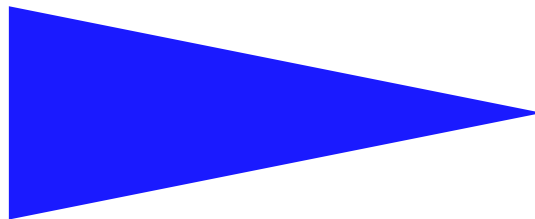


PUBLICATION
INTERNE
N° 1838



**FAILURE DETECTORS AS SCHEDULERS
(AN ALGORITHMICALLY-REASONED
CHARACTERIZATION)**

S. RAJSBAUM M. RAYNAL C. TRAVERS

Failure Detectors as Schedulers (An Algorithmically-Reasoned Characterization)

S. Rajsbaum* M. Raynal** C. Travers***

Systèmes communicants
Projet ASAP

Publication interne n° 1838 — Mars 2007 — 38 pages

Abstract: This paper presents a new approach to study unreliable failure detectors. It uses the *iterated immediate snapshot model* (IIS) to capture the precise amount of synchrony achievable by a failure detector. The IIS model is a round-based model consisting of one-shot objects that provide processes with a single operation to atomically write and snapshot the shared memory. In a wait-free asynchronous manner, processes access a predefined sequence of one-shot immediate snapshot objects. This model is equivalent (for wait-free task solvability) to the usual read/write shared memory model, but its runs are more structured and easier to analyze. It has already been instrumental in other works.

The paper studies three known failure detector classes $\{\diamond S_x\}_{1 \leq x \leq n}$, $\{\Omega^z\}_{1 \leq z \leq n}$, and $\{\diamond \psi^y\}_{1 \leq y \leq n}$, via the IIS model (x, y or z are parameters that specify the scope of the corresponding failure detector class). It identifies restrictions of the IIS model that characterize the power of each of these classes. These restrictions are captured as additional properties that the underlying immediate snapshot objects have to satisfy, in essence, obtaining a subset of IIS runs. For each failure detector class C , it is shown that the basic read/write model enriched with C and a restricted IIS model have the same computational power for wait-free solvable tasks. Immediate applications of these results are new lower bound proofs for k -set agreement in asynchronous systems enriched with a failure detector of each one of these classes. The proofs are simple, using novel distributed simulations, and shed light both to the IIS model and to the nature of the failure detectors.

Key-words: Asynchronous system, Atomic read/write registers, Distributed algorithm, Distributed Computability, Failure detectors, Fault-tolerance, Immediate snapshot, Process crash, Round-based computation, Shared memory, Snapshot operation.

(Résumé : *tsvp*)

* Instituto de Matemáticas, Universidad Nacional Autónoma de México, D. F. 04510, Mexico rajsbaum@math.unam.mx

** IRISA, Université de Rennes 1, Campus de Beaulieu, 35042 Rennes Cedex, France, raynal@irisa.fr

*** IRISA, Université de Rennes 1, Campus de Beaulieu, 35042 Rennes Cedex, France, travers@irisa.fr



Les détecteurs de fautes sont des ordonnanceurs

Résumé : Ce rapport étudie les détecteurs de fautes à l'aide du modèle IIS (*Iterated Immediate Snapshot model*). Il montre que les détecteurs de fautes sont en fait des ordonnanceurs.

Mots clés : Système asynchrone, registre atomique, algorithme distribué, calculabilité distribuée, détecteur de fautes, instantané atomique, crash de processus, modèle de calcul fondé sur les rondes, mémoire partagée, algorithme sans attente.

1 Introduction

Consider an asynchronous distributed system of n processes where any number of them can fail by crashing, communicating via standard read/write shared memory (a similar discussion holds for message passing), and solving some *task* in a *wait-free* manner. When solving a task, such as *consensus*, processes start with private input values, and must decide on output values, that satisfy the task's specification. The most famous task is *consensus* [13]. More generally, in the *k-set agreement* task [12], processes must agree on at most k of their input values. However, *k-set agreement* is not solvable in such an asynchronous system even for $k = n - 1$ [8, 22, 33].

Failure detectors Given the previous impossibility results, researchers have considered various ways of enriching the system, in particular, with a *failure detector* [10], that is, a distributed oracle that provides each process with hints on process failures. The oracle controls a local variable at each process p_i , that can be read but not modified by the processes. According to the type and the quality of the hints, several classes of failure detectors can be defined. The difficulty of the problems that can be solved in a distributed system can be measured as a function of the quality of the information on failures provided by the underlying failure detector. For example, a failure detector of the class Ω [11] controls a local variable LEADER_i at each p_i , such that eventually all these variables are equal to the identity of some correct process. The information provided by such a failure detector is sufficient to solve consensus [11], and hence to solve *any* task [19]. More generally, in an Ω^z failure detector [31], eventually the LEADER_i variables of all correct processes are equal to a set L of process identifiers, $|L| \leq z$, that includes the identifier of a correct process. In this case z -set agreement can be solved, but not $(z - 1)$ -set agreement [26].

It is clear from the previous discussion that the addition of a failure detector such as Ω^z is restricting the asynchrony of the system. But in what way? As z goes from n to 1, we go from a purely asynchronous system to a synchronous system. How should we characterize the synchrony that we get for each value of z ? Moreover, other failure detectors also restrict the asynchrony of the system. Consider $\diamond\mathcal{S}_x$ and $\diamond\psi^y$. The failure detector class $\diamond\mathcal{S}_x$ [18, 27, 34] is a simple generalization of the class $\diamond\mathcal{S}$ introduced in [10] ($\diamond\mathcal{S}_n$ is $\diamond\mathcal{S}$). Informally, the variable TRUSTED_i provided by a failure detector of the class $\diamond\mathcal{S}_x$ contains the identities of the processes that are believed to be currently alive. A failure detector of the class $\diamond\psi^y$ [26] outputs at each process p_i an integer NBC_i that estimates the number of processes that have crashed.

It is known that $\diamond\mathcal{S}_x$, Ω^z and $\diamond\psi^y$, have the same power as far as solving the set agreement problem, however, they are not equivalent [26]. For example, while it is possible to build a failure detector of the class Ω^z from a failure detector of the class $\diamond\mathcal{S}_x$ iff $x + z > t + 1$ (where t is an upper bound on the number of processes that may crash), it is impossible to build a failure detector of the class $\diamond\mathcal{S}_x$ from a failure detector of the class Ω^z for $1 < x, z \leq t$. Reductions among the other classes are studied in detail in [26]. Thus, while $\diamond\mathcal{S}_x$, Ω^z and $\diamond\psi^y$ restrict the synchrony of the system, it appears that they don't do it in the same way.

The iterated immediate snapshot model Designing correct distributed algorithms, as well as proving impossibility results, is a difficult task, that is complicated by the wide spectrum of models that arise when considering the synchrony, the type and number of failures, and the communication means available in the system. Thus, as noticed in [7],

“Identifying high level constructs is essential for advancing the art of distributed algorithms”.

In the basic shared memory model there is a shared array with one entry per process, such that each process can read any entry of the array, but can write only to the entry of the array it is associated with. The first fundamental step in the identification of high level constructs that can be built on top of this basic shared memory model has been the introduction of the `snapshot()` operation [1], that allows a process to read the shared array in a single atomic step. Such an operation simplifies the design of algorithms.

A next step has been the definition of the *immediate snapshot* shared memory abstraction [7]. Here the shared array is encapsulated in an object called *immediate snapshot*, that is accessed by the processes with an operation denoted `write_snapshot()`. That operation writes the value provided by the invoking process and returns to it a snapshot of the shared array, as encapsulated by the object. Concurrent `write_snapshot()` invocations write their values concurrently, and return the same snapshot of the array. The immediate

atomic snapshot model has been useful not only to prove fundamental impossibility results such as the ones [5, 8, 33], but also to design simple and elegant wait-free algorithms [7].

The definition of the *IIS model*, an *Iterated* version of the *Immediate Snapshot* shared memory abstraction [9], was the next step. In the IIS model, the processes proceed in consecutive asynchronous rounds, and at each round, a new immediate snapshot object is used. While in the immediate snapshot model, the shared memory consists in a single immediate snapshot object that the processes access repeatedly, in the IIS model there is an infinite number of one shot immediate snapshot objects, accessed by the processes in the same order, in an asynchronous manner. Both have the same computational power: a task is wait-free solvable in the snapshot model if and only if it is wait-free solvable in the IIS model [9].

While the snapshot and the immediate snapshot models provide higher abstraction levels to the programmer, the IIS model imposes a restriction to the programmer, namely, a shared object can be accessed only once by each process. The noteworthy added value provided of the IIS model is the regular and structured view that processes have of runs. More precisely, the structure of the process views in all the k -round runs, is exactly the same as in all $(k + 1)$ -round runs. Topologically speaking, it is easy to see that the latter is a subdivision of the former. This regularity is the essence of an elegant IIS-based characterization of wait-free task solvability [9]. More recently, a generalized version of the IIS model that includes objects other than read/write registers was instrumental in proving another fundamental result: renaming is strictly less powerful than set agreement [16]. Hence, the IIS model allows for an easier exploration of the computational limit of the basic read/write model [3], as well as other more general models.

Contributions This paper shows that the IIS model has yet another fundamental advantage, namely, it allows studying the computability power of the read/write shared memory model equipped with a failure detector. More specifically, the paper presents several results in that direction.

1. Given that we would like to study the read/write model equipped with a failure detector, a natural question is the following. As the read/write shared memory model and the IIS model are equivalent for wait-free task solvability, are they still equivalent when both of them are enriched with a failure detector of the same class C ? We show that (not surprisingly) the answer to this question is “no”, which means that enriching the IIS model with a failure detector does not increase its wait-free computational power.
2. The previous negative answer leads us, instead of augmenting the IIS model with a failure detector, to consider partially synchronous versions of the IIS model. Such an IIS model is defined by restricting the possible interleavings of the `write_snapshot()` operations issued by the processes.

Given a failure detector of a class C , we define a corresponding restricted IIS model. This model is denoted $IRIS(PR_C)$. *IRIS* stands for *Iterated Restricted Immediate Snapshot* model. PR_C denotes a property, derived from the failure detector class C , that is encapsulated in the `write_snapshot()` operation (that is consequently denoted `restricted_write_snapshot()`). The $IRIS(PR_C)$ model is induced by the runs in which the `restricted_write_snapshot()` operations satisfy the corresponding PR_C property. Every run of $IRIS(PR_C)$ is a run of the IIS model enriched with C , but the opposite is not necessarily true.

To illustrate the approach, the paper considers three families of failure detector classes. These are the classes denoted $\{\diamond S_x\}_{1 \leq x \leq n}$, $\{\Omega^z\}_{1 \leq z \leq n}$, and $\{\diamond \psi^y\}_{1 \leq y \leq n}$. For a failure detector C in each one, it defines a corresponding $IRIS(PR_C)$ model.

3. The paper shows that the synchrony exhibited by the $IRIS(PR_C)$ model characterizes the power of the read/write model with C . It presents a simulation from the shared memory model with C to the $IRIS(PR_C)$ model, that generates every one of its possible runs. Conversely, it shows how to extract C from the $IRIS(PR_C)$, and then simulate the read/write model with C (generalizing the simulation of [9]). A noteworthy corollary follows from that simulation, namely, a task is wait-free solvable in the read/write model with C if and only if it is wait-free solvable in the $IRIS(PR_C)$ model.
4. As an application of the previous simulations, are derived new, simple proofs of the impossibility of solving k -set agreement in the read/write model equipped with a failure detector from the above classes. Such direct proofs were known only for the $\{\diamond S_x\}_{1 \leq x \leq n}$ family [20], using combinatorial

topology techniques from [21]. Impossibility proofs for the other families are by reduction to this result [26].

Conversely, the results presented in the paper open the possibility of designing new set agreement (and in particular consensus) algorithms: design an algorithm in an $IRIS(PR_C)$ model, and then using the simulation mentioned above, transform it into an algorithm for the read/write model with C .

We remark that the definition of an $IRIS(PR_C)$ model is not in terms of process failures or failure detectors. The characterization of a failure detector class C appears as a restriction of the set of runs that would be produced if the corresponding failure detector was used in a certain canonical way in the IIS model. So, the $IRIS(PR_C)$ model captures the synchronization/scheduling power of the corresponding failure detector class. In that sense, a failure detector is a scheduler with specific fairness properties¹.

Roadmap The paper is composed of 10 sections. The base computation models (read/write, snapshot, immediate snapshot and iterated immediate snapshot) are presented in Section 2. The eventual failure detector classes we are interested in are introduced in Section 3. Then, the impossibility to solve consensus in the iterated immediate snapshot model equipped with an eventual leader failure detector is proved in Section 4. The additional properties the base immediate snapshot operation of the iterated model has to satisfy when we want to have the same power as a base snapshot model equipped with a failure of a given class C are described in Section 5. This defines the $IRIS(PR_C)$ model. Then, Section 6 shows how to build an immediate snapshot operation suited to the iterated model from base snapshots and a failure detector of a given class C . Section 7 shows how to build a failure detector of the class C from the corresponding $IRIS(PR_C)$ model. Section 8 presents a general simulation from $IRIS(PR_C)$ model to the read/write model enriched with a failure detector of the class C . Section 9 shows that the proposed framework allows designing simple lower bound proofs for set agreement problems (the important point here lies in the fact that these proofs are based on algorithmic reductions only; they do not involve topology notions). Finally, section 10 concludes the paper.

2 Computational models

2.1 Basic shared memory model

We consider a standard asynchronous system made up of n processes, p_1, \dots, p_n , of which at most t can crash, $1 \leq t < n$. A process has *crashed* when it halts prematurely, otherwise it is *correct* (only crash failures are considered). In this paper we concentrate on the *wait-free* case (i.e., $t = n - 1$). We sometimes use f , $1 \leq f \leq t$, to denote the actual number of processes that are faulty in a run. More details about these and the following notions can be found in standard textbooks (e.g., [6, 25]). The integer i , $1 \leq i \leq n$, is called the index of the process p_i .

The processes communicate by reading and writing simple shared memory registers. It is often useful to consider higher level abstractions constructed out of such registers. We describe first the simple registers, and then the other abstractions used in the paper.

Single writer/multi readers atomic registers model The processes communicate by reading and writing a shared memory made up of single-writer/multi-reader (1W*R) atomic registers [19, 24]. This means that the shared memory is structured as an array $SM[1..n]$, such that only p_i can write $SM[i]$, and p_i can read any entry.

Uppercase letters are used to denote shared registers. A process can have local variables. Those are denoted with sub-indexed lowercase letters, e.g., $level_i[1..n]$ is a local array of p_i .

¹This is similar to the *linearizability* consistency criterion [23] that restricts the set of runs generated by processes that access concurrently shared objects.

Atomic snapshot model In this model processes communicate via *snapshot objects*, that can be accessed with a `write()` operation and a `snapshot()` operation [1]. When a process p_i executes a $SM.snapshot()$ operation, it obtains the value of the entire array as if that array was read atomically at some point between its invocation and return events.

A snapshot object can be wait-free implemented from base 1W*R atomic registers. The best wait-free implementation proposed so far requires $O(n \log n)$ invocations of base read and write operations [4]. Consequently, it is possible to assume, without loss of generality, that the higher level abstraction provided by snapshot objects is available in the basic shared-memory model.

Immediate snapshot model A higher level of abstraction is provided by the immediate snapshot model [7, 33]. This model includes *immediate snapshot* objects, accessed by the processes through a single operation, denoted `write_snapshot()` that replaces both the `write()` and the `snapshot()` operations. Intuitively, when a process p_i invokes $SM.write_snapshot(v)$, it is as if it instantaneously executes a $SM[i] \leftarrow v$ operation followed by an $SM.snapshot()$ operation. If several processes execute a `write_snapshot()` operation simultaneously, then their corresponding write operations are executed concurrently, and then their corresponding snapshot operations are executed concurrently (each of the concurrent operations sees the values written by the other concurrent operations).

It is shown in [7] that the immediate snapshot model and the snapshot model are equivalent in the following sense: if a problem can be wait-free solved in one of these models, it can also be wait-free solved in the other model.

An algorithm, due to Borowski and Gafni [7], that implements immediate snapshot objects is described in Figure 1. That algorithm considers a one-shot immediate snapshot object (a process invokes $SM.write_snapshot()$ at most once). It uses two arrays of 1W*R 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 $[n+1, \dots, n+1]$, 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 behavior is described in Figure 1 [7].

```

operation write_snapshot( $v_i$ ):
   $REG[i] \leftarrow v_i$ ;
  repeat  $LEVEL[i] \leftarrow LEVEL[i] - 1$ ;
    for  $j \in \{1, \dots, n\}$  do  $level_i[j] \leftarrow LEVEL[j]$  end for;
     $view_i \leftarrow \{j : level_i[j] \leq LEVEL[i]\}$ ;
  until ( $|view_i| \geq LEVEL[i]$ ) end repeat;
  return ( $\{(j, REG[j]) \text{ such that } j \in view_i\}$ ).

```

Figure 1: Borowsky-Gafni's one-shot `write_snapshot()` algorithm (code for p_i)

This very elegant algorithm satisfies the following properties [7]. The sets $view_i$ of the processes that terminate the algorithm, satisfy the following main property: if $|view_i| = \ell$, then p_i stopped at the level ℓ , and there are ℓ processes whose current level is $\leq \ell$. From this property, follow the self-inclusion, containment and immediacy properties (stated in Section 2.2) that define the one-shot immediate snapshot object.

2.2 Iterated immediate snapshot (IIS) model

One-shot immediate snapshot A *one-shot immediate snapshot* object is an immediate snapshot object that can be accessed only once by each process. In this case, the semantics of the `write_snapshot()` operation is characterized by the three following properties, where v_i is the value written by p_i and sm_i , the value (or *view*) it gets back from the operation, for each p_i invoking the operation. To simplify the statement of the properties, we consider sm_i as a set of pairs (k, v_k) , where v_k corresponds to the value in p_k 's entry of the array. By definition we have $sm_j = \emptyset$, if the process p_j never invokes `write_snapshot()` on the corresponding object.

- Self-inclusion. $\forall i : (i, v_i) \in sm_i$.
- Containment. $\forall i, j : sm_i \subseteq sm_j \vee sm_j \subseteq sm_i$.
- Immediacy. $\forall i, j : (i, v_i) \in sm_j \Rightarrow sm_i \subseteq sm_j$.

The first property states that a process sees the value it has written. The second property states that the *views* (i.e. the contents of the sm_i sets) obtained by the processes can be ordered by containment. The last property states that when a process invokes `write_snapshot()`, the snapshot is scheduled immediately after the write. The `write_snapshot()` issued by the processes are *set-linearizable* [30]. This means that each `write_snapshot()` issued by a process appears as if it has been instantaneously executed at a single point of the time line, without preventing several `write_snapshot()` to appear at the same point of time.

The iterated immediate snapshot model (IIS) In the IIS model [9] the shared memory is made up of an infinite number of one-shot immediate snapshot objects $IS[1], IS[2], \dots$. These objects are accessed sequentially (and asynchronously) by the processes according to the following round-based pattern:

```
 $r_i \leftarrow 0;$   
loop forever  $r_i \leftarrow r_i + 1;$   
                  local computations; compute  $v_i;$   
                   $sm_i \leftarrow IS[r_i].write\_snapshot(v_i);$   
                  local computations  
end loop.
```

Figure 2: Generic algorithm in Iterated Immediate Snapshot model

Let us observe that the IIS model requires that each correct process executes an infinite number of rounds. Moreover, it is possible in that model, that a correct process p_2 be unable to send information to another correct process, p_1 . Consider a run where both execute an infinite number of rounds, but p_1 is scheduled before p_2 in every round. Thus, p_1 never reads a value written to an immediate snapshot object by p_2 . Of course, in the usual (non-iterated read/write shared memory) asynchronous model, two correct processes can always eventually communicate with each other. Thus, it may be surprising that, despite the use of such a strong constraint on the behavior of the processes, it is still possible to derive simulations between the IIS model and the base read/write non-iterated model.

