entails that, at any round r, the simulation of the operation $IS[r]$.write_snapshot() is wait-free). Finally, a faulty simulator p_i trivially gives rise to a faulty simulated process q_i which concludes the proof of Item (b).

Proof of (c). To show that the behavior of the local failure detector variables at each simulated process q_x is the one defined by Ω_k^*, let us first observe that, it follows directly from lines 2 and 4 that the outputs of Ω_k^* are outputs of Ω_k. Hence, we have only to show that, after some time, Ω_k outputs forever a set L such that there is a simulator p_i with $i \in L \cap \mathcal{C}$ (let us remember that $\mathcal{C}$ is the set of correct simulators) and the simulated process q_i associated with the simulator p_i is strongly correct.

Let r be a round such that all the faulty simulators have crashed before r and, after r, all correct simulators obtain forever the same set of leaders L from Ω_k. Due to PROP_{Ω_k}, in all rounds $r' \geq r$, each correct simulator p_j, $j \notin L$, has to wait (at each round it owns) until a correct simulator p_i, $i \in L$, has written into $S[r']$ (execution of $S[r']$.arw_write_snapshot() by p_i at line 6 and execution of $S[r']$.collect() by p_j at line 2). It follows that, in the simulated system, the invocation of $IS[r']$.write_snapshot() issued by any simulated process q_j, $j \notin L$, is set-linearized after the invocation of $IS[r']$.write_snapshot() issued by a simulated process p_i such that $i \in L$.

Let q_j be a strongly correct simulated process (since there are some correct simulated processes, there is at least one strongly correct one). As q_j executes an infinite number of rounds and $1 \leq |L \cap \mathcal{C}| \leq k$, it follows that there is a correct simulated process q_ℓ such that $\ell \in L \cap \mathcal{C}$ and there is an infinite number of rounds r'' such that the invocation of $IS[r'']$.write_snapshot() issued by q_j is set-linearized after the one of q_ℓ. It follows that q_ℓ is a (simulated process which) is strongly correct.[4]

Proof of (d). The fact that, at any round r, the invocations of $IS[r]$.write_snapshot() satisfy the self-inclusion, containment and immediacy properties follow directly (as already indicated) from the fact that invocations of the underlying operation $S[r]$.arw_write_snapshot() satisfy these properties. $\square_{Theorem\ 2}$

The following corollary is an immediate consequence of the previous theorem.

Corollary 1 *A colorless task T is solvable in $\mathcal{ARW}_n[\Omega_k]$ iff it is solvable in $\mathcal{IIS}_n[\Omega_k^*]$.*

4.3 Instantiating the simulation with $C = \diamond P$

The failure detector class $C = \diamond P$ When $C = \diamond P$, the property $\text{PROP}_{\diamond P}(arw\_view)$ can be instantiated at each simulator p_i as follows:

$$\text{PROP}_{\diamond P}(arw\_view_i) \ = \ \big(\forall\, j \in FD\_VAL[i] : \ (j = i \ \vee \ \langle j, - \rangle \in arw\_view_i)\big).$$

This property forces the corresponding simulator p_i, at each round r it owns, to wait until all the simulators p_j that it currently trusts (i.e., any $j \in fd\_val_i(i)$) have invoked $S[r]$.arw_write_snapshot() (i.e., have written a pair $\langle j, - \rangle$ into $S[r]$).

Theorem 3 *Let A be an algorithm solving a colorless task in the $\mathcal{IIS}_n[\diamond P^*]$ model. The simulation of A on top of $\mathcal{ARW}_n[\diamond P]$ where the invocations of $IS[r]$.write_snapshot() and $\diamond P^*$.read() are implemented by the algorithms described in Figure 3 and the predicate PROP_C is instantiated by $\text{PROP}_{\diamond P}$, produces an execution of A that could have been obtained in $\mathcal{IIS}_n[\diamond P^*]$. Moreover, there is a one-to-one correspondence between the correct (simulated) processes in $\mathcal{IIS}_n[\diamond P^*]$ and the correct simulators in $\mathcal{ARW}_n[\diamond P]$ and all correct simulated processes are strongly correct.*

Proof The proof is similar to the previous one. We show here only that each correct simulator p_i gives rise to a strongly correct (simulated) process q_i.

[4] Let us observe that, while this proves that there is a correct simulator that gives rise to a strongly correct simulated process, we cannot determine how many simulated processes are strongly correct. This number depends on the execution.

The proof is by contradiction. Let us suppose that a correct simulator loops forever in the **repeat** (lines 2-3) when it executes the algorithm simulating $IS[r]$.write_snapshot(). Let r denote the first round during which this happens and let p_i be the simulator that owns this round (hence, p_i is the first simulator that loops forever).

