
VNIVERSIDAD NACIONAL AVTNOMA DE MEXICO

UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

**POSGRADO EN CIENCIA E INGENIERÍA DE LA COMPUTACIÓN**

# STUDIES ON RECURSIVE DISTRIBUTED ALGORITHMS

T E S I S

QUE PARA OBTENER EL GRADO DE:

**MAESTRO EN CIENCIAS**

**P R E S E N T A:**

**DIEGO RÁBAGO ARREDONDO**

**DIRECTOR DE TESIS: DR. SERGIO RAJSBAUM**

**MÉXICO, D.F.** **2010.**

# Table of contents

# List of Figures

# List of Algorithms

**Resumen**

El uso de la recursión en el diseño y análisis de los algoritmos distribuidos ha sido muy escaso a pesar de estar relacionado con los modelos iterados de cómputo distribuido; en los que diferentes procesos corren por rondas invocando distintas instancias de objetos compartidos. En este trabajo se presenta un estudio sobre la utilización de la recursión como método para implementar las tareas conocidas como *atomic snapshot* e *immediate atomic snapshot*, analizando las particularidades de un conjunto de algoritmos propuestos que buscan obtener una implementación óptima. En primer lugar, se caracterizan las vistas de salida que se obtienen al realizar una lectura seguida de un escaneo sobre un arreglo compartido de registros atómicos de lectura/escritura por parte de un conjunto de procesos asíncronos, las cuales representan un bloque constructor fundamental para muchos protocolos sobre el modelo iterado de lectura/escritura, y también para los algoritmos presentados aquí. Dichas vistas se caracterizan primero con un enfoque conjuntista y después, por medio de una representación matricial, que está a su vez asociada a las propias ejecuciones.

También presentamos un análisis sobre los algoritmos recursivos que nos lleva a probar que una implementación para la tarea de snapshot en el modelo iterado de lectura/escritura no puede ser tan eficiente –en términos de su complejidad en accesos compartidos– como las implementaciones existentes en la versión no iterada del mismo modelo. La prueba que se presenta para el caso particular con tres procesos, utiliza un argumento de conectividad sobre el complejo simplicial del protocolo; herramienta que pudiera ser adecuada para obtener una futura generalización para cualquier número de procesos.

**Abstract**

The use of recursion in the design and analysis of distributed algorithms has been scarce even though it's deeply related to the iterated models of distributed computing, where processes run in rounds accessing different instances of shared object implementations. This work presents a study on the use of recursion as a method to implement the *atomic snapshot* and *immediate atomic snapshot* tasks, observing particularities on a set of attempts to obtain a most efficient implementation. We first characterize the output views of a write/scan task over a shared array of atomic registers invoked by a set of asynchronous processes, which is a basic constructing block not only in many read/write iterated distributed executions but also in the recursive algorithms presented here. Such views are first characterized in a set-wise fashion and later by matrix representations associated to the executions themselves.

We also present an analysis on recursive algorithms leading to a proof that an implementation of the snapshot task on the read/write iterated model cannot be as efficient in terms of shared access complexity as the implementations on the non iterated version. The presented proof is for the particular case with three processes and uses a connectivity argument on the protocol complex, tool that may be adequate for a future generalization to any number of processes.

# 1 Introduction

Even though distributed computing has been a recurrent subject of research for many years, in recent times, the study of distributed systems has reemerged as one of the mainstream theoretical fields for current and future IT development. The appearance of multiprocessor architectures has significantly incremented interest in understanding, analyzing and developing distributed tasks and problems, as well as their possible implementations [2, 4, 17, 26, 36].

In accordance, this study focuses in a set of problems considered fundamental in the area, adopting in most cases a paradigm which has been mildly used: recursion. This paradigm has been explored recently by [22]. The main objects of study are the *atomic snapshot* task [1] and its refinement, the *immediate atomic snapshot* task [13, 41], although other common tasks, like *renaming* [5] are explored as well.

## 1.1 Context

In distributed systems, processes communicate through shared objects that comply to a certain specification, depending of the functionality of the particular object. In the simplest case, the shared object is an array of multi-reader/single-writer (or *MRSW*) registers which a process can access, either to write a value into its assigned register or to read any of the registers in the array. Other shared objects can then be implemented by constructing them from MRSW registers or by the use of more powerful primitives. Any object, however powerful, can be described by its sequential specification together with its safety and liveness properties. Safety properties guarantee that potentially dangerous scenarios will not occur and are therefore associated to the correctness of the implementation, liveness properties specify the way in which invocations

to the object's operations make progress.