3 Failure detector classes

A *failure detector* [10] is a device that provides the processes with information on process failures. Several classes of failure detectors can be defined according to the kind and the quality of that information. This section presents three known classes of failure detectors $\diamond_{\mathcal{S}_x}$, \diamond_{ψ^y} and Ω^z that, while having the same power as far as solving the set agreement problem, are not equivalent.

A failure detector provides each process p_i with a local variable that p_i can read, but not modify. We will use small capital letters for these variables, to distinguish them from p_i 's locally controlled variables.

3.1 The family $\{\diamond\mathcal{S}_x\}_{1 \leq x \leq n}$

The failure detector class $\diamond\mathcal{S}_x$ is a simple generalization of the class $\diamond\mathcal{S}$ introduced in [10]. In particular, $\diamond\mathcal{S}_n$ is $\diamond\mathcal{S}$. Informally, the variable TRUSTED_i provided by a failure detector of the class $\diamond\mathcal{S}_x$ contains the identities of the processes that are believed to be currently alive. When $j \in \text{TRUSTED}_i$ we say “ p_i trusts p_j .”² By definition, a crashed process trusts all processes. The failure detector class $\diamond\mathcal{S}_x$ is defined by the following properties:

- **Strong completeness.** There is a time after which every faulty process is never trusted by every correct process.
- **Limited scope eventual weak accuracy.** There is a set Q of x processes containing a correct process p_ℓ , and a (finite) time after which each process of Q trusts p_ℓ .

The time τ , the set Q and the process p_ℓ are not explicitly known. Moreover, some processes of Q could have crashed. The parameter x , $1 \leq x \leq n$, defines the scope of the eventual accuracy property. When $x = 1$, the failure detector provides no information on failures, when $x = n$ the failure detector can be used to solve consensus.

We will sometimes use the following equivalent formulation of $\diamond\mathcal{S}_x$ [26], assuming the local variable controlled by the failure detector is REPR_i .

- **Limited eventual common representative.** There is a set Q of x processes containing a correct process p_ℓ , and a (finite) time after which, for any correct process p_i , we have $i \in Q \Rightarrow \text{REPR}_i = \ell$ and $i \notin Q \Rightarrow \text{REPR}_i = i$.

Clearly, a failure detector that satisfies the previous property can be transformed into one of the class $\diamond\mathcal{S}_x$ (define $\text{TRUSTED}_i = \{\text{REPR}_i\}$). Conversely, an algorithm that transforms any failure detector of $\diamond\mathcal{S}_x$ into a failure detector satisfying the limited eventual common representative property is described in [26].

3.2 The family $\{\diamond\psi^y\}_{1 \leq y \leq n}$

A failure detector of the class $\diamond\psi^y$ outputs at each process p_i an integer NBC_i that is an estimate of the number of processes that have crashed. The class $\diamond\psi^y$ is defined by the following property, where f is the number of actual crashes in a run.

- **Eventual accuracy.** There is a time from which $\text{NBC}_i = \max(n - y, f)$ at each correct process p_i .

The family $\{\diamond\psi^y\}_{1 \leq y \leq n}$ was introduced in [26] although with a different formulation³. It is shown in [26] that $\diamond\psi^n$ is equivalent to $\diamond\mathcal{P}$, the class of eventually perfect failure detectors [10] (a failure detector of that class is strictly stronger than Ω), while $\diamond\psi^1$ provides no information on failures.

3.3 The family $\{\Omega^z\}_{1 \leq z \leq n}$

The failure detector class Ω^z [31] is a generalization of the class Ω [11]; in particular, Ω^1 is the class Ω . A failure detector of the class Ω^z controls a local variable LEADER_i containing a set process identities, and satisfies the following property.

- **Eventual multiple leadership.** There is a set L , of size at most z and containing a correct process, and a (finite) time after which the set LEADER_i of every correct process p_i remains forever equal to L .

Let us notice that when $z = n$ a failure detector of the class Ω^z provides no information on failures; when $z = 1$, Ω^z is equivalent to $\diamond\mathcal{S}$ [11], and hence powerful enough to solve consensus. However, as shown in [26], while it is possible to build a failure detector of the class Ω^z from a failure detector of the class $\diamond\mathcal{S}_x$ iff $x + z > t + 1$ (where t is an upper bound on the number of processes that may crash), it is impossible to build a failure detector of the class $\diamond\mathcal{S}_x$ from a failure detector of the class Ω^z for $1 < x, z \leq t$. On another side, while $\diamond\psi^y$ can be transformed into Ω^z iff $y + z > t$, Ω^z cannot be transformed into $\diamond\psi^y$ [26].

²The original definition of the failure detector calls $\diamond\mathcal{S}$ [10] provides each process p_i with a set denoted SUSPECTED_i . Using the set TRUSTED_i is equivalent to using the set SUSPECTED_i . We use TRUSTED_i to emphasize the fact that what is important to ensure progress is the set of processes that are alive.

³The Chandra-Toueg original definition of failure detector required that the local output of a failure detector is a function of the failure pattern, while the failure detectors of $\diamond\psi^y$ as defined in [26] allowed processes to interact with the failure detector providing a parameter o a query.

4 An impossibility result about adding a failure detector to the IIS model

This section considers the question described in the first contribution of the Introduction. Are the base read/write shared memory model and the IIS model equivalent for wait-free solvable tasks, when both are equipped with a failure detector of the same class?

An IIS model with failure detector Assume a failure detector of some class C is available in the IIS model, that provides each process p_i with a local variable F_i . During each round r , p_i can read F_i any number of times, and eventually, access the next immediate snapshot object. Also, assume some task is being solved, so that each process starts with a private input value in a local variable $input_i$, and must eventually put its decision in a write-once variable dec_i .

The framework of the IIS model defined in Section 2.2 is refined as described in Figure 3, with a *full information* algorithm \mathcal{A} solving the task. In line (4), $compute(sm_i^{(r_i-1)}, F_i)$ is a shorthand for a loop that is executed by p_i , where in each iteration it can use the current value of the failure detector obtained from F_i , to make local computations and decide weather to execute one more iteration or to exit the loop returning a value to be placed in the variable fd_i . It is assumed that the number of iterations executed is always finite. Thus, when line (5) is executed, p_i invokes $IS[r].write_snapshot()$ to write its view $sm_i^{(r-1)}$ (obtained from the previous $write_snapshot()$ invocation) together with its latest failure detector information (or any additional desired information), and the current decision dec_i . After it has obtained a view $sm_i^{(r)}$ during the round r (line 5), a process p_i checks if it can decide by applying a decision function, denoted $g()$, to that view $sm_i^{(r)}$.

Recall that correct processes keep on taking steps forever, even after having decided. We say that a *task* T is *solvable in the IIS model with* C if there is an algorithm \mathcal{A} of the form in Figure 3, such that for any failure detector of the class C , in any (infinite) run where the input values are in the domain of T , every correct process eventually decides, and the decisions satisfy the input/output relation of T .

```

(1) init  $r_i \leftarrow 0$ ;  $dec_i \leftarrow \perp$ ;  $sm_i^{(0)} \leftarrow \{ \langle input_i, \emptyset, dec_i \rangle \}$ 
(2) loop forever
(3)    $r_i \leftarrow r_i + 1$ ;
(4)    $fd_i \leftarrow compute(sm_i^{(r_i-1)}, F_i)$ ;
(5)    $sm_i^{(r_i)} \leftarrow IS[r_i].write\_snapshot(\langle sm_i^{(r_i-1)}, fd_i, dec_i \rangle)$ ;
(6)   if ( $dec_i = \perp$ ) then  $dec_i \leftarrow g(sm_i^{(r_i)})$  end if
(7) end loop.
```

Figure 3: Full information “IIS + Failure detector” code for p_i

The impossibility The idea of the proof is to show that C does not restrict the set of possible interleavings of the IIS model. Thus, if T is solvable in the IIS model with C , in particular it is solvable in the set of runs of the IIS model, and hence solvable in the read/write shared memory model, by the simulation of [9]. The crucial step is to group together all operations of a round related to the failure detector, in a fixed predetermined order, before executing shared memory operations of that round. And only then, considering all interleavings of the shared memory operations.

Theorem 1 *For any failure detector class C and task T , if T is solvable in the IIS model with C then T is wait-free solvable in the base read/write shared memory model with no failure detector.*

Proof To facilitate the proof, we consider a single input configuration of T (e.g., [9]), and hence a single input initial configuration of the system, denoted S^0 .

We consider the following subset of runs of the IIS model with C , where no process fails, defined inductively, starting with S^0 . Consider some reachable configuration of the system after $r - 1$ rounds, say S^{r-1}

(for the basis, we take S^0). We schedule the steps of the processes following the same round structure of the algorithm \mathcal{A} , by having all processes execute their round r before proceeding to round $r + 1$. Moreover, we schedule first all local computations of the processes corresponding to line (4) of round r , before any process starts executing its line (5). We schedule all those local operations in a fixed order, first all those of p_1 , then all those of p_2 , until all those of p_n , and we get a specific partial run

$$\text{compute}(sm_1^{(r-1)}, F_1), \text{compute}(sm_2^{(r-1)}, F_2), \dots, \text{compute}(sm_n^{(r-1)}, F_n).$$

Let us denote the system configuration at the end of this partial run by S_1^{r-1} . Now, we consider all possible interleavings of executions of line (5) for all processes. After such an interleaving, we execute (in an arbitrary order) line (6) of every process, and end up in a configuration denoted S^r (abusing notation, as for each such interleaving the system ends up in a different configuration).

Let us observe that any failure detector output change at a process p_i after p_i returned from its invocation $\text{compute}(sm_i^{(r-1)}, F_i)$ does not affect the execution of its operation of line (5), because the value to be written by the $\text{write_snapshot}()$ invocation is fixed. That is, the views obtained as a result of such invocations, in the $sm_i^{(r)}$ variables, on all possible interleavings, are equivalent to the views of the IIS model with no failure detector. In other words, given two set-linearizations of the $\text{write_snapshot}()$ operation, the view of a process p_i is the same in both, iff in the IIS model with no failure detector, p_i has the same view in both set-linearizations (or, in topological terms, the complex of views in both models are isomorphic).

We have constructed a subset of runs of IIS with C where the views of the processes at the end of each round have the same structure as the views of the original IIS model with no failure detector. As we are assuming algorithm \mathcal{A} solves T in the IIS with C , it solves T also in the IIS model, and we can use the simulation in [9] to solve T in the basic shared-memory model. $\square_{\text{Theorem 1}}$

It follows from this theorem that the read/write model with C and the IIS model with C are equivalent only if C does not provide any additional information on failures.

Remark. The previous theorem means that à la Paxos agreement algorithms designed for the shared memory model e.g. [14, 17, 32] cannot be expressed in the iterated immediate snapshot model enriched with an eventual leader failure detector. A consensus algorithm suited to a restricted version of the IIS model is described in Appendix 5.4.

5 The IRIS model

This section shows how to define an *Iterated Restricted Immediate Snapshot* model, $IRIS(PR_C)$, for each of the failure detector classes $\{\diamond S_x\}_{1 \leq x \leq n}$, $\{\diamond \psi^y\}_{1 \leq y \leq n}$, and $\{\Omega^z\}_{1 \leq z \leq n}$. The $IRIS(PR_C)$ model is induced by the runs that satisfy a corresponding PR_C property. We call $\text{restricted_w_snapshot}()$ the operation that satisfies the self-inclusion, containment and immediacy properties (as defined in Section 2.2), plus the additional property PR_C (although this is really a property on runs, not on individual immediate snapshot objects).

Preliminaries The additional properties PR_{Ω^z} , $PR_{\diamond S_x}$ and $PR_{\diamond \psi^y}$ all assume that, whatever the round r , the index i of a process p_i that invokes $IS[r].\text{restricted_w_snapshot}()$, always appears in the value that is written. So, if due to the specific task that is solved in the corresponding enriched $IRIS$ model other values are also written, they are ignored in the following definitions as they are irrelevant for stating the properties.

Let sm_j^r be the value obtained by the process p_j when it returns from the $IS[r].\text{restricted_w_snapshot}()$ invocation (this value is called a *view*). If at least one process executes the round r , we have $\{sm_j^r : sm_j^r \neq \emptyset\} \neq \emptyset$ (due to the self-inclusion property). The additional properties are on the sets $sm_i^{r'}$ obtained by the processes from some round $r \leq r'$. As each process p_i is assumed to execute rounds forever, $sm_i^{r'} = \emptyset$ means that p_i never executes the round r' (it is consequently faulty).

5.1 $IRIS(PR_{\diamond S_x})$

The property $PR_{\diamond S_x}$ is defined as follows (where Q is a set of process indexes).

- $PR_{\diamond S_x} \equiv \exists Q, \ell: |Q| = x \wedge \ell \in Q: \exists r: \forall r' \geq r: (sm_{\ell}^{r'} \neq \emptyset) \wedge (i \in Q \setminus \{\ell\} \Rightarrow (sm_i^{r'} = \emptyset \vee sm_i^{r'} \subsetneq sm_i^{r'}))$.

This property states that there are a set Q of x processes containing a correct process p_{ℓ} , and a round r , such that at any round $r' \geq r$, each process $p_i \in Q \setminus \{\ell\}$ either has crashed ($sm_i^{r'} = \emptyset$) or obtains a view $sm_i^{r'}$ that contains strictly $sm_{\ell}^{r'}$.

From the set-linearization point of view, this means that, from some round r and for any round $r' \geq r$, the invocation $IS[r'].restricted_w_snapshot(\ell)$ is set-linearized before the invocations on the same object $IS[r']$ issued by the other processes of Q .

As a simple case, let us consider the strongest case $x = n$. $PR_{\diamond S_n}$ states that in every run, there exists a process p_{ℓ} and a round r , such that every correct process sees p_{ℓ} in every round $r' \geq r$ (so p_{ℓ} must be correct).

5.2 $IRIS(PR_{\diamond \psi^y})$

Let us recall that f ($0 \leq f \leq n - 1$) denotes the actual number of processes that crash in a given run. The property $PR_{\diamond \psi^y}$ is defined as follows.

- $PR_{\diamond \psi^y} \equiv \exists r: \forall r' \geq r: ((i - 1 = (r' - 1) \bmod n) \wedge (sm_i^{r'} \neq \emptyset)) \Rightarrow |sm_i^{r'}| \geq n - \max(n - y, f)$.

The intuition that underlies this property is the following: there is a logical time (round number) after which each correct process obtains infinitely often a view that misses at most $\max(n - y, f)$ processes. As we can see, when $f < n - y$ such views can miss correct processes. As a particularly simple case, let us consider the instance $y = n$ (as already noticed, $PR_{\diamond \psi^n}$ equivalent to $PR_{\diamond \mathcal{P}}$): $PR_{\diamond \psi^n}$ states that after some round there is an infinite number of rounds at which p_i obtains a view containing the $(n - f)$ correct processes.

5.3 $IRIS(PR_{\Omega^z})$

The property PR_{Ω^z} is defined as follows (where L is a set of process indexes).

- $PR_{\Omega^z} \equiv \exists L: |L| \leq z$ and $\exists r: \forall r' \geq r: \exists i \in L: (sm_i^{r'} \neq \emptyset) \wedge (sm_i^{r'} \subseteq L)$.

This property states that there are a round r and a set L (including at least one and at most z correct processes) such that, at any round $r' \geq r$, any process that executes $IS[r'].restricted_w_snapshot()$ sees a non-empty subset of L in its view, and there are processes of L that see only processes of L in their view. Let us notice that nothing prevent $IS[r'].restricted_w_snapshot()$ invocations to be concurrent or not.

Let us consider the case $z = 1$, i.e., the simplest instance of PR_{Ω^z} . We have $PR_{\Omega^1} \equiv \exists \ell: \exists r: \forall r' \geq r: sm_{\ell}^{r'} = \{\ell\}$, which means that there is a round r and a process p_{ℓ} such that, at each round $r' \geq r$, any process p_i that executes $sm_i \leftarrow IS[r'].restricted_w_snapshot()$ sees ℓ in its view sm_i (i.e., $\ell \in sm_i$). Said differently, whatever the concurrency degree among the $IS[r'].restricted_w_snapshot()$ invocations issued by the processes during r' , the invocation issued by p_{ℓ} is always set-linearized alone and before the other invocations. So, p_{ℓ} always obtains the same view that contains only itself ($\forall r' \geq r: sm_{\ell}^{r'} = sm_{\ell}^r = \{\ell\}$). It then follows from the containment property of the immediate snapshot operation, that the view $sm_j^{r'}$ of any process p_j that executes a round $r' \geq r$ is such that $\ell \in sm_j^{r'}$. The instance $z = 1$ ensures that the invocation $IS[r'].restricted_w_snapshot()$ by p_{ℓ} is set-linearized alone and before the others. The instances $z > 1$ are weaker in the sense that they allow several $IS[r'].restricted_w_snapshot()$ invocations issued by the processes of a subset of L to be set-linearized together and before the invocations issued by the other processes. Moreover this subset of L can differ from one round to another. (This property is close to, but different from, the notion of z -bounded concurrency [15]).

5.4 An example: solving consensus in $IRIS(PR_{\Omega'})$

In order to get a better insight of an $IRIS(PR_C)$ model, this section presents an algorithm that solves the consensus problem in $IRIS(PR_{\Omega'})$. This algorithm is described in Figure 4⁴. A more general (and more involved) z -set agreement algorithm suited to the $IRIS(PR_{\Omega^z})$ model is described in Appendix A.

The value proposed by p_i is v_i ; \perp is a default value that cannot be proposed by a process. In addition to r_i (the current round number), a process p_i manages four local variables:

- The local variables est_i and dec_i are directly related to the decision value: est_i (initialized to v_i) contains p_i 's current estimate of the decision value, while dec_i is a write-once local variable (initialized to \perp) that eventually contains the single valued decided by p_i .
- sm_i and tm_i are two local variables used to contain the snapshot value returned by the invocations to the operation `restricted_w_snapshot()` at the odd and even round numbers, respectively. The variable sm_i contains a set of triples, while tm_i contains a set of set of triples (i.e., a set of tm_j values).

A process p_i executes a sequence of pairs of rounds, namely, (1, 2), then (3, 4), etc. During the first round ($r - 1$), p_i writes the triple $\langle i, est_i, dec_i \rangle$ in the one-shot immediate snapshot object $IS[r - 1]$, from which it obtains a set of such triples (lines 1-2). During the second round, p_i writes into $IS[r]$, the set of triples sm_i it has just obtained, and obtains a corresponding set of set of triples tm_i (lines 3-4).

