

# The Iterated Restricted Immediate Snapshot Model

Sergio Rajsbaum, Michel Raynal, Corentin Travers

► **To cite this version:**

Sergio Rajsbaum, Michel Raynal, Corentin Travers. The Iterated Restricted Immediate Snapshot Model. [Research Report] PI 1874, 2007, pp.22. <inria-00193683v3>

**HAL Id: inria-00193683**

**<https://hal.inria.fr/inria-00193683v3>**

Submitted on 13 Dec 2007

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

PUBLICATION  
INTERNE  
N° 1874

**THE ITERATED RESTRICTED IMMEDIATE SNAPSHOT MODEL**

S. RAJSBAUM   M. RAYNAL   C. TRAVERS



## The Iterated Restricted Immediate Snapshot Model

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

Systèmes communicants  
Projet ASAP

Publication interne n° 1874 — Décembre 2007 — 38 pages

**Abstract:** In the *Iterated Immediate Snapshot* model (*IIS*) the memory consists of a sequence of one-shot *Immediate Snapshot* (*IS*) objects. Each *IS* object can be accessed with an operation that atomically writes a value and returns a snapshot of its contents. Each process can access each *IS* object at most once. Processes access the sequence of *IS* objects, one-by-one, asynchronously, in a *wait-free* manner; any number of processes can crash. It has been shown by Borowsky and Gafni and others that this model is very useful to study the usual read/write shared memory model. Its interest lies in the elegant recursive structure of its runs, hence of the ease to analyze it round by round. In a very interesting way, Borowsky and Gafni have shown that the *IIS* model and the read/write model are equivalent for the wait-free solvability of decision tasks.

In this paper we extend the benefits of the *IIS* model to partially synchronous systems. Given a shared memory model enriched with a failure detector, what is an equivalent *IIS* model? The paper shows that an elegant way of capturing the power of a failure detector and other partially synchronous systems in the *IIS* model is by restricting appropriately its set of runs, giving rise to the *Iterated Restricted Immediate Snapshot* model (*IRIS*).

The benefit of the proposed approach is new results (including new proofs of existing results) when we consider the *IRIS* model instead of the equivalent read/write model enriched with a given failure detector directly. As a study case, the paper considers a system enriched with *limited-scope accuracy* failure detectors, where there is a cluster of processes such that eventually some correct process is eventually never suspected by any process in that cluster. The paper provides a new proof of the *k*-set agreement Herlihy and Peaso's lower bound for shared memory system augmented with a limited-scope accuracy failure detector. The proof is based on an extension of the Borowsky-Gafni *IIS* simulation to encompass failure detectors, followed by a very simple topological argumentation.

With the *IRIS* model we have succeeded in capturing the partial synchrony of a failure detector enriched system via a fully asynchronous, round by round system. We thus hope to have contributed to a better understanding of fault-tolerant distributed computing.

**Key-words:** Algorithmic reduction, Asynchronous system, Distributed algorithm, Distributed Computability, Failure detectors, Fault-tolerance, Round-based computation, Shared memory, Topology.

(Résumé : tsyp)

\* Instituto de Matemáticas, Universidad Nacional Autónoma de México, D. F. 04510, Mexico [rajsbaum@math.unam.mx](mailto:rajsbaum@math.unam.mx)

\*\* IRISA, Université de Rennes 1, Campus de Beaulieu, 35042 Rennes Cedex, France, [raynal@irisa.fr](mailto:raynal@irisa.fr)

\*\*\* IRISA, Université de Rennes 1, Campus de Beaulieu, 35042 Rennes Cedex, France, [travers@irisa.fr](mailto:travers@irisa.fr)



## **Le modèle de calcul IRIS**

**Résumé :** Ce rapport présente le modèle de calcul réparti IRIS. Ce modèle considère des calculs asynchrones, “sans-attente” en présence de fautes.

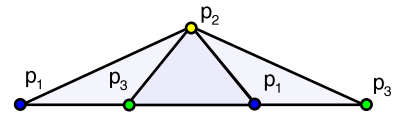
**Mots clés :** Système asynchrone, réduction algorithmique, algorithme distribué, calculabilité distribuée, détecteur de fautes, instantané atomique, crash de processus, modèle de calcul fondé sur les rondes, mémoire partagée, algorithme sans attente, topologie.

## 1 Introduction

A distributed model of computation consists of a set of  $n$  processes communicating through some medium (some form of message passing or shared memory), satisfying specific timing assumptions (process speeds and communication delays), and failure assumptions (their number and severity). A major obstacle in the development of a theory of distributed computing is the wide variety of models that can be defined – many of which represent real systems – with combinations of parameters in both the (a)synchrony and failure dimensions [4, 26, 27]. Thus, an important line of research is concerned with finding ways of unifying results, impossibility techniques, and algorithm design paradigms of different models.

An early approach towards this goal has been to derive direct simulations from one model to another; e.g., to show how to transform a protocol running in an asynchronous message passing model to one for a shared memory model [2], or from an asynchronous model to a synchronous model [3], or from a protocol tolerating some number of failures to one tolerating more failures [8]. A more recent approach has been to devise models of a higher level of abstraction, where results about various more specific models can be derived (e.g., [16, 23, 28]). Two main ideas are at the heart of the approach, which has been studied mainly for crash failures only, and is the topic of this paper.

**Two bedrocks: wait-freedom and round-based execution** It has been discovered [6, 24, 34] that the *wait-free* case is fundamental. In a system where any number of processes can crash, each process must complete the protocol in a finite number of its own steps, and “wait statements” to hear from another process are not useful. In a wait-free system it is easy to consider the *simplicial complex of global states* of the system after a finite number of steps, and various papers have analyzed topological invariants about the structure of such a complex, to derive impossibility results. Such invariants are based on the notion of *indistinguishability*, which has played a fundamental role in nearly every lower bound in distributed computing. Two global states are indistinguishable to a set of processes if they have the same local states in both. In the figure on the right, there is a complex with three triangles, each one is a *simplex* representing a global state; the corners of a simplex represent local states of processes in the global state. The center simplex and the rightmost simplex represent global states that are indistinguishable to  $p_1$  and  $p_2$ , which is why the two triangles share an edge. Only  $p_3$  can distinguish between the two global states. This complex is a manifold because each edge is contained in at most two triangles.



Results about  $t$ -resilient systems are derived by reduction to the wait-free case [8], or using bivalency arguments (e.g., [15, 28]) which do not seem to be generalizable from consensus to set agreement. The 1-resilient characterization of [9] is by reduction to the consensus impossibility of [15], and in general dealing with  $t$ -resilient executions is more difficult than the wait-free case; compare for example the wait-free consensus impossibility proof in [21] with the one of [15].

Most attempts at unifying models of various degrees of asynchrony restrict attention to a subset of well-behaved, *round-based* executions. The approach in [7] goes beyond that and defines an *iterated* round-based model (*IIS*), where each communication object can be accessed only once by each process. These objects, called *Immediate Snapshot* objects [5], are accessed by the processes with a single operation denoted `write_snap()`, that writes the value provided by the invoking process and returns to it a snapshot [1] of its content. An *IS* object can be accessed at most once by each process, and the sequence of *IS* objects are accessed asynchronously, and one after the other by each process. It is shown in [7] that the *IIS* model is equivalent (for bounded wait-free task solvability) to the usual read/write shared memory model.

Thus, the runs of the *IIS* model are not a subset of the runs of a standard (non-iterated) model as in other works, and the price that has to be paid is an ingenious simulation algorithm showing that the model is equivalent to a read/write shared memory model (w.r.t. wait-free task solvability). But the reward is a model that has an elegant recursive structure: the complex of global states after  $i$  rounds is a manifold, and the complex after  $i + 1$  rounds is obtained by replacing each simplex by a one round complex (see Figure 1). Indeed, the *IIS* model was the basis for the proof in [7] of the main characterization theorem of [24], and was instrumental for the results in [19].

**Context and goals of the paper** We introduce the *IRIS model*, which consists of a subset of runs of the *IIS* model of [7], to obtain the benefits of the round by round and wait-freedom approaches in one model, where processes run wait-free but the executions represent those of a partially synchronous model. As an application, we derive new, simple impossibility results for set agreement in several partially synchronous systems, as described in more detail below.

In the construction of a distributed computing theory, a central question has been understanding how the degree of synchrony of a system affects its power to solve distributed tasks. The degree of synchrony has been expressed in various ways, typically either by specifying a bound  $t$  on the number of processes that can crash, as bounds on delays and process steps [14], by a failure detector [10], or by using powerful shared memory objects [21]. It has been shown multiple times that systems with more synchrony can solve more tasks. Previous works in this direction have mainly considered an asynchronous system enriched with a failure detector that can solve consensus. Some works have identified this type of synchrony in terms of fairness properties [35]. Other works have considered round-based models with no failure detectors [16]. Some other works [25] focused on performance issues mainly about consensus. Also, in some cases, the least amount of synchrony required to solve some task has been identified, within some paradigm. A notable example is the weakest failure detector to solve consensus [11] or  $k$ -set agreement [37]. Set agreement [12] represents a desired coordination degree to be achieved in the system, requiring processes to agree on at most  $k$  different values (consensus is 1-set agreement), and hence is natural to use it as a measure for the *synchrony degree* in the system. The fundamental result of the area is that  $k$ -set agreement is not solvable in a wait-free, i.e. fully asynchronous system even for  $k = n - 1$  [6, 24, 34]. However, we are still lacking a clear view of what exactly “degree of synchrony” means. For example, the same power as far as solving  $k$ -set agreement can be achieved in various ways, such as via different failure detectors [29] or  $t$ -resilience assumptions. A second goal for introducing the *IRIS* model, is to have a mean of precisely representing the degree of synchrony of a system, and this is achieved with the *IRIS* model by considering particular subsets of runs of the *IIS* model.

**Capturing partial synchrony with a failure detector** As previously observed, a way of defining a partially synchronous system is with a *failure detector* [10], i.e., a distributed oracle that provides each process with hints on process failures. According to the type and the quality of the hints, several classes of failure detectors have been defined (e.g., [13, 18, 29, 32, 37]).