Let us notice that, as $i = (r \mod n) + 1$, it is the only simulator which loops at round r. Hence, all other correct simulators eventually invoke $S[r]$.arw_write_snapshot() at line 6 and consequently return from their simulation of $IS[r]$.write_snapshot(). As the failure detector $\diamond P$ eventually outputs a set including only correct simulators, the evaluation of the predicate $\text{PROP}_{\diamond P}(r, -, -)$ eventually returns $true$ to p_i which terminates its simulation of $IS[r]$.write_snapshot(). A contradiction. It follows from this contradiction that every correct simulator executes an infinite number of rounds, which means that, at any round r, the implementation of the operation $IS[r]$.write_snapshot() is wait-free.

Finally, as, at each round it owns, each correct simulator p_i waits for all other correct simulators, each correct simulator sees the simulation of $IS[r]$.write_snapshot() by the other correct simulators infinitely often. Hence, each correct simulator p_i simulates a strongly correct process q_i. $\square_{Theorem\ 3}$

4.4 Instantiating the simulation with $C = P,\ \Sigma,\ S,\ \diamond S,\ S_x,\ \diamond S_x$

The same predicate PROP_C as the one used for $\diamond P$ works for these failure detector classes.

The failure detector classes $C = P,\ \Sigma,\ S,\ \diamond S$ The previous proof can be easily translated for the the failure detector classes $C = P,\ \Sigma,\ S,\ \diamond S$. This follows from the observation that, as $\diamond P$, each of these failure detector classes permanently ($C = P$) or eventually ($C = \Sigma,\ S,\ \diamond S$) output only sets of correct processes.

The failure detector classes $C = S_x,\ \diamond S_x$ These failure detector classes (introduced in [2]) extend the classes S and $\diamond S$. Intuitively, they restrict the properties defining S and $\diamond S$ to be only on a dynamically determined subset of processes Q such that $|Q| = x$ (hence their name: limited scope failure detector classes).

It is possible to show that the algorithms of Figure 3 instantiated with the previous predicate $\text{PROP}_{\diamond P}()$ allows for a correct simulation of $\mathcal{IIS}_n[C^*]$ in $\mathcal{ARW}_n[C]$ for $C = S_x,\ \diamond S_x$. Let us also remark that, for any r, the simulation of the operation $IS[r]$.write_snapshot() is wait-free (each simulated process q_i whose simulator p_i is correct executes an infinite number of IIS rounds). However, while each correct simulator simulates a strongly correct process when $C \in \{P, \diamond P\}$, no conclusion can be drawn from the number of strongly correct simulated processes (except that there is at least one) when $C \in \{\Sigma, S_x, \diamond S_x\}$.

5 From wait-freedom to t-resilience

Notation Let $\mathcal{IIS}_{n,t}[C]$ denote the extended $\mathcal{IIS}_n[C]$ model in which at least $n - t$ processes are strongly correct, i.e., $|\mathcal{SC}| \geq n - t$ and $|\mathcal{WC}| + |\mathcal{F}| \leq t$. Similarly, let $\mathcal{ARW}_{n,t}[C]$ denote the extended $\mathcal{ARW}_n[C]$ model in which at most t processes are faulty.

From $\mathcal{IIS}_{n,t}[C^*]$ to $\mathcal{ARW}_{n,t}[C]$ Theorem 1 has shown that the simulation described in Figure 2 (which is a simple extension to failure detectors of the simulation described in [4]) ensures that (a) any strongly correct simulator in IIS gives rise to a correct simulated process in ARW and (b) a weakly or faulty simulator gives rise to a faulty simulated process. It follows that if $|\mathcal{SC}| \geq n - t$ in $\mathcal{IIS}_{n,t}[C^*]$ we have at most t faulty process in the simulated system $\mathcal{ARW}_{n,t}[C]$.

From $\mathcal{ARW}_{n,t}[C]$ to $\mathcal{IIS}_{n,t}[C^*]$ In this direction, the simulation from $\mathcal{ARW}_n[C]$ in $\mathcal{IIS}_n[C^*]$ presented in Figure 3 can be easily adapted in order to simulate $\mathcal{ARW}_{n,t}[C]$ in $\mathcal{IIS}_{n,t}[C^*]$.

It is indeed sufficient to replace PROP_C by $\text{PROP}_C \wedge |arw\_view_i| \geq (n - t - 1)$ (it is of course assumed that we do not have $|arw\_view_i| \geq (n - t - 1) \Rightarrow \neg\ \text{PROP}_C$). In this way, at every round r it owns, each correct simulator p_i is constrained to wait until at least $n - t - 1$ processes have invoked $S[r]$.arw_write_snapshot() before being allowed to invoke its own. The correction of this extended simulation is captured in the following theorem.