Then, p_i considers the values it has obtained from the one-shot immediate snapshot objects $IS[r - 1]$ and $IS[r]$. If p_i sees that a value (dec) has already been decided (line 5) while it has not yet decided, it decides that value (line 6). Otherwise, if the set of set of triples (tm_i) it has obtained from $IS[r]$ contains a set sm with a single triple (line 7), p_i adopts the estimate value of that triple sm (line 8) as its new estimate. Moreover, if additionally, tm_i contains a single triple and that triple is from p_i itself ($tm_i = \{\langle i, est_i, - \rangle\}$, line 9), then p_i decides its current estimate. Let us observe that $tm_i = \{\langle i, est_i, - \rangle\}$ means that p_i was the only “winner” of both the rounds $r - 1$ and r (the invocations $IS[r - 1].restricted_w_snapshot()$ and $IS[r].restricted_w_snapshot()$ issued by p_i have been set-linearized alone and before the invocations from the other processes).

```

init  $r_i \leftarrow 0$ ;  $est_i \leftarrow v_i$ ;  $dec_i \leftarrow \perp$ 

loop forever
(1)  $r_i \leftarrow r_i + 1$ ;
(2)  $sm_i \leftarrow IS[r_i].restricted\_w\_snapshot(\langle i, est_i, dec_i \rangle)$ ;
(3)  $r_i \leftarrow r_i + 1$ ;
(4)  $tm_i \leftarrow IS[r_i].restricted\_w\_snapshot(sm_i)$ ;
(5) if  $(\exists sm : (sm \in tm_i) \wedge (\langle -, -, dec \rangle \in sm \text{ with } dec \neq \perp))$ 
(6)     then if  $dec_i = \perp$  then  $est_i \leftarrow dec$ ;  $dec_i \leftarrow dec$  end if
(7) elsif  $(\exists sm : (sm \in tm_i) \wedge (sm = \{\langle -, est, - \rangle\}))$ 
(8)     then  $est_i \leftarrow est$ ;
(9)         if  $tm_i = \{sm\} \wedge sm = \{\langle i, est, - \rangle\}$  then  $dec_i \leftarrow est$  end if
(10) end if
end loop.
```

Figure 4: A consensus algorithm for the $IRIS(PR_{\Omega})$ model (code for p_i)

Theorem 2 *The algorithm described in Figure 4 wait-free solves the consensus problem in the $IRIS(PR_{\Omega})$ model.*

Proof We have to show that, whatever the number of processes that crash, a decided value is a proposed value (validity), no two different values are decided (agreement), and each correct process decides (wait-free termination). The proof of the validity property follows directly from the code of the algorithm. So, we

⁴This algorithm can be seen as an “instance” of the transformation described in Section 8 customized for the consensus problem.

concentrate on the proof of the agreement and termination properties.

Agreement⁵. If no process decides, the agreement property is trivially satisfied. So, let r be the first round during which a process decides (this occurs at line 9). Let p_i be a process that decides. It follows that the test $tm_i = \{sm\} \wedge sm = \{< i, est, - >\}$ is true at line 9, just before p_i decides. This means that p_i “won” alone the rounds $r - 1$ (because $sm = \{< i, est, - >\}$) and r (because $tm_i = \{sm\}$). (Here “win” means that p_i was the first to access $IS[r - 1]$ and $IS[r]$, and these accesses were not concurrent with other accesses to these objects.) It follows that p_i is the only process that decides during the round r .

As p_i executed $IS[r].restricted_w_snapshot()$ without concurrency, it follows from the containment property that, for any process p_j that executes $IS[r].restricted_w_snapshot()$, we have $tm_i \subseteq tm_j$ at line 4 of the round r . As, at round r , we have $tm_i = \{\{< i, est_i, - >\}\}$, it follows that the predicate of line 7 is satisfied when p_j executes that line (namely, $sm_i \in tm_j \wedge sm_i = \{< i, est_i, - >\}$). The process p_j consequently executes line 8, and we then have $est_j = est_i$. It follows that all the processes p_j that execute the round r are such that $est_j = est_i$ when they terminate that round. Hence, from round r , no value different from est_i is present in an estimate value of a process, which completes the proof of the agreement property.

Wait-free termination. As soon as a correct process executes $dec_i \leftarrow est_i$ at line 9 during a round r , all the correct processes decide at line 6 of the round $r + 2$. So, let us assume (by contradiction) that no correct process ever executes $dec_i \leftarrow est_i$ at line 9. Due to the property $PR_{\Omega^!}$, there are a process p_ℓ , and a round r , such that for any round $r' \geq r$, we have $sm_\ell = \{< \ell, est_\ell, - >\}$ at round r' and $tm_\ell = \{sm_\ell\} = \{\{< \ell, est_\ell, - >\}\}$ at the round $r' + 1$. It follows from the text of line 9 that p_ℓ decides at the first round $r' + 1$ that occurs after r , from which the wait-free termination property follows. $\square_{Theorem 2}$

5.5 The case of synchronization operations

When one has to solve the consensus problem, instead of considering a system enriched with a failure detector, one can instead consider a system with synchronization operations more sophisticated than read/write operations [19].

In a way similar to what has been previously done with failure detectors, the iterated immediate snapshot model can allow characterizing the runs of the read/write shared memory model enriched with some synchronization operations. As an illustration, this section considers the `test&set()` operation (the consensus number of which is 2). Let us recall that this operation provides a process with the value 1 (this process is the winner) and the other processes with the value 0 (they are losers). The property associated with *Test&Set* is the following:

- $PR_{Test\&Set} \equiv (\forall r : \exists i : sm_i^r = \{i\})$.

This property states that, independently of its actual concurrency degree, each round has a single winner process.

It is interesting to compare $PR_{Test\&Set}$ and $PR_{\Omega^!}$. While the former states that each round has a winner, the latter property states that (from some round), the same process is always the winner. If we consider the “perpetual” version of Ω (a correct process is the common leader from the very beginning), it is easy to see that it is a much stronger property than $PR_{Test\&Set}$.

6 From the read/write model with C to $IRIS(PR_C)$

This section presents simulations of the $IRIS(PR_C)$ model from the snapshot model equipped with a failure detector of the class C when C is $\diamond S_x$, $\diamond \psi^y$ and Ω^z . Three simulations are presented that build the operation $IS[r].restricted_w_snapshot(v_i)$, where $IS[1..+\infty)$ defines sequence of immediate snapshot objects used in the IRIS model we want to build. In addition to this shared array, each construction uses appropriate additional shared registers and local variables.

⁵As a faulty process is not allowed to decide a value different from a correct process, that property is sometimes called *uniform agreement*.

Each of the constructions described in Figure 5, 6, and 7, associates with each $IS[r]$ of the iterated model an underlying immediate snapshot object $R[r]$, provided with two operations. The first is $R[r].write_snapshot()$ that returns views that satisfy the self-inclusion, containment and immediacy properties. The other one is $R[r].snapshot()$ that satisfies the containment property. These operations can easily be constructed from base read/write operations as indicated in Section 2.1 [7]. Let us recall that, given any object $R[r]$, the $R[r].snapshot()$ operations are totally ordered, and the $R[r].write_snapshot()$ operations are consistently set-linearized with respect to the $R[r].snapshot()$ operations.

In each construction, the last shared memory operation issued by a process is an $R[r].write_snapshot()$ operation. It consequently follows that the constructed $IS[r].restricted_w_snapshot(v_i)$ automatically benefits from the self-inclusion, containment and immediacy properties. This means that only the property PR_C has to be proved for each construction.

6.1 Building $IRIS(PR_{\diamond S_x})$ in the read/write model equipped with $\diamond S_x$

An algorithm that simulates the $IRIS(PR_{\diamond S_x})$ model from one-shot immediate snapshot objects is described in Figure 5. This construction considers the definition of $\diamond S_x$ based on the representative variable $REPR_i$. Each process p_i manages two local variables, denoted m_i and rp_i ; $R[r]$ is the one-shot immediate snapshot object associated with the round r .

```

operation  $IS[r].restricted\_w\_snapshot(< i, v_i >)$ :
(1)  repeat  $m_i \leftarrow R[r].snapshot(); rp_i \leftarrow REPR_i$ 
(2)    until  $((< rp_i, - > \in m_i) \vee rp_i = i)$  end repeat;
(3)   $sm_i \leftarrow R[r].write\_snapshot(< i, v_i >)$ ;
(4)  return  $(sm_i)$ .

```

Figure 5: From the R/W shared memory model with $\diamond S_x$ to $IRIS(PR_{\diamond S_x})$ (code for p_i)

When it invokes $IS[r].snapshot(< i, v_i >)$, a process p_i repeatedly (1) issues a snapshot operation on $R[r]$ in order to know which processes have already written $R[r]$, and (2) reads the value locally output by the underlying failure detector ($REPR_i$), until it discovers that it is its own representative ($rp_i = i$) or its representative has already written $R[r]$ ($< rp_i, - > \in m_i \neq \perp$). When this occurs, p_i invokes $R[r].write_snapshot(< i, v_i >)$ to write the one-shot immediate snapshot object $R[r]$. It finally returns the snapshot value obtained by that $write_snapshot()$ invocation.

Theorem 3 *Assuming a base model with atomic $1W^*R$ atomic registers, equipped with a failure detector of the class $\diamond S_x$, the algorithm described in Figure 5 is a simulation of the $IRIS(PR_{\diamond S_x})$ computation model.*

Proof Due to the properties defining $\diamond S_x$, there is a set Q of x processes including a correct process p_ℓ , such that, after some arbitrary but finite time τ , we have for any correct process p_i : $i \in Q \Rightarrow REPR_i = \ell$ and $i \notin Q \Rightarrow REPR_i = i$. Let us take the set Q and the process p_ℓ that appears in the statement of the property $PR_{\diamond S_x}$ as the set and the process that are denoted the same way in the definition of $\diamond S_x$. As p_ℓ is correct, and there is a time after which the local predicate $REPR_\ell = \ell$ remains true forever, it follows that p_ℓ executes rounds forever (line 2). Moreover, due to the self-inclusion property of the immediate snapshot object we have $sm_i^r \neq \emptyset$ at any round $r \geq 1$.

Let us now show that any correct process executes rounds forever. Let τ be a time such that all the faulty processes have crashed before τ , and for any correct process p_i we have $i \in Q \Rightarrow REPR_i = \ell$ and $i \notin Q \Rightarrow REPR_i = i$ (due to $\diamond S_x$, τ does exist).

- $i \notin Q \vee i = \ell$. It follows from the exit predicate of the **repeat** loop that, after τ , no process p_i blocks forever within the **repeat** loop. It follows that p_i executes round forever.
- $i \in Q \setminus \{\ell\}$. It follows from the exit predicate of the **repeat** loop that among the processes of Q , p_ℓ is the first that exits from the loop, and the other processes of Q do exit from that loop after p_ℓ has executed $R[r].write_snapshot(< \ell, v_\ell >)$ at line 3. Consequently, no correct process blocks forever in a round. (Said another way, when we consider the processes of Q , p_ℓ is set-linearized before the other processes of Q .)

Let r be a round that occurs after τ , $r' \geq r$ and p_i be a correct process in $Q \setminus \{\ell\}$. The fact that $sm_i^{r'} \subseteq sm_i^{r'}$ follows directly from the previous observation that states that the invocation $R[r].write_snapshot(\langle \ell, v_\ell \rangle)$ is set-linearized before the invocations $R[r].write_snapshot()$ issued by the other processes of Q .

□*Theorem 3*

6.2 Building $IRIS(PR_{\diamond\psi^y})$ in the read/write model equipped with $\diamond\psi^y$

The construction (that has some ℓ -mutual exclusion flavor [2]) uses a deterministic function $order(r)$, where the parameter r is a round number. This function orders the process indexes as follows: $order(r)$ returns a sequence of the indexes $1, \dots, n$ in which the last element is the index i such that $(i - 1) = (r - 1) \bmod n$.

operation $IS[r].restricted_w_snapshot(\langle i, v_i \rangle)$:

- (1) $sequence_i \leftarrow order(r_i)$;
- (2) $pred_i \leftarrow \{j : j \text{ appears before } i \text{ in } sequence_i\}$;
- (3) **repeat** $m_i \leftarrow R[r_i].snapshot()$;
- (4) $seen_i \leftarrow m_i \cap pred_i$;
- (5) $nbc_i \leftarrow NBC_i$;
- (6) **until** $(|pred_i| - nbc_i \leq |seen_i|)$ **end repeat**;
- (7) $sm_i \leftarrow R[r].write_snapshot(\langle i, v_i \rangle)$;
- (8) **return** (sm_i) .

Figure 6: From the base shared memory model with $\diamond\psi^y$ to $IRIS(PR_{\diamond\psi^y})$ (code for p_i)

The simulation is described in Figure 6. It uses the same array R of one-shot immediate snapshot objects as the previous simulations. When it invokes $IS[r].restricted_w_snapshot()$, a process p_i first computes the sequence ($sequence_i$) of process indexes associated with the round r (line 1), and determines the set of processes ($pred_i$) that are ordered before it in that sequence (line 2). Then, p_i enters a loop during which it determines the set ($seen_i$) including the processes that (1) have already written into $R[r]$ (those are the processes of m_i) and (2) precede it in $sequence_i$ (those are the processes of $pred_i$). It also reads the value (nbc_i) currently provided by the underlying failure detector (line 5), that is an approximation of the number of crashed processes. This set of statements is repeated by p_i until the processes of $pred_i$ that it perceives as not crashed have written in $R[r]$ (line 6); p_i locally estimates there are at least $(|pred_i| - nbc_i)$ such processes. As in the previous simulations, when this predicate becomes true, p_i writes $R[r]$ (line 7) and returns the associated snapshot value it has just obtained (line 8).

The proof considers the more general t -resilient case (let us recall that t denotes an upper bound on the number of faulty processes in any run, and wait-free means $(n - 1)$ -resilient). The definitions of the eventual accuracy of the class $\diamond\psi^y$ and the property $IRIS(PR_{\diamond\psi^y})$ become:

- **Eventual accuracy.** There is a time from which $NBC_i = \max(t + 1 - y, f)$ at each correct process p_i .
- $PR_{\diamond\psi^y} \equiv \exists r: \forall r' \geq r: ((i - 1 = (r' - 1) \bmod n) \wedge (sm_i^{r'} \neq \emptyset)) \Rightarrow |sm_i^{r'}| \geq n - \max(t + 1 - y, f)$.

Lemma 1 *For any round $r \geq 1$, if a correct process p_i invokes $IS[r].restricted_w_snapshot(\langle i, - \rangle)$, it returns from that invocation.*

Proof The proof is by contradiction. Assuming that there are rounds at which correct processes block forever in the repeat loop (lines 3- 6), let r be the smallest round number at which this happens, and B the set of correct processes that remain blocked forever at that round. This means that, for any $p_i \in B$, the predicate $(|pred_i| - nbc_i \leq |seen_i|)$ is never satisfied when these processes invoke $IS[r].restricted_w_snapshot(\langle i, - \rangle)$ (line 5). Let $sequence[r]$ be the sequence returned by $order(r)$ (let us recall that as $order(r)$ is deterministic, there is a single $sequence[r]$). Let p_s be the process of B whose index s has the smallest rank in $sequence[r]$. We show that p_s returns from its $IS[r].restricted_w_snapshot(\langle i, - \rangle)$ invocation, which contradicts the initial assumption and consequently proves the lemma.

Let us consider $pred_s$ (i.e., the set of process indexes that are before s in $sequence[r]$). We consider two cases:

- $|pred_s| \leq \max(t + 1 - y, f)$. It follows from the eventual accuracy property of $\diamond\psi^y$, that there is a time τ after which we always have $nb_{c_s} = \max(t + 1 - y, f)$. Consequently, after time τ , we have $|pred_s| - nb_{c_s} \leq 0$. As $|seen_s| = |m_s \cap pred_s| \geq 0$, it follows that, after τ , the predicate of line 6 is always true. Hence, the invocation $IS[r].restricted_w_snapshot(< s, - >)$ by process p_s returns.
- $|pred_s| > \max(t + 1 - y, f)$. Let $faulty(S)$ be the set of faulty processes in the set S . We have $|faulty(pred_s)| \leq |faulty(\{1, \dots, n\})| = f \leq \max(t + 1 - y, f)$. Let α be the number of correct processes in $pred_s$. We have $\alpha = |pred_s| - |faulty(pred_s)| \geq |pred_s| - \max(t + 1 - y, f)$. Let us recall that these α processes have a rank smaller than s in $sequence[r]$. Moreover, it follows from the definition of p_s that all the correct processes whose index is smaller than s in $sequence[r]$ return from their $IS[r].restricted_w_snapshot()$ invocation. There is consequently a finite time τ_0 after which each correct processes p_i whose index belongs to $pred_s$ has returned from its $R[r].write_snapshot(< i, - >)$ invocation (line 7). Hence, there is a finite time $\tau_1 \geq \tau_0$ after which we permanently have $|seen_s| = |m_s \cap pred_s| \geq \alpha$. On another side, due to the eventual accuracy property of $\diamond\psi^y$, there is time τ_2 after which the predicate $nb_{c_s} = \max(t + 1 - y, f)$ remains permanently true. So, after time $\tau = \max(\tau_1, \tau_2)$, we always have $|seen_s| = |m_s \cap pred_s| \geq \alpha$ and $\alpha \geq |pred_s| - nb_{c_s}$, from which we conclude that the **repeat** loop eventually terminates. Consequently, the invocation $IS[r].restricted_w_snapshot(< s, - >)$ terminates.

□_{Lemma 1}

Theorem 4 *Assuming a base model with atomic $1W^*R$ atomic registers, equipped with a failure detector of the class $\diamond\psi^y$, the algorithm described in Figure 6 is a simulation of the $IRIS(PR_{\diamond\psi^y})$ computation model.*

Proof It follows from Lemma 1 that each correct process executes an infinite number of rounds (a requirement of any $IRIS(PR_C)$ model). So, it remains to show that the property $PR(\diamond\psi^y)$ is satisfied.

Let r be a round such that, $\forall r' \geq r$, we have $nb_{c_i} = \max(t + 1 - y, f)$ for each correct process p_i that invokes $IS[r'].restricted_w_snapshot(< i, - >)$. Due to Lemma 1 and the eventual accuracy property of $\diamond\psi^y$, such a round does exist.

Let $r(i) \geq r$ be any round number such that the index i satisfies $(i - 1) = ((r(i) - 1) \bmod n)$, and p_i obtains the snapshot value sm_i from its $R[r(i)].write_snapshot(< i, - >)$ invocation. We show that $|sm_i| \geq n - \max(t + 1 - y, f)$.

1. Due to the definition of $order(r(i))$, the rank of i in $sequence[r(i)]$ is n . This means that $|pred_i| = n - 1$ during that round. As p_i returns from $R[r(i)].write_snapshot(< i, - >)$ (Lemma 1), we conclude that the predicate $(|pred_i| - nb_{c_i} \leq |seen_i|)$ was true at line 7, from which we have $|pred_i| - nb_{c_i} = (n - 1) - \max(t + 1 - y, f) \leq |seen_i|$.
2. Let m_i be the last value obtained by p_i at line 3. It follows from (1) the $R[r(i)]$ immediate snapshot object (2) the fact that m_i is computed before sm_i , that sm_i contains the write of p_i in $R[r(i)]$ while m_i does not contains it, and consequently $m_i \subsetneq sm_i$, i.e., $|m_i| < |sm_i|$. Moreover, we have $|seen_i| = |m_i \cap pred_i| \leq |m_i| < |sm_i|$.

It follows from the previous items that $|pred_i| - nb_{c_i} = (n - 1) - \max(t + 1 - y, f) \leq |seen_i| \leq |m_i| < |sm_i|$, from which we have $n - \max(t + 1 - y, f) \leq |sm_i|$, which completes the proof. □_{Theorem 4}

6.3 Building $IRIS(PR_{\Omega^z})$ in the read/write model equipped with Ω^z

The algorithm This construction is described in Figure 7. As previously, a one-shot immediate snapshot object $R[r]$ is associated with each round r . When p_i invokes $IS[r].restricted_w_snapshot(v_i)$ it waits until either a process has written in the one-shot immediate snapshot object $R[r]$, or its index belongs to the output $LEADER_i$ managed by its local failure detector. When one of these conditions becomes true, p_i writes $R[r]$ by invoking $R[r].write_snapshot(< i, v_i >)$. This invocation returns a snapshot value of $R[r]$ that p_i returns as the result of its the invocation $IS[r].restricted_w_snapshot(v_i)$.

Irisa

operation $IS[r].restricted_w_snapshot(< i, v_i >)$: (1) repeat $m_i \leftarrow R[r].snapshot(); ld_i \leftarrow LEADER_i$ (2) until $((m_i \neq \emptyset) \vee (i \in ld_i))$ end repeat ; (3) $sm_i \leftarrow R[r].write_snapshot(< i, v_i >)$; (4) return (sm_i) .

