The Iterated Restricted Immediate Snapshot Model

Sergio Rajsbaum[1], Michel Raynal[2], and Corentin Travers[3]

[1] Instituto de Matemáticas, UNAM, D.F. 04510, Mexico
[2] IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France
[3] Facultad de Informática, UPM, Madrid, Spain
rajsbaum@math.unam.mx, raynal@irisa.fr, ctravers@fi.upm.es

Abstract. In the *Iterated Immediate Snapshot* model (*IIS*) the memory consists of a sequence of one-shot *Immediate Snapshot* (*IS*) objects. Processes access the sequence of *IS* objects, one-by-one, asynchronously, in a *wait-free* manner; any number of processes can crash. 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.

This paper extends 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*).

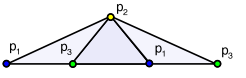
1 Introduction

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].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., [3],[2],[6]. 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., [12],[15]).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.

X. Hu and J. Wang (Eds.): COCOON 2008, LNCS 5092, pp. 487–497, 2008.

Two bedrocks: wait-freedom and round-based execution It has been discovered [6],[16],[21] 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.



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. 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 payed 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 $i + 1$ 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 [16], and was instrumental for the results in [13].

Context and Goals of the Paper. The paper introduces 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, new, simple impossibility results for set agreement in several partially synchronous systems are derived.

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 [11], or by a failure detector [8]. 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 [22]. Other works have considered round-based models with no failure detectors [12]. Some other works [17] 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 [9] or k-set agreement [24]. Set agreement [10] 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],[16],[21]. However, a clear view of what exactly "degree of synchrony" means is still lacking. For example, the same power as far as solving k-set agreement can be achieved in various ways, such as via different failure detectors [18] 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. A *failure detector* [8] is 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., [18],[24]).

As an example, this paper focuses on the family of *limited scope* accuracy failure detectors, denoted $\diamond\mathcal{S}_x$ [14],[23]. 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 [8] ($\diamond\mathcal{S}_n$ is $\diamond\mathcal{S}$). Informally, a failure detector $\diamond\mathcal{S}_x$ ensures that there is a correct process that is eventually never erroneously suspected by any process in a cluster of x processes.

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 [14] 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 [14] combinatorial topology techniques introduced in [15] are used to derive the topological properties of the runs of the system enriched with C directly[1].

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],[8].

2.1 Shared Memory Model Enriched with a F.D. of the Class $\diamond\mathcal{S}_x$

The paper considers a standard asynchronous system made up of n processes, $p_1, \ldots, p_n$, 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. 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 failure detector of the class $\diamond\mathcal{S}_x$, where $1 \leq x \leq n$, provides each process p_i 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 $j \in \text{TRUSTED}_i$ we say "p_i trusts p_j". 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 and, (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 following equivalent formulation of $\diamond\mathcal{S}_x$ [18] is used in Section 4, 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 $i \notin Q \Rightarrow \text{REPR}_i = i$.

2.2 The Iterated Immediate Snapshot (*IIS*) Model

A *one-shot immediate snapshot* object *IS* is accessed with a a single operation denoted write_snap(). Intuitively, when a process p_i invokes write_snap(v) it is

[1] A companion technical report [19] extends the results presented here to other failure detector classes.

as if it instantaneously executes a write $IS[i] \leftarrow v$ operation followed by an IS.snapshot() operation.

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. A view sm_i is 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 $sm_i = \emptyset$, if the process p_i never invokes write_snap() on the corresponding object. The three properties are :(Self-inclusion) $\forall i : (i, v_i) \in sm_i$, (Containment) $\forall i, j : sm_i \subseteq sm_j \ \vee \ sm_j \subseteq sm_i$ and, (Immediacy) $\forall i, j : (i, v_i) \in sm_j \ \Rightarrow sm_i \subseteq sm_j$.

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.