As an example, this paper focuses on the family of *limited scope* accuracy failure detectors, denoted  $\diamond\mathcal{S}_x$  [20, 30, 36]. These capture the idea that a process may detect failures reliably on the same local-area network, but less reliably over a wide-area network. They are a generalization of the class denoted  $\diamond\mathcal{S}$  that has been introduced in [10] ( $\diamond\mathcal{S}_n$  is  $\diamond\mathcal{S}$ ). Informally, a failure detector of the class  $\diamond\mathcal{S}_x$  is for a system made up of a single cluster of processes; it states that there is a correct process that is eventually never erroneously suspected by any process in that cluster. The technical report [33] describes the extension to  $q$  disjoint clusters and the circumstances under which  $k$ -set agreement can be solved in this model, which were proved first in [22].

**Results of the paper** The paper starts by describing the read/write computation model enriched with a failure detector  $C$  of the class  $\diamond\mathcal{S}_x$ , and the *IIS* model, in Section 2. Then, in Section 3, it describes an *IRIS* model that precisely captures the synchrony provided by the asynchronous system equipped with  $C$ . To show that the synchrony is indeed captured, the paper presents two simulations in Section 4. The first is a simulation from the shared memory model with  $C$  to the *IRIS* model. The second shows how to extract  $C$  from the *IRIS* model, and then simulate the read/write model with  $C$ . From a technical point of view, this is the most difficult part of the paper. We had to develop a generalization of the wait-free simulation described in [7] that preserved consistency with the simulated failure detector.

The simulations prove Theorem 1: an agreement task is wait-free solvable in the read/write model enriched with  $C$  if and only if it is wait-free solvable in the corresponding *IRIS* model. Then, using a simple topological observation, it is easy to derive the lower bound of [22] for solving  $k$ -set agreement in a system enriched with  $C$ . In the approach presented in this paper, the technically difficult proofs are encapsulated in algorithmic reductions between the shared memory model and the *IRIS* model, while in the proof of [22] combinatorial topology techniques introduced in [23] are used to derive the topological properties of the runs of the system enriched with  $C$  directly.

A companion technical report [33] extends the results presented here to other failure detector classes and in the full version, to  $t$ -resilient computability.

## 2 Computation model and failure detector class

This section presents a quick overview of the background needed for the rest of the paper, more detailed descriptions can be found elsewhere, e.g., [4, 7, 10, 27]. We describe here the two main models we are concerned with, in Section

2.1 the standard shared memory model with a failure detector, and in Section 2.2 the *IIS* model. In Section 2.3 we define tasks, and the known equivalence between these models.

## 2.1 Shared memory model enriched with a failure detector of the class $\diamond S_x$

We consider a standard asynchronous system made up of  $n$  processes,  $p_1, \dots, p_n$ , of which any of them can crash. A process is *correct in a run* if it takes an infinite number of steps. The shared memory is structured as an array  $SM[1..n]$  of atomic registers, such that only  $p_i$  can write to  $SM[i]$ , and  $p_i$  can read any entry. Uppercase letters are used to denote shared registers. However, it is often useful to consider higher level abstractions constructed out of such registers, that are implementable on top of them, such as snapshots objects. In this case, a process can read the entire memory  $SM[1..n]$  in a single atomic operation, denoted `snapshot()` [1]. 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  $p_i$ .

Several classes of failure detectors can be defined according to the kind and the quality of failures information they provide. The presentation is centered on the class  $\{\diamond S_x\}$ , where  $1 \leq x \leq n$ , a simple generalization of the class  $\diamond S$  introduced in [10] (in particular,  $\diamond S_n$  is  $\diamond S$ ). Each process  $p_i$  is endowed with a variable  $TRUSTED_i$  that contains identities of processes that are believed to be currently alive. The process  $p_i$  can only read  $TRUSTED_i$ . When  $j \in TRUSTED_i$  we say “ $p_i$  trusts  $p_j$ ”<sup>1</sup>. By definition, a crashed process trusts all processes. The failure detector class  $\diamond S_x$  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  $p_\ell$ , and a (finite) time after which each process of  $Q$  trusts  $p_\ell$ .

The time  $\tau$ , the set  $Q$  and the process  $p_\ell$  are not known by the processes. 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 sometimes use the following equivalent formulation of  $\diamond S_x$  [29], 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  $p_\ell$ , and a (finite) time after which, for any correct process  $p_i$ , we have  $i \in Q \Rightarrow REPR_i = \ell$  and  $i \notin Q \Rightarrow REPR_i = i$ .

## 2.2 The Iterated immediate snapshot (*IIS*) model

A *one-shot immediate snapshot* object  $IS$  is accessed with a single operation denoted `write_snap()`. Intuitively, when a process  $p_i$  invokes `write_snap(v)` it is as if it instantaneously executes a `write IS[i] ← v` operation followed by an `IS.snapshot()` operation. If several processes execute simultaneously `IS.write_snap()`, 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): they are set-linearizable [31].

The semantics of the `write_snap()` 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. If  $SM[k] = \perp$ , the pair  $(k, \perp)$  is not placed in  $sm_i$ . Moreover, we have (by definition)  $sm_i = \emptyset$ , if the process  $p_i$  never invokes `write_snap()` on the corresponding object. The three properties are<sup>2</sup>:

- **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.**  $\forall i, j : (i, v_i) \in sm_j \Rightarrow sm_i \subseteq sm_j$ .

<sup>1</sup>The original definition of the failure detector calls  $\diamond S$  [10] provides each process  $p_i$  with a set denoted  $SUSPECTED_i$ . Using the set  $TRUSTED_i$  is equivalent to using the set  $SUSPECTED_i$ .

<sup>2</sup>For completeness, a wait-free implementation of the `write_snap()` operation is presented in Appendix B. This implementation is due to Borowsky and Gafni [5].



These properties are represented in the first image of Figure 1, for the case of three processes. The image represents a *simplicial complex*, i.e. a family of sets closed under containment; each set is called a *simplex*, and it represents the views of the processes after accessing the *IS* object. The *vertices* are the 0-simplexes, of size one; edges are 1-simplexes, of size two; triangles are of size three (and so on). Each vertex is associated with a process  $p_i$ , and is labeled with  $sm_i$  (the *view*  $p_i$  obtains from the object).

The highlighted 2-simplex in the figure represents a run where  $p_1$  and  $p_3$  access the object concurrently, both get the same views seeing each other, but not seeing  $p_2$ , which accesses the object later, and gets back a view with the 3 values written to the object. But  $p_2$  can't tell the order in which  $p_1$  and  $p_3$  access the object; the other two runs are indistinguishable to  $p_2$ , where  $p_1$  accesses the object before  $p_3$  and hence gets back only its own value or the opposite. These two runs are represented by the corner 2-simplexes. Thus, the vertices at the corners of the complex represents the runs where only one process  $p_i$  accesses the object, and the vertices in the edges connecting the corners represent runs where only two processes access the object. The triangle in the center of the complex, represents the run where all three processes access the object concurrently, and get back the same view.

In the *iterated immediate snapshot model (IIS)* 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 each process.

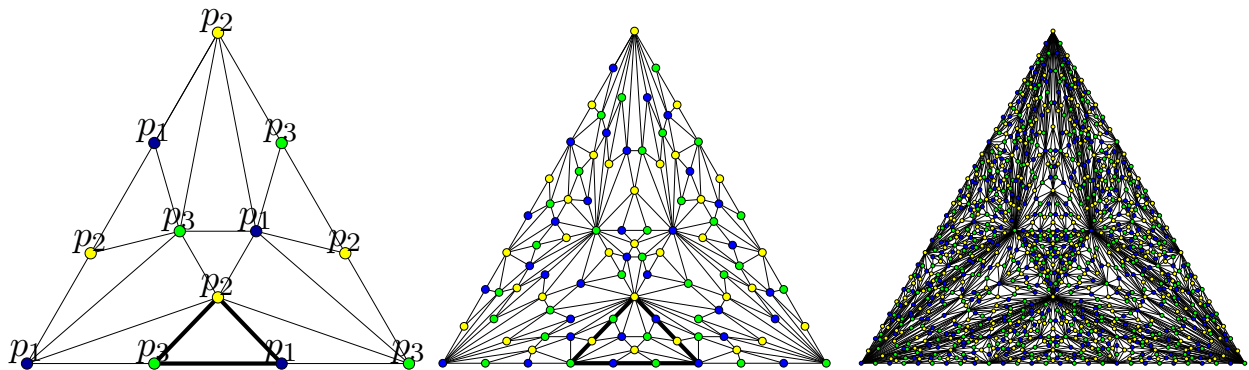


Figure 1: One, two and three rounds in the *IIS* model

In Figure 1 one can see that the *IIS* complex remains a manifold with no holes round after round, and is constructed recursively by replacing each simplex by the one round complex.

**On the meaning of failures in the *IIS* model** Consider a run where processes,  $p_1, p_2, p_3$ , execute an infinite number of rounds, but  $p_1$  is scheduled before  $p_2, p_3$  in every round. The triangles at the left-bottom corners of the complexes in Figure 1 represent such a situation;  $p_1$ , at the corner, never hears from the two other processes. Of course, in the usual (non-iterated read/write shared memory) asynchronous model, two correct processes can always eventually communicate with each other. Thus, in the *IIS* model, the set of *correct processes* of a run,  $Correct_{IIS}$ , is defined as the set of processes that observe each other directly or indirectly infinitely often (a formal definition of the set  $Correct_{IIS}$  is given in Appendix A).

### 2.3 Tasks and equivalence of the two models

An algorithm *solves a task* if each process starts with a private input value, and correct processes (according to the model) eventually decide on a private output value satisfying the task's specification. In an *agreement task*, the specification is such that, if a process decides  $v$ , it is valid for any other process to decide  $v$  (or some other function of  $v$ ). The  $k$ -set agreement task is an agreement task, where processes start with input values of some domain of at least  $n$  values, and must decide on at most  $k$  of their input values.

It was proved in [7] that a task (with a finite number of inputs) is solvable wait-free in the read/write memory model if and only if it is solvable in the *IIS* model. As can be seen in Figure 1, the *IIS* complex of global states at

any round is a subdivided simplex, and hence Sperner's Lemma implies that  $k$ -set agreement is not solvable in the *IIS* model if  $k < n$ . Thus, it is also unsolvable in the wait-free read/write memory model.

### 3 The IRIS model

This section presents the *IRIS* model associated with a failure detector class  $C$ , denoted  $IRIS(PR_C)$ . It consists of a subset of runs of the *IIS* model, that satisfy a corresponding  $PR_C$  property. In order to distinguish the write-snapshot operation in the *IIS* model and its more constrained counterpart of the *IRIS* model, the former is denoted  $R[r].write\_snap()$ , while the latter is denoted  $IS[r].WRITE\_SNAPSHOT()$ .