Figure 7: From the R/W shared memory model with Ω^z to $IRIS(PR_{\Omega^z})$ (code for p_i)

Theorem 5 *Assuming a base model with atomic 1W*R atomic registers, equipped with a failure detector of the class Ω^z , the algorithm described in Figure 7 is a simulation of the $IRIS(PR_{\Omega^z})$ computation model.*

Proof Let us first observe that snapshot and immediate snapshot operations on any object $R[r]$ can be wait-free implemented from atomic 1W*R atomic registers [1, 7]. The proof is made up of two parts: (1) any correct process executes an infinite number of rounds; and (2) the property PR_{Ω^z} is satisfied.

To prove that any correct process p_i executes an infinite number of rounds, we have to show that the local predicate $(m_i \neq \emptyset) \vee (i \in ld_i)$ evaluated by p_i at line 2 is eventually true at each round $r \geq 1$.

Let us proceed by contradiction. Let r be the first round at which a correct process p_i remains blocked forever, i.e., $(m_i \neq \emptyset) \vee (i \in ld_i)$ remains forever false once p_i has entered the round r . This means that, after some time, i never belongs to $LEADER_i$ when p_i reads this set, and m_i remains always empty. As m_i remains empty, we conclude that no process p_j ever returns from its $R[r].write_snapshot(< j, v_j >)$ invocation (Observation O1).

On another side, due to the eventual multiple leadership property of Ω^z , there is a set L of size at most z containing at least one correct process p_ℓ such that, after some arbitrary (but finite) time τ , the predicate $LEADER_\ell = L$ is true forever at p_ℓ . It follows that while p_ℓ is blocked at round r , the local predicate $\ell \in ld_\ell$ becomes eventually true. Consequently, p_ℓ executes $R[r].write_snapshot(< \ell, v_\ell >)$ and proceeds to the round $r + 1$ (Observation O2). The observations O1 and O2 contradict each other, from which we conclude that any correct process executes an infinite number of rounds.

Let us now show that the property PR_{Ω^z} is satisfied. Due to the property defining Ω^z , there are a set L containing at least one correct process (and at most z processes) and a time τ such that, after τ , we always have $LEADER_i = L$ at any correct process p_i . Due to the very existence of τ , and the fact that the correct processes execute rounds infinitely often, we conclude that there is a round r such that, at any round $r' \geq r$, we have $ld_i = L$ for any correct process p_i .

Let $L(r')$ be the subset of the processes of L that stop waiting at line 1 because the predicate $i \in ld_i$ is true while the predicate $m_i \neq \emptyset$ is false. We have $1 \leq |L(r')| \leq |L| \leq z$. Let us also notice that the invocations $R[r].write_snapshot()$ issued by the processes of $L(r')$ are set-linearized before the invocations issued by the processes that do not belong to $L(r')$.

As only the correct processes execute an infinite number of rounds, it follows that there is a round $r_c \geq r$, such that, for all $r'' \geq r_c$, $L(r'')$ always contains a correct process. Let us define $L' = \cup_{r' \geq r_c} L(r')$. We have $L' \neq \emptyset$, $L' \subseteq L$, and L' contains at least one correct process⁶. Finally, let us consider L' as the set that appears in the definition of the property PR_{Ω^z} . As L' contains at least one correct process, it follows that, during each round r' , there is at least one process p_k , $k \in L'$, such that $sm_k^{r'} \neq \emptyset$. Moreover, due to the fact that the invocations of the processes p_k , $k \in L(r')$, are set-linearized before the invocations from the other processes, we have $sm_k^{r'} \subseteq L$, which completes the proof. $\square_{Theorem 5}$

7 From $IRIS(PR_C)$ to a failure detector of the class C

Given the read/write model equipped with a failure detector of the class C , the previous section has shown how to simulate the $IRIS(PR_C)$ model. This section presents simulations building a failure detector of a class

⁶Let us notice that it is possible that there are distinct rounds r_1 and r_2 , such that $L(r_1) \cap L(r_2) = \emptyset$.

C from $IRIS(PR_C)$. The next section will provide complete simulation from $IRIS(PR_C)$ to the read/write model equipped with a failure detector of the class C .

7.1 From $IRIS(PR_{\diamond S_x})$ to a failure detector of the class $\diamond S_x$

A trivial algorithm implementing $\diamond S_x$ from $IRIS(PR_{\diamond S_x})$ is described in Figure 8. The set $TRUSTED_i$ is permanently updated to sm_i , where sm_i is the last invocation $restricted_w_snapshot(i)$.

```

(1) init  $r_i \leftarrow 0$ ;  $TRUSTED_i \leftarrow \Pi$ 

(2) loop forever  $r_i \leftarrow r_i + 1$ ;
(3)            $sm_i \leftarrow IS[r_i].restricted\_w\_snapshot(i)$ ;
(4)            $TRUSTED_i \leftarrow sm_i$ 
(5) end loop.
```

Figure 8: From $IRIS(PR_{\diamond S_x})$ to $\diamond S_x$ (code for p_i)

Theorem 6 *The algorithm described in Figure 8 constructs a failure detector of the class $\diamond S_x$ in the $IRIS(PR_{\diamond S_x})$ model.*

Proof The property $PR_{\diamond S_x}$ states that there is a set Q of x processes containing a correct process p_ℓ and a round r , such that, for any round $r' \geq r$, $sm_i^{r'}$ is empty (then p_i has crashed), or $sm_\ell^{r'} \subsetneq sm_i^{r'}$. Due to the assignment $TRUSTED_i \leftarrow sm_i^{r'}$ executed during each round $r' \geq r$, this immediately translates as “there is a set Q of x processes containing a correct process p_ℓ and a time τ after which p_ℓ is always trusted by the correct processes of Q ”. $\square_{Theorem 6}$

7.2 From $IRIS(PR_{\diamond \psi^y})$ to a failure detector of the class $\diamond \psi^y$

Figure 9 builds a failure detector of the class $\diamond \psi^y$ from $IRIS(PR_{\diamond \psi^y})$. It has the same structure as the previous algorithm. The only lines that is modified are the initialization line and line 4. The aim of this new line is to take into account the property of $PR_{\diamond \psi^y}$ (recall that wait-free is when $t = n - 1$).

```

(1) init  $r_i \leftarrow 0$ ;  $NB\_C_i \leftarrow (t + 1 - y)$ 

(2) loop forever  $r_i \leftarrow r_i + 1$ ;
(3)            $sm_i \leftarrow IS[r_i].restricted\_w\_snapshot(i)$ ;
(4)           if  $(i - 1) = ((r_i - 1) \bmod n)$  then  $NB\_C_i \leftarrow \max(t + 1 - y, n - |sm_i|)$  end if
(5) end loop.
```

Figure 9: From $IRIS(PR_{\diamond \psi^y})$ to $\diamond \psi^y$ (code for p_i)

Theorem 7 *The algorithm described in Figure 9 constructs a failure detector of the class $\diamond \psi^y$ in the $IRIS(PR_{\diamond \psi^y})$ model.*

Proof The proof is nearly the same as for Theorem 6. It is left to the reader. $\square_{Theorem 7}$

7.3 From $IRIS(PR_{\Omega^1})$ to a failure detector of the class Ω^1

Figure 10 considers the case $z = 1$. It consequently builds a failure detector of the class Ω (single leader) from $IRIS(PR_{\Omega})$. Its structure is a little bit more involved than the previous algorithms. At each round, a process p_i writes in the corresponding one-shot immediate snapshot object (line 3) the processes it has seen as participating to the previous round (their indexes are kept in $prev_sm_i$). Then, p_i computes (lines 4-5)

Irisa

the smallest set (sm_{in}) among the the $prev_sm_j$ sets that have been seen by the processes p_j it perceives as participating in the current round (these sets are ordered by the containment property). It then elects as its current leader (whose index is kept in $LEADERS_i$) the process with the smallest index in sm_{in} (line 6), and updates $prev_sm_i$ for the next round (line 7).

```

(1) init  $r_i \leftarrow 0$ ;  $prev\_sm_i \leftarrow \Pi$ ;  $LEADERS_i \leftarrow \{i\}$ 
(2) loop forever  $r_i \leftarrow r_i + 1$ ;
(3)            $sm_i \leftarrow IS[r_i].restricted\_w\_snapshot(< i, prev\_sm_i >)$ ;
(4)           let  $sm_{in}$  be the set such that
(5)             ( $< -, sm_{in} > \in sm_i$ )  $\wedge$  ( $\forall < j, sm_j > \in sm_i : sm_{in} \subseteq sm_j$ );
(6)            $LEADERS_i \leftarrow \min(sm_{in})$ ;
(7)            $prev\_sm_i \leftarrow \{j : < j, - > \in sm_i\}$ 
(8) end loop.

```

Figure 10: From $IRIS(PR_\Omega)$ to Ω (code for p_i)

Theorem 8 *The algorithm described in Figure 10 constructs a failure detector of the class Ω in the $IRIS(PR_\Omega)$ model.*

Proof Due to the property PR_Ω , there are a process p_ℓ and a round r after which, at any round $r' \geq r$, the $IS[r].restricted_w_snapshot()$ invocation issued by p_ℓ is always set-linearized before all the other $IS[r].restricted_w_snapshot()$ invocations. Consequently, at each round $r' > r$, we have $sm_\ell = \{< \ell, \{\ell\} >\}$. It follows from the containment property of the immediate snapshot that each process p_j that executes the round r' is such that $\{< \ell, \{\ell\} >\} \in sm_j^{r'}$. Consequently, when p_j executes line 6, we have $LEADERS_j = \ell$. It follows that after r , all the processes have the same leader p_ℓ . The observation that, as p_ℓ executes all the rounds, it is correct, completes the proof of the theorem. $\square_{Theorem\ 8}$

As it is more involved, the construction of a failure detector of the class Ω^z from $IRIS(PR_{\Omega^z})$ is presented in Appendix B.

8 From $IRIS(PR_C)$ to the read/write model with C

This section presents a simulation from $IRIS(PR_C)$ to the read/write model equipped with a failure detector of the class C , for any pair (C, PR_C) such that there is an algorithm A that builds a failure detector of the class C in the $IRIS(PR_C)$ model. (Examples of such algorithms A have been described in Section 7 for the three failure detector classes $\diamond\mathcal{S}_x$, $\diamond\psi^y$ and Ω^z .) Although the simulation is not wait-free (it is only non-blocking, Theorem 9), it is then used to derive that the base read/write mode equipped with a failure detector of class C and the corresponding $IRIS(PR_C)$ have the same computational power, as far as wait-free agreement tasks are concerned (Theorem 10).

Notation In the rest of the paper, the function $\max_{cw}()$ denotes the component-wise maximum of the size n vectors passed as input parameters. *Correct* denotes the set of processes that are correct in the considered run.

8.1 The general simulation

Principles of the simulation The algorithm extends the simulation given in [9] to the context of the read/write model equipped with failure detectors. (That construction can be seen as an “appropriate merge” of the algorithm described in [9] with a generalization of the algorithm described in Figure 15.)

An algorithm B in this model performs local computations, `write()`, `snapshot()` and `fd_query()` operations. Without loss of generality, we assume that (i) B is a full information deterministic algorithm⁷, (ii) (as in [9])

⁷In a full information algorithm, when a process writes, it writes all its causal past, and when it reads, it becomes aware of the causal past associated with the value it reads.

the k th value written by a process is k (consequently, a snapshot of the shared memory can be represented as a vector made up of n integers) and, (3), a `write()` is always followed by a `snapshot()`. So we have to simulate three kinds operation, namely `write_snapshot()`, `snapshot()` and `fd_query()`. A simulation of `write_snapshot()` is a shortcut for a simulation of the sequence `write(); snapshot()`. Each invocation `fd_query()` returns the current value of the failure detector, according to the underlying failure detector class C .

```

init    $r_i \leftarrow 0$ ;  $sm\_snap_i[1..n] \leftarrow [-1, \dots, -1]$ ;  $sm\_est_i[1..n] \leftarrow [0, \dots, 0]$ ;
        for each  $\rho \geq 1$  do  $view_i[\rho] \leftarrow \emptyset$  end for

function simulate( $op()$ )
(1)   if  $op() = write\_snapshot()$  then  $sm\_est_i[i] \leftarrow sm\_est_i[i] + 1$  end if;  $r\_start_i \leftarrow r_i$ ;
(2)   repeat  $r_i \leftarrow r_i + 1$ ;
(3)      $is_i \leftarrow IS[r_i].restricted\_w\_snapshot(< i, sm\_est_i, view_i[1..(r_i - 1)] >)$ ;
(4)      $view_i[r_i] \leftarrow \{ < i, \{ < j, sm\_est_j > : < j, sm\_est_j, - > \in sm_i \} > \}$ ;
(5)     for each  $\rho \in \{1, \dots, r_i - 1\}$  do  $view_i[\rho] \leftarrow \bigcup_{view_j : < j, - > \in is_i} view_j[\rho]$  end for;
(6)      $fd\_output_i \leftarrow$  current value of the failure detector computed from  $view_i[1..r_i]$ ;
(7)      $sm\_est_i \leftarrow \max_{cw} \{ sm\_est_j : < j, sm\_est_j, - > \in is_i \}$ ;
(8)     if  $(\exists \rho > r\_start_i : \exists < -, sm\_min > : \forall j \in sm\_min : < j, sm\_min > \in view_i[\rho])$ 
        % there is a smallest snapshot in  $view_i[r\_start_i..r_i]$  that is known by  $p_i$  %
(9)       then let  $\rho'$  be the greatest round  $\leq r_i$  that satisfies the previous predicate;
(10)       $sm\_min_i \leftarrow$  the smallest snapshot in  $view_i[\rho']$ ;
(11)       $sm\_snap_i \leftarrow \max_{cw} \{ sm\_est_j : < j, sm\_est_j > \in sm\_min_i \}$ ;
(12)    end if;
(13)    if  $op()$  is snapshot() or write_snapshot()
(14)      then case  $(\forall j \mid < j, sm\_est_j, - > \in is_i : sm\_est_j = sm\_est_i)$  then return  $(sm\_est_i)$ 
(15)         $(sm\_snap_i[i] = sm\_est_i[i])$  then return  $(sm\_snap_i)$ 
(16)        other cases then skip
(17)      end case
(18)    else return  $(fd\_output_i)$ 
(19)    end if
(20)  end repeat.

```

Figure 11: From $IRIS(PR_C)$ to the read/write model equipped with C (code for p_i)

Variables The algorithm described in Figure 11 implements the following three operations `snapshot()`, `fd_query()`, and `write_snapshot()` in $IRIS(PR_C)$. To attain this goal, it uses an infinite number of immediate snapshot objects $IS[1], IS[2], \dots$ shared by the processes. Each process manages the following local variables:

- r_i (initialized to 0) is a round number. fd_output_i is a local variable that contains the last failure detector value obtained from $IRIS(PR_C)$.
- Each process maintains a vector of integers, denoted $sm_est_i[1..n]$, that represents its current estimate of the state of the shared memory. More explicitly, $sm_est_i[j]$ denotes the value (a sequence number) associated with the last write announced by p_j (as known by p_i). It is important to notice that this write has been announced by p_j but maybe has not yet been committed in the shared memory. That array is initialized to $[0, \dots, 0]$.

Similarly, the array $sm_snap_i[1..n]$ represents the last snapshot of the shared memory, obtained by p_i . Its value is $[-1, \dots, -1]$ (i.e., undefined) until p_i has computed a snapshot value.

The aim of the function `simulate()` is to provide p_i with an estimate of the shared memory that (1) takes into account its last write operation (if it is `write_snapshot()`), and (2) is as recent as possible.

- $view_i[1..+\infty)$ is an infinite array (each entry of which is initialized to \emptyset) such that $view_i[\rho]$, $1 \leq \rho \leq r_i$, contains a set of pairs, i.e.,

$$view_i[\rho] = \{ \langle j1, sm_est_{j1} \rangle, \langle j2, sm_est_{j2} \rangle, \dots \},$$

and each $sm_est_{j_1}$ is in turn such that

$$sm_est_{j_1} = \{ \langle k_1, sm_est_{k_1} \rangle, \langle k_2, sm_est_{k_2} \rangle, \dots \}.$$

The local variable $view_i[\rho]$ aggregates the “knowledge” that p_i has obtained as far as the values sm_j returned to the processes p_j from $IS[\rho]$ are concerned. During each round r , p_i improves its “knowledge” of $view_i[\rho]$ for each $\rho < r$ (the values of $view_i[\rho]$ are computed at lines 4 and 5).

Process behavior If the current operation to simulate is a `write_snapshot()` operation, p_i increases $sm_est_i[i]$ to announce that it wants to write the shared memory (line 1). In all cases, it updates r_start_i , that records the round at which p_i starts simulating the operation.

Then, p_i enters a loop (lines 2-19). It will exit that loop when it executes a `return()` statement at line 14, 15 or 17. Each execution of a loop body corresponds to a new round. The behavior of a process p_i during a round r , can be decomposed into three parts.

1. Part 1: lines 3-6.

A process p_i first invokes $IS[r].restricted_w_snapshot(\langle i, sm_est_i, view_i[1, \dots, r_i - 1] \rangle)$ (line 3) in order to write in the shared immediate snapshot object $IS[r]$ (1) its current estimate of the shared memory (sm_est_i) and (2) everything it knows concerning the previous rounds ($view_i[1..(r-1)]$). Then, according to the value it has obtained from the $IS[r]$ object, p_i computes its current view associated with the round r (line 4), updates its knowledge on immediate snapshots returned in the previous rounds (line 5), and finally computes the current output of the failure detector (line 6).

2. Part 2: lines 7-12.

During this part, a process p_i determines its view of the shared memory. First, according to what it has learned during this round (kept in is_i), p_i updates its estimate of the shared memory (line 7). To do so, it computes the maximum component-wise (denoted \max_{CW}) of the estimate vectors sm_est_j it sees in its immediate snapshot of round r . While there is a smallest immediate snapshot for each object $IS[\rho]$, it is possible that p_i does not know such a snapshot. (Let us recall, that, due to the containment property provided by the immediate snapshot objects, the smallest snapshot provided by $IS[\rho]$ is included in the snapshot obtained by each process that invokes $IS[\rho]$, but that process does not necessarily know it explicitly). So, the next thing that p_i does is the determination of the last smallest snapshot that it can know. If there is a more recent one, p_i keeps it in sm_min_i (lines 8-9), and computes in sm_snap_i the corresponding value of the shared memory (line 11). If p_i cannot improve its smallest snapshot (i.e., find a more recent one), it keeps the previous one.

3. Part 3: lines 13-18.

In this last part, a process p_i terminates the current round r . If the simulated operation was `fd_query()`, p_i simply returns the current output of the failure detector (line 18). Otherwise, the operation was `snapshot()` or `write_snapshot()` (line 13). There are three cases.

- If the immediate snapshot is_i that p_i has obtained from the $IS[r]$ object is such that all the processes p_j that appear in is_i have the same estimate of the shared memory as p_i (test of line 14), then the operation issued by p_i succeeds: it is committed in the shared memory and its local estimate sm_est_i is consequently a valid snapshot (i.e., a snapshot that can be totally ordered -in a consistent way- with the other operations). Hence, sm_est_i is returned as the result of the simulation of the operation $op()$.

When the estimate vectors in is_i satisfy the predicate of line 14, it follows from the containment property of immediate snapshots that the estimates vectors in the smallest immediate snapshot of round r also satisfy the predicate. Due to fact that an est_sm_i is a component-wise maximum, this implies that each estimate computed at rounds greater or equal to r is component-wise greater or equal to the vector returned by p_i . Hence, any operation that starts at a round $\geq r$ may only return the same vector or a vector that is component-wise greater or equal to it.