Theorem 4 *Let A be an algorithm solving a colorless task in the $\mathcal{IIS}_n[C^*]$ model. For the failure detector classes studied in this paper, The simulation of A on top of $\mathcal{ARW}_n[C]$ where the invocations of $IS[r]$.write_snapshot() and C^*.read() are implemented by the algorithms described in Figure 3 and the predicate PROP_C is replaced by $\text{PROP}_C \wedge |arw\_view_i| \geq (n - t - 1)$, produces a correct execution of A in $\mathcal{IIS}_n[C^*]$ in which $n - t$ processes are strongly correct.*

Proof The proof consists in showing that (a) each correct simulator that would simulate an infinite number of IIS rounds when considering only PROP_C does simulate an infinite number of rounds when considering $\text{PROP}_C \wedge (arw\_view_i \geq n - t - 1)$; and (b) there is at least $(n - t)$ strongly correct processes in the simulated IIS model.

Proof of (a). The proof by contradiction. Let us suppose that a correct simulator p_i that, at some round r, blocks forever because the predicate $(arw\_view_i \geq n-t-1)$ is never satisfied. Let r be the first round at which this occurs. By assumption, at least $n-t-1$ simulators p_j eventually invoke $S[r]$.arw_write_snapshot(), it follows that the predicate $(arw\_view_i \geq n-t-1)$ eventually becomes true. Hence the predicate $\text{PROP}_C \wedge (arw\_view_i \geq n-t-1)$ eventually becomes true which concludes the proof of item (a).

Proof of (b). Due to the previous item, there is a correct simulator p_i that that simulates a strongly correct process. Let p_i such a simulator. This simulator p_i simulates an infinite number of rounds and, in each round r it owns, it simulates $IS[r]$.write_snapshot() on behalf of q_i, after (or simultaneously with) the invocations of $IS[r]$.write_snapshot() issued by at least $n-t$ processes ($n-t-1$ other processes plus itself). As there is a bounded number of processes, p_i consequently simulates its write-snapshots infinitely often after (or simultaneously with) those of at least $n-t$ simulators. Hence at least $n-t$ simulated processes are strongly correct and (b) follows.
$\square_{Theorem\ 4}$

6 Conclusion

This paper has addressed the respective power of the base asynchronous read/write model ARW and the iterated immediate snapshot model IIS when both are enriched with the same non-trivial failure detector (non-trivial means that the failure detector cannot be implemented in the base ARW model). The paper has shown that, once enriched with the same failure detector, both models have the same computational power when the notion of a *correct* process used in the ARW model is replaced by the notion of a *strongly correct* when considering the IIS model. This notion is related to the writes seen by a process at each round of the IIS model, more precisely, a process is strongly correct if all its writes into the sequence of immediate snapshot objects are eventually propagated to all the processes that do not crash.

The paper has presented a formal definition of a strongly correct process and two simulations, a first one from IIS to ARW and a second one from ARW to IIS. It has also proved these simulations and shown how they can be extended when the wait-freedom is replaced by t-resilience. An appendix of the paper has also presented a consensus algorithm suited to the IIS model enriched with an eventual leader failure detector.

Acknowledgments

This work has been partially supported by the French ANR project DISPLEXITY devoted to the computability and complexity in distributed computing.

References

[1] Afek Y., Attiya H., Dolev D., Gafni E., Merritt M. and Shavit N., Atomic snapshots of shared memory. *Journal of the ACM*, 40(4):873-890, 1993.

[2] Anceaume E., Fernandez A., Mostefaoui A., G. Neiger G. and Raynal M., A necessary and sufficient condition for transforming limited accuracy failure detectors. *Journal of Computer and System Sciences*, 68:123-133, 2004.

[3] Borowsky E. and Gafni E., Immediate atomic snapshots and fast renaming. *Proc. 12th ACM Symposium on Principles of Distributed Computing (PODC'93)*, pp. 41-51, 1993.

[4] Borowsky E. and Gafni E., A simple algorithmically reasoned characterization of wait-free computations. *Proc. 16th ACM Symposium on Principles of Distributed Computing (PODC'97)*, pp. 189-198, 1997.

[5] Castañeda A., Rajsbaum S. and Raynal M., The renaming problem in shared memory systems: an introduction. *Elsevier Computer Science Review*, 5:229-251, 2011.

[6] Chandra T. and Toueg S., Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225-267, 1996.

[7] Chandra T., Hadzilacos V. and Toueg S., The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685-722, 1996.

[8] Chaudhuri S., More *Choices* Allow More *Faults:* Set Consensus Problems in Totally Asynchronous Systems. *Information and Computation,* 105:132-158, 1993.

[9] Cornejo A., Rajsbaum S., Raynal M., Travers C., Failure Detectors as Schedulers (Brief Announcement). *Proc. 26th ACM Symposium on Principles of Distributed Computing (PODC)*, ACM Press, pp.308-309, 2007.

[10] Delporte-Gallet C., Fauconnier H., Guerraoui R., Hadzilacos V., Kouznetsov P. and Toueg S., The weakest failure detectors to solve certain fundamental problems in distributed computing. *Proc. 23th ACM Symposium on Principles of Distributed Computing (PODC'04)*, ACM Press, pp. 338-346, 2004.