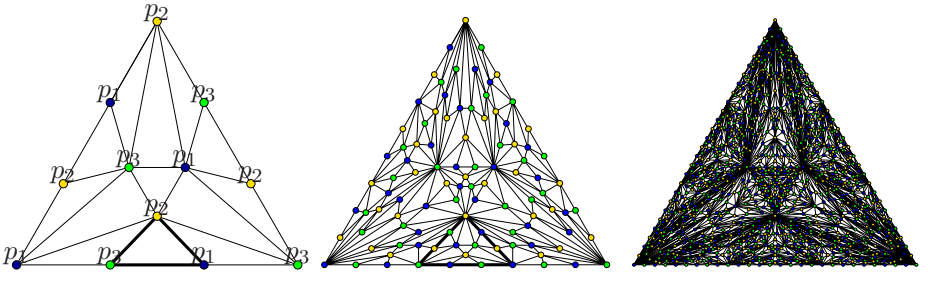


Fig. 1. One, two and three rounds in the *IIS* model

In the *iterated immediate snapshot model* (*IIS*) the shared memory is made up of an infinite number of one-shot immediate snapshot objects $IS[1], IS[2], \ldots$ These objects are accessed sequentially and asynchronously by each process. In Figure 1 one can see that the *IIS* complex 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 is given in [20]).

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. 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 $C = \diamond\mathcal{S}_x$

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. The property states that there is a set Q of x processes containing a process p_ℓ 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 $PR_{\diamond\mathcal{S}_x}$ is defined as follows:

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

Figure 2 shows runs of the $IRIS(PR_{\diamond\mathcal{S}_x})$ model for $x = 2$. The complex remains connected in this case and consequently consensus is unsolvable in that model.

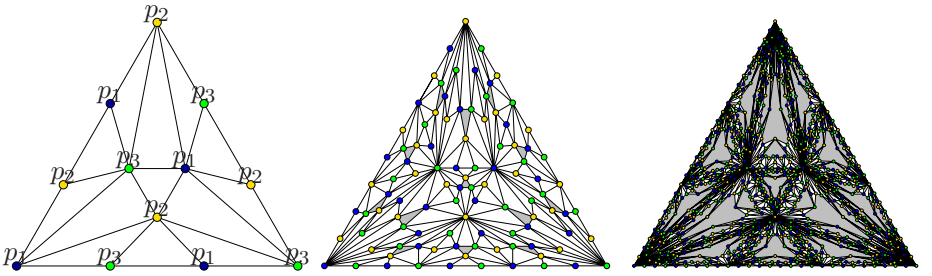


Fig. 2. One, two and three rounds in $IRIS(PR_{\diamond\mathcal{S}_x})$ with $x = 2$ and $r = 2$

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

We prove this theorem in Section 4 by providing a transformation from the read/write model enriched with $\diamond\mathcal{S}_x$ to the $IRIS(PR_{\diamond\mathcal{S}_x})$ model and the inverse transformation from the $IRIS(PR_{\diamond\mathcal{S}_x})$ model to the read/write model with $\diamond\mathcal{S}_x$.

3.2 The k-Set Agreement with $\diamond\mathcal{S}_x$

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 $\diamond\mathcal{S}_x$.

Theorem 2. *In the read/write shared memory model, in which any number of processes may crash, there is no $\diamond\mathcal{S}_x$-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 $L = \{p_1, \ldots, p_{n-x+1}\}$ and the high-order processes $H = \{p_{n-x+2}, \ldots, p_n\}$, and consider all *IIS* runs where the processes in H never take any steps, these runs trivially satisfy the $PR_{\diamond\mathcal{S}_x}$ 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 $IRIS(PR_{\diamond\mathcal{S}_x})$ model. By Theorem 1 it is unsolvable in the read/write shared memory model equipped with a failure detector of the class $\diamond\mathcal{S}_x$.[2]

4 Simulations

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

[2] In the full paper, we show how to re-derive the more general result of [14] in the *IRIS* framework.

```
operation IS[r].WRITE_SNAPSHOT()(< i, v_i >):
(1)   repeat m_i ← R[r].snapshot(); rp_i ← REPR_i
(2)      until ((< rp_i, − >∈ m_i) ∨ rp_i = i)) end repeat;
(3)   sm_i ← R[r].write_snap(< i, v_i >);
(4)   return (sm_i).
```

Fig. 3. From the read/write model with $\diamond\mathcal{S}_x$ to the $IRIS(PR_{\diamond\mathcal{S}_x})$ model (code for p_i)

described in Figure 3. It uses the $\diamond\mathcal{S}_x$ version based on the representative variable REPR_i. Each round r is associated with an immediate snapshot object $R[r]$ that can in addition be read in snapshot. Sets returned by $R[r]$.snapshot() and $R[r]$.write_snap() are ordered by containment, and the operations can be consistently ordered. Objects $R[r]$ can be wait-free implemented from base read/write operations [1],[5].