- It is possible that the previous test (line 14) be false for p_i^8 , but p_i observes it last operation announced (that is identified $sm_est_i[i]$) in an estimate that appears in some smallest immediate snapshot sm_{in}_i (line 15). p_i can safely return the component-wise maximum of the estimates sm_snap_i in this immediate snapshot since, as in the previous case, any operation that starts at a round $\geq r$ may only return the same vector or a vector that is component-wise greater or equal to it. Moreover, no operation that terminates at a round $\leq r_start_i$ (p_i starts simulating its operation at round $r_start_i + 1$) can return a vector strictly greater than sm_snap_i . This is because sm_{in}_i is returned at some round during which p_i is simulating its current operation.
- In the other cases (line 16), the simulated operation cannot terminate. The process p_i starts consequently a new round to try to terminate the current operation.

Let us observe that a failure detector query always terminates. Differently, the termination of a `snapshot()` operation or a `write_snapshot()` operation depends on the property PR_C embedded in $IRIS(PR_C)$.

8.2 Proof of the general simulation

Preliminaries Let x_i^r denote the value of local variable x at process p_i by the end of round r (i.e., before p_i executes $r_i \leftarrow r + 1$). Among all the immediate snapshots returned by `restricted_w_snapshot()` invocations on object $IS[r]$, sm_{in}^r is the smallest immediate snapshot returned. As the immediate snapshots returned by $IS[r].write_snapshot()$ invocations are ordered by containment, sm_{in}^r is well defined.

Let sm_value be a snapshot of the simulated shared memory returned at line 14 or at line 15 by an of `simulate(op)`, where $op \in \{write(), write_snapshot()\}$. sm_value is associated with the pair (r, sm_{in}^r) if one of the following conditions holds:

- sm_value is returned at line 14 and when this happens, $r_i = r$.
- sm_value is returned at line 15 and the smallest snapshot from which sm_value is computed (lines 8-12) is returned by `restricted_w_invocation()` on object $IS[r]$.

Let us observe that, in both cases, sm_value is the component-wise maximum of all the estimates in sm_{in}^r , i.e., $sm_value = \max_{cw}\{sm_est : \langle -, sm_est, - \rangle \in sm_{in}^r\}$. If sm_value is returned at line 14 by process p_i at round r , all the estimates observed by p_i in is_i^r are equal to sm_value . It then follows from the containment property of immediate snapshots that all estimates in sm_{in}^r are also equal to sm_value .

The first part of the proof show that the operations on the shared memory can be totally ordered.

Lemma 2 Let sm_{in}^r (resp., $sm_{in}^{r'}$) be the smallest snapshot returned at round r (resp. r') and let $sm_value = \max_{cw}\{sm_est : \langle -, sm_est, - \rangle \in sm_{in}^r\}$ (resp., $sm_value' = \max_{cw}\{sm_est : \langle -, sm_est, - \rangle \in sm_{in}^{r'}\}$). $r \leq r' \Rightarrow sm_value \leq sm_value'$.

Proof If $r = r'$, the Lemma directly follows from the fact that, given a round, the smallest immediate snapshot returned at that round is uniquely defined. Therefore, let us assume that $r < r'$. The proof is based on the two following observations: (O1) $\forall \rho, \forall i : \max_{cw}\{sm_est : \langle -, sm_est, - \rangle \in sm_{in}^\rho\} \leq sm_est_i^\rho$ and, (O2) $\forall \rho, \forall i : sm_est_i^\rho \leq sm_est_i^{\rho+1}$.

Let p_i be a process such that $\langle i, -, - \rangle \in sm_{in}^{r'}$. Due to observation O1, $sm_value \leq sm_est_i^r$. Moreover, it follows from observation O2 that $sm_est_i^r \leq sm_est_i^{r'-1}$. As p_i writes $sm_est_i^{r'-1}$ at round r' and p_i belongs to the smallest immediate snapshot $sm_{in}^{r'}$, we obtain have $sm_est_i^{r'-1} \leq sm_value'$. Since this is true $\forall j : \langle j, -, - \rangle \in sm_{in}^{r'}$, we conclude that $sm_value \leq sm_value'$.

Proof of Observation O1. Let p_i be a process that obtains an immediate snapshot is_i^ρ at round ρ . Due to the containment property, $sm_{in}^\rho \subseteq is_i^\rho$. As p_i updates its estimate vector by taking the maximum component-wise of all estimate vectors it observes in is_i^ρ , it follows that $\max_{cw}\{sm_est : \langle -, sm_est, - \rangle \in sm_{in}^\rho\} \leq sm_est_i^\rho$. End of the proof of Observation O1.

⁸If both conditions at lines 14 or 15 are simultaneously satisfied, one of them is arbitrarily selected.

Proof of Observation O2. Let p_i be a process that updates its estimate at round $\rho + 1$. Due to the self inclusion property of immediate snapshots, p_i observes $sm_est_i^\rho$ in the immediate snapshot it obtains at round $\rho + 1$. As p_i updates sm_est_i by taking the component-wise maximum over all the estimates it observes, it follows that $sm_est_i^\rho \leq sm_est_i^{\rho+1}$. *End of the proof of Observation O2.* $\square_{Lemma\ 2}$

Lemma 3 *Let sm_val be the value returned by the k -th invocation of `simulate(write_snapshot())` by process p_i . $sm_val[i] = k$.*

Proof Let us first observe that for each `simulate(write_snapshot())` invocation issued by p_i , p_i increments $sm_est_i[i]$ (line 1) and no process $\neq p_i$ increments $est_sm_j[i]$. When p_i returns, either $sm_val[i] = sm_est_i[i]$ (line 15) or $sm_val \geq sm_est$ (line 14), from which we conclude that $sm_val[i] = k$. $\square_{Lemma\ 3}$

Lemma 4 *Let us assume that sm_val is returned at round r as a result of a simulation of a `write_snapshot()` or `snapshot()`. If p_j has not started simulating its k -th `write_snapshot()` at round r , then $sm_val[j] < k$.*

Proof At process p_i , $sm_est_i[j] = k$ is always due to the fact that process p_j has executed $sm_est_j[j] \leftarrow k$. This occurs only when p_j starts simulating a new `write_snapshot()` operation (line 1). $\square_{Lemma\ 4}$

The next lemma shows that the `write()` and `snapshot()` operations can be totally ordered. Moreover, in this order, each `snapshot()` operations returns the values written by the last write operations that precede it. In addition, this order is consistent with a notion of time denoted “*round-time*” defined below. Relevant events are the beginning and the end of `write()` or `snapshot()` operations. Such an event occurs when the corresponding invocation of `simulate()` starts or ends. The round-time of an event is a pair (r, id) where r is a round number and id is the identity of the process that writes or reads. The simulation of an operation op by p_i involves invocations of `restricted_w_snapshot()` on objects $IS[r_s], IS[r_s + 1], \dots, IS[r_e]$. Let τ_{op}^s and τ_{op}^e be the time at which op starts and ends at process p_i , respectively. We define $\tau_{op}^s = (r_s, i)$ and $\tau_{op}^e = (r_e, i)$. Round times are totally ordered in the obvious way.

Lemma 5 *There is a total order on the simulated `write()` and `snapshot()` operations that (1) respects their round-time occurrence, and (2) such that any `snapshot()` operation obtains the values written by the last write operations that precede it in this sequence.*

Proof Let us first remind that an invocation of `simulate(write_snapshot())` simulates a `write()` followed by a `snapshot()`. The k -th write operations of process p_i is *effective* if there exists a vector sm_value returned as a result of a `simulate(op)` invocation such that $sm_value[i] = k$. All write operations whose associated `simulate(write_snapshot())` invocation terminates are effective. Some of the write operations whose associated `simulate(write_snapshot())` invocation does not terminate are effective, others are not. Intuitively, an effective write is a write whose value is seen by other processes. Finally, let an *effective* snapshot be a snapshot operation such that the associated `simulate(snapshot())` invocation terminates.

For each simulated effective operation $op \in \{\text{write}(), \text{snapshot}\}$, we associate a timestamp $ts(op) = (v_{op}, r_{op})$ where v_{op} is a vector of integers and r_{op} a round number. A `snapshot()` operation op is associated with the vector sm_value returned as the result of the corresponding `simulate(snapshot())` or `simulate(write_snapshot())`. Such a vector is uniquely associated with a pair (sm_min^r, r) (see the preliminaries). We define $r_{op} = r$. For a write operation, let us us consider the k -th `write()` issued by p_i and let us assume that this operation is effective. Then there is a vector sm_value returned such that $sm_value[i] = k$. Moreover, as we observed in the preliminaries, there exists r such that $sm_value = \max_{CW} \{sm_est : < -, sm_est, - > \in sm_min^r\}$. Let r_m be the smallest round number such that $sm_value = \max_{CW} \{sm_est : < -, sm_est, - > \in sm_min^r\}$. The write operation is associated with (v_{op}, r_{op}) such that $v_{op} = \max_{CW} \{sm_est : < -, sm_est, - > \in sm_min^{r_m}\}$ and $r_{op} = r_m$.

As each v_{op} is the component-wise maximum over all the estimates in the smallest immediate snapshot of round r_{op} , it follows from Lemma 2 that that all timestamps can be totally ordered. Moreover, we have $r_s \leq r_{op} \leq r_e$ where r_s and r_e are the rounds at which the simulation of the operation starts and ends. (If the simulation does not terminate, $r_e = \infty$).

Let us define a total order S on all the effective operations as follows. The operations are first ordered in S according to their timestamp. If several operations have the same timestamp, the writes are ordered before the reads. Writes (resp. reads) are then ordered according to the round-time at which they start. The next two claims establish that S respects the round-time occurrence of write and snapshot operation and snapshots return values consistent with S .

Claim C1: S respects the round-time occurrence of operations.

Proof of the claim C1 Let us consider two operations $op1$ and $op2$ such that $op1$ precedes $op2$ in the round-time order. Let $\tau_1 = (r_1, id_1)$ be the round time at which $op1$ ends and $\tau_2 = (r_2, id_2) (> \tau_1)$ be the round time at which $op2$ starts and let $(v_{op1}, r_{op1}), (v_{op2}, r_{op2})$ be their associated timestamp. Let us observe that we have $r_{op1} \leq r_1 \leq r_2 \leq r_{op2}$. It consequently follows from the definition timestamps and lemma 2 that $(v_{op1}, r_{op1}) \leq (v_{op2}, r_{op2})$. The only case to discuss is then $(v_{op1}, r_{op1}) = (v_{op2}, r_{op2})$. If $op1$ and $op2$ are two operations of the same type or $op1$ is a write and $op2$ is a snapshot, $op1$ is ordered before $op2$ in S . In the remaining case ($op1$ is a read and $op2$ is a write, say the k -th simulated write of p_i), we have (1) $v_{op2}[i] = k$ (Lemma 3) and (2) $v_{op1}[i] < k$ (Lemma 4): contradiction with $v_{op1} = v_{op2}$. *End of the proof of claim C1.*

Claim C2: Any snapshot operation obtains the values written by the last writes that precede it in S .

Proof of the claim C2 Let us consider a snapshot operation s and let (v, r) be its timestamp. The result of this snapshot operation is then v . Let us assume that $v[i] = k$. It follows from lemma 4 that p_i writes at least k times. Let w_k be this operation and w_{k+1} be the $k+1$ -th write operation of p_i (if any). According to the definition of timestamp and Lemma 4, all operations op that precede w_k in S have a timestamp (v_{op}, r_{op}) such that $v_{op}[i] < k$. Consequently, w_k is ordered before s in S . The vector w_{k+1} in the timestamp of w_{k+1} is such that $w_{k+1}[i] = k+1$. As all vectors in timestamps are ordered, $v < w_{k+1}$, i.e., w_{k+1} is ordered after s in S . *End of the proof of claim C2.* $\square_{Lemma 5}$

The first part of the proof has addressed the “safety” part of simulation. In the following we address the liveness part. We show that the simulation is non-blocking. To do so, we consider runs in which each process invokes infinitely often `simulate(op)`, $op \in \{\text{snapshot}, \text{write_snapshot}, \text{fd_query}\}$ (until it possibly crashes). In such run each correct process invokes `restricted_w_snapshot()` infinitely many often. Given a run in the $IRIS(PR_C)$ model in which some processes invoke infinitely many often `restricted_w_snapshot()`, we define a relation \rightsquigarrow between processes as follows: $p_i \rightsquigarrow p_j$ if p_i sees p_j taking an infinite number of steps. Consequently, $p_i \rightsquigarrow p_j$ if $\{r : < j, -, - > \in is_i^r\}$ is infinite. Let us observe that the relation \rightsquigarrow is reflexive, transitive and antisymmetric, from which we have that $(Correct, \rightsquigarrow)$ is a partially ordered set. Let Cl_{\min} be the set of correct processes that are seen infinitely many often by all other correct processes. More precisely, $Cl_{\min} = \{p_i : \forall p_j \in Correct, p_j \rightsquigarrow p_i\}$. It follows from the properties of the relation \rightsquigarrow that $Cl_{\min} \neq \emptyset$.

The next lemmas consider processes p_i that belong to Cl_{\min} and establish that (1) each operation simulated by p_i terminates, and (2) the values returned by the (simulated) operations of p_i may have been returned in some run of the read/write model equipped with a failure detector of class C in which each process $p_j \notin Cl_{\min}$ is faulty.

Lemma 6 $\exists R : \forall p_i \in Cl_{\min}, \forall r \geq R : (< j, -, - > \in smin^r) \vee (< j, -, - > \in is_i^r) \Rightarrow p_j \in Cl_{\min}$.

Proof Let $p_i \in Cl_{\min}$ and $p_j \notin Cl_{\min}$. It follows from the definition of the class Cl_{\min} that p_j is seen (directly or indirectly) by p_i finitely many often. More precisely, $\exists r_j$ such that $\forall r \geq r_j : < j, -, - > \notin is_i^r$. Moreover, for $r \geq r_j$, $< j, -, - > \notin smin^r$. Otherwise, due to the containment property of immediate snapshots, we would have $< j, -, - > \in is_i^r$. Taking $R = \max\{r_j : p_j \notin Cl_{\min}\}$ completes the proof. $\square_{Lemma 6}$

Lemma 7 Let $r \geq n$ be a round number and p_i be a process that completes the round r . When $r_i = r$, $\exists r' : r - n < r' \leq r$ such that the predicate of line 8 is satisfied for $\rho = r'$ (i.e., by the end of the round r , p_i knows the smallest immediate snapshot returned by the `restricted_w_snapshot()` invocations on the object $IS[r']$).

Proof We prove the lemma by induction. Let $RA_1(k)$ the following property:

$$\left| \bigcup_{r-k < \rho \leq r} is_i^\rho \right| \leq k \Rightarrow \exists r', r_k < r' \leq r : smin_i^r = smin_i^{r'}$$

Irisa

where $sm_{in}_i^r$ is the smallest immediate snapshot computed by p_i during round r .

- $RA_I(1)$. In that case $|is_i^r| = 1$. Thus, the smallest snapshot returned at round r has cardinality 1, is returned by p_i and therefore known by p_i .
- $RA_I(k) \Rightarrow RA_I(k+1)$. We consider two cases:
 - $\forall j \in sm_{in}^{r-k} : \exists \rho, r-k+1 \leq \rho \leq r : j \in is_i^\rho$. Let $j \in sm_{in}^{r-k}$. There exists a round $r', r-k < r' \leq r$ such that $j \in is_i^{r'}$. At round r' , p_j writes all the snapshots it has obtained in the previous rounds (line 3). In particular, when it invokes $IS[r'].restricted_w_snapshot(< *, view_j >)$, $< j, sm_{in}^{r-k} > \in view_j[r-k]$. Consequently, as $j \in is_i^{r'}$, we have $< j, sm_{in}^{r-k} > \in view_i^{r'}[r-k]$ after p_i has updated $view_i$ at round r' (line 5). It then follows that, at round r , $\forall j \in sm_{in}^{r-k} : < j, sm_{in}^{r-k} > \in view_i^r[r-k]$, from which we conclude that the predicate of line 8 is satisfied for $\rho = r-k$.
 - $\exists j \in sm_{in}^{r-k} : \forall \rho, r-k+1 \leq \rho \leq r : j \notin is_i^\rho$. The case assumption implies that $|\bigcup_{r-k < \rho \leq r} is_i^\rho| \leq k$. Consequently, due to $RA_I(k)$, $\exists r' : r-k < r' \leq r$ such that $sm_{in}_i^r = sm_{in}^{r'}$. $\square_{Lemma 7}$

Lemma 8 *Let $p_i \in Cl_{min}$. Each invocation of $simulate(op)$ issued by p_i terminates.*

Proof Let $p_i \in Cl_{min}$ and let us consider an operation op simulated by p_i . It directly follows from the protocol text that any invocation of $simulate(fd_query())$ terminates. So let us consider that $op \in \{write_snapshot, snapshot\}$ and let us assume that the simulation of op does not terminate. Let k be the value of $sm_est_i[i]$ when p_i starts the repeat loop while simulating op . r_0 denotes the round at which p_i starts simulating op .

First, as processes update their estimate by taking the maximum component-wise of the estimates they see in their immediate snapshots, it follows from the definition of the class Cl_{min} that there is a round r_1 such that $\forall r \geq r_1, \forall p_j \in Cl_{min} : sm_est_j^r[i] = k$. Second, after some round r_2 , the smallest immediate snapshot contains only estimates written by processes $\in Cl_{min}$ (Lemma 6). Finally, with a large enough number of invocations of $restricted_w_snapshot()$, p_i can discover the smallest immediate snapshot returned at some round $r \geq \max(r_1, r_2, r_0)$ (Lemma 7). By piecing together these three observations, we obtain that p_i eventually computes a snapshot of the shared memory sm_snap_i such that $sm_snap_i[i] = k$. Consequently, p_i eventually evaluates the predicate of line 15 to true and completes the simulation of op : a contradiction. $\square_{Lemma 8}$

The next theorem shows that the algorithm depicted in Figure 11 provides a non-blocking simulation of the read/write model equipped with a failure detector of class C in the $IRIS(PR_C)$ model.

Theorem 9 *By invoking sequentially infinitely often (until they possibly crash) $simulate(op)$, processes simulate a run in the read/write model equipped with a failure detector of class C in which at least one process is correct. Moreover, each process that is correct in the simulated read/write run is correct in the $IRIS(PR_C)$ run.*

Proof In the simulated read/write run, relevant events are $fd_query()$, the end and the beginning of $write()$ and $snapshot()$ operations. Such an event e is associated with a round time (r_e, id_e) (see the discussion that precedes Lemma 5) where r_e is the round in the $IRIS(PR_C)$ at which the simulation of the corresponding operation starts or ends. id_e is the identity of the process that simulates the operation. In the simulated read/write run, let us define the “real time” rt_e at which e occurs to be $n.r_e + (id_e - 1)$.

According to this notion of “real time”, let us define a failure pattern fp_{rw} as follows. A process is rw -correct if it terminates infinitely many operations. Otherwise it is rw -faulty. Let r_{ns} be a round such that no rw -faulty process starts simulating an operation after round r_{ns} and no operations issued by a rw -faulty process terminates after r_{ns} . Let $R_f = \max(R+1, r_{ns})$ where R is the round number introduced in Lemma 6. We define the real time at which rw -faulty processes crash to be $n.R_f$. This failure pattern fp_{rw} induces a set of correct process $Correct_{rw}$. According to the definition of fp_{rw} , there are infinitely many operations issued by each process $\in Correct_{rw}$ in the simulated read/write run and finitely many operations issued by processes $\notin Correct_{rw}$.