[11] Gafni E. and Rajsbaum S. Distributed Programming with Tasks. *Proc. 14th Int'l Conference On Principles Of Distributed Systems (OPODIS)*, Springer Verlag LNCS #6490, pp. 205-218, 2010.

[12] Herlihy M.P., Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124-149, 1991.

[13] Herlihy M. and Shavit N., The topological structure of asynchronous computability. *Journal of the ACM*, 46(6):858-923, 1999.

[14] Herlihy M.P. and Wing J.M., Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463-492, 1990.

[15] Lo W.-K. and Hadzilacos V., Using failure detectors to solve consensus in asynchronous shared-memory systems. *Proc. 8th Int'l Workshop on Distributed Algorithms (WDAG'94, now DISC)*, Springer-Verlag LNCS #857, pp. 280-295, 1994.

[16] Loui M. and Abu-Amara H., Memory requirements for for agreement among Unreliable Asynchronous processes. *Advances in Computing Research*, 4:163-183, JAI Press Inc., 1987.

[17] Mostefaoui A., Rajsbaum S., Raynal M. and Travers C., On the Computability Power and the Robustness of Set Agreement-oriented Failure Detector Classes. *Distributed Computing*, 21(3):201-222, 2008.

[18] Neiger G., Set Linearizability. *Brief Announcement, Proc. 13th ACM Symposium on Principles of Distributed Computing (PODC)*, ACM Press, pp. 396, 1994.

[19] Neiger G., Failure detectors and the wait-free hierarchy. *Proc. 14th ACM Symposium on Principles of Distributed Computing (PODC'95)*, ACM Press, pp. 100-109, 1995.

[20] Pike S.M., Sastry S. and Welch J.L., Failure detectors encapsulate fairness. *Proc. 14th Int'l Conference On Principles Of Distributed Systems (OPODIS)*, Springer Verlag LNCS #6490, pp. 173-188, 2010.

[21] Rajsbaum S., Iterated shared memory models. *Proc. 9th Latin American Symposium Theoretical Informatics (LATIN'10)*, Springer Verlag LNCS #6343, pp.407-416, 2010.

[22] Rajsbaum S., Raynal M., Travers C., The iterated restricted immediate snapshot model. *Proc. 14th Annual Int'l Conference Computing and Combinatorics (COCOON 2008)*, Springer Verlag LNCS #5092, pp. 487-497, 2008.

[23] Rajsbaum S., Raynal M., Travers C., An impossibility about failure detectors in the iterated immediate snapshot model. *Information Processing Letters*, 108(3):160-164, 2008.

[24] Raynal M., Failure detectors for asynchronous distributed systems: an introduction. *Wiley Encyclopdia of Computer Science and Engineering*, Vol. 2, pp. 1181-1191, 2009 (ISBN 978-0-471-38393-2).

A Proof of Theorem 1

Theorem 1 Let A be an algorithm designed for the $\mathcal{ARW}_n[C]$ model. Let us consider an execution of A simulated in the $\mathcal{IIS}_n[C^*]$ model by the algorithms described in Figure 2. A process p_i is correct in the simulated execution if and only if its simulator q_i is strongly correct.

Proof Let us first observe that a simulated process whose simulator eventually crashes in $\mathcal{IIS}_n$ is faulty (it cannot issue an infinite number of steps since its simulator crashes after a finite number of steps). Hence, it remains to show that (a) all simulators in $\mathcal{WC}$ simulate faulty processes and (b) all simulators in $\mathcal{SC}$ simulate correct processes .

Proof of (a). Let us first remark that, as there is a bounded number of simulators and (by assumption) at least one of them is correct, $|\mathcal{SC}| \geq 1$. If $\mathcal{C} = \mathcal{SC}$ (i.e., $\mathcal{WC} = \emptyset$), (a) trivially follows. Hence, let us consider that there is at least one weakly correct simulator q_i.

Let $rmax$ be a round number such that (1) $\forall i \in \mathcal{WC}, \forall j \in \mathcal{SC}, \forall r \geq rmax \ : \ \langle i, - \rangle \notin is\_view_j[r]$ and (2) no faulty simulator starts round $rmax$. At any round $r \geq rmax$, any strongly correct simulator issues its invocation of $IS[r]$.write_snapshot() strictly before those of any weakly correct simulator. Consequently, when after round $rmax$, q_i, $i \in \mathcal{WC}$, executes the algorithm that simulates the next invocation of S.write() or S.snapshot() issued by p_i, encoded in the triple (i, sn, x), the only simulators that can see this triple are weakly correct simulators. Then, as after $rmax$ the simulator q_i sees all the triples written or known by the strongly correct simulators, its predicate of line 5 will never be verified and q_i will loop forever. Hence the simulated process p_i never ends its invocation of S.write() or S.snapshot(): it does no longer progress and appears as a crashed process in the simulated execution in $\mathcal{ASW}_n[C]$.