#### 3.1 The model $IRIS(PR_C)$ with $C = \diamond S_x$

Let  $sm_j^r$  be the view obtained by the process  $p_j$  when it returns from the  $IS[r].WRITE\_SNAPSHOT()$  invocation. As each process  $p_i$  is assumed to execute rounds forever,  $sm_i^r = \emptyset$  means that  $p_i$  never executes the round  $r$ , and is consequently faulty.

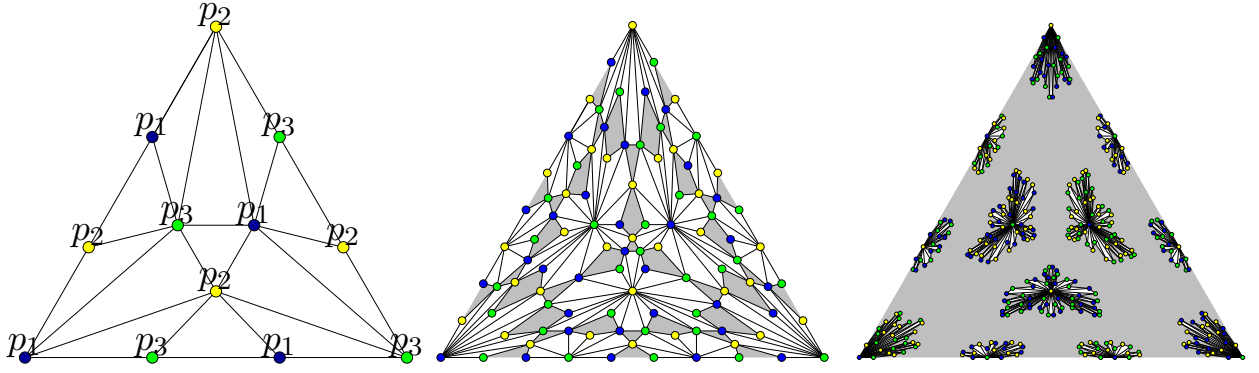


Figure 2: One, two and three rounds in  $IRIS(PR_{\diamond S_x})$  with  $x = 3$  ( $PR_{\diamond S_3}$  is satisfied from round 2)

When  $x = n$ ,  $PR_{\diamond S_n}$  states that in every run, there exists a process  $p_\ell$  and a round  $r$ , such that every process that does not crash sees  $p_\ell$  in every round  $r' \geq r$ . More generally, the property states that there is a set  $Q$  of  $x$  processes containing a process  $p_\ell$  that does not crash, 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  $sm_\ell^{r'}$ . Formally, the property  $PR_{\diamond S_x}$  is defined as follows<sup>3</sup>:

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

Figure 2 shows runs of the  $IRIS(PR_{\diamond S_x})$  model for  $x = 3$ , while Figure 3 shows runs of the  $IRIS(PR_{\diamond S_x})$  model for  $x = 2$ . Let us notice that the complex remains connected in the case  $x = 2$  (Figure 3) and consequently consensus is unsolvable in that model. Differently, in the case  $x = 3$  (Figure 2), consensus is unsolvable in 2 rounds, but it is solvable, as in the 3rd round the complex gets disconnected.

**Theorem 1 (main)** *An agreement task is solvable in the read/write model equipped with a failure detector of the class  $\diamond S_x$  if and only if it is solvable in the  $IRIS(PR_{\diamond S_x})$  model.*

We prove this theorem in Section 4 by providing a transformation from the read/write model enriched with  $\diamond S_x$  to the  $IRIS(PR_{\diamond S_x})$  model and the inverse transformation from the  $IRIS(PR_{\diamond S_x})$  model to the read/write model with  $\diamond S_x$ . The restriction of the theorem to agreement tasks comes from the fact that the first transformation does not preserve faultiness. A correct process may be perceived faulty in the simulated iterated run.

<sup>3</sup>The definition implicitly assumes that, each invocation of  $IS[r].WRITE\_SNAPSHOT()$  issued by  $p_i$  takes the index  $i$  of  $p_i$  as an input parameter.

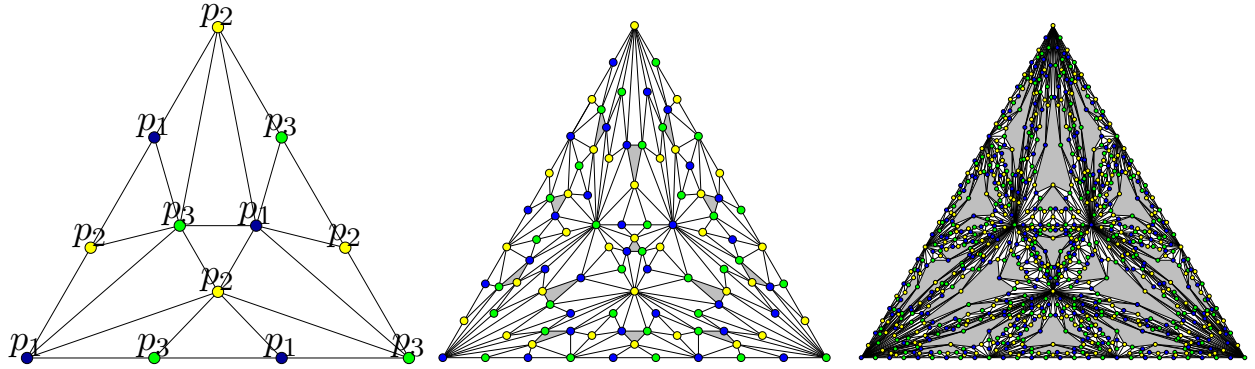


Figure 3: One, two and three rounds in  $IRIS(PR_{\diamond S_x})$  with  $x = 2$  ( $PR_{\diamond S_2}$  is satisfied from round 2)

### 3.2 The $k$ -set agreement with $\diamond S_x$

The power of the  $IRIS$  model becomes evident when we use it to prove the lower bound for  $k$ -set agreement in the shared memory model equipped with a failure detector of the class  $\diamond S_x$ .

**Theorem 2** *In the read/write shared memory model, in which any number of processes may crash, there is no  $\diamond S_x$ -based algorithm that solves  $k$ -set agreement if  $k < n - x + 1$ .*

The proof consists of first observing that, if we partition the  $n$  processes in two sets: the low-order processes  $L = \{p_1, \dots, p_{n-x+1}\}$  and the high-order processes  $H = \{p_{n-x+2}, \dots, p_n\}$ , and consider all  $IIS$  runs where the processes in  $H$  never take any steps, these runs trivially satisfy the  $PR_{\diamond S_x}$  property. Therefore, as noticed at the end of Section 2.3,  $k$ -set agreement is unsolvable in the  $IIS$  model when  $k < n - x + 1$ , and hence unsolvable in our  $IRIS(PR_{\diamond S_x})$  model. By Theorem 1 it is unsolvable in the read/write shared memory model equipped with a failure detector of the class  $\diamond S_x$ .

The argument is illustrated in Figure 4. It depicts the first three rounds of a subset of legal executions in the  $IRIS(PR_{\diamond S_2})$  model. More precisely, Figure 4 pictures all executions that satisfy property  $PR_{\diamond S_2}$  with the following parameters:  $Q = \{p_2, p_3\}$  and  $\ell = p_2$ . At the heart of the proof lies the observation that these set of executions contains all possible wait-free executions of processes  $p_1$  and  $p_2$  (these executions are highlighted in the picture). Moreover, we observe that in these executions neither  $p_1$  nor  $p_2$  see  $p_3$  in their successive views. Therefore, an algorithm designed for the  $IRIS(PR_{\diamond S_2})$  model that solves some task  $T$  can be directly used to wait-free solve the same task among  $p_1$  and  $p_2$ .

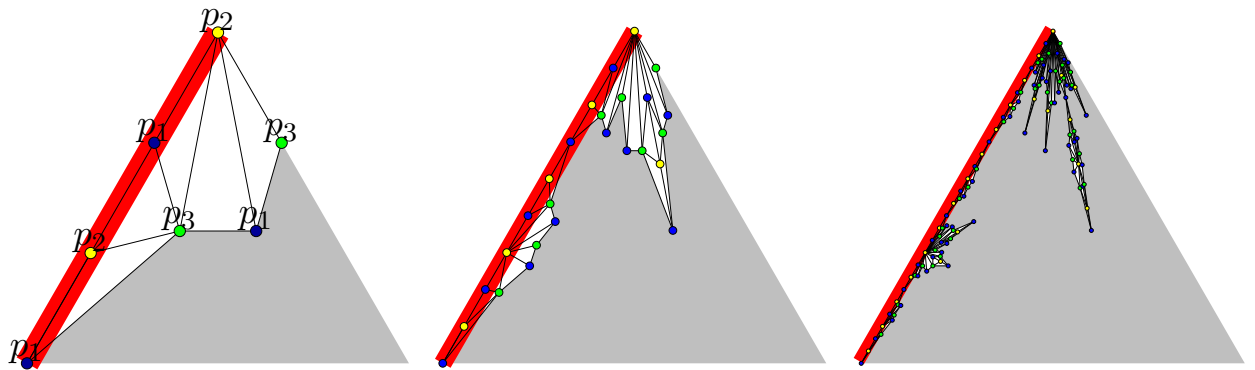


Figure 4: Subsets of  $IRIS(PR_{\diamond S_2})$  that contain all executions by  $p_1$  and  $p_2$

Herlihy and Penso [22] have established a more general result. They consider an extension of the family  $\diamond\mathcal{S}_x$ , namely  $\diamond\mathcal{S}_{x,q}$  and work in the message passing model, assuming at most  $t$  crash failure in any execution. In the full paper we show how to prove this in our *IRIS* framework.

## 4 Simulations

This section proves Theorem 1 with the two simulations between  $IRIS(PR_{\diamond\mathcal{S}_x})$  and the read/write model with  $\diamond\mathcal{S}_x$ . Due to space limitations, the proofs of this section are in appendix C.