The simulation is correct and non-blocking if we can show that, according to the time notion defined above and the failure pattern fp_{rw} , the simulated (infinite) read/write run satisfies the following properties:

1. `write()` and `snapshot()` operations are linearizable according to their occurrence order in the simulated read/write run.
2. The occurrence order of the simulated operations is consistent with fp_{rw} (i.e., each $p_i \in Correct_{rw}$ issues -and completes- infinitely many operations, and no process $\notin Correct_{rw}$ starts or terminates an operation after it has crashed).
3. $Correct_{rw} \neq \emptyset$ (no trivial simulation).
4. At each process, the failure detector output is valid according to the specification of class C with respect to the failure pattern fp_{rw} .
5. A process $\in Correct_{rw}$ is correct from the point of view of the $IRIS(PR_C)$ run.

Proof of properties 1, 2, 3, 4 and 5.

1. This is stated and proved in Lemma 5.
2. This directly follows from the definition of fp_{rw} .
3. Let $p_i \in Cl_{\min}$. It follows from Lemma 8 that p_i simulates infinitely many operations from which we have $p_i \in Correct_{rw}$. Then $Cl_{\min} \subseteq Correct_{rw}$, and, as $Cl_{\min} \neq \emptyset$, $Correct_{rw} \neq \emptyset$.
4. Let us first consider the simulation of the failure detector output at a process $p_i \in Cl_{\min}$. It follows from Lemma 6 that, after round $R_f > R$, no process $\notin Cl_{\min}$ is seen by processes $\in Cl_{\min}$. For those processes, there is no mean to distinguish the actual $IRIS(PR_C)$ run from a run in which processes $\notin Cl_{\min}$ fail before starting round R_f . Consequently, the value returned by a `fd_query()` that occurs at a time $\geq n.R_f$ (simulated at a round $\geq R_f$ in $IRIS(PR_C)$) is valid with respect to fp_{rw} . (Simulating `fd_query()` at round $\geq R_f$, p_i cannot distinguish between its current view and a view in which processes $\notin Correct_{rw}$ fail before executing round R_f , i.e., fails at time $R_f.n$ in the corresponding simulated read/write run).

Finally, due to the correctness of the algorithm that emulates the failure detector, the output of the failure detector at a process $\notin Cl_{\min}$ is consistent with the output at processes $\in Cl_{\min}$.

5. For a rw -correct process p_i , there are infinitely many operations in the simulated read/write run. This implies that p_i executes an infinite number of rounds in $IRIS(PR_C)$, i.e., p_i is correct.

End of the Proof of properties 1, 2, 3, 4 and 5.

□*Theorem 9*

Finally, observing that a non-blocking simulation is equivalent to wait-free solvability for agreement tasks, we obtain the following theorem.

Theorem 10 *Let C be a failure detector class and $IRIS(PR_C)$ be the corresponding iterated restricted immediate snapshot model. Let us assume that there are two algorithm $A1$ and $A2$ such that (1) $A1$ implements $IRIS(PR_C)$ in the read/write model equipped with a failure detector of class C and, (2) $A2$ builds a failure detector of the class C in $IRIS(PR_C)$. An agreement task T is solvable in $IRIS(PR_C)$ if and only if it is wait free solvable in the read/write model equipped with a failure detector of class C .*

Proof Let us first consider the \Rightarrow direction. Let A be an algorithm that solves T in the $IRIS(PR_C)$ model. It follows that by stacking A on top of the algorithm $A1$ we obtain an algorithm that solves T in the read/write model equipped with a failure detector of the class C .

Let us now consider the \Leftarrow direction. Let A be an algorithm that solves the task T in the read/write model equipped with a failure detector of the class C . As the simulation is non-blocking, at least one process will eventually decide.

On another side, some correct processes may be failed in the simulated read/write execution. In order to help those processes to decide, a process that has decided executes forever the repeat loop in the code of `simulate()`, writing (at line 3) its decision value (in addition to the value specified). As we are concerned with agreement tasks, a process that observes such a decision value can then locally compute its own decision. More precisely, let us consider a process $p_i \in Cl_{\min}$. Such a process will eventually decide, either directly (Lemma 8), or indirectly (observing the decision value of another process). As a process $\in Cl_{\min}$ is seen infinitely often by all the correct processes, it follows that all the correct processes eventually see the decision of p_i .

□*Theorem 10*

9 Benefiting from the $IRIS(PR_C)$ model

9.1 Characterizing wait-free solvable tasks

The previous characterization in the $IRIS(PR_C)$ framework of the synchrony achievable by the failure detector families $\{\diamond\mathcal{S}_x\}_{1 \leq x \leq n}$, $\{\Omega^z\}_{1 \leq z \leq n}$, and $\{\diamond\psi^y\}_{1 \leq y \leq n}$ can be used to study their computational power in the read/write shared memory model. As a particular example, we have the following.

Theorem 11 *The k -set agreement problem is not solvable in a read/write shared memory system with a failure detector of the class Ω^z if $k < z$.*

This result was proved in [26] by reduction to a similar impossibility for $\{\diamond\mathcal{S}_x\}$ proved in [20] using combinatorial topology techniques from [21]. A simple proof of the theorem is described next.

Consider the $IRIS(PR_{\Omega^z})$ model. Notice that all runs of the IIS model where are most z processes are correct (and the others crash initially) are runs of the $IRIS(PR_{\Omega^z})$ model. This is because these processes do not see a write by any other process (i.e., their views are always contained in a set L of size at most z , as required by property PR_{Ω^z} property). But it is known that in the IIS model of z processes, $(z - 1)$ -set agreement is not solvable [9] (because it is similar to a wait-free system of z processes).

More generally, thanks to Theorem 10, the $IRIS(PR_C)$ allows characterizing the agreement tasks wait-free solvable in the read/write model enriched with a failure detector of the class C .

9.2 The failure detector classes $\mathcal{S}_{x,q}$ and $\diamond\mathcal{S}_{x,q}$

To illustrate the advantage of the $IRIS(PR_C)$ framework when one is interested in lower bounds, this section gives a new proof of the lower bound for the k -set agreement problem. That lower bound, conjectured in [29], has been proved in [20] in the context of t -resilient message-passing systems, using techniques borrowed from combinatorial topology. The new proof is on the *wait-free* case ($t = n - 1$) in the read/write model enriched with a failure detector of the class $\diamond\mathcal{S}_{x,q}$. Technically speaking, the problem is reduced to the question of the k -set agreement wait-free solvability. No topology notion is required.

The family $\{\diamond\mathcal{S}_{x,q}\}_{1 \leq x \leq n, 1 \leq q \leq x}$ extends the notion of limited scope failure detector to a system where the processes are partitioned into multiple disjoint clusters. There are q disjoint clusters denoted X_1, \dots, X_q , where $|X_i| = x_i$, $X = \bigcup_{1 \leq i \leq q} X_i$ and $x = \sum_{i=1}^q x_i$. Informally, there is a process that is never suspected in each cluster X_i . More specifically, the variable TRUSTED_i provided by a failure detector of the class $\diamond\mathcal{S}_{x,q}$ contains the identities of the processes that are believed to be currently alive. When $j \in \text{TRUSTED}_i$ we say “ p_i trusts p_j .” By definition, a crashed process trusts all the processes. The failure detector class $\diamond\mathcal{S}_{x,q}$ is defined by the following properties:

- **Strong completeness.** There is a time after which every faulty process is never trusted by every correct process.
- **Eventual weak (x, q) -accuracy.** There are q disjoint sets X_1, \dots, X_q of cumulatively x processes, q processes $p_{\ell_1} \in X_1, \dots, p_{\ell_q} \in X_q$ and a (finite) time τ such that each process of X_i trusts p_{ℓ_i} .

The time τ , the set X_1, \dots, X_q and the processes p_{ℓ_i} are not explicitly known. Moreover, some or all processes of X_i may be faulty (A cluster X_i of faulty processes trivially satisfies (x, q) -accuracy).

As in Section 3.1, we use the following equivalent formulation of $\diamond\mathcal{S}_{x,q}$ [26], assuming the local variable controlled by the failure detector is REPR_i .

- **Eventual (x, q) -common representative.** There are q disjoint sets X_1, \dots, X_q of cumulatively x processes, q processes $p_{\ell_1} \in X_1, \dots, p_{\ell_q} \in X_q$, and a (finite) time τ after which, for any correct process p_j , we have $j \in X_i \Rightarrow \text{REPR}_j = \ell_i$ and $j \notin \bigcup_{1 \leq i \leq q} X_i \Rightarrow \text{REPR}_j = j$.

Clearly, a failure detector that satisfies the previous property can be transformed into one of the class $\diamond\mathcal{S}_{x,q}$ (define $\text{TRUSTED}_i = \{\text{REPR}_i\}$). Conversely, one can easily extend the algorithm in [26] that transforms a failure detector of class $\diamond\mathcal{S}_x$ into a failure detector satisfying the limited eventual common representative property to the context of the family $\{\diamond\mathcal{S}_{x,q}\}_{1 \leq x \leq n, 1 \leq q \leq x}$.

9.3 The lower bounds

The lower bounds established in [20] are on t -resilient asynchronous message-passing systems (i.e., systems prone to up to t process crashes). They are the following.

- If the system is equipped with $\mathcal{S}_{x,q}$, any k -set agreement protocol must satisfy $t < k + x - q$ if $q \leq k$ and $t < x$ otherwise.
- If the system is equipped with $\diamond\mathcal{S}_{x,q}$, any k -set agreement protocol must satisfy $t < \min(\frac{n}{2}, k + x - q)$ if $q \leq k$ and $t < \min(\frac{n}{2}, x)$ otherwise. (In the shared memory context, the requirement $t < \frac{n}{2}$ is no longer needed, and the lower bound becomes $t < \min(k + x - q)$.)

9.4 $IRIS(PR_{\diamond\mathcal{S}_{x,q}})$

The property $PR_{\diamond\mathcal{S}_{x,q}}$ extends the property $PR_{\diamond\mathcal{S}_x}$ in a natural way. Informally, $PR_{\diamond\mathcal{S}_{x,q}}$ is satisfied if $PR_{\diamond\mathcal{S}_{x_i}}$ is satisfied for each cluster X_i .

$$\begin{aligned} PR_{\diamond\mathcal{S}_{x,q}} \equiv & \exists X_1, \dots, X_q : \left| \bigcup_{1 \leq j \leq q} X_j \right| \geq x \wedge \forall 1 \leq j < k \leq q : X_j \cap X_k = \emptyset, \\ & \exists \ell_1, \dots, \ell_q : \forall 1 \leq j \leq q : \ell_j \in X_j, \\ & \exists r : \forall r' \geq r, \forall 1 \leq j \leq q : (i \in X_j - \{\ell_j\}) \Rightarrow (sm_i^{r'} = \emptyset \vee sm_{\ell_j}^{r'} \subsetneq sm_i^{r'}). \end{aligned}$$

This property states that, for each cluster X_i , there is a process p_{ℓ_i} that, from some round r , always belongs to the view of the processes of X_i that have not crashed.

9.5 $IRIS(PR_{\diamond\mathcal{S}_{x,q}})$ vs read/write model equipped with $\diamond\mathcal{S}_{x,q}$

Building $IRIS(PR_{\diamond\mathcal{S}_{x,q}})$ An algorithm that simulates the $IRIS(PR_{\diamond\mathcal{S}_x})$ model from one-shot immediate snapshot objects is described in Figure 5. One can easily check that this algorithm, when used with the representative variable $REPR_i$ based definition of the class $\diamond\mathcal{S}_{x,q}$ builds $IRIS(PR_{\diamond\mathcal{S}_{x,q}})$. Thus, we obtain:

Theorem 12 *Assuming a basic R/W model equipped with a failure detector of the class $\diamond\mathcal{S}_{x,q}$, the algorithm described in Figure 5 is a simulation of the $IRIS(PR_{\diamond\mathcal{S}_{x,q}})$ computation model.*

From $IRIS(PR_{\diamond\mathcal{S}_{x,q}})$ to a failure detector of the class $\diamond\mathcal{S}_{x,q}$ A very simple algorithm implementing $\diamond\mathcal{S}_x$ from $IRIS(PR_{\diamond\mathcal{S}_x})$ has been described in Figure 8. The set $TRUSTED_i$ is permanently updated to sm_i , where sm_i is the last invocation `restricted_w_snapshot(i)`. Again, one can easily check that executing this algorithm in the $IRIS(PR_{\diamond\mathcal{S}_{x,q}})$ model implements a failure detector of the class $\diamond\mathcal{S}_{x,q}$.

Theorem 13 *The algorithm described in Figure 8 constructs a failure detector of the class $\diamond\mathcal{S}_{x,q}$ in the $IRIS(PR_{\diamond\mathcal{S}_{x,q}})$ model.*

Proof The property $PR_{\diamond\mathcal{S}_{x,q}}$ states that there are q disjoint sets X_1, \dots, X_q of cumulatively x processes and q processes $p_{\ell_1} \in X_1, \dots, p_{\ell_q} \in X_q$ and a round r after which, $\forall j : 1 \leq j \leq q, \ell_j$ belongs to the views $sm_i^{r'}$ of the processes p_i of X_j that have not yet crashed. Due to the assignment $TRUSTED_i \leftarrow sm_i^{r'}$ executed during each round $r' \geq r$, this immediately translates as “there are q disjoint sets X_1, \dots, X_q of cumulatively x processes, q processes $p_{\ell_1} \in X_1, \dots, p_{\ell_q} \in X_q$ and a time τ after which, for each $1 \leq j \leq q, p_{\ell_j}$ is not suspected by the processes of X_j ”. $\square_{Theorem 13}$

9.6 Lower Bound

To prove the lower bound the following strategy is used. Given $k < n - x + q$, let us assume that there is an algorithm \mathcal{A} that solves wait-free solves the k -set agreement problem in the basic read/write model equipped with a failure detector of the class $\diamond\mathcal{S}_{x,q}$. From the Theorems 12 and 13, the conditions required by the Theorem 10 hold. We can consequently conclude that there is an algorithm \mathcal{B} that solves k -set agreement in

the $IRIS(PR_{\diamond\mathcal{S}_{x,q}})$ model. Then, analyzing a class of admissible runs in $IRIS(PR_{\diamond\mathcal{S}_{x,q}})$ model, it is possible to derive (from the algorithm \mathcal{B}) a solution to the k -set agreement problem for $(k <) n - x + q$ processes in the IIS model, which is known to be impossible ([8, 9, 22, 33]).

Theorem 14 *There is no algorithm that wait-free solves k -set agreement for n processes in the read/write model equipped with a failure detector of the class $\diamond\mathcal{S}_{x,q}$, for $k < n - x + q$.*

Proof From the previous discussion, there is an algorithm \mathcal{B} that solves the k -set agreement task in the $IRIS(PR_{\diamond\mathcal{S}_{x,q}})$ model. We restrict our attention to a particular class of executions E defined iterated models. Let us partition the set of processes in two sets: the low-order processes $L = \{p_1, \dots, p_{n-x+q}\}$ and the high-order processes $H = \{p_{n-x+q+1}, \dots, p_n\}$. E is a subset of all (infinite) executions admissible in the IIS model. Moreover, in an execution $e \in E$, there is at least one low-order process that is correct and, at each round, low-order processes that have not yet crashed are scheduled before any high-order process. In other words, a low order process p_i never observes a high order process in its view sm_i . More formally, an iterated execution e belongs to E iff the two following conditions hold:

- $\exists p_i \in L : \forall r : sm_i^r \neq \emptyset$.
- $\forall r, \forall p_i \in L, \forall p_j \in H : (sm_i^r \neq \emptyset \wedge sm_j^r \neq \emptyset) \Rightarrow sm_i^r \subsetneq sm_j^r$.

Let us observe (*observation O1*) that all wait-free runs in which only a subset of low-ordered processes participate are included in E . We next show that all executions that belong to E are admissible in the $IRISPR_{\diamond\mathcal{S}_{x,q}}$ model (*observation O2*).

Let $e \in E$. There is a low-order process p_α that takes infinitely many steps in e . W.l.o.g., let us assume that $p_\alpha = p_q$ (as $n - x \geq 0, n - x + q \geq q$, i.e., p_q is a low-order process). Consider the following q sets of processes: $X_1 = \{p_1\}, \dots, X_{q-1} = \{p_{q-1}\}$ and $X_q = \{p_q\} \cup H$. These sets are disjoint and $|\bigcup_{1 \leq i \leq q} X_i| = q - 1 + 1 + |H| = q + x - q = x$. Define $\ell_1 = 1, \ell_2 = 2, \dots, \ell_q = q$. Finally, observe that $\forall r, \forall j \in X_q - \{p_q\} = H, sm_{\ell_q}^r \subsetneq sm_j^r$. The later follows from the fact that the low-order process p_q is always set linearized before any high-order process. To summarize, $\forall r, \forall j, 1 \leq j \leq q, \forall i \in X_j - \{\ell_j\} : sm_i^r = \emptyset \vee sm_{\ell_j}^r \subsetneq sm_i^r$, from which we conclude that the property $PR_{\diamond\mathcal{S}_{x,q}}$ is satisfied in the execution e .

It follows from *O1* that in all the wait-free runs in which only low-ordered processes participate are included in E . Moreover, *O2* establishes that algorithm \mathcal{B} is a wait-free solution to k -set agreement in E . Consequently, \mathcal{B} is solution to k -set agreement in the IIS model for $n - (x - q)$ processes. This would imply a wait-free solution for $n - (x - q) > k$ processes to the k -set agreement problem in the read/write model [9], which is known to be impossible [8, 22, 33]. $\square_{Theorem 14}$

Wait-free algorithms for solving k -set agreement for n processes in a message-passing system equipped with a failure detector of the class $\mathcal{S}_{x,q}$, such that $q \leq k \wedge n - x + q \leq k$, are given in [20, 29]. Such algorithms can easily be translated in the read/write model equipped with a failure detector of the class $\mathcal{S}_{x,q}$. Then, using the techniques developed in [34], these algorithms can be transformed to obtain solutions in the read/write model equipped with $\diamond\mathcal{S}_{x,q}$. We consequently obtain the following corollary.

Corollary 1 *Let $q \leq k$. There is a wait-free algorithm for solving k -set agreement among n processes in the read/write model equipped with a failure detector of the class $\diamond\mathcal{S}_{x,q}$ iff $n - x + q \leq k$.*

10 Conclusion

This paper has shown that failure detectors are schedulers, the aim of which is to prevent some runs from occurring. To that end, the paper has investigated the Iterated Immediate Snapshot (IIS) model equipped with failure detectors. It has first shown that enriching such a model with a failure detector does not increase its computational power with respect to wait-free solvable tasks. Then, given a failure detector of a class C (where C is $\{\diamond\mathcal{S}_x\}_{1 \leq x \leq n}, \{\Omega^z\}_{1 \leq z \leq n}$, or $\{\diamond\psi^y\}_{1 \leq y \leq n}$), it has shown that the power of C can be added to the iterated model as soon as its base write-snapshot primitive satisfies an additional requirement, giving rise to the Iterated Restricted Immediate Snapshot model denoted $IRIS(PR_C)$. The paper has then shown that that model and the classical read/write model enriched with a failure detector of three class C have the same computational power for wait-free solvable tasks.

In addition to providing a better insight on the very nature of failure detectors, the approach followed in the paper allows designing novel impossibility proofs, entirely based on an algorithmic reasoning (reductions).

Acknowledgments

The authors would like to thank Alejandro Cornejo and Eli Gafni for discussions on distributed computing models and asynchronous computability.