At each round r, each process p_i repeatedly reads $R[r]$ until it observes its representative has already written or it discovers that it is its own representative ($rp_i = i$). This simple rule guarantees that processes that share the same representative p_ℓ eventually always return a view sm that contains the view returned by p_ℓ. Thus, the set of sequences of views produced by the algorithm satisfies the property $PR_{\diamond\mathcal{S}_x}$.

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 write() and 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 write() *and* snapshot() *Operations.* The algorithm described in Figure 4 is based on the ideas of the simulation of [7]. Without loss of generality, we assume that (as in [7]) the kth 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 simulate(snapshot()) should be ordered and contain the integers written by the last simulate(write()) that precedes it.

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 $\max_{\text{CW}}$, 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

init $r_i \leftarrow 0$; $last\_snap_i[1..n] \leftarrow [-1,\ldots,-1]$; $est_i[1..n] \leftarrow [0,\ldots,0]$; $view_i[1..] \leftarrow [\emptyset,..]$
function simulate($op()$) % $op \in \{$write(), snapshot()$\}$
(1) **if** $op()$ = write() **then** $est_i[i] \leftarrow est_i[i]+1$ **endif**; $r\_start_i \leftarrow r_i$;
(2) **repeat** $r_i \leftarrow r_i + 1$;
(3) $sm_i \leftarrow IS[r_i]$.WRITE_SNAPSHOT$(< i, est_i, view_i[1..(r_i-1)] >)$;
(4) $view_i[r_i] \leftarrow \{ < i, \{< j, est_j > \textbf{ such that } < j, est_j, - >\in sm_i\} > \}$;
(5) **for each** $\rho \in \{1,\ldots,r_i-1\}$ **do**
$view_i[\rho] \leftarrow \bigcup_{view_j \textbf{ such that } <j,-,view_j>\in sm_i} view_j[\rho]$ **endfor**;
(6) $est_i \leftarrow \max_{\mathrm{CW}}\{est_j \textbf{ such that } < j, est_j, - >\in sm_i\}$;
(7) **if** $\big(\exists \rho > r\_start_i \mid \exists < -, smin >: \forall j \in smin : \ < j, smin > \in view_i[\rho]\big)$
% there is a smallest snapshot in $view_i[r\_start_i+1..r_i]$ known by p_i
(8) **then let** ρ' **be** the greatest round $\leq r_i$ that satisfies predicate of line 7;
(9) $smin_i \leftarrow$ the smallest snapshot in $view_i[\rho']$;
(10) $last\_snap_i \leftarrow \max_{\mathrm{CW}}\{est_j \textbf{ such that } < j, est_j >\in smin_i\}$;
(11) **if** $last\_snap_i[i] = est_i[i]$ **then**
(12) **if** op = snapshot() **then** return $(last\_snap_i)$ **else** return() **endif endif**
(13) **endif endrepeat**

Fig. 4. Simulation of the write() and snapshot() operations in $IRIS(PR_{\diamond\mathcal{S}_x})$ (p_i's code)

that appear in the smallest view (denoted $smin^r$) returned by $IS[r]$. Due to the fact that estimates are updated maximum component-wise, it follows from the containment property of views that $\forall r, r' : r < r' \Rightarrow sm\_min^r \leq sm\_min^{r'}$. 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.

In order to determine smallest views, each process p_i maintains an array $view_i[1,..]$ 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 $smin_i$ (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.

From $IRIS(PR_{\diamond\mathcal{S}_x})$ to a Failure Detector of the Class $\diamond\mathcal{S}_x$. 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 $\diamond\mathcal{S}_x$-queries in the $IRIS(PR_{\diamond\mathcal{S}_x})$ is described in the figure on the right.

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$_i$)

Simulation of fd_query() in $IRIS(PR_{\diamond\mathcal{S}_x})$

General Simulation. Given an algorithm $\mathcal{A}$ that solves a task T in the read/write model equipped with $\diamond\mathcal{S}_x$, we show how to solve T in the $IRIS(PR_{\diamond\mathcal{S}_x})$ model. Algorithm $\mathcal{A}$ performs local computation, write(), snapshot() and fd_query(). In the $IRIS(PR_{\diamond\mathcal{S}_x})$ model, processes run in parallel the algorithms described in Figures 4 and below in order to simulate these operations. More precisely, whatever the operation $op \in \{\mathsf{write}(), \mathsf{snapshot}(), \mathsf{fd\_query}()\}$ being simulated, each immediate snapshot object is used to update both the estimate of the shared memory and the output of the failure detector.

The simulations are proved correct in [20]. Theorem 1 then follows from the two simulations presented in Section 4.1 and Section 4.2.