### 4.1 From the read/write model with $\diamond\mathcal{S}_x$ to $IRIS(PR_{\diamond\mathcal{S}_x})$

This section presents a simulation of the  $IRIS(PR_{\diamond\mathcal{S}_x})$  model from the read/write model equipped with a failure detector  $\diamond\mathcal{S}_x$ . The aim is to produce subsets of runs of the *IIS* model that satisfy the property  $PR_{\diamond\mathcal{S}_x}$ . The algorithm is described in Figure 5. It uses the  $\diamond\mathcal{S}_x$  version based on the representative variable  $REPR_i$ .  $R[r]$  is the immediate snapshot object associated with the round  $r$  that supports an additional operation:  $R[r].\text{snapshot}()$ . Given any object  $R[r]$ , the  $R[r].\text{snapshot}()$  operations are ordered by containment, and the  $R[r].\text{write\_snap}()$  operations are consistently ordered with respect to the  $R[r].\text{snapshot}()$  operations. These operations can be wait-free implemented from base read/write operations [1, 5].

```

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

```

Figure 5: From the read/write model with  $\diamond\mathcal{S}_x$  to the  $IRIS(PR_{\diamond\mathcal{S}_x})$  model (code for  $p_i$ )

When it invokes  $IS[r].\text{WRITE\_SNAPSHOT}(< i, v_i >)$ , 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 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  $R[r].\text{write\_snap}(< i, v_i >)$  to write in the object  $R[r]$ . It finally returns the snapshot value obtained by that  $\text{write\_snap}()$  invocation.

In infinite executions in which the underlying failure detector belongs to the class  $\diamond\mathcal{S}_x$ , the set of sequences of views ( $sm_i$ ) produced by the algorithm satisfies the property  $PR_{\diamond\mathcal{S}_x}$ . Yet, in order to solve in the read/write model with  $\diamond\mathcal{S}_x$  a task known to be solvable in  $IRIS(PR_{\diamond\mathcal{S}_x})$ , the set of correct processes  $Correct_{IIS}$  in the simulated execution should be related with the correct processes (denoted  $Correct_{rw}$ ) in the base read/write model. It can be shown that  $Correct_{IIS} \subseteq Correct_{rw}$ . This condition is sufficient as far as we are interested in agreement tasks: when a process has decided in the simulated run, it writes its decision in a register in order to allow the processes simulated as faulty to decide.

### 4.2 From $IRIS(PR_{\diamond\mathcal{S}_x})$ to the read/write model equipped with $\diamond\mathcal{S}_x$

We first show how to simulate the basic operations of an *IIS* model, namely  $\text{write}()$  and  $\text{snapshot}()$ . This simulation works for any  $IRIS(PR)$  model, as its runs are a subset of the *IIS* runs. Then a complete simulation that encompasses the failure detector  $\diamond\mathcal{S}_x$  is given.

**Simulating the  $\text{write}()$  and  $\text{snapshot}()$  operations** The algorithm described in Figure 6 is based on the ideas of the simulation of [7]. A process simulates an operation  $op \in \{\text{write}(), \text{snapshot}()\}$  by invoking  $\text{simulate}(op)$ . Without loss of generality, we assume that (as in [7]) the  $k$ th value written by a process is  $k$  (consequently, a snapshot of the shared memory is a vector made up of  $n$  integers). To respect the semantics of the shared memory, vectors  $v$  returned as result of  $\text{simulate}(\text{snapshot}())$  should be ordered and contain the integers written by the last  $\text{simulate}(\text{write}())$  that precedes it.

```

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

function simulate( $op()$ )                                     %  $op \in \{write(), snapshot()\}$ 
(1) if  $op() = write()$  then  $est_i[i] \leftarrow est_i[i] + 1$  end if;  $r\_start_i \leftarrow r_i$ ;
(2) repeat  $r_i \leftarrow r_i + 1$ ;
(3)    $sm_i \leftarrow IS[r_i].WRITE\_SNAPSHOT(\langle i, est_i, view_i[1..(r_i - 1)] \rangle)$ ;
(4)    $view_i[r_i] \leftarrow \{ \langle i, \{ \langle j, est_j \rangle \text{ such that } \langle j, est_j, - \rangle \in sm_i \} \rangle \}$ ;
(5)   for each  $\rho \in \{1, \dots, r_i - 1\}$  do  $view_i[\rho] \leftarrow \bigcup_{view_j \text{ such that } \langle j, -, view_j \rangle \in sm_i} view_j[\rho]$  end for;
(6)    $est_i \leftarrow \max_{cw} \{ est_j \text{ such that } \langle j, est_j, - \rangle \in sm_i \}$ ;
(7)   if  $(\exists \rho > r\_start_i \text{ such that } \exists \langle -, sm_{min} \rangle \text{ such that } \forall j \in sm_{min} : \langle j, sm_{min} \rangle \in view_i[\rho])$ 
        % there is a smallest snapshot in  $view_i[r\_start_i + 1..r_i]$  that is known by  $p_i$  %
(8)     then let  $\rho'$  be the greatest round  $\leq r_i$  that satisfies the previous predicate;
(9)      $sm_{min_i} \leftarrow$  the smallest snapshot in  $view_i[\rho']$ ;
(10)     $last\_snap_i \leftarrow \max_{cw} \{ est_j \text{ such that } \langle j, est_j \rangle \in sm_{min_i} \}$ ;
(11)    if  $last\_snap_i[i] = est_i[i]$  then if  $op = snapshot()$  then return ( $last\_snap_i$ )
(12)    else return() end if end if
(13)  end if
(14) end repeat

```

Figure 6: Simulation of the write() and snapshot() operations in  $IRIS(PR_C)$  (code for  $p_i$ )

As in [7], each process  $p_i$  maintains an estimate vector  $est_i$  of the current state of the simulated shared memory. When  $p_i$  starts simulating its  $k$ -th write(), it increments  $est_i[i]$  to  $k$  to announce that it wants to write the shared memory (line 1). At each round  $r$ ,  $p_i$  writes its estimate in  $IS[r]$  and updates its estimate by taking the maximum component-wise, denoted  $\max_{cw}$ , of the estimates in the view  $sm_i$  it gets back (line 6). The main difference with [7] is the way processes compute valid snapshots of the shared memory. In [7],  $p_i$  returns a snapshot when all estimates in its view are the same. Here, for any round  $r$ , we define a valid snapshot as the maximum component-wise (denoted  $sm\_min^r$ ) of the estimates that appear in the smallest view (denoted  $sm_{min}^r$ ) returned by  $IS[r]$ . Due to the fact that estimates are updated maximum component-wise, it follows from the containment property of views that  $\forall r, r' : r < r' \Rightarrow sm\_min^r \leq sm\_min^{r'}$ .

In order to determine smallest views, each process  $p_i$  maintains an array  $view_i[1, \dots]$  that aggregates  $p_i$ 's knowledge of the views obtained by other processes. This array is updated at each round (lines 4-5) by taking into account the knowledge of other processes (that appear in  $sm_i$ ).

Then,  $p_i$  tries to determine the last smallest view that it can know by observing the array  $view_i$  (line 7). If there is a recent one (it is associated with a round greater than the round  $r\_start_i$  at which  $p_i$  has started simulating its current operation),  $p_i$  keeps it in  $sm_{min_i}$  (lines 8-9), and computes in  $last\_snap_i$  the corresponding snapshot value of the shared memory (line 10). Finally, if  $p_i$  observes that its last operation announced (that is identified  $est_i[i]$ ) appears in this vector, it returns  $last\_snap_i$  (line 11). In the other cases,  $p_i$  starts a new iteration of the loop body.

For any round  $r$ , let  $sm\_min^r = \max_{cw} \{ est \text{ such that } \langle -, est, - \rangle \in sm_{min}^r \}$ . As observed earlier, the fact that estimate vectors are component-wise maximum and the inclusion property of views imply that the sequence of vectors ( $sm\_min^r$ ) is increasing. As each snapshot returned is equal to  $sm\_min^r$  for some  $r$ , it follows that any two snapshots of the shared memory are equal or one is greater than the other. Given an operation  $op$ , let us also observe that  $r_s \leq r$ , where  $r_s$  is the round at which the simulation of  $op$  starts. Hence, any simulate(snapshot()) that starts after a completed write returns the value written or a most recent one. Finally, it can be induced from the notion of correctness in the iterated model that correct processes may simulate infinitely many write operations while faulty processes can simulate only a finite number of them.

**From  $IRIS(PR_{\diamond S_x})$  to a failure detector of the class  $\diamond S_x$**  In a model equipped with a failure detector, each process can read at any time the output of the failure detector. We denote  $fd\_query()$  this operation. A trivial algorithm that simulates  $\diamond S_x$ -queries in the  $IRIS(PR_{\diamond S_x})$  is described in Figure 7.

```

init  $r_i \leftarrow 0$ ; TRUSTED $_i \leftarrow \Pi$ 

function simulate(fd_query())
(1)  $r_i \leftarrow r_i + 1$ ;  $sm_i \leftarrow IS[r_i].WRITE\_SNAPSHOT(i)$ ;
(2) TRUSTED $_i \leftarrow \{j : j \in sm_i\}$ ; return( TRUSTED $_i$ )

```

Figure 7: Simulation of fd\_query() in  $IRIS(PR_{\diamond S_x})$  (code for  $p_i$ )

**General simulation** Given an algorithm  $\mathcal{A}$  that solves a task  $T$  in the read/write model equipped with  $\diamond S_x$ , we show how to solve  $T$  in the  $IRIS(PR_{\diamond S_x})$  model. Algorithm  $\mathcal{A}$  performs local computation, write(), snapshot() and fd\_query(). In the  $IRIS(PR_{\diamond S_x})$  model, processes run in parallel the algorithms described in Figures 6 and 7 in order to simulate these operations. More precisely, whatever the operation  $op \in \{\text{write}(), \text{snapshot}(), \text{fd\_query}()\}$  being simulated, each immediate snapshot object is used to update both the estimate of the shared memory and the output of the failure detector.

A main difficulty in the proof consists in establishing that the successive states of the simulated shared memory and the failure detector outputs are consistent with respect to failures. E.g., a process that is perceived faulty through the simulated failure detector does not change the state of the shared memory infinitely often. To that end, we show that any infinite run of the simulation produces an infinite run of the shared memory in which all operations are linearizable. Then, we show that there exists a failure pattern  $FP$  such that (1) the failure detector outputs are admissible for  $FP$  according to the failure detector specification, and (2) the successive states of the shared memory are consistent with  $FP$ . Moreover, the correct processes in  $FP$  are exactly the correct processes of the base model. The equivalence theorem (Theorem 1 that has been announced in Section 3.1) then follows from the two simulations presented in Section 4.1 and Section 4.2.

**Acknowledgments** The authors wish to thank Alejandro Cornejo and Eli Gafni for many discussions on wait-free distributed computing.