One-shot objects are a particular type of shared objects that receive at most one invocation by each participating process, these objects can be specified by a *task*, which is basically an input/output relation between input vectors and related output vectors. Tasks are defined precisely in section 2.1. This work is particularly interested in this kind of objects as they conform much better to recursive implementations. When a set of processes make use of a one-shot shared object, each process invokes the object with an associated private input, the set of these inputs is referred to as the *input vector*, and each process receives in return an output value that complies to the specification of the particular object, the set of outputs compose the *output vector*. As an example, a one-shot object that implements the well known *consensus* task [35, 19], receives a different private input value proposed from each invoking process, suppose that each process proposes its own unique identifier. Then, in order to comply with the task specification, the object should return the same output value to every invoking process and this value should be contained in the proposed input vector.

**Atomic snapshots** The atomic snapshot object was introduced in [1, 3] and has become an important primitive for shared memory distributed systems as it is deemed fundamental for the design and verification of wait-free algorithms. A snapshot object is a shared data structure consisting of various segments, usually atomic MRSW registers, which processors can access to either update one of these segments or scan the whole structure atomically. The main contribution of this object is that it allows a process to obtain a trustworthy image of the global state of an execution by avoiding possible interference from other updating processes.

Many wait-free algorithms implementing the atomic snapshot task using MRSW atomic registers have been proposed. When taking into account deterministic implementations which only use MRSW registers as primitives, Anderson [3] presented an algorithm to solve the atomic snapshot task with exponential complexity of $O(2^n)$. In [1], the authors present an elegant wait-free algorithm with $O(n^2)$ complexity that is still used in recent textbooks as introduction to atomic snapshots [26]. Later, Attiya and Rachman [8] lowered the bound to $O(nlogn)$, which after applying the transformation by Israeli et al. [29] stays even today as the most efficient deterministic implementation with MRSW registers. This transformation implements

the atomic snapshot task with $O(n)$ steps for updates and $O(nlogn)$ for scans by using a general technique to reduce an implementation using $O(f(n))$ operations for either a scan or an update into one that uses $O(f(n))$ for a scan and $O(n)$ for an update (or viceversa, i.e. $O(f(n))$ per update and $O(n)$ per scan). Kirousis, Spirakis and Tsigas [30] present a variation to the problem which separates processes into scanners and updaters and give an implementation which uses linear space on the number of processes, $O(n)$ operations for each scan and $O(1)$ operations for updates. Randomized solutions have also been implemented, starting with the algorithm in [7] by Attiya, Herlihy and Rachman and the one by Chandra and Dwork [16], both with a complexity of $O(nlog^2n)$. Finally, many implementations have been presented with linear or even sublinear complexity bounds by using more powerful primitives instead of MRSW registers, *e.g.* [7, 40]. Another version of the problem allows each process to write any of the segments, this are referred as multiple writer snapshot algorithms [1, 3, 18], or take partial snapshots which only scan a subset of the segments [6, 28]. This works restricts itself to the MRSW version of the problem.

**Immediate atomic snapshot** This task is a refinement of the atomic snapshot in which a scan is scheduled *immediately* after every update. Similarly to the atomic snapshot object, the immediate atomic snapshot object has also been subject of intensive study. However, in this case, much more interest has been shown towards the *immediate snapshot memory model* (described in section 2.4) and not so much in the implementations of the task itself. This refinement of the atomic snapshot reduces the problem of interference even more by offering just one operation to any invoking process, namely the $ImmSnap()$ operation, which implements the abstraction that when a process updates a value in the shared memory it also, instantaneously, obtains a snapshot of the whole array. The task was first proposed simultaneously by Borowsky and Gafni in [13] as the *participating set problem* and by Saks and Zaharoglou in [41] as *block executions*. In [13], the authors present a protocol to solve the task in $O(n^2)$. This protocol was later adapted by Gafni and Rajsbaum into a recursive setting in [22] leaving the operation's complexity unaltered in terms of shared memory accesses. It is clear that, by being a solution to the restriction of the atomic snapshot task, these implementations are also solutions for this last, more general task. We are particularly interested in the one-shot setting of the task, which is a particularization in which each process can access the implementing object only once, *i.e.* can only invoke $ImmSnap$ once. The use of this one-shot implementations lead to the *Iterated*

*Immediate Snapshot Model* (IS) of distributed computing which is also described later in more depth (see section 2.4). The immediate snapshot task has resisted all attempts to obtain an implementation with a complexity below $O(n^2)$ steps.