Proof of (b). Let now q_i be a strongly correct simulator. By definition of $\mathcal{SC}$, there is a set $\mathcal{SC}_k$ such that $i \in \mathcal{SC}_k$ and there is an infinite sequence of invocations of $IS[r]$.write_snapshot() issued by q_i which occur before those of some simulators of $\mathcal{SC}_{k-1}$. It follows that, each time q_i simulates a S.write() or S.snapshot() invocation on behalf of p_i, after a finite number of rounds, the corresponding triple (i, sn, x) is added to the set of its known simulated operations by a simulator of $\mathcal{SC}_{k-1}$. Then, in the same way and in a finite number

of rounds, that simulator propagates this triple to a simulator $\mathcal{SC}_{k-2}$. This continues recursively and, in a finite number of rounds, the triple (i, sn, x) written by q_i is added to the set ops_y of known operations by a simulator q_y, $y \in \mathcal{SC}_0$. Then, it follows from the definition of $\mathcal{SC}_0$ that q_y propagates the triple that becomes eventually known by all correct simulators. After that, the next invocation of $S[r']$.write_snapshot() issued by q_i is such that the predicate of line 5 is satisfied and q_i can terminate its simulation of the operation S.write() or S.snapshot() issued by the process p_i. It follows that the simulated process p_i always progresses and it is consequently correct with respect to the simulated execution in $\mathcal{ARW}_n[C]$.

$\square_{Theorem\ 1}$

B An implementation of the arw_write_snapshot() operation

For a completeness purpose, this appendix presents a wait-free implementation of the operation write_snapshot() (called arw_write_snapshot(in Section 4) suited to $\mathcal{ARW}_n[\emptyset]$.

```
operation write_snapshot(v_i) is
    REG[i] ← v_i;
    repeat LEVEL[i] ← LEVEL[i] − 1;
           for j ∈ {1, ..., n} do level_i[j] ← LEVEL[j] end for;
           view_i ← {j : level_i[j] ≤ LEVEL[i]};
    until (|view_i| ≥ LEVEL[i]) end repeat;
    return({⟨j, REG[j]⟩ such that j ∈ view_i}).
```

Figure 4: Borowsky-Gafni's wait-free algorithm for write_snapshot() in $\mathcal{ARW}_n[\emptyset]$: code for p_i

.

This algorithm, described in Figure 4. is due to Borowsky and Gafni [3]. It considers that each process invokes the operation at most once and uses two arrays of single-writer/multi-reader atomic registers denoted $REG[1..n]$ and $LEVEL[1..n]$ (only p_i can write $REG[i]$ and $LEVEL[i]$). A process p_i first writes its value in $REG[i]$. Then the core of the implementation of write_snapshot() is based on the array $LEVEL[1..n]$. That array, initialized to $[n+1, \ldots, n+1]$, can be thought of as a ladder, where initially a process is at the top of the ladder, namely, at level $n+1$. Then it descends the ladder, one step after the other, according to predefined rules until it stops at some level (or crashes). While descending the ladder, a process p_i registers its current position in the ladder in the atomic register $LEVEL[i]$.

After it has stepped down from one ladder level to the next one, a process p_i computes a local view (denoted $view_i$) of the progress of the other processes in their descent of the ladder. That view contains the processes p_j seen by p_i at the same or a lower ladder level (i.e., such that $level_i[j] \leq LEVEL[i]$). Then, if the current level ℓ of p_i is such that p_i sees at least ℓ processes in its view (i.e., processes that are at its level or a lower level) it stops at the level ℓ of the ladder. Finally, p_i returns a set of pairs determined from the values of $view_i$. Each pair is a process index and the value written by the corresponding process. This algorithm satisfies the self-inclusion, containment and immediacy properties stated in Section 2.4.

For the interested reader, a recursive algorithm implementing the operation write_snapshot() is described in [5, 11].

C A consensus algorithm for the IIS model

For completeness, this appendix presents an algorithm designed for an $\mathcal{IIS}_n[C^*]$ model, namely, a consensus algorithm for $\mathcal{IIS}_n[\Omega^*]$.

C.1 Description of the algorithm

Global variables As we consider the IIS model, there is an infinite array IS of immediate snapshot objects, such that $IS[r]$ is accessed by a process only when it executes round r.

Local variables Each process p_i manages the following local variables.

- r_i is the local round number.
- est_i is the local estimate of the decision value. It is initialized to v_i, the value proposed by p_i.
- r_leader_i is the leader known by p_i at the current round. It is initialized to the default value $\perp$.
  cur_leader_i is an auxiliary variable that contains the leader id obtained by p_i at the beginning of the current round.
- dec_i is a one-write boolean variable initialized to $false$. It is set to $true$ when p_i decides and keeps then that value forever.