## References

- [1] Afek Y., Attiya H., Dolev D., Gafni E., Merritt M. and Shavit N., Atomic Snapshots of Shared Memory. *Journal of the ACM*, 40(4):873-890, 1993.
- [2] Attiya H., Bar-Noy A. and Dolev D., Sharing Memory Robustly in Message Passing Systems. *Journal of the ACM*, 42(1):124-142, 1995.
- [3] Awerbuch, B., Complexity of network synchronization. *Journal of the ACM*, 32, pp. 804–823, 1985.
- [4] Attiya H. and Welch J., *Distributed Computing: Fundamentals, Simulations, and Advanced Topics*, Wiley, 2004.
- [5] Borowsky E. and Gafni E., Immediate Atomic Snapshots and Fast Renaming. *Proc. 12th ACM Symposium on Principles of Distributed Computing (PODC'93)*, ACM Press, pp. 41-51, 1993.
- [6] 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.
- [7] Borowsky E. and Gafni E., A Simple Algorithmically Reasoned Characterization of Wait-free Computations. *Proc. 16th ACM Symposium on Principles of Distributed Computing (PODC'97)*, ACM Press, pp. 189-198, 1997.
- [8] Borowsky E., Gafni E., Lynch N. and Rajsbaum S., The BG distributed simulation algorithm. *Distributed Computing*, 14(3):127-146, 2001.
- [9] Biran, O., Moran, S., and Zaks, S., A Combinatorial Characterization of the Distributed 1-solvable Tasks. *J. Algorithms*, 11, pp. 420-440, 1990.
- [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] Chen W., Zhang J., Chen Y. and Liu X., Weakening Failure Detectors for  $k$ -Set Agreement via the Partition Approach *Proc. 21st Int'l Symposium on Distributed Computing (DISC'07)*, Springer Verlag LNCS #4731, pp. 123-138, 2007.
- [14] Dwork, C., Lynch, N. and Stockmeyer, L., Consensus in the presence of partial synchrony, *J. ACM*, 35(2):288-323, 1988.
- [15] 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.
- [16] Gafni E., Round-by-round Fault Detectors: Unifying Synchrony and Asynchrony. *Proc. 17th ACM Symposium on Principles of Distributed Computing (PODC'00)*, ACM Press, pp. 143-152, 1998.
- [17] Gafni E.,
- [18] Guerraoui R., Herlihy M.P., Kouznetsov P., Lynch N. and Newport C. On the weakest failure detector ever *Proc. 26th ACM Symposium on Principles of Distributed Computing (PODC'07)*, ACM Press, pp. 235-243, 2007.
- [19] 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.
- [20] 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.
- [21] Herlihy M.P., Wait-Free Synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124-149, 1991.
- [22] 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.
- [23] 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.

- [24] Herlihy M.P. and Shavit N., The Topological Structure of Asynchronous Computability. *Journal of the ACM*, 46(6):858-923,, 1999.
- [25] Keidar I., Shraer A., Timeliness, Failure-detectors, and Consensus Performance. *Proc. 25th ACM Symposium on Principles of Distributed Computing (PODC'06)*, ACM Press, pp. 169-178, 2006.
- [26] Lamport, L. and Lynch, N., Distributed Computing: Models and Methods. *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, 1157-1199, 1990.
- [27] Lynch, N. A., *Distributed Algorithms*, Morgan Kaufmann, 872 pages, 1997.
- [28] Moses, Y. and Rajsbaum, S., A Layered Analysis of Consensus. *SIAM Journal of Computing*, 31(4): 989-1021, 2002.
- [29] 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. Extended abstract: Irreducibility and Additivity of Set Agreement-oriented Failure Detector Classes, in *Proc. PODC'06*, ACM Press, pp. 153-162, 2006.
- [30] Mostefaoui A. and Raynal M., Unreliable Failure Detectors 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.
- [31] Neiger G., Set Linearizability. *Brief Announcement, Proc. 13th ACM Symposium on Principles of Distributed Computing (PODC'94)*, ACM Press, pp. 396, 1994.
- [32] 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.
- [33] Rajsbaum S., Raynal M., Travers C., Failure Detectors as Schedulers (An Algorithmically-Reasoned Characterization). *Tech Report # 1838*, 38 pages, IRISA, Université de Rennes, France, March 2007.
- [34] 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.
- [35] Völzer H., On Conspiracies and Hyperfairness in Distributed Computing. *Proc. 19th Int'l Symposium on Distributed Computing (DISC'05)*, Springer-Verlag LNCS, pp. 33-47, 2005.
- [36] 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)*, ACM Press, pp.297-308, 1998.
- [37] Zieliński P., Anti-Omega: the weakest failure detector for set agreement. *Tech Report # 694*, University of Cambridge, July 2007.

## A Notion of correct process in the *IIS* model

This section defines precisely the notion of what is a *correct* process in the *IIS* model. Intuitively, given an infinite execution in this model, the set of correct processes  $Correct_{IIS}$  is the set of processes that see each other directly or indirectly infinitely often. (A similar definition of what is a correct process in the *IIS* model is given in [17].)

At round  $r$ , a process  $p_i$  “sees directly” a process  $p_j$  if  $j$  appears in the view  $sm_i^r$ . Yet, even if  $\forall r : j \notin sm_i^r$ , a process  $p_i$  can “see indirectly” a process  $p_j$  by observing past views of some process  $p_k$  that has seen directly  $p_j$ . For example, let us consider the following run defined for three processes:  $\forall r : sm_1^{2r} = \{1\}$ ,  $sm_2^{2r} = \{1, 2\}$ ,  $sm_3^{2r} = \{1, 2, 3\}$  and  $sm_1^{2r+1} = \{1, 3\}$ ,  $sm_2^{2r+1} = \{1, 2, 3\}$ ,  $sm_3^{2r+1} = \{3\}$ .  $p_1$  never sees directly  $p_2$ . However, in each odd round  $r$ ,  $p_1$  can learn the value written by  $p_2$  in the previous round if  $p_3$  writes its last view in  $IS[r]$ . The next paragraph formalizes the relations “seen directly” and “seen indirectly”. Given an infinite execution, the smallest equivalence class induced by these relations define the set  $Correct_{IIS}$ .

A run in the Iterated Immediate Snapshot model is entirely defined by the sequences  $(sm_i^r)_{r \geq 1}$  of each process  $p_i$ , where  $sm_i^r \subseteq \{1, \dots, n\}$  ( $(sm_i^r)_{r \geq 1}$  is the sequence of the consecutive views obtained by  $p_i$ ). As seen in Section 2.2, it is possible that for some processes  $p_j$  we have a round  $R_j$  such  $\forall r \geq R_j : sm_j^r = \emptyset$  (those processes are the processes that “crashed” in that run, according to the usual meaning).



Given an infinite run in the *IIS* model, let  $\xrightarrow{s}$  be the relation over the set of processes  $p_i$  such that  $\forall r \geq 1 : sm_i^r \neq \emptyset$ , defined as follows:  $p_i \xrightarrow{s} p_j$  if the set of rounds  $\{r : j \in sm_i^r\}$  is infinite. This relation captures the fact that  $p_i$  observes *directly*  $p_j$  infinitely many times. Due to the self-inclusion property of the immediate snapshot objects, the relation  $\xrightarrow{s}$  is reflexive. Moreover, due to the containment property of the immediate snapshot objects, we have  $\forall p_i, p_j : p_i \xrightarrow{s} p_j \vee p_j \xrightarrow{s} p_i$ . Let  $\xrightarrow{\sim}$  be the transitive closure of  $\xrightarrow{s}$ .

Let  $obs(p_i)$  be the set of processes observed directly or indirectly infinitely many often by  $p_i$ , i.e.,  $obs(p_i) = \{p_j : p_i \xrightarrow{\sim} p_j\}$ . From the properties of  $\xrightarrow{s}$ , it follows that (1)  $\forall p_i : obs(p_i) \neq \emptyset$  and (2),  $\forall p_i, p_j : obs(p_i) \subseteq obs(p_j) \vee obs(p_j) \subseteq obs(p_i)$ . Consequently, there exists a smallest set  $obs_{\min}$ . We define the set of correct processes in a run as the associated set  $obs_{\min}$ .

The following lemma follows directly from the definition of correct processes in the *IIS* model.

**Lemma 1** *Let  $e$  be an infinite run of the *IIS* model. Let  $sm_{\min}^r$  be the smallest immediate snapshot returned at round  $r$ .  $\exists R$  such that (1)  $\forall p_i \in Correct_{IIS}, \forall r \geq R : sm_i^r \subseteq Correct_{IIS}$  and, (2)  $\forall r \geq R : sm_{\min}^r \subseteq Correct_{IIS}$ .*

## B A wait-free implementation of the write\_snap() operation

For a completeness purpose, this appendix presents a one-shot write\_snap() construction. This algorithm, due to Borowski and Gafni [5], is described in Figure 8. That algorithm considers a one-shot immediate snapshot object (a process invokes  $SM.write\_snap()$  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\_snap() 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  $\ell$  of  $p_i$  is such that  $p_i$  sees at least  $\ell$  processes in its view (i.e., processes that are at its level or a lower level) it stops at the level  $\ell$  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 8 [5].

```

operation write_snap(v):
  REG[i] ← v;
  repeat LEVEL[i] ← LEVEL[i] - 1;
    for j ∈ {1, ..., n} do level_i[j] ← LEVEL[j] end for;
    view_i ← {j : level_i[j] ≤ LEVEL[i]};
  until (|view_i| ≥ LEVEL[i]) end repeat;
  return ({(j, REG[j]) such that j ∈ view_i}).

```

Figure 8: Borowsky-Gafni's one-shot write\_snap() algorithm (code for  $p_i$ )

This very elegant algorithm satisfies the following properties [5]. 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  $\ell$ , and there are  $\ell$  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.

## C Missing proofs of Section 4

### C.1 From the read/write model with $\diamond S_x$ to $IRIS(PR_{\diamond S_x})$

This section shows that, as far as agreement tasks are concerned, the  $IRIS(PR_{\diamond S_x})$  model can be simulated in the base wait-free read/write model enriched with  $\diamond S_x$ . More precisely, it shows that any agreement task  $T$  that is solvable in  $IRIS(PR_{\diamond S_x})$  is solvable in the read/write model equipped with a failure detector of the class  $\diamond S_x$  (Lemma 4).