**Recursion** In [22], the authors propose recursive reasoning as a mechanism to create distributed algorithms which can, additionally, be easily analyzed and understood. This methodology, recognized and used commonly in sequential computing, has found little or no employment in the different distributed settings even though the benefits of recursive structuring have been proposed for a long time [39]. Research in this direction has been encouraged by a number of papers which also propose different recursive algorithms [20, 42]. Some of the recursive distributed algorithms include the Byzantine agreement of Lamport, Shostak and Pease [32], the cloture voting for Byzantine Agreement by Berman, Garay and Perry [11] and, more recently, implementations for the immediate snapshot, renaming and swap tasks by Gafni and Rajsbaum [22].

The recursive approach also allows researchers to study the algorithm's inherent properties by using topological tools and results, which create a link between distributed algorithms and topology, enhancing a new focus for analysis [25]. Additionally, iterated task executions have simple geometrical descriptions when these topological notions are used. If tasks $T_1$ and $T_2$ have topological descriptions as protocol complexes $C_1$ and $C_2$ respectively, then an iterated execution where processes first invoke task $T_1$ and only afterwards invoke task $T_2$, also has a simple topological description which can be obtained by replacing the simplices of complex $C_1$ with the complex $C_2$ [25, 22]. A section with a brief description of some of this preliminary topological notions and their link to distributed computing is offered in section 2.5.

## 1.2 Contents of this work

This work assumes a standard MRSW register shared memory model in an asynchronous wait-free setting where any number of processes participating in an execution can fail by crashing. As we mainly analyze the snapshot and immediate snapshot tasks we, evidently, cannot use objects which implement these tasks as primitives, therefore we rely on a most basic task denoted write/scan or $WScan$. This task abstracts a shared array divided in so many segments as there

are processes; when a process $p_i$ invokes $WScan(v)$ with $v$ as input, the value $v$ is written into the $i$-th segment of the shared array and later $p_i$ reads, in an arbitrary order, the contents of every single segment in the array. To this respect, we present a set-wise characterization of the views returned to the different processes by such task. This simple abstraction of a shared array of registers which may be concurrently written and scanned by a number of asynchronous processes, offers only very general guarantees on the characteristics of the individual views and the relation between different views is not very clear. The views obtained as a result of a $WScan$ invocation can be thought of as pre-snapshot views, as there's no mechanism to avoid interference between concurrent *scanning* and *updating* processes. Some properties over these sets of views are devised in various theorems in section 3. Accordingly, section 3.3 introduces a method to generate matrix representations of executions over a $WScan$ object which can also serve as a characterization for the corresponding views.

Later, in section 4, we present a step by step study of a series of algorithms intended to reduce the complexity bound on the known implementations of the immediate snapshot task in a recursive setting. The original atomic snapshot is also studied under the characteristics of the iterated model using read/write atomic registers, leading to a conjecture that states that snapshots cannot be taken in the IS model in $O(nlogn)$ complexity. The main result is to prove a particular case of this conjecture using a connectivity proof over the simplicial complex that represents the executions for three processes. The contribution of this conjecture to distributed computing is to show that, although the iterated models and its non iterated versions have been shown to be equivalent in terms of computability (*i.e.* a problem solvable in the former model is also solvable in the latter model, and viceversa), there is an associated tradeoff to the use of the iterated models: although the iterated versions of the different protocols may be easier to understand and analyze, its complexity in terms of shared accesses may be higher to that of its non iterated equivalent. This tradeoff stresses on the importance of the *double-collect* used in most of the non iterated implementations of the atomic snapshot, as it is the only section of the usual implementations that cannot be trivially simulated in an iterated context.

# 2 Preliminaries

This chapter presents a very general review of the main issues examined throughout this work, starting from the snapshot objects and tasks up to some introductory topological concepts necessary to understand some of the forthcoming proofs. There are various very good textbooks to extend an understanding of standard notions of distributed computing, *e.g.* [10, 26, 33] are some good sources among many others. As to the topological concepts, some good recommendations include [31, 34]. Finally, the article by Herlihy and Shavit [25] is a source to understand the link between topology and distributed computing. The next section extends and formalizes the description of a *task*.