- $val\_ld\_seen_i$ is a boolean initialized to $false$. It is set to $true$ during a round r if and only if p_i knows the current estimate value of the leader of the current round and all processes known by p_i have the same leader.

```
operation propose (v_i)
(1)   init: r_i ← 0; est_i ← v_i; r_leader_i ← ⊥; val_ld_seen_i ← false; dec_i ← false;
(2)   repeat forever (asynchronous IIS rounds)
(3)     r_i ← r_i + 1;
(4)     cur_leader_i ← Ω*.read();
(5)     if (cur_leader_i ≠ r_leader_i) then val_ld_seen_i ← false; r_leader_i ← cur_leader_i end if;
(6)     iis_view_i ← IS[r_i].write_snapshot(⟨est_i, r_leader_i, val_ld_seen_i, dec_i⟩);
(7)     if (¬ dec_i) then
(8)       if  (∃⟨−, ⟨v, −, −, true⟩⟩ ∈ iis_view_i)
(9)         ∨(∃v : ∀⟨j, ⟨est, −, val_ld_seen, −⟩⟩ ∈ iis_view_i : (val_ld_seen = true ∧ est = v))
(10)        then est_i ← v; dec_i ← true; decide (est_i)
(11)        else if (∃ℓ : ∀⟨j, ⟨−, r_leader, −, −⟩ ∈ iis_view_i : r_leader = ℓ)
(12)                then if (∃ v : (⟨ℓ, ⟨v, −, −, −⟩⟩ ∈ iis_view_i) ∨ (∃⟨−, ⟨v, −, true, −⟩⟩ ∈ iis_view_i))
(13)                     then est_i ← v; val_ld_seen_i ← true
(14)                     end if
(15)                else if (∃⟨−, ⟨v, −, true, −⟩⟩ ∈ iis_view_i) then est_i ← v end if; val_ld_seen_i ← false
(16)                end if
(17)      end if
(18)    end if
(19)  end repeat
end operation.
```

Figure 5: Consensus in $\mathcal{IIS}_n\Omega^*$

The algorithm is described in Figure 5. When a process invokes propose (v_i) where v_i is the value it proposes, it first initializes its local variables (line 1). Then it enters an infinite loop (lines 2-19). Within the loop, the behavior of a process p_i can be decomposed into three phases.

- A process p_i first increases r_i (line 3) and reads the current value of its local failure detector output (line 4). If the leader has changed since the last round, p_i resets accordingly $val\_ld\_seen_i$ to $false$ and r_leader_i (line 5).
- Then, p_i invokes $IS[r]$.write_snapshot() to write its local state which abstracted as a tuple of 4 values $\langle est_i, r\_leader_i, val\_ld\_seen_i, de$ When it returns from that invocation, it obtains a view made up of pairs $\langle j, \langle est, r\_leader, val\_ld\_seen, dec\rangle\rangle$ deposited by other processes (line 6).
- Then the behavior of p_i depends on dec_i. If it has already decided, p_i proceeds directly to the next round. Otherwise it checks a decision predicate (lines 9-10). This predicate is true if the view iis_view_i just obtained by p_i contains a value v already decided by a process (line 9) or all the processes p_j that appear in iis_view_i have written the same estimate value $est = v$ and this value comes from the same leader, namely $val\_ld\_seen = true$ (line 10).
  - It the predicate is satisfied, p_i adopts v as new estimate, decides (line 10) and proceeds to the next round.
  - If the decision predicate is false, p_i strives to progress to a decision. To that end, it checks if all the processes in its view have the same leader p_ℓ (line 11).
    - If all the processes in its view have the same leader p_ℓ and p_i knows the estimate v of p_ℓ (first predicate of line 12) or knows that a process knows that value (first predicate of line 12), then it adopts v as its new estimate value est_i and sets $val\_ld\_seen_i$ to $true$ (line 13). If none of this predicates is satisfied, est_i and $val\_ld\_seen_i$ keep their previous values.
    - If the processes in its view do not share the same leader, p_i sets $val\_ld\_seen_i$ to $false$ and, if its view contains a tuple $\langle v, -, true, -\rangle$, it updates its estimate est_i to v (line 15).

As we can see, the principle of the algorithm is the following. A process that is considered as a leader tries to impose its estimate value as the decision value. As Ω^* does not guarantee a common leader since the very beginning, the processes are required to "vote" when they see a common leader in their view (this is captured by the boolean variables $val\_ld\_seen_i$). This idea is made operational by the lines 8-17.

C.2 Proof of the consensus algorithm

Lemma 1 *When executed in $\mathcal{IIS}_n[\Omega^*]$, the algorithm described in Figure 5 satisfies the consensus validity property.*

Proof All the decided values are read from the local variables est_i (line 10). These variables are initialized to proposed values (line 1). They are then updated with values v read from tuples contained in iis_view_i (lines 10, 13 and 15). The views iis_view_i are returned

from the immediate snapshot objects (line 6). The rest of the proof shows that any tuples written in an immediate snapshot object contains a proposed values.