References

1. Afek, Y., Attiya, H., Dolev, D., Gafni, E., Merritt, M., Shavit, N.: Atomic Snapshots of Shared Memory. J. ACM 40(4), 873–890 (1993)
2. Attiya, H., Bar-Noy, A., Dolev, D.: Sharing Memory Robustly in Message Passing Systems. J. ACM 42(1), 124–142 (1995)
3. Awerbuch, B.: Complexity of network synchronization. J. ACM 32, 804–823 (1985)
4. Attiya, H., Welch, J.: Distributed Computing: Fundamentals, Simulations, and Advanced Topics. Wiley, Chichester (2004)
5. Borowsky, E., Gafni, E.: Immediate Atomic Snapshots and Fast Renaming. In: Proc. of PODC 1993, pp. 41–51 (1993)
6. Borowsky, E., Gafni, E.: Generalized FLP Impossibility Results for t-Resilient Asynchronous Computations. In: Proc. 25th ACM STOC, pp. 91–100 (1993)
7. Borowsky, E., Gafni, E.: A Simple Algorithmically Reasoned Characterization of Wait-free Computations. In: Proc. 16th ACM PODC, pp. 189–198 (1997)
8. Chandra, T., Toueg, S.: Unreliable Failure Detectors for Reliable Distributed Systems. J. ACM 43(2), 225–267 (1996)
9. Chandra, T., Hadzilacos, V., Toueg, S.: The Weakest Failure Detector for Solving Consensus. J. ACM 43(4), 685–722 (1996)
10. Chaudhuri, S.: More Choices Allow More Faults: Set Consensus Problems in Totally Asynchronous Systems. Information and Computation 105, 132–158 (1993)
11. Dwork, C., Lynch, N., Stockmeyer, L.: Consensus in the Presence of Partial Synchrony. J. ACM 35(2), 288–323 (1988)
12. Gafni, E.: Round-by-round Fault Detectors: Unifying Synchrony and Asynchrony. In: Proc. 17th ACM Symp. on Principles of Distributed Computing, pp. 143–152 (1998)
13. Gafni, E., Rajsbaum, S., Herlihy, M.: Subconsensus Tasks: Renaming is Weaker than Set Agreement. In: Dolev, S. (ed.) DISC 2006. LNCS, vol. 4167, pp. 329–338. Springer, Heidelberg (2006)
14. Herlihy, M., Penso, L.D.: Tight Bounds for k-Set Agreement with Limited Scope Accuracy Failure Detectors. Distributed Computing 18(2), 157–166 (2005)
15. Herlihy, M.P., Rajsbaum, S., Tuttle, M.: Unifying Synchronous and Asynchronous Message-Passing Models. In: Proc. 17th ACM PODC, pp. 133–142 (1998)
16. Herlihy, M., Shavit, N.: The Topological Structure of Asynchronous Computability. J. ACM 46(6), 858–923 (1999)
17. Keidar, I., Shraer, A.: Timeliness, Failure-detectors, and Consensus Performance. In: Proc. 25th ACM PODC, pp. 169–178 (2006)

18. Mostefaoui, A., Rajsbaum, S., Raynal, M., Travers, C.: Irreducibility and Additivity of Set Agreement-oriented Failure Detector Classes. In: Proc. PODC 2006, pp. 153–162 (2006)
19. Rajsbaum, S., Raynal, M., Travers, C.: Failure Detectors as Schedulers. Tech Report # 1838, IRISA, Université de Rennes, France (2007)
20. Rajsbaum, S., Raynal, M., Travers, C.: The Iterated Restricted Immediate Snapshot Model. Tech Report # 1874, IRISA, Université de Rennes, France (2007)
21. Saks, M., Zaharoglou, F.: Wait-Free k-Set Agreement is Impossible: The Topology of Public Knowledge. SIAM Journal on Computing 29(5), 1449–1483 (2000)
22. Völzer, H.: On Conspiracies and Hyperfairness in Distributed Computing. In: Fraigniaud, P. (ed.) DISC 2005. LNCS, vol. 3724, pp. 33–47. Springer, Heidelberg (2005)
23. Yang, J., Neiger, G., Gafni, E.: Structured Derivations of Consensus Algorithms for Failure Detectors. In: Proc. 17th ACM PODC, pp. 297–308 (1998)
24. Zieliński, P.: Anti-Omega: the Weakest Failure Detector for Set Agreement. Tech Rep # 694, University of Cambridge (2007)