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] Afek Y., Dolev D., Gafni E., Merritt M. and Shavit N., A Bounded First-In, First-Enabled Solution to the l-Exclusion Problem. *ACM Transactions on Programming Languages and Systems*, 16(3):939-953, 1994.
- [3] Afek Y., Gafni E., Rajsbaum S., Raynal M. and Travers C., Simultaneous consensus tasks: a tighter characterization of set consensus. *Proc. 8th Int'l Conference on Distributed Computing and Networking (ICDCN'06)*, Springer Verlag LNCS #4308, pp. 331-341, 2006.
- [4] Attiya H. and Rachman O., Atomic Snapshots in $O(n \log n)$ Operations. *SIAM Journal on Computing*, 27(2):319-340, 1998.
- [5] Attiya H. and Rajsbaum S., The Combinatorial Structure of Wait-Free Solvable Tasks, *SIAM Journal of Computing*, 31(4):1286-1313, 2002.
- [6] Attiya H. and Welch J., *Distributed Computing: Fundamentals, Simulations, and Advanced Topics*, Wiley, (2nd Edition) 2004.
- [7] 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.
- [8] Borowsky E. and Gafni E., Generalized FLP Impossibility Results for t -Resilient Asynchronous Computations. *Proc. 25th ACM Symposium on Theory of Computing (STOC'93)*, ACM Press, pp. 91-100, 1993.
- [9] Borowsky E. and Gafni E., A Simple Algorithmically Reasoned Characterization of Wait-free Computations. *Proc. 16th ACM Symp. on Principles of Distributed Computing (PODC'97)*, ACM Press, pp. 189-198, 1997.
- [10] Chandra T. and Toueg S., Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM*, 43(2):225-267, 1996.
- [11] Chandra T., Hadzilacos V. and Toueg S., The Weakest Failure Detector for Solving Consensus. *Journal of the ACM*, 43(4):685-722, 1996.
- [12] Chaudhuri S., More Choices Allow More Faults: Set Consensus Problems in Totally Asynchronous Systems. *Information and Computation*, 105:132-158, 1993.
- [13] Fischer M.J., Lynch N.A. and Paterson M.S., Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, 32(2):374-382, 1985.
- [14] Gafni E. and Lamport L., Disk Paxos. *Distributed Computing*, 16(1):1-20, 2003.
- [15] Gafni E., Merritt M. and Taubenfeld G., The concurrency hierarchy and algorithms for unbounded concurrency. *Proc. 20th ACM Symp. on Principles of Distributed Computing (PODC'00)*, ACM Press, pp. 161-169, 2001.
- [16] Gafni E., Rajsbaum S. and Herlihy M., Subconsensus Tasks: Renaming is Weaker than Set Agreement. *Proc. 20th Int'l Symposium on Distributed Computing (DISC'06)*, Springer-Verlag #4167, pp.329-338, 2006.
- [17] Guerraoui R. and Raynal M., The Alpha of indulgent consensus. *The Computer Journal*, 50(1):53-67, 2007.
- [18] Guerraoui R. and Schiper A., Gamma-accurate Failure Detectors. *Proc. 10th Int'l Workshop on Distributed Algorithms (WDAG'96)*, Springer Verlag LNCS #1151, pp. 269-286, 1996.
- [19] Herlihy M.P., Wait-Free Synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124-149, 1991.

- [20] Herlihy M.P. and Penso L. D., Tight Bounds for k -Set Agreement with Limited Scope Accuracy Failure Detectors. *Distributed Computing*, 18(2):157-166, 2005.
- [21] Herlihy M.P., Rajsbaum S., and Tuttle M., Unifying Synchronous and Asynchronous Message-Passing Models, *Proc. 17th ACM Symposium on Principles of Distributed Computing (PODC)*, ACM Press, pp. 133-142, 1998.
- [22] Herlihy M.P. and Shavit N., The Topological Structure of Asynchronous Computability. *Journal of the ACM*, 46(6):858-923,, 1999.
- [23] 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.
- [24] Lamport. L., On Interprocess Communication, Part II: Algorithms. *Distributed Computing*, 1(2):86-101, 1986.
- [25] Lynch, N. A., *Distributed Algorithms*, Morgan Kaufmann, 872 pages, 1997.
- [26] Mostefaoui A., Rajsbaum S., Raynal M. and Travers C., On the Computability Power and the Robustness of Set Agreement-oriented Failure Detector Classes. *Tech Report # 1819*, 31 pages, IRISA, Université de Rennes, France, October 2006. An extended abstract appeared as: Irreducibility and Additivity of Set Agreement-oriented Failure Detector Classes. *Proc. 25th ACM Symposium on Principles of Distributed Computing (PODC'06)*, ACM Press, pp. 153-162, 2006.
- [27] Mostefaoui A. and Raynal M., Unreliable Failure Detector with Limited Scope Accuracy and an Application to Consensus. *Proc. 19th Int'l Conference on Foundations of Software Technology and Theoretical Computer Science (FST&TCS'99)* Springer Verlag LNCS #1738, pp. 329-340, 1999.
- [28] Mostefaoui A. and Raynal M., Solving Consensus Using Chandra-Toueg's Unreliable Failure Detectors: a General Quorum-Based Approach. *Proc. 13th Int'l Symposium on Distributed Computing (DISC'99)*, Springer-Verlag LNCS 1693, pp. 49-63, 1999.
- [29] Mostefaoui A. and Raynal M., k -Set Agreement and Limited Accuracy Failure Detectors. *19th ACM Symposium on Principles of Distributed Computing (PODC'00)*, pp. 143-152, 2000.
- [30] Neiger G., Set Linearizability. *Brief Announcement, Proc. 13th ACM Symposium on Principles of Distributed Computing (PODC'94)*, ACM Press, pp. 396, 1994.
- [31] 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.
- [32] Raynal M. and Travers C., In search of the holy grail: looking for the weakest failure detector for wait-free set agreement. *Proc. 10th Int'l Conference On Principles Of Distributed Systems (OPODIS'06)*, Springer Verlag LNCS #4305, pp. 1-17, 2006.
- [33] Saks M. and Zaharoglou F., Wait-Free k -Set Agreement is Impossible: The Topology of Public Knowledge. *SIAM Journal on Computing*, 29(5):1449-1483, 2000.
- [34] Yang J., Neiger G. and Gafni E., Structured Derivations of Consensus Algorithms for Failure Detectors. *Proc. 17th ACM Symposium on Principles of Distributed Computing (PODC'98)*, pp.297-308, 1998.

A A z -set agreement algorithm for the $IRIS(PR_{\Omega^z})$ model

This section presents a simple algorithm that solves the z -set agreement problem in $IRIS(PR_{\Omega^z})$. This algorithm, made up of two parts (the sub-algorithms denoted z -adopt() and z -converge() in Figure 12), combines ideas from [3, 28, 32, 34]. The first part addresses safety (at most k values are decided), while the second part guarantees eventual decision.

The algorithm proceeds as follows. The underlying immediate snapshot objects are partitioned into subsets of $(z + 1)$ consecutive objects. In the first part (z -adopt() algorithm, Figure 13), a process p_i invokes sequentially the operation `restricted_w_snapshot()` on a partition on the shared immediate snapshot objects $IS[a], \dots, IS[a + (z + 1)]$, each invocation writing what it has obtained in the previous invocations (of the same partition). Based on its partial view of immediate snapshots returned from $IS[a], \dots, IS[a + z]$, p_i tries

to identify a smallest snapshot returned from one of the immediate snapshot objects $IS[a], \dots$ or $IS[a+z]$. If it succeeds in finding such a smallest snapshot, it keeps it in $smin$.

Let us observe that, as at each immediate snapshot object $IS[\ell]$, $a \leq \ell \leq (a+z)$, the smallest immediate snapshot is uniquely defined, at most z distinct values can be adopted by the processes. As we will see in the proof, if the property $PR(\Omega^z)$ is satisfied during the rounds a from $a+(z+1)$, each process can identify a smallest snapshot. It follows that, when embedding the previous algorithm $IRIS(PR_{\Omega^z})$, eventually at most z different values remain in the system. As the processes do not know when this happens, it is necessary to enrich the algorithm to allow them to decide (this constitutes the second part of the algorithm). This can be obtained using a z -converge algorithm as proposed in [28, 34]. A translation of that algorithm, suited to the $IRIS(PR_C)$, is described in figure 14.

```

init  $r_i \leftarrow 1; est_i \leftarrow v_i; dec_i \leftarrow \perp$ 

loop forever
  (1)  $est_i \leftarrow z\text{-adopt}(est_i, r_i);$ 
  (2)  $r_i \leftarrow r_i + (z + 1);$  %  $z\text{-adopt}$  uses  $z + 1$  consecutive  $IS$  objects %
  (3)  $\langle est_i, commit_i \rangle \leftarrow z\text{-converge}(est_i, r_i);$ 
  (4)  $r_i \leftarrow r_i + 2;$  %  $z\text{-converge}$  uses 2 consecutive  $IS$  objects %
  (5) if  $(commit_i) \wedge (dec_i = \perp)$  then  $dec_i \leftarrow est_i$  end if
end loop.

```

Figure 12: A z -set agreement algorithm for the $IRIS(PR_{\Omega^z})$ model (code for p_i)

The z -adopt() algorithm Each invocation of z -adopt() has an input value, and returns an output value. Moreover, if processes execute an infinite sequence of invocations, there is an invocation number after which at most z distinct values are returned by the invocations [17, 32]. More precisely, z -adopt() satisfy the following properties:

- **Termination.** Any invocation of z -adopt() by a correct process returns.
- **Validity.** If a process gets back v from an invocation z -adopt, then there is a process that invokes z -adopt() with value v .
- **Eventual z -adoption.** Let $(inv1, inv2, \dots)$ be an infinite sequence of z -adopt() invocations. There exists an invocation sequence number s such that all the invocations whose sequence number is $s' \geq s$, return at most z distinct values. ⁽⁹⁾.

```

function  $z\text{-adopt}(v_i)$ 
  (1)  $h_i[1 : z] \leftarrow [\emptyset, \dots, \emptyset]; est_i \leftarrow v_i;$ 
  (2) for  $r_i = 1$  to  $z + 1$  do
  (3)    $s_i \leftarrow IS[r_i].restricted\_w\_snapshot(\langle i, v_i, h_i \rangle);$ 
  (4)   for  $m$  from 1 to  $r_i - 1$  do  $h_i[m] \leftarrow \bigcup_{j: \langle j, v_j, h_j \rangle \in s_i} h_j[m]$  end for;
  (5)    $h_i[r_i] \leftarrow \{ \langle i, \{ \langle j, v_j \rangle : \langle j, v_j, h_j \rangle \in s_i \} \}$ 
  (6) end for;
  (7) for  $m$  from 1 to  $z$  do
  (8)   if  $(\exists \langle -, smin \rangle \in h_i[m] : \forall \langle j, \_ \rangle \in smin : \langle j, smin \rangle \in h_i[m])$ 
  (9)     then  $est_i \leftarrow \min(v_j : \langle \_, v_j \rangle \in smin)$  end if
  (10) end for
  (11) return $(est_i)$ 

```

Figure 13: z -adopt algorithm (code for p_i)

⁹Another way to state this property is as follows: if the underlying iterated model satisfies PR_{Ω^z} , then at most z distinct values are adopted.

Proof of the algorithm Validity and Termination directly follow from the algorithm text. To prove eventual z -adoption, we show that when the property $PR(\Omega^z)$ is established, a process is able to identify a smallest snapshot in $z + 1$ rounds of `restricted_w_snapshot`. We say that an invocation of z -adopt starts at round r if this invocation uses objects $IS[r], \dots, IS[r+z]$. Let R be a round such that, after R , the property $PR(\Omega^z)$ is satisfied.

Lemma 9 *The set of values returned by invocations that start at a round $\geq R$ is of cardinality at most z .*

Proof We claim (claim C) that each process is able to identify a smallest snapshot while executing the **for** loop at lines 7-10. Assuming the claim, each process adopts a value taken from a smallest snapshot returned at round $r, r+1, \dots$ or $r+z-1$. Moreover, given a smallest snapshot, at most one value can be adopted from it. Consequently at most z distinct values are returned.

Let us denote $smin^\rho$ the smallest snapshot returned by invocations of `restricted_w_snapshot()` on object $IS[\rho]$. To facilitate the exposition, $smin_i$ is a fictitious local variable of p_i initially set to the value \emptyset . If the predicate of line 8 is evaluated to true by p_i for round ρ , $smin_i$ is set to $smin^\rho$. The claim statement considers an execution of z -adopt() that starts at round r .

Claim C: Let p_i be a process that terminates the execution of the **for** loop (lines 7-10). $\exists \rho : r \leq \rho \leq r+z-1$ such that $smin_i = smin^\rho$.

Proof of the Claim C. We prove the claim by induction. Let $RA_2(k)$ be the following property:

$$\left| \bigcup_{r+z-k \leq \rho \leq r+z} smin^\rho \right| \leq k \Rightarrow \exists \rho, r+z-k \leq \rho < r+z : smin_i = smin^\rho.$$

- $RA_2(1)$. In that case, $\exists j$ such that $smin^{r+z-1} = smin^{r+z} = \{ \langle j, _, _ \rangle \}$. At the end of round $r+z-1$, $h_j[r+z-1]$ contains the immediate snapshot obtained by p_j at that round (line 5). Moreover, due to the inclusion property of immediate snapshots, this array is seen by p_i at round $m+1$, from which we conclude that p_i identifies $s_j^{r+z-1} = smin^{r+z-1}$ as the smallest snapshot returned at round $r-z+1$ (lines 8-9).
- $RA_2(k) \Rightarrow RA_2(k+1)$. We consider two cases:
 - $\forall j \in smin^{r+z-(k+1)} : \exists \rho, r+z-k \leq \rho \leq r+z : j \in smin^\rho$. Let $j \in smin^{r+z-(k+1)}$. There exists a round ρ such that $j \in smin^\rho$. At the end of round $\rho-1$, h_j contains the immediate snapshots obtained by p_j at rounds $r+z-(k+1), r+z-k, \dots, \rho-1$. In particular, $h_j[z-k]$ contains $s_j^{r+z-(k+1)} = smin^{r+z-(k+1)}$. Moreover, due to the containment property of immediate snapshots, p_i sees h_j at round ρ , and consequently “learns” that p_j belongs to the smallest snapshot of round $r+z-(k+1)$. Since this is true for all $j \in smin^m$, $RA_2(k+1)$ follows.
 - $\exists j \in smin^{r+z-(k+1)} : \forall \rho, r+z-k \leq \rho \leq r+z : j \notin smin^\rho$. In that case, we have $|\bigcup_{r+z-k \leq \rho \leq r+z} smin^\rho \cup \{j\}| = |\bigcup_{r+z-k \leq \rho \leq r+z} smin^\rho| + 1 \leq k+1$, i.e., $|\bigcup_{r+z-k \leq \rho \leq r+z} smin^\rho \cup \{j\}| \leq k$. Consequently, we can apply $RA_2(k)$ and $RA_2(k+1)$ follows.

Finally, let us observe that the property $PR(\Omega^z)$ can be restated as follows:

- $PR(\Omega^z) \equiv \exists L : |L| \leq z$ and $\exists r : \forall r' \geq r : (smin^{r'} \subseteq L) \wedge \exists i : sm_i^{r'} \neq \emptyset$, where $smin^{r'}$ is the smallest snapshot returned by the invocation $IS[r'].restricted_w_snapshot()$.

As we assume that during the rounds ρ considered, $PR(\Omega^z)$ is satisfied it follows that $|\bigcup_{r \leq \rho \leq r+z} smin^\rho| \leq z$. Applying $RA_2(z)$ allows to conclude the proof. *End of the Proof of Claim C* \square *Lemma 9*

The z -converge() algorithm Processes invoke z -converge() with a value as input. It returns a pair $\langle c, v \rangle$ where c is a boolean and v a value. Following [34], we say that a process *picks* a value v if its invocation returns $\langle c, v \rangle$. Moreover, if $c = true$, the process *commits* to the value v . Invocations of z -converge() satisfy the following properties [17, 28, 32, 34]:

```

function:  $z$ -converge( $v_i$ )
(1)   $est_i \leftarrow v_i, ok_i \leftarrow false$ ;
(2)   $s_i \leftarrow IS[r_i].restricted\_w\_snapshot(< i, est_i >)$ ;
(3)  if  $\{ \{ est_j : < j, est_j > \in s_i \} \} \leq k$  then  $ok_i \leftarrow true$  end if;
(4)   $r_i \leftarrow r_i + 1$ ;
(5)   $t_i \leftarrow IS[r_i].restricted\_w\_snapshot(< i, est_i, ok_i >)$ ;
(6)  case  $\forall j : < j, est_j, ok_j > \in t_i : ok_j$  then  $return(< est_i, true >)$ 
(7)     $\exists j : < j, est_j, ok_j > \in t_i : ok_j$  then  $return(< est_j, false >)$ 
(8)     $\forall j : < j, est_j, ok_j > \in t_i : \neg ok_j$  then  $return(< est_i, false >)$ 
(9)  end case

```

Figure 14: z -converge algorithm (code for p_i)

- **Termination.** Any invocation by a correct process returns a pair $\langle c, v \rangle$.
- **Validity.** If a process picks v , then some process invokes z -converge with value v .
- **Convergent z -agreement.** If some process commits to v then at most z distinct values are picked.
- **z -Convergence.** If all process invoke z -converge() with values from a set of size at most z , then each process that returns commits to a value.

The algorithm described in figure 14 translates the original z -converge() algorithm into the iterated model.

B From $IRIS(PR_{\Omega^z})$ to a failure detector of the class Ω^z

This section considers the case where $1 \leq z \leq n$. The algorithm described in Figure 15 provides each process p_i with a local variable $LEADERS_i$ containing set of z process identities that eventually includes at least one correct process. The transformation extends the ideas of the “wheel” algorithms (introduced in [26]) to the context of $IRIS(PR_{\Omega^z})$. It uses a sequence \mathcal{L} that contains all the possible sets of size z generated from the n processes composing the system. \mathcal{L} is known by all the processes. Let nb_L be the length of this sequence. The elements of \mathcal{L} are indexed between 0 and $nb_L - 1$ and $\mathcal{L}[k]$ denotes its k -th element.