## 2.1 Tasks

We recall from the introduction that shared objects to which processes can only issue a single invocation are called one-shot objects. The specification of a one-shot object is itself defined by a task, which is a relation between the set of possible inputs to the object and the set of related outputs which are returned by the object. Therefore, a task is a triplet composed by a set $I$ of input vectors, a set $O$ of output vectors and a relation $\Delta$ between both sets[1]. If there are $n$ processes that can invoke the object, each with a respective private input, the relation $\Delta$ associates to the input vector, created from the set of private individual inputs, a compliant output vector. By a compliant output vector we mean the set of output values that the processes decide, one for each process, and as there may be more than one feasible output vector, we say that $\Delta$ is a point-to-set relation. In an execution where the set of participating processes is a subset of the total number of processes, the input vector $\mathcal{I}$ includes the input values $v_i$ of

[1]The concept of task was introduced by Biran, Moran and Zaks in [12]

every participating process $i$ and a default input value, $\perp$, for every non participating process. Analogously, the output vector $\mathcal{O}$ includes a decision value for each participating process and the default value $\perp$ for the non participating processes. In a correct implementation for the task specified by $\Delta$ we should then have that $\mathcal{O} \in \Delta(\mathcal{I})$. For instance, if a shared one-shot object is to implement the consensus task between processes $p_1$, $p_2$, and $p_3$, each of which invoke the object with inputs $x$, $y$, and $z$ respectively, then every correctly deciding process would have to decide on the same value and this must be one of the three proposed ones. So for an execution in which the three processes correctly decide, the input vector is $(x, y, z)$ and the related output vectors are $(x, x, x)$, $(y, y, y)$ and $(z, z, z)$. In a given distributed system, there might be several instances of the same task provided by different shared objects which processes may invoke in a given order.

## 2.2 Atomic snapshots

The atomic snapshot, already mentioned in the introduction, is a shared object that provides two operations, *update(v)* and *scan()*. The object abstracts a shared array divided into $n$ segments with one segment being associated to each different process. A call by a process $p_i$ to the first operation *update(v)* writes the value $v$ into the segment reserved to process $i$ in the shared array. The second operation returns an immediate view of the whole shared array as if every segment had been read simultaneously. Also, the object needs to satisfy a safety property known as *linearizability* which states that every operation on the object must look as if it was executed instantly at some moment between its invocation and return [27], and so, any scan performed after an update must reflect this event consistently in the returned view. It also needs to satisfy a progress property known as *wait-freedom* which was introduced by Herlihy in [24], and states that any operation by a process which doesn't crash must eventually return. These safety and progress properties are standard in this work and are a requirement for every correct algorithm.

The one-shot version of the atomic snapshot object is an implementation of the atomic snapshot task. This in turn is similar to the usual atomic snapshot abstraction only that it allows just one update operation followed certain time later by one scan operation. If we assume that $i$ is the value written by $p_i$ as private input value for its update operation, we have that the set of views obtained by the processes as a result of their scan invocation satisfy two properties

(where we refer to the result of $i$'s scan result as its view and we denote it by $view_i$). The first property, called self-inclusion, states that every process must *see* itself in its obtained view.[2] In symbols:

i) Self-Inclusion: $\forall i, \quad i \in view_i$

The second property known as containment states that the set of returned views is ordered by set inclusion:

ii) Containment: $\forall i, j, \quad view_i \subseteq view_j \quad \vee \quad view_j \subseteq view_i$

## 2.3 Immediate atomic snapshots

The immediate atomic snapshot object is a restriction of the preceding atomic snapshot object. Once again, the object abstracts a shared array with $n$ segments, *i.e.* one segment for each process. The difference is that an object that implements this task should only provide a single operation, the immediate snapshot or *ImmSnap(v)*. The abstraction is that when a process $p_i$ invokes this task with input value $v$, it is as if the value $v$ gets instantaneously written into the $i$-th segment of the shared array and immediately after the write the process $p_i$ obtains a snapshot of the whole shared array. Different invocations should then be linearizable to look as if an update of a register and a scan of the whole array were both made immediately one after the other and at some instant between the operation's invocation and response, and the resulting view should reflect this linearization coherently. Therefore, if two processes concurrently invoke the operation, it is as if they concurrently make an update and immediately after they both, concurrently, perform a scan rendering both processes' identifiers in both views.