Let us assume that each process  $p_i$  invokes sequentially first  $IS[1].WRITE\_SNAPSHOT(< i, - >)$ , then  $IS[2].WRITE\_SNAPSHOT(< i, - >)$ , etc., until it possibly fails.  $sm_i^r \subseteq \{1, \dots, n\}$  denotes the view returned from the invocation  $IS[r].WRITE\_SNAPSHOT()$  (the value that  $p_i$  is assumed to write together with its identity in  $IS[r]$  is ignored). Let us remind that, if  $p_i$  fails before invoking  $IS[r].WRITE\_SNAPSHOT()$  or does not return from that invocation, we have  $sm_i^r = \emptyset$ . Let  $\mathcal{S}$  be the set of the sequences of views obtained by the processes, i.e.,  $\mathcal{S} = \{(sm_i^r)_{r \geq 1}, i \in \{1, \dots, n\}\}$ . This first lemma shows that the algorithm described in Figure 5 produces sequences that satisfy the property  $PR_{\diamond \mathcal{S}_x}$ .

**Lemma 2** *Let  $\mathcal{S} = \{(sm_i^r)_{r \geq 1}, i \in \{1, \dots, n\}\}$  be a set of infinite sequences of views produced by algorithm 5 in the read/write model equipped with  $\diamond \mathcal{S}_x$ .  $\mathcal{S}$  represents an infinite execution of the IIS model that satisfies the  $PR_{\diamond \mathcal{S}_x}$  property.*

**Proof** Given a round  $r$ , views  $sm_i$  are returned from the object  $R[r]$  associated with round  $r$ . The properties of this “readable” immediate snapshot object guarantees that the views  $\{sm_i^r, i \in \{1, \dots, n\}\}$  satisfy the self-inclusion, containment and immediacy properties.  $\mathcal{S}$  is an admissible execution in the IIS model.

The rest of the proof is divided in two parts. The first part establishes that each correct process  $p_i$  in the base model ( $Correct_{rw}$  denotes the set of correct process in the read/write model), obtains infinitely many views  $sm_i \neq \emptyset$ . The second part proves that  $\mathcal{S}$  satisfies  $PR_{\diamond \mathcal{S}_x}$ .

1.  $\forall i \in Correct_{rw}, \forall r : sm_i^r \neq \emptyset$ . Let us assume for contradiction that  $\exists i \in Correct_{rw}, \exists r$  such that  $sm_i^r = \emptyset$ . Let  $m$  be the smallest round at which a correct process  $p_i$  is such that  $sm_i = \emptyset$ . As  $p_i$  is correct, this can only happen if  $p_i$  never gets an answer from the invocation  $IS[m].WRITE\_SNAPSHOT()$ . This means that  $p_i$  never exit from the repeat loop.

At  $p_i$ , there is a time after which  $REPR_i$  contains permanently the same identity  $j$ . Moreover,  $j$  is the identity of a correct process and there is a time after which  $REPR_j = j$  forever (definition of the class  $\diamond \mathcal{S}_x$ , see Section 2.1). Due to the definition of  $m$ , all correct processes invoke  $IS[m].WRITE\_SNAPSHOT()$ . Since eventually  $REPR_j = j$  forever, it follows that the predicate of line 2 (of the algorithm described in Figure 5) eventually becomes true at  $p_j$ . Therefore,  $p_j$  eventually executes  $R[r].write\_snap(< j, v_j >)$  (line 3, Figure 5). Consequently, as  $p_i$  snapshots  $R[m]$  forever, it eventually obtain a set  $m_i$  that contains its representative  $j$ . It follows that  $p_i$  then computes a views  $sm_i$  for round  $r$ : a contradiction.

2.  $PR_{\diamond \mathcal{S}_x}$  is satisfied in  $\mathcal{S}$ . Due to the properties defining  $\diamond \mathcal{S}_x$ , there is a set  $Q$  of  $x$  processes including a correct process  $p_\ell$ , such that, after some arbitrary but finite time  $\tau$ , 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  $p_\ell$  that appears in the statement of the property  $PR_{\diamond \mathcal{S}_x}$  as the set and the process that are denoted the same way in the definition of  $\diamond \mathcal{S}_x$ . Let  $R$  be the first round that starts after  $\tau$  (i.e., in the given execution, any  $IS[R].WRITE\_SNAPSHOT()$  starts after  $\tau$ ). Let  $r \geq R$  and  $p_i$  such that  $i \in Q - \{\ell\} \wedge sm_i^r \neq \emptyset$ . As  $p_\ell$  is correct, it writes in  $R[r]$  (as shown in Item 1 above). When  $p_i$  executes the repeat loop of round  $r$ , it is such that  $REPR_i = \ell (\neq i)$ . The only possibility for  $p_i$  to exit the repeat loop is to observe that  $p_\ell$  has executed  $R[r].write\_snap()$ , from which we conclude that  $sm_i^r \subsetneq sm_i^r$ . It follows that the property  $PR_{\diamond \mathcal{S}_x}$  is satisfied in  $\mathcal{S}$ .

□ Lemma 2

The previous proof shows that at least one process obtains infinitely many non-empty views in  $\mathcal{S}$ . The set of correct processes  $Correct_{IIS}$  in the simulated execution is consequently non-empty. Moreover, one can easily check that a correct process in the simulated execution is a correct process in the base read/write model.

**Lemma 3** *Let  $\mathcal{S}$  be an infinite  $IRIS(PR_{\diamond \mathcal{S}_x})$  execution produced by the algorithm described in Figure 5. Let  $Correct_{rw}$  and  $Correct_{IIS}$  be respectively the sets of correct processes in the execution of the base model and in the simulated execution  $\mathcal{S}$ .  $Correct_{IIS} \subseteq Correct_{rw}$ .*

A simple counter example that in general  $Correct_{IIS} \neq Correct_{rw}$  is as follows. In an execution such that  $|Correct_{rw}| \geq 2$ , a process  $p_i \in Correct_{rw}$  that is eventually its own representative forever may always be the first to write each object  $R[r]$ . In that case, there is only one correct process in the simulated execution, i.e.,  $Correct_{IIS} = \{i\}$ .

Finally, we show that any agreement task solvable in the  $IRIS(PR_{\diamond \mathcal{S}_x})$  model is also solvable in the base read/write model with  $\diamond \mathcal{S}_x$ .

**Lemma 4** *Let  $T$  be an agreement task. If  $T$  is solvable in  $IRIS(PR_{\diamond\mathcal{S}_x})$  model, then it is solvable in the read/write model enriched with  $\diamond\mathcal{S}_x$ .*

**Proof** Let  $\mathcal{A}$  be an algorithm that solves  $T$  in the  $IRIS(PR_{\diamond\mathcal{S}_x})$  model. Let  $D$  be a shared array intended to contain decided values. To solve  $T$  in the read/write model with  $\diamond\mathcal{S}_x$ , each process simulates each `WRITE_SNAPSHOT()` operation of  $\mathcal{A}$  (with the algorithm described in Figure 5) and periodically scans  $D$ . When a process decides through the simulation, it writes its decision value in  $D$ . A process that observes a decided value in  $D$  decides this value.

It follows from Lemma 2 that values that are decided through the simulation of  $\mathcal{A}$  respect the specification of  $T$  (this is because  $\mathcal{A}$  solves  $T$  in  $IRIS(PR_{\diamond\mathcal{S}_x})$  and the simulation produces executions that are admissible in this model). As  $T$  is an agreement task, any value written in  $D$  can be safely decided by any process. Moreover, it follows from Lemma 3 that at least one correct process (in the read/write model) decides through the simulation of  $\mathcal{A}$ . Such a process writes its decision in  $D$ . Consequently, all correct processes decide (either directly by simulating  $\mathcal{A}$  or by observing a decision written in  $D$ ). □ Lemma 4

## C.2 From $IRIS(PR_{\diamond\mathcal{S}_x})$ to the read/write model with $\diamond\mathcal{S}_x$

This section proves that the general simulation (Section 4.2) produces valid runs of the read/write model augmented with a failure detector  $\diamond\mathcal{S}_x$ .

### C.2.1 Simulation of `write()` and `snapshot()` operations

In the  $IRIS(PR_{\diamond\mathcal{S}_x})$  model, let us assume that processes simulate `write()` and `snapshot()` operations by invoking `simulate()` (algorithm described in Figure 6). We show that by doing so, processes simulate a valid run of the read/write model. Moreover, every `simulate(op)` invocation issued by any correct process in the  $IRIS(PR_{\diamond\mathcal{S}_x})$  run terminates.

Let  $x_i^r$  denotes the value of the local variable  $x_i$  of process  $p_i$  at the end of round  $r$  (i.e., before  $p_i$  executes  $r_i \leftarrow r+1$ ). Among all the immediate snapshots returned by the invocations of `WRITE_SNAPSHOT()` 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  $Mm\_est^r$  be the vector that is the component-wise maximum of the estimates  $est$  that appear in  $sm_{in}^r$ , i.e.,  $Mm\_est^r = \max_{c_w}\{est \text{ such that } \langle -, est, - \rangle \in sm_{in}^r\}$ . ( $Mm\_est^r$  stands for Maximum of the  $est$  vectors that appear in the smallest immediate snapshot of the round  $r$ .) Lemma 5 states that all vectors  $sm\_sm_{in}$  are ordered.

**Lemma 5**  $\forall r : Mm\_est^r \leq Mm\_est^{r+1}$ .

**Proof** Let  $i$  such that  $\langle i, est_i, - \rangle \in sm_{in}^{r+1}$ . Due to the containment property of immediate snapshots,  $sm_{in}^r \subseteq sm_i^r$ . As  $p_i$  updates its estimate vector by taking the maximum component-wise of all estimate vectors it observes in  $sm_i^r$ , it follows that  $Mm\_est^r \leq est_i^r$ . At the beginning of round  $r+1$ ,  $est_i$  may change but it remains greater than  $Mm\_est^r$  (line 1). Consequently,  $Mm\_est^r \leq est_i \leq Mm\_est^{r+1}$ . □ Lemma 5

When a process  $p_i$  simulates an operation  $op \in \{\text{write}(), \text{snapshot}()\}$ , it accesses a sequence of immediate snapshot objects  $IS[r], IS[r+1], \dots, IS[r+\alpha]$  or, in other words, execute rounds  $r, r+1, \dots, r+\alpha$ . Let  $\tau_s(op)$  and  $\tau_e(op)$  be respectively the ranks of the first and the last objects accessed by  $p_i$  while simulating  $op$ . If the invocation `simulate(op)` never terminates whereas  $p_i$  keeps executing rounds forever, let  $\tau_e(op) = +\infty$ .