Local variables The processes scan in the same order the infinite sequence of sets $\mathcal{L}[0], \mathcal{L}[1], \dots, \mathcal{L}[nb_L], \mathcal{L}[0], \dots$. In order to eventually converge towards the same set, each process p_i manages the following local variables:

- The infinite sequence $\mathcal{L}[0], \mathcal{L}[1], \dots, \mathcal{L}[nb_L], \mathcal{L}[0], \dots$ can be seen as a ring around which processes are turning. The successive positions of p_i along the ring are represented by a non-decreasing sequence of integers. When p_i is at position α , the set $\mathcal{L}[\alpha \bmod nb_L]$ is considered by p_i to be the common leader set. Each process maintains a vector of integers, denoted $pos_i[1 : n]$, that represents its current estimate of the position of processes on the ring. $pos_i[i]$ is the current position of p_i .
- can_move_i is a boolean value set to true when the current position of p_i is known by all other participating processes. The process p_i is allowed to progress along the ring only when all the processes know its current position.
- The local variable smi_n is intended to contain a smallest immediate snapshot. When p_i is able to identify such a smallest snapshot (by examining its view of immediate snapshots returned during the last z rounds), the n -vector of integers $pmin_i$ contain the maximum component-wise (denoted \max_{cw}) of all the estimates pos_j that appear in smi_n (line 9); ℓ_i is then set to the maximum value that appears in $pmin_i$ (line 10).
- $view_{(r),i}$ is a sliding “window” array that contains the knowledge of p_i concerning the immediate snapshots returned during the last z previous rounds. For $1 \leq k \leq z$, $view_{(r),i}[k]$ is a set of pairs $\langle j, s_j \rangle$ where s_j is the immediate snapshot obtained by p_j at round $r - k$. Each s_j is in turn such that $s = \{ \langle j_1, pos_{j_1} \rangle, \langle j_2, pos_{j_2} \rangle, \dots \}$. During each round r , p_i improves its “knowledge” of the values s_j returned to the processes p_j from $IS[\rho]$ for each $r - z \leq \rho \leq r - 1$ (the values of $view_i[\rho]$ are computed at line 5).

```

Initially:  $r_i \leftarrow 0$ ;  $can\_move_i \leftarrow true$ ;  $pos_i[1 : n] \leftarrow [0, \dots, 0]$ ;  $\ell_i \leftarrow 0$ ;
 $view_{(1),i}[1 : z] \leftarrow [\emptyset, \dots, \emptyset]$ ;
for  $m$  from  $z$  to 1 do for each  $j \in \{1, \dots, n\}$  do
     $view_{(1),i}[m] \leftarrow view_i[m] \cup \{ \langle i, \{ \langle j, [0, \dots, 0] \rangle : j \in \{1, \dots, n\} \} \}$ 
end for end for

(1) repeat forever
(2)  $r_i \leftarrow r_i + 1$ ;  $s_i \leftarrow IS[r_i].restricted\_w\_snapshot(\langle i, pos_i, view_i \rangle)$ ;
(3)  $pos_i \leftarrow \max_{cw} \{ pos_j : \langle j, pos_j, - \rangle \in s_i \}$ ;  $smi_n \leftarrow \emptyset$ ;
(4) for  $m$  from  $z$  to 1 do
(5)  $view_i[m] \leftarrow \bigcup_{j: \langle j, -, h_j \rangle \in s_i} h_j(r_i, m)$ ;
(6) if  $(\exists \langle -, s \rangle : \forall \langle j, - \rangle \in s : \langle j, s \rangle \in view_i[m])$ 
(7) then  $smi_n \leftarrow s$  end if
(8) end for;
(9) if  $(smi_n \neq \emptyset)$  then  $pmin_i \leftarrow \max_{cw} \{ pos_j : \langle j, pos_j \rangle \in smi_n \}$ ;
(10)  $\ell_i \leftarrow \max \{ pmin_i[j] : j \in \{1, \dots, n\} \}$ ;
(11)  $can\_move_i \leftarrow (pmin_i[i] = pos_i[i])$ 
(12) end if;
(13)  $LEADERS_i \leftarrow \mathcal{L}[\ell_i \bmod nb_L]$ ;
(14) if  $can\_move_i \wedge (\mathcal{L}[\ell_i \bmod nb_L] \cap \{ j : \langle j, -, - \rangle \in s_i \} = \emptyset)$ 
(15) then  $pos_i[i] \leftarrow \max(\ell_i, pos_i[i]) + 1$ ;  $can\_move_i \leftarrow false$  end if;
(16)  $view_i \leftarrow shift(view_i)$ ;  $view_i[1] \leftarrow \{ \langle i, \{ \langle j, pos_j \rangle : \langle j, pos_j, - \rangle \in s_i \} \rangle \}$ 
(17) end repeat

```

Figure 15: From $IRIS(PR_{\Omega^z})$ to Ω^z (code for p_i)

Such a window array supports a `shift()` operation that moves the window one step ahead. When $view_{(r+1),i} \leftarrow shift(view_{(r),i})$ is executed, the information on round $r - z$ is lost and $view_{(r+1),i}[1]$ is set to \emptyset (there is no information on round $(r + 1) - 1$ in $view_{(r),i}$). At the end of a round r , preparing for the next round, p_i shifts $view_i$ and stores its r -th immediate snapshot in $view_i[1]$ (line 16). In this way, at the beginning of round $r + 1$, $view_i$ contains all the knowledge that p_i has aggregated on immediate snapshots returned at round $r, r - 1, \dots, r - (z - 1)$.

Rounds $-(z + 1), \dots, 0$ are fictitious rounds such that each process observes the n processes in its immediate snapshots. (the fictitious objects $IS[-(z + 1)], \dots, IS[0]$ return at each process the same immediate snapshot in which all the processes appear). $view_{(1),i}$ is initialized accordingly.

Processes behavior The behavior of a process p_i is described in Figure 15. At each round r , a process p_i writes in the corresponding one-shot immediate snapshot object $IS[r]$ (line 2) the immediate snapshots that, according to its current knowledge, have been returned in the z previous round (these immediate snapshots are kept in $view_i$) and its current estimate of processes positions pos_i . Then, p_i updates its estimate pos_i of the positions of the other processes, taking the component-wise maximum of the vectors it observes in s_i (line 3).

The **for** loop (lines 4-8) has two aims. First, p_i improves its knowledge concerning the immediate snapshots returned from the objects $IS[r_i - z], IS[r_i - (z - 1)], \dots, IS[r_i - 1]$ (line 5). Second, p_i tries to identify a smallest immediate snapshot smi_n for one of the objects $IS[r_i - z], \dots, IS[r_i - 1]$ (lines 6-7). A pair (s, m) that satisfies the predicate of line 6 is such that s is the smallest snapshot for the object $IS[r_i - m]$. The proof shows that, due to the property PR_{Ω^z} , examining the last z rounds, p_i is eventually always able to determine a smallest snapshot.

If p_i has determined a smallest snapshot smi_n , it computes the component-wise maximum $pmin_i$ of the estimates vectors pos_j that appear in it. Such a vector is “common knowledge”: as the estimate pos are component-wise maximum, any pos vector computed at round r or later is equal or greater to it. Then, if $pos_i[i] = pmin_i[i] = \alpha$, p_i discovers that the processes will know that it has reached position α in the ring. p_i is consequently allowed to move ahead (line 11). Finally, p_i computes the greatest position ℓ_i that

appears in $pmin_i$ (line 10) and the new value of the local failure detector output is the corresponding set in the sequence \mathcal{L} (line 13).

Then, if p_i is allowed to move, it checks if there is a process (1) that belongs to the set corresponding to position ℓ_i in \mathcal{L} , and (2) appears in its immediate snapshot s_i (line 14). When this predicate is not satisfied, p_i moves to position $\ell_i + 1$ and updates $pos_i[i]$ accordingly (line 15).

Correctness proof The proof uses the following notations:

- $smin^r$: the smallest immediate snapshot returned at object $IS[r]$.
- $pmin^r = \max_{\text{cw}}\{pos : < -, pos, - > \in smin^r\}$.
- $\ell^r = \max\{pmin^r[i] : i \in \{1, \dots, n\}\}$.
- x_i^r denotes the value of p_i 's local variable at the end of round r .

The next lemmas show that the sequence (ℓ^r) is non decreasing (Lemma 11) and converges towards a value denoted ℓ_M (Lemma 12 and Lemma 13). Then, as a direct corollary of Lemma 13, the sets LEADERS_i of the correct processes p_i remain forever equal to the same set $L = \mathcal{L}[\ell_M \bmod nb_L]$. Finally, Theorem 15 establishes the correctness of the algorithm. Lemma 10 is a “technical” lemma used several times in the proof.

Lemma 10 *When p_i executes line 15, $\max(pos_i[i], \ell_i) = \ell_i$.*

Proof let us assume that at round $r + 1$, p_i sets the value of $pos_i[i]$ to $\ell_i + 1$ and let α be the previous value of $pos_i[i]$ (line 15). This means that the predicate of line 14 is satisfied. In particular, $can_move_i = true$. Let $r' \leq r$ be the greatest round at which can_move_i is modified. (At round r' , p_i executes $can_move_i \leftarrow true$ and, $\forall r' \leq \rho \leq r$, $can_move_i^\rho = true$). This implies that, at rounds r', \dots, r , p_i does not modify $pos_i[i]$ at line 15. In particular, $pos_i^{r'}[i] = \alpha$ and the last round at which ℓ_i is modified is r' . I.e., at round r' , p_i identifies a smallest snapshot $smin_i$ such that $pmin_i[i] = pos_i^{r'}[i] = \alpha$ (lines 9-12). Moreover, $\ell_i = \max\{pmin_i[j]\} \geq pmin_i[i] = \alpha$. Consequently, at round r , $\ell_i \geq \alpha$. $\square_{\text{Lemma 10}}$

Lemma 11 $\forall r \geq 0 : \ell^{r+1} = \ell^r \vee \ell^{r+1} = \ell^r + 1$.

Proof We show the lemma by induction on round numbers r .

- *Base case:* we show that $\ell^0 = \ell^1 = 0$. During the fictitious round 0, all pos_i vectors are equal to $[0, \dots, 0]$. It then follows that $\ell^0 = 0$. At round 1, each process p_i writes $pos_i^0 = [0, \dots, 0]$. Consequently the smallest immediate snapshots of round 1 contains only 0 vectors, from which we obtain that $\ell^1 = 0 = \ell^0$.
- *Induction case:* Let us assume that the lemma is true up to round r . We establish that (a) $\ell^{r+1} \geq \ell^r$ and, (b) $\ell^r + 1 \geq \ell^{r+1}$, which prove the case.
 - *Proof of property (a):* Let p_i be a process that appears in the smallest immediate snapshot of round $r + 1$. At round $r + 1$, p_i writes its estimate pos_i^r it has computed at round r . At round r , pos_i is the component-wise maximum of the estimate vectors observed by p_i in its immediate snapshot s_i . Due to the containment property of immediate snapshot and line 15, we have $pos_i^r \geq pmin^r$. Consequently, $\max_{\text{cw}}\{< i, pos, - > : i \in smin^{r+1}\} \geq pmin^r$ from which we obtain that $\ell^{r+1} \geq \ell^r$.
 - *Proof of property (b):* Assume for contradiction that $\alpha = \ell^{r+1} > \ell^r + 1$. During round $r + 1$, a process p_i writes the value pos_i^r of its array pos_i computed during round r . Due to the definition of ℓ^{r+1} , there must exist a pair (j, k) such that $pos_j^r[k] = \alpha$ and p_j appears in the smallest snapshot for object $IS[r + 1]$. This can only happens if at some round $r' \leq r$, p_k executes $pos_k[k] \leftarrow \alpha$ (line 15), where $\alpha = 1 + \ell_k^{r''}$ (Lemma 10; $\ell_k^{r''}$ is computed at some round $r'' \leq r'$ at line 10). Finally, let us observe that $\exists r''' \leq r''$ such that $\ell_k^{r''} = \ell_j^{r'''}$ (line 10).

Thus, we have $\ell^{r+1} = \alpha = \ell_j^{r''} + 1 > \ell^r + 1$, from which we conclude that $(\ell^{r''} > \ell^r) \wedge (r''' < r)$: a contradiction with the assumption that the lemma is satisfied up to round r .

$\square_{\text{Lemma 11}}$

Lemma 12 $\exists R_\ell, \ell_M : \forall r \geq R_\ell : \ell^r = \ell_M$.

Proof Let us assume for contradiction that the lemma is not true. The property PR_{Ω^z} states that there exists a round r_1 and a set $L = \mathcal{L}[x]$ (x being the position of L in the sequence \mathcal{L} .) such that, $\forall r \geq r_1, \forall s$ immediate snapshot returned at round r : $L \cap \{< j, -, - > \in s\} \neq \emptyset$. It follows from the initial assumption and lemma 11 that the set $S = \{r : \ell^r \bmod nb_L = x\}$ is infinite. Let us consider a round $r_2 > r_1$ such that (1) $\ell^{r_2} = \alpha$, (2) $\ell^{r_2-1} = \alpha - 1$ and, (3) $\alpha \bmod nb_L = x$. As, at each round the value of ℓ is unchanged or is increased by one unit (Lemma 11), it follows from the fact that the set S is infinite that such a round r_2 exists.

We show that, after round r_2 , the value ℓ never change: a contradiction. We proceed by induction. Let $RA_\beta(r)$ be the following property: $RA_\beta(r) : \ell^r = \alpha$.

- $RA_\beta(r_2)$. By definition of round r_2 , $\ell^{r_2} = \alpha$.
- $RA_\beta(r_2 + 1)$. We claim that $\forall r \geq 1, i, j, y : (\ell^r \leq y) \Rightarrow pos_i[j]^{r+1} \leq y + 1$ (*Claim C1*). Let p_i be a process that appears in the smallest immediate snapshot of round $r_2 + 1$. By definition, $\ell^{r_2-1} = \alpha - 1$. It then follows from claim *C1* that, at the end of round r , $\forall j : pos_i^{r_2}[j] \leq \alpha$. At round $r_2 + 1$, p_i writes the value of its estimate pos_i it has computed in the previous round. As ℓ^{r_2+1} is the maximal entry among all estimate pos_i that appears in smi^{r_2+1} , we have $\ell^{r_2+1} \leq \alpha$. As the sequence (ℓ^r) is non-decreasing (Lemma 11), $\ell^{r_2+1} = \alpha$.
- Let $r \geq r_2 + 1$. ($\forall r', r_2 \leq r' \leq r : RA_\beta(r') \Rightarrow RA_\beta(r' + 1)$). We follow the same line of reasoning. Let p_i be a process that appears in the smallest immediate snapshot smi^{r+1} at round $r + 1$. We show that the value of p_i 's estimate pos_i^r written at round $r + 1$ is such that $\forall j : pos_i^r[j] \leq \alpha$. It then follows that $\ell^{r+1} \leq \alpha$. As the sequence (ℓ^r) is non-decreasing, it follows that $\ell^{r+1} = \alpha$.

Let us assume for contradiction that there exists an entry j such that $pos_i^r[j] = k > \alpha$. Let us observe that this can happens only if there is a round $r' \leq r$ during which p_j executes $pos_j[j] \leftarrow k$ (line 15). Moreover, as $\forall \rho \leq r_2 - 1 : \ell^\rho \leq \alpha - 1$ (case assumption), we obtain by applying claim *C1* that $r_2 < r'$. As $r_2 < r' \leq r$, $RA_\beta(r')$ holds, i.e., $\ell^{r'-1} = \alpha$, from which we obtain $k = \alpha + 1$ (applying again *C1* at round number $r' - 1$).

To summarize, at round r' , $r_2 < r' \leq r$, p_j executes $pos_j[j] \leftarrow \ell_j + 1 = \alpha + 1$ (Lemma 10). From the protocol text, this implies that the predicate of line 14 is satisfied. In particular, at round r' no process that appears in the immediate snapshot of p_j is in $\mathcal{L}[\alpha \bmod nb_L] = \mathcal{L}[x]$. This is not possible since, after round $r_1 < r'$, the property $PR(\Omega^z)$ holds for the set $\mathcal{L}[x]$.

Claim C1: $\forall r \geq 1, i, j, y : (\ell^r \leq y) \Rightarrow pos_i^{r+1}[j] \leq y + 1$.

Proof of the Claim C1: Let α be the value of $pos_i^{r+1}[j]$. Then, there is a round $r' \leq r$ at which p_j executes $pos_j[j] \leftarrow \ell_j + 1 = \alpha$ at line 15 (Lemma 10). Let us observe that there is a round $r'' \leq r'$ such that $\ell_j = \ell^{r''}$ (line 10). Moreover, as $r'' \leq r$, it follows from Lemma 11 that $\ell^{r''} \leq \ell^r$. We obtain $pos_i^{r+1}[j] = \ell^{r''} + 1 \leq \ell^r + 1$.
end of the proof of the Claim C1. $\square_{\text{Lemma 12}}$

Lemma 13 $\exists R$ such that, for any correct process p_i , $\forall r \geq R : \ell_i^r = \ell_M$ (ℓ_M is introduced in Lemma 12).

Proof The proof of the lemma is based on the following claim (*Claim C2*): $|\bigcup_{r-z \leq \rho \leq r} smi^\rho| \leq z \Rightarrow smi_i^r \neq \emptyset$. The claim states that if all the processes that appear in the smallest snapshots at rounds $r - z, r - z + 1, \dots, r$ belong to a set L of size at most z , each process is able to determine one of these smallest snapshots at round r .

Let us observe that the property $PR(\Omega^z)$ can be restated as follows: $\exists L : |L| \leq z$ and $\exists \rho : \forall r \geq \rho : (smi^r \subseteq L)$. Consequently, there is a round R_1 such that at each round $r \geq R_1$, $\forall p_i : \ell_i^r = \ell^{r'}$ where $r - z \leq r' \leq r - 1$ (*Claim C2* and lines 9-10). Moreover, due to Lemma 12, there is a round R_2 such that $\forall r \geq R_2 : \ell^r = \ell_M$. We choose $R = \max(R_1, R_2) + z$ to complete the proof.

Claim C2: Let p_i be a process that executes round $r \geq z + 1$. $|\bigcup_{r-z \leq \rho \leq r} smi^\rho| \leq z \Rightarrow smi_i^r \neq \emptyset$. *Proof of the Claim C2* We prove the claim by induction. Let $RA_\beta(k)$ be the following property:

$$\left| \bigcup_{r-k \leq \rho \leq r} smi^\rho \right| \leq k \Rightarrow \exists \rho, r - k \leq \rho < r : smi_i = smi^\rho.$$

- $RA_4(1)$. In that case, $\exists j$ such that $smi^{r-1} = smi^r = \{< j, _, _ >\}$. At the end of round $r-1$, $view_j$ contains the immediate snapshot obtained by p_j at that round (at position 1, line 16). Moreover, due to the inclusion property of immediate snapshots, this array is seen by p_i at round r , from which we conclude that p_i identifies $s_j^{r-1} = smi^{r-1}$ as the smallest snapshot returned at round $r-1$ (lines 5-7).
- $RA_4(k) \Rightarrow RA_4(k+1)$. We consider two cases:
 - $\forall j \in smi^{r-(k+1)} : \exists \rho, r-k \leq \rho \leq r : j \in smi^\rho$. Let $j \in smi^{r-(k+1)}$. There exists a round ρ such that $j \in smi^\rho$. At the end of round $\rho-1$, $view_j$ contains the immediate snapshots obtained by p_j at rounds $r-(k+1), r+z-k, \dots, \rho-1$. In particular, $view_{(\rho),j}[\rho-(r-k)]$ contains $s_j^{r-(k+1)} = smi^{r-(k+1)}$. Moreover, due to the containment property of immediate snapshots, p_i sees $view_j$ at round ρ , and consequently “learns” that p_j belongs to the smallest snapshot of round $r-(k+1)$. Since this is true for all $j \in smi^{r-(k+1)}$, $RA_4(k+1)$ follows.
 - $\exists j \in smi^{r-(k+1)} : \forall \rho, r-k \leq \rho \leq r : j \notin smi^\rho$. In that case, we have $|\bigcup_{r-k \leq \rho \leq r} smi^\rho \cup \{j\}| = |\bigcup_{r-k \leq \rho \leq r} smi^\rho| + 1 \leq k+1$, i.e., $|\bigcup_{r+z-k \leq \rho \leq r} smi^\rho \cup \{j\}| \leq k$. Consequently, we can apply $RA_4(k)$ and $RA_4(k+1)$ follows.

End of the Proof of Claim C2.

□_{Lemma 13}

Theorem 15 *The algorithm described in Figure 15 constructs a failure detector of the class Ω^z in IRIS(PR_{Ω^z}).*

Proof For any correct process p_i , it follows from Lemma 13 (and line 13) that there is a set L , namely, $L = \mathcal{L}[\ell_M \bmod nb_L]$, such that eventually $LEADERS_i = L$ remains true forever. It remains to show that L contains a correct process. Let us assume for contradiction that there is no correct process in L . Observe that there is a round r_1 after which no immediate snapshot that contains a process $\in L$ is returned. Let us consider a process p_i that appears infinitely often in smallest immediate snapshots. For such a process, there are infinitely many rounds such that $can_move_i = true$. Consequently, there is a round $r \geq r_1$ such that (1) $\ell_i = \ell_M$ and (2) $can_move_i = true$. As $r \geq r_1$, the predicate of line 14 is evaluated to false, after which we have $pos_i^r[i] = \ell_M + 1$. Since p_i appears infinitely often in smallest immediate snapshots, eventually it updates ℓ_i with a value $\geq pos_i^r[i] > \ell_M$: a contradiction with Lemma 13. □_{Theorem 15}