Let us consider the value est contained in a tuple written by a given in $IS[r]$ by a process p_i, with $r > 1$ (line 6). It is either a tuple p_i has written in $IS[r-1]$ (if one of its predicates at line 7, 12 or 15 was false during round $r-1$), or the one it has adopted from $IS[r-1]$ (through iis_view_i at line 10, 13 or 15). In both cases that tuple has been written in the immediate snapshot object during the previous round. Since only proposed values belong to tuples written in $IS[1]$ (est_i is not modified between line 1 and 6), the only values inside tuples written in $IS[r], r > 0$ are proposed values and the validity follows. $\square_{Lemma\ 1}$

Lemma 2 *When executed in $\mathcal{IIS}_n[\Omega^*]$, the algorithm described in Figure 5 satisfies the consensus termination property, i.e., any strongly correct process decide.*

Proof Let us first observe that, as soon as a process decide (invocation of decide() at line 10), it repeatedly write tuples carrying the value it has decided together with the flag $dec_i = true$ (line 6). Moreover, when a process retrieves such a tuple from an invocation $IS[r]$.write_snapshot() it also immediately decides (lines 8 and 10) and consequently writes tuples with the same decided value in the following rounds (if it does not crash).

For any process p_i and any round number r such that p_i invokes $IS[r]$.write_snapshot(), let P_i^r be the non-empty set of processes that see this write in the view they obtain from their own invocation of $IS[r]$.write_snapshot(). If a process p_i decides at round r, the processes of P_i^{r+1} that have not decided yet decide during round $r+1$. Then, at the beginning of round $r+2$, all the processes of P_i^{r+1} have decided, and it follows that all the processes of $\bigcup_{j \in P_i^{r+1}} P_j^{r+2}$ have decided by the beginning of round $r+3$ and recursively. It follows that all processes that would have been informed of p_i state at round r in a full information algorithm following the same schedule eventually decide. By definition of the strongly correct processes, it follows that, if one of them decide, then, in a finite number of round, all alive processes decide.

Suppose now that no strongly correct process ever decide. Let then p_ℓ be the strongly correct leader elected by Ω^*. It follows from the definition of "strongly correct process" and Ω^* that there is a round number $rmax$ such that: (1) all faulty processes have crashed before invoking $IS[rmax]$.write_snapshot(), (2) for all $r \geq rmax$, the invocations of $IS[r]$.write_snapshot() issued by the strongly correct processes are set-linearized strictly before those of weakly correct processes, and (3) all the reads of Ω^* issued by strongly correct processes after the beginning of $rmax$ return ℓ. Consequently, after the beginning of $rmax$, the predicate of line 11 is always true for all the strongly correct processes.

Since p_ℓ is correct and never decides, it executes an infinite number of rounds without ever invoking decide()(line 10). When it executes the round $rmax$, the view it obtains from $IS[rmax]$.write_snapshot() contains only pairs written by strongly correct processes. As it does not decide, it checks the predicate line 11 and according to the definition of $rmax$, this predicate returns $true$. Process p_ℓ then makes $true$ the first part of the predicate of line 12 and (according to the self-inclusion property of the immediate snapshot object $IS[rmax]$) p_ℓ consequently sets $val\_ld\_seen_\ell$ to true at the end of round $rmax$.

According to the definition of $rmax$, it follows that, after the beginning of round $rmax+1$, the predicate of line 5 is never satisfied while the predicate of line 11 is always satisfied by any strongly correct process. These processes consequently never reset their variables val_ld_seen to false. Hence, after round $rmax$, each strongly correct process p_k that eventually verifies the predicate of line 12 sets $val\_ld\_seen_k$ to true and keep doing it forever.

After $rmax+1$, p_ℓ and all the processes that obtain the tuple in their view repeatedly write tuples carrying the flag $val\_ld\_seen = true$. Like in the case of tuples carrying $dec = true$, these flags eventually reach all alive processes. Consequently, all strongly correct processes eventually set $val\_ld\_seen = true$ and then keep it so forever.

Let us consider now that the following claim (proved below in Lemma 3) is verified: (C1) if two processes write tuples with $val\_ld\_seen = true$ and $dec = false$ in the same $IS[r]$ object, then these tuples carry also the same est. It follows from this claim that, as soon as all strongly correct processes have (1) reached the round $rmax+1$ and (2) set their variable val_ld_seen to true, they all verify the predicate line 9 and decide, what contradicts the initial hypothesis. This ends the proof of termination under the assumption that C1 is true. $\square_{Lemma\ 2}$

Lemma 3 *When executed in $\mathcal{IIS}_n[\Omega^*]$, the algorithm described in Figure 5 satisfies the consensus agreement property.*

Proof Let us consider an execution of the algorithm of Figure 5, where p_i a process that terminates its invocation of $IS[r]$.write_snapshot(), $r > 0$, and iis_view_i the view it obtains. Let $\ell_i[r]$ be a process identity or the default value $\perp$ according to the following rules:

- $\big(\forall\ \langle j, \langle -, \ell_j, -, - \rangle\rangle \in iis\_view_i\ :\ \ell_j = \ell\big) \Rightarrow \ell_i[r] = \ell$,
- $\big(\exists\ \langle j, \langle -, \ell_j, -, - \rangle\rangle,\ \langle k, \langle -, \ell_k, -, - \rangle\rangle \in iis\_view_i\ :\ \ell_j \neq \ell_k\big) \Rightarrow \ell_i[r] = \perp$.