If we restrict ourselves to the one-shot version of the object, we can describe it as the immediate snapshot task. The abstraction offered by such a task is similar to that of the multi-shot object with the limitation that each process can invoke the task only once. If we again assume that $i$ is the input value delivered by process $p_i$ and that $view_i$ is its resulting snapshot, then the set of all the views obtained by the different processes in a single execution must satisfy the previous properties of *self-inclusion* and *containment* together with a new property known

[2]Throughout this work we say that process $p_i$ sees process $p_j$ if $p_j \in view_i$

as *immediacy*, which states that if a process has seen another process included in its view, then the whole view of the contained process must be itself totally included in the containing process' view. In symbols:

iii) Immediacy: $\forall i, j, \quad i \in view_j \implies view_i \subseteq view_j$

A useful equivalent restatement of this last property asserts that if two processes see each other in their respective views, then their views must necessarily be identical:

iii') $\forall i, j, \quad ( \ i \in view_j \ \wedge \ j \in view_i \ ) \implies view_i = view_j$

## 2.4 Models

We work on top of the base read/write asynchronous shared memory model of distributed computing, in which a set of sequential and deterministic processes can make use of any number of MRSW atomic registers[3] in a shared memory to communicate with other processes, in addition to their local computations. We denote the set of processes in the model as $\Pi = \{p_1, p_2, \ldots, p_n\}$ where every process is uniquely identified by its subindex. The processes are *asynchronous* which means that there is no bound in the relative processing speed and, given an execution, any number of processes can fail by crashing. We assume that a process that crashes executes its algorithm correctly up to the crashing point and doesn't take more steps after it.

In the *snapshot model* [1] the processes can invoke operations *update*($v$) and *scan*() on a $\infty$-shot[4] snapshot object abstracting the idea that they share a MRSW register array on which they can update their respective registers and take immediate snapshots of the whole array as described earlier. As snapshot objects can be constructed from MRSW atomic registers, this model delivers the exact same computational power as the base model does. However, by hiding the implementation details which take care of the interference generated by concurrency in an asynchronous setting, it provides a useful higher abstraction for solving other problems over

[3]The atomic property of such registers refers to the fact that every read or write operation on them appears to take effect instantaneously and sequentially in between the operation's invocation and response.

[4]On an $\infty$-shot snapshot there is no bound on the number of update and scan operations that a process can invoke.

the same base model. We then consider an iterated version of this model denoted the *iterated snapshot model (IS)* [15], in which processes can update or scan a snapshot object only once. Therefore executions are organized in a round by round basis, in which processes invoke different one-shot snapshot objects in an orderly manner, by iteration number. Therefore, a process will invoke an update and a scan operations on the $i$-th one-shot atomic snapshot object, $i \geq 1$, before issuing invocations to the $(i+1)$-th one-shot snapshot object. It is clear that any problem solvable in IS model is readily solvable in the snapshot model, and the equivalence of both models in terms of wait-free problem solvability was proven by exposing simulations from the snapshot model to the IS model, such simulations are described by Borowski and Gafni in [15] and by Gafni and Rajsbaum in [23]. The benefits of using this model as an abstraction for the design and analysis of distributed algorithms are well studied and thoroughly applied [14, 21, 37, 38] We refer to a particular execution as a *full-information* execution if the value given as input for the first update is a private value but all subsequent update invocations use the result of the previous snapshot operation as input.

The *immediate snapshot model* [1] and the *iterated immediate snapshot model (IIS)* [15] are similar to the snapshot models with the modification obtained by exchanging the snapshot objects for immediate snapshot objects. Therefore, in the IIS model, there is a sequence of immediate snapshot shared objects $IS_1, IS_2, \ldots$ which can receive at most a single $ImmSnap$ invocation by each participating process in the order given by the iteration number. So once again, no process invokes it's operation on the $i$-th immediate snapshot object $IS_i$ if it hasn't returned from it's invocation to all previous snapshot objects $IS_j,\ j < i$. Full-information executions are also defined on these models as executions in which the first input to an $ImmSnap$ operation is a private value but every successive invocations uses the previously obtained snapshot as input.