Each process  $p_i$  simulates a sequence of operation  $op_i^1, op_i^2, \dots, op_i^x \in \{\text{write}(), \text{snapshot}()\}$ . This sequence may be infinite and the last operation may not terminate. Let us consider the following read/write run denoted  $e_{rw}$ . In  $e_{rw}$ , processes perform exactly the same sequence of `write()` and `snapshot()` operations. Moreover, each operations  $op_i^x$  starts at time  $\tau_s(op_i^x)$  and ends at time  $\tau_e(op_i^x)$ . In the following, we establish that  $e_{rw}$  is a valid execution in the read/write model.

We first observe that the timing of the relevant events (i.e., the starts and the ends of `write()` and `snapshot()` operations) in  $e_{rw}$  is consistent with causality.

**Lemma 6** *Let  $op_1$  a `snapshot()` operation that returns  $v$  and  $op_2$  the  $k$ -th `write()` operations issued by  $p_i$ .  $v[i] = k \Rightarrow \tau_s(op_2) \leq \tau_e(op_1)$ .*

**Proof** At any process  $p_j$ ,  $est_j^r[i] = k$  is always due to the fact that process  $p_i$  has executed  $est_i[i] \leftarrow k$  at line 1 at the beginning of some round  $r$ . This occurs only when  $p_i$  starts simulating its  $k$ -th write() operation. As the  $k$ -th write of  $p_i$  starts at round  $\tau_s(op_2)$ , it follows that  $\forall r < \tau_s(op_2), \forall p_j : est_j^r[i] < k$ . Consequently,  $\forall r < \tau_s(op_2) : Mm\_est^r[i] < k$ .

Let us observe that  $v$  is equal to  $Mm\_est^r$ , for some  $r$  such that  $\tau_s(op_1) \leq r \leq \tau_e(op_1)$  (lines 7-11). Since  $v[i] = Mm\_est^r[i] = k$ , it follows that  $\tau_s(op_2) \leq \tau_e(op_1)$ .  $\square_{Lemma 6}$

To show that the simulation respects the semantic of the read/write model, let us build a total order  $\mathcal{S}$  on the simulated snapshot() and write() operations. We consider all *effective* operations. An operation is effective if its simulation completes. Moreover, the  $k$ -th write() operation of process  $p_i$  is effective if there exists a vector  $v$  returned as a result of a simulate(snapshot()) invocation such that  $v[i] = k$ . Thus, some of the write operations whose associated simulation does not terminate are effective, others are not. Intuitively, an effective write is a write whose value is seen by other processes. In order to build  $\mathcal{S}$ , we associate a timestamp  $ts(op)$  to each effective operation.

Each simulated shared memory operation  $op$  is uniquely associated with a vector  $v$ . This vector is the value of the local variable  $last\_snap_i$  when  $p_i$  returns from the invocation simulate( $op$ ) (line 11). It directly follows from the code that every such vector is equal to  $Mm\_est^r$  for some  $r$  (lines 7-13). Moreover, we have  $\tau_s(op) \leq r \leq \tau_e(op)$ .

If  $op$  is a snapshot() that returns  $v$ , then  $ts(op)$  is the integer  $r$  such that  $v = Mm\_est^r$ . More precisely,  $r$  is the value of the local variable  $\rho'$  when the invoking process returns. Let us assume that  $op$  is a write() operation, say the  $k$ -th write() of process  $p_i$ . If the invocation simulate( $op$ ) terminates, let  $r$  be the round identified by the algorithm such that  $last\_snap_i = Mm\_est^r$  when  $p_i$  returns. Otherwise,  $r = +\infty$ .  $ts(op)$  is the minimum between  $r$  and the set of timestamps of snapshot() operations that return a vector whose  $i$ -th entry is equal to  $k$ , i.e.,  $ts(op) = \min(\{r\} \cup \{ts(snapshot()) : simulate(snapshot()) \text{ returns } v \text{ with } v[i] = k\})$ . Let us notice that we have  $Mm\_est^{ts(op)}[i] = k$ .

The operations are first ordered in  $\mathcal{S}$  according to their timestamps. If several operations have the same timestamp, write() are ordered before snapshot(); write() and snapshot() are then ordered according to the round at which they start. Finally, two operations of the same type that have the same timestamp and start at the same round are ordered according to the identity of the invoking process.

**Lemma 7**  $\mathcal{S}$  is a linearization of the snapshot() and write() operations.

**Proof** The following facts establish that the simulated snapshot() and write() operations are linearizable.

1.  $\mathcal{S}$  is consistent with the timing of the beginning and the end of write() and snapshot() operations in the simulated run. More precisely, let  $op_1$  and  $op_2$  such that  $\tau_e(op_1) < \tau_s(op_2)$ .  $op_1$  appears before  $op_2$  in  $\mathcal{S}$ .

It directly follows from the definition of timestamps that  $ts(op_1) \leq \tau_e(op_1)$ . If  $op_2$  is a snapshot() operation,  $\tau_s(op_2) \leq ts(op_2)$ . If  $op_2$  is a write() operation, say the  $k$ -th write() of  $p_i$ , we consider two cases :

- There is a snapshot() operation  $op$  that returns  $v$  such that  $v[i] = k$  and  $ts(op_2) = ts(op) = r$ . In that case,  $v = Mm\_est^r$ , i.e.,  $v$  is the maximum component-wise of the estimates  $est_j$  contained in the smallest view of round  $r$ .  $est_j[i] = k$  is always due to the fact that  $p_i$  has executed  $est_i[i] \leftarrow k$  at line 1. Moreover, this occurs only when  $p_i$  starts simulating its  $k$ -th write() operation. Consequently,  $\tau_s(op_2) \leq r$ , i.e.,  $\tau_s(op_2) \leq ts(op_2)$ .
- There is no such snapshot() operation. In that case, the timestamp associated with  $op_2$  is a round  $r$  such  $\tau_s(op_2) \leq r \leq \tau_e(op_2)$ . Therefore,  $\tau_s(op_2) \leq ts(op_2)$ .

We have shown that  $ts(op_1) \leq \tau_e(op_1) < \tau_s(op_2) \leq ts(op_2)$ , from which we conclude that  $op_1$  appears before  $op_2$  in  $\mathcal{S}$ .

2. The vectors returned by snapshot() operations are ordered. More precisely, let  $op_1$  and  $op_2$  be two snapshot() operations that return  $v_1$  and  $v_2$  respectively.  $op_1$  appears before  $op_2$  in  $\mathcal{S} \Rightarrow v_1 \leq v_2$ .

Since  $op_1$  appears before  $op_2$  in  $\mathcal{S}$ ,  $r_1 = ts(op_1) \leq ts(op_2) = r_2$ . Moreover, by definition of timestamps  $v_1 = Mm\_est^{r_1}$  and  $v_2 = Mm\_est^{r_2}$ . It then follows from Lemma 5 that  $v_1 \leq v_2$ .

3. Let  $op$  be a snapshot() operation that returns  $v$ .  $v[i] = k \Rightarrow$  there are  $k$  write() operations of  $p_i$  that precede  $op$  in  $\mathcal{S}$ .

Let  $op^\ell$  be the  $\ell$ -th write of  $p_i$ . By definition of timestamps,  $ts(op^k) \leq ts(op)$ . Since write() are ordered before snapshot() that have the same timestamp,  $op^k$  is ordered before  $op$  in  $\mathcal{S}$ . Moreover, due to the first item above,  $op^1, \dots, op^{k-1}$  are ordered before  $op^k$  in  $\mathcal{S}$ .

4. Let  $op$  be a  $\text{snapshot}()$  operation that returns  $v$ .  $op$  appears after  $k$   $\text{write}()$  operations of  $p_i$  in  $\mathcal{S}$  implies that  $v[i] \geq k$ .  
 Let  $op'$  be the  $k$ -th write of  $p_i$ . Since  $op'$  appears before  $op$  in  $\mathcal{S}$ ,  $r' = ts(op') \leq ts(op) = r$ . Due to the definition of the timestamps of the  $\text{write}()$  operations,  $r'$  is such that  $Mm\_est^{r'}[i] = k$ . Similarly, due to the definition of the timestamps of the  $\text{snapshot}()$  operations,  $Mm\_est^r = v$ . As  $r' \leq r$ , it follows from Lemma 5 that  $Mm\_est^{r'} \leq Mm\_est^r$ , from which we conclude that  $k \leq v[i]$ .

□*Lemma 7*

The first part of the proof has addressed the “safety” of simulation. In what follows, we address the “liveness” part by showing that all invocations  $\text{simulate}(op)$  issued by the processes  $\in \text{Correct}_{IIS}$  terminate.

**Lemma 8** *Let  $r \geq n + 1$ .  $\exists \rho' : r - n \leq \rho' \leq r$  such that  $\text{smi}n_i^r = \text{smi}n_i^{\rho'}$ .*

**Proof** The proof is by induction on the size of the union of the smallest snapshots. Let  $HR(k)$  be the following property :

$$\left| \bigcup_{r-k \leq \rho \leq r} \text{smi}n_i^\rho \right| \leq k \Rightarrow \exists \rho' \in \{r-k, \dots, r\} : \text{smi}n_i^r = \text{smi}n_i^{\rho'}$$

This property states that when the smallest snapshots of  $k$  consecutive rounds are included in a set of size  $k$ ,  $p_i$  is able to identify at least one of them by the end of round  $k + 1$ . Hence, if we prove  $HR(n)$  we prove the Lemma.

- $HR(1)$ .  $\exists j$  such that  $\text{smi}n_i^{r-1} = \text{smi}n_i^r = \{j\}$ . At the end of round  $r - 1$ ,  $\text{view}_j[r - 1] = \langle j, \{j\} \rangle$ . At round  $r$ ,  $\text{view}_j$  is observed by all processes since the smallest round of that round is  $\{j\}$ . Consequently,  $p_i$  includes  $\langle j, \{j\} \rangle$  in  $\text{view}_i[r - 1]$  and identifies  $\{j\}$  as the smallest snapshot of round  $r - 1$ .
- $HR(k) \Rightarrow HR(k + 1)$ .
  - $\forall j \in \text{smi}n_i^{r-k-1} : \exists \rho \in \{r-k, \dots, r\}$  such that  $j \in \text{smi}n_i^\rho$ . In other words, each  $j$  in the smallest snapshot of round  $r - k - 1$  appears again least once in the smallest snapshots of rounds  $r - k, \dots, r$ .  
 During such a round,  $p_i$  observes  $p_j$ 's snapshot of round  $r - k - 1$ . It then follows that by the end of round  $r$ ,  $\text{view}_i[r - k - 1]$  contains  $\langle j, \text{sm}_j^{r-k-1} \rangle$  for each  $j \in \text{smi}n_i^{r-k-1}$ . Consequently,  $p_i$  identifies the smallest snapshot of round  $r - k - 1$  while executing line 7 during round  $r$ .
  - In the other case,  $\exists j \in \text{smi}n_i^{r-k-1}$  such that  $\forall \rho \in \{r-k, \dots, r\} : j \notin \text{smi}n_i^\rho$ . Thus,  $j \notin \bigcup_{r-k \leq \rho \leq r} \text{smi}n_i^\rho$ , from which we obtain  $|\bigcup_{r-k \leq \rho \leq r} \text{smi}n_i^\rho| \leq k$ . Consequently, in that case  $HR(k + 1)$  follows from  $HR(k)$ .