Namely, if all processes seen by p_i during round r agree on the same round leader, then $\ell_i[r]$ is this leader, otherwise it is $\perp$. (Let us remark that one could replace the predicate of line 11 by $\exists \ell \neq \perp\ :\ \ell_i[r_i] = \ell$.)

The set-linearizability of the invocations of write_snapshot(), implies these two properties, where $iis\_view_i[r]$ denotes the view obtained by a process p_i during round r:

- $\forall\, i\, :\, \big((\ell_i[r] = \bot) \Rightarrow (\forall j\, :\, (\langle i, -\rangle \in iis\_view_j[r] \Rightarrow \ell_j[r] = \bot))\big)$,
- $\forall\, i, j\, :\, \big((\ell_i[r] \neq \bot \wedge \ell_j[r] \neq \bot) \Rightarrow (\ell_i[r] = \ell_j[r])\big)$.

The first property one states that, if a process sees at least two different leaders in the view it obtains from $IS[r]$.write_snapshot(), then all processes that have their invocation of $IS[r]$.write_snapshot() set-linearized after p_i's one see also at least two different leaders. (The proof follows directly from the containment property of the write_snapshot() operation.) On the other hand, the second property ensures that there can be at most one round leader for a given round. (This follows from both the containment and immediacy properties of the write_snapshot() operation.) It is thus possible to define $\ell[r]$ as the unique non-$\bot$ leader of the round r or as $\bot$ if $\forall i\, :\, \ell_i[r] = \bot$.

Remark that a process p_i that writes a tuple carrying the flag $val\_ld\_seen = true$ during round r necessarily verifies (1) $\ell_i[r-1] \neq \bot$ and (2) the value it retrieves from Ω^* at the beginning of round r is $\ell[r-1]$. It follows from that remark and from the second property of $\ell_i[r]$ that all the tuples written at a given round with the tag $val\_ld\_seen = true$ carry the same value of r_leader.

Let now p_ℓ be a process such that (1) a process p_i writes $\langle -, \ell, true, false\rangle$ during a round r and (2) no process writes $\langle -, \ell, true, false\rangle$ during the round $r-1$ (note that all tuples written into $WS[1]$ are such that $val\_ld\_seen = false$). The previous observations entail that $\ell[r-1] = \ell$ and that all the tuples of the type $\langle -, \ell, true, false\rangle$ written during the round r carry the same estimate. Indeed, the processes that wrote these tuples have necessarily executed line 13 at the end of round $r-1$, hence their first predicate of line 12 was verified (this cannot be due to the second predicate of line 12 since their predicate line 11 was verified and no process writes $\langle -, \ell, true, false\rangle$ during round $r-1$). They consequently all verified the first part of the predicate line 12 and all adopted the unique estimate value written by p_ℓ at round $r-1$.

Similarly, all the processes that write a tuple $\langle -, \ell, true, false\rangle$ during the round $r+1$ (if there are some) have adopted their estimate at line 13, so it is the only one present in the immediate snapshot object $IS[r]$ (and recursively while there are such processes). It follows that during a given round r, all the tuples carrying a flag $val\_ld\_seen = true$ also contain the same estimate and the same round leader, which proves Claim C1.

Consider the first round r during which a process decides, let p_i be such a process and v the value it decides. Necessarily p_i decides because the predicate line 9 is true (otherwise another process would have decided during round $r-1$ which would contradicts the definition of r). Hence, according to Claim C1 and the containment property of the immediate snapshot objects, all the processes that decide during round r decide the same value. Moreover, since p_i obtains a view where all tuples are such that $\langle v, -, true, false\rangle$, all other processes that terminates round r obtain a view where there is at least one such tuple. They consequently execute line 13 or line 15 according to the value of the predicate line 11. In both cases they adopt v at the end of round r. The only value written in all the objects $IS[r'], r' > r$, is then v and the agreement property follows. $\Box_{Lemma\ 3}$

Theorem 5 *When executed in $\mathcal{IIS}_n[\Omega^*]$, the algorithm described in Figure 5 solves the the consensus problem.*

Proof The proof follows directly Lemma 1, Lemma 2 and Lemma 3. $\Box_{Theorem\ 5}$