## 2.5 Topological preliminaries

This section presents a minimal set of definitions from algebraic topology necessary to understand some proofs in later chapters. For a much more extensive and profound treatment of this area and its applications in distributed computing, refer to [9, 25, 31]. A *simplex* is a finite set $\sigma$ of $n+1$ affinely independent vertices. Its *dimension* is given by one less than the number of vertices and is denoted as $dim(\sigma) = n$ or directly as a superscript as in $\sigma^n$. Any simplex

that can be formed from a subset of the vertices in $\sigma$ is called a *face* of $\sigma$, and a face that is formed by a proper subset of vertices is denoted as a proper face. A *simplicial complex* is a set of simplices closed under containment and intersection, the dimension of which is the same as that of its contained simplex with higher dimension. Figure 1 shows a simplicial complex of dimension 2 which is formed by the set of simplices $\{x, y, z\}$, $\{y, z, w\}, \{u, w\}$ and $\{u, v, r\}$. In particular the simplex $\{x, y, z\}$ has dimension 2 and has proper subsimplices $\{x\}, \{y\}, \{z\}$ of dimension 0, $\{x, y\}, \{y, z\}$ and $\{x, z\}$ of dimension 1. Simplices of dimension 2 in figure 1 and in the remaining of this work are colored in light gray to make clear that the simplex is formed by the border (the three edges that form the triangle) and the interior (the light gray colored area). A simplex of dimension 3 is a tetrahedron together with its interior, and so on for higher dimensions.

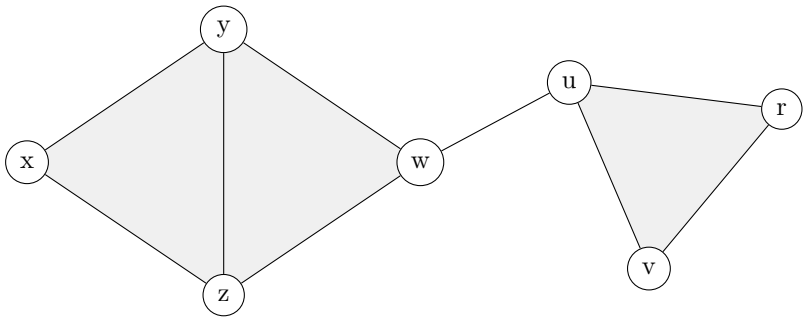


Figure 1: 2-dimensional simplicial complex.

A complex $\phi(\sigma)$ is a *subdivision* of a complex $\sigma$ if every simplex in $\phi(\sigma)$ is contained in a simplex of $\sigma$ and every simplex of $\sigma$ is the union of finitely many simplices of $\phi(\sigma)$. So the complex of figure 2 can be viewed as a subdivision of the complex defined by the three corners of the triangle. We denote by $V(\sigma)$ the set of vertices in $\sigma$. A *vertex map* $\psi$ between two complexes $\Gamma$ and $\Omega$ is a mapping $\psi : V(\Gamma) \to V(\Omega)$, and is additionally called a *simplicial map* if it maps simplices of $\Gamma$ to simplices of $\Omega$. Maps are dimension preserving if the image of a simplex has its same dimension. A coloring of a complex $\Gamma$ is a dimension preserving simplicial map from $\Gamma$ to $\sigma^n$, where $\sigma^n$ is an $n$-simplex. A complex together with a coloring is denoted as a chromatic complex, which can be intuitively understood as a complex with labels associated to vertices in such a way that no two neighboring vertices are labeled with the same label (as the complex in figure 2).

We use this topological notions to describe protocols in distributed computing iterated (or round by round) models. Recalling that a task specification is given by a triplet $(I, O, \Delta)$ where $I$ is a set of input vectors, $O$ a set of output vectors and $\Delta$ a relation between the previous two sets. We can construct an *input complex* $\mathcal{I}$ related to the set $I$ so that every input vector in $I$ is related to a simplex in $\mathcal{I}$; and an *output complex* $\mathcal{O}$ in which for each element of the set $O$ there is a simplex in $\mathcal{O}$. Finally, the relation $\Delta$ specifies how simplices in the input complex are related to the simplices in the output complex in a way that is consistent with the relation over the sets $I$ and $O$. It is important to note that the task specification $\Delta$ is not necessarily simplicial.