□*Lemma 8*

**Lemma 9** *Let  $p_i \in \text{Correct}_{IIS}$ . Each invocation of  $\text{simulate}(op)$  issued by  $p_i$  terminates.*

**Proof** Let us assume for contradiction that the invocation  $\text{simulate}(op)$  issued by  $p_i$  does not terminate. Let  $r_0$  be the round at which  $\text{simulate}(op)$  starts and let  $k = \text{est}_i[i]$  when  $p_i$  starts the repeat loop while simulating  $op$ .

As the processes update their estimates by taking the component-wise maximum of the estimates they see, it follows from the definition of  $\text{Correct}_{IIS}$  (see section A) that after some round  $r_1$  all processes  $p_j \in \text{Correct}_{IIS}$  are such that  $\text{est}_j[i] = k$ . Moreover, there exists a round  $r_2$  such that the smallest snapshots that are returned from  $IS[r_2], IS[r_2 + 1], \dots$  contain only identities of processes that belongs to  $\text{Correct}_{IIS}$  (Lemma 1). Finally, there is some round  $r \geq r_0$  during which  $p_i$  discovers the smallest immediate snapshot returned at some round  $r' \geq \max(r_1, r_2, r_0)$  (Lemma 8). Consequently,  $p_i$  computes during round  $r$  a snapshot of the shared memory  $\text{last\_snap}_i$  such that  $\text{last\_snap}_i[i] = k$  (lines 7-10) and then completes the simulation of  $op$  (line 11): a contradiction. □*Lemma 9*

### C.2.2 Simulation of the failure detector queries $\text{fd\_query}()$

Let us assume that in the  $IRIS(PR_{\diamond \mathcal{S}_x})$  model, processes simulate  $\text{fd\_query}()$  by invoking  $\text{simulate}(\text{fd\_query}())$  (Algorithm 7). The simulation is correct if there exists a failure pattern  $\text{fp}_{rw}$  and a timing of operations such that the outputs of  $\text{fd\_query}()$  respect the specification of the class  $\diamond \mathcal{S}_x$ .

To simulate an operation  $op$ , a process accesses one object  $IS[r]$  and computes the output of the failure detector from the view it gets back from that object. Hence, let us define the time  $\tau(op)$  at which the query occurs to be  $r$ , the rank of the immediate snapshot object accessed to compute the failure detector output.

Let us define failure pattern  $fp_{rw}$  as follows. If the  $IRIS(PR_{\diamond S_x})$  run is finite, all processes are correct in  $fp_{rw}$ . If the run is infinite (processes keep simulating  $fd\_query()$  operations), let  $R$  be the smallest round such that  $\forall r \geq R$  : (1) the smallest view returned at round  $r$  contained only identities of processes in  $Correct_{IIS}$  and, (2) at round  $r$  the view obtained by each process in  $Correct_{IIS}$  contains only identities of processes in  $Correct_{IIS}$ . Due to Lemma 1, such a round exists. In  $fp_{rw}$ , the set of correct processes is exactly  $Correct_{IIS}$ . All faulty processes fail at time  $\tau_f = R$  (let us remind that the round numbers in  $IRIS(PR_{\diamond S_x})$  do correspond to the time instants in the simulated read/write model).

**Lemma 10** *The outputs generated by the algorithm described in Figure 7 satisfy the specification of the class  $\diamond S_x$  with respect to  $fp_{rw}$ .*

**Proof** As the properties of the class  $\diamond S_x$  are eventual, the outputs of the algorithm are always valid in any finite run. Let us assume that some processes simulate infinitely many  $fd\_query()$  operations.

We first checks that, at each correct process  $p_i$ , the set  $TRUSTED_i$  satisfies strong completeness. After time  $\tau_f = R$ , each correct process observes only correct process in its view  $sm_i$  (by definition of  $fp_{rw}$  and Lemma 1). Therefore, the sets  $TRUSTED_i, i \in Correct_{IIS}$  eventually contains only identities of correct processes.

For limited scope eventual weak accuracy, the property  $PR_{\diamond S_x}$  states that there is a set  $Q$  of  $x$  processes containing a process  $p_\ell$  and a round  $r$ , such that, for any round  $r' \geq r$ , for any  $p_i \in Q - \{p_\ell\}$ ,  $sm_\ell^{r'} \subsetneq sm_i^{r'}$ . We consider two cases.

- $Q \cap Correct_{IIS} \neq \emptyset$ . In that case, the eventual limited accuracy property directly follows from  $PR_{\diamond S_x}$  and the code of the algorithm.
- $Q \cap Correct_{IIS} = \emptyset$ . Let  $p_i$  be an arbitrary correct process. The limited scope weak accuracy property is satisfied for the pair  $\{p_i, Q\}$ . (This property requires that each correct process in  $Q$  eventually trusts forever the same correct process  $p_i$ . As  $Q$  contains only faulty processes, the property is trivially satisfied.)  $\square_{Lemma 10}$

### C.2.3 General simulation

This final part of the proof shows that any task solvable in the read/write model with  $\diamond S_x$  is solvable in the  $IRIS(PR_{\diamond S_x})$ . Let us assume that processes simulate an algorithm  $\mathcal{A}$  designed for the read/write model equipped with  $\diamond S_x$ . In the first part of the proof, we have established that simulated  $write()$  and  $snapshot()$  operations are linearizable (Lemma 7) and every invocation of  $simulate(op)$  by processes in  $Correct_{IIS}$  terminates (Lemma 9). In the second part, we have shown that there is a failure pattern  $fp_{rw}$  such that the outputs of  $fd\_query()$  are valid with respect to that failure pattern. Moreover, the correct processes in  $fp_{rw}$  are exactly the correct processes of the  $IRIS(PR_{\diamond S_x})$  run.

It remains to show that the view of failures through  $snapshot()$  operations and  $fd\_query()$  are consistent.

**Lemma 11** *Let  $p_i$  be a faulty processes according to  $fp_{rw}$ . Let  $R$  be the time at which  $p_i$  fails. Assuming  $p_i$  starts simulating its  $k$ -th write operation at some round  $\geq R$ . For every  $v$  returns as a result of  $simulate(snapshot())$ ,  $v[i] < k$ .*

**Proof** Assume for contradiction that there exists a vector  $v$  returned by a  $snapshot()$  operation such that  $v[i] \geq k$ . It then follows that there is a round  $r_0$  such that  $Mm\_est^{r_0}[i] \geq k$ , where  $Mm\_est^{r_0}$  is the maximum component-wise of the estimates contained in  $sm_i^{r_0}$  (lines 7-13). However,  $Mm\_est^r[i] \geq k$  is always due to the fact that  $p_i$  has executed  $est_i[i] \leftarrow k$  at line 1. This only happens when  $p_i$  starts simulating its  $k$ -th  $write()$  operation. Consequently  $r_0 \geq R$  (observation O1).

By definition of  $fp_{rw}$ ,  $R$  is such that (observation O2)  $\forall r \geq R$  :  $sm_i^r \subseteq Correct_{IIS}$ , and (observation O3)  $\forall r \geq R, \forall j \in Correct_{IIS} : sm_j^r \subseteq Correct_{IIS}$ . As  $r_0 \geq R$  (O1), the smallest view  $sm_i^{r_0}$  contains only estimate of correct processes (O2). It then follows that between the round  $\geq R$  at which  $p_i$  introduces the new value  $k$  and round  $r_0$ ,  $p_i$  has been observed directly or indirectly by a correct process  $p_j$ . As a correct process observes only correct processes in their views after round  $R$  (O3), this cannot happen.  $\square_{Lemma 11}$

**Lemma 12** *Let  $T$  be a decision task. If  $T$  is wait-free solvable in the read/write model with  $\diamond S_x \Rightarrow$ , it is solvable in  $IRIS(PR_{\diamond S_x})$ .*

**Proof** Let  $\mathcal{A}$  be an algorithm that solves  $T$  in the read/write model. To solve  $T$  in the  $IRIS(PR_{\diamond\mathcal{S}_x})$  model, each process simulates  $\text{write}()$ ,  $\text{snapshot}()$  and  $\text{fd\_query}()$  operations of  $\mathcal{A}$  by running in parallel Algorithms 6 and 7.

The simulation produces an admissible run of the read/write model equipped with a failure detector of the class  $\diamond\mathcal{S}_x$  in which the correct processes are exactly the correct processes in the run of the  $IRIS(PR_{\diamond\mathcal{S}_x})$  model. More precisely, this follows from the following facts :

- There is a linearization of  $\text{write}()$  and  $\text{snapshot}()$  operations that respects the semantic of the read/write shared memory model (Lemma 7).
- Every operation simulated by each process  $\in \text{Correct}_{IIS}$  terminates (Lemma 9).
- There is a failure pattern in which correct processes are exactly processes  $\in \text{Correct}_{IIS}$  such that the outputs of  $\text{fd\_query}()$  satisfy the specification of the class  $\diamond\mathcal{S}_x$  with respect to that failure pattern (Lemma 10).
- The successive states of the simulated shared memory is consistent with that failure pattern (Lemma 11).

Consequently, each process  $\in \text{Correct}_{IIS}$  decides and the decisions obtained satisfy the specification of  $T$ .  $\square_{\text{Lemma 12}}$

By piecing together Lemma 4 and Lemma 12, we obtain the equivalence theorem.

**Theorem 1** Let  $T$  be an agreement task.  $T$  is solvable in the read/write model with  $\diamond\mathcal{S}_x$  if and only if  $T$  is solvable in  $IRIS(PR_{\diamond\mathcal{S}_x})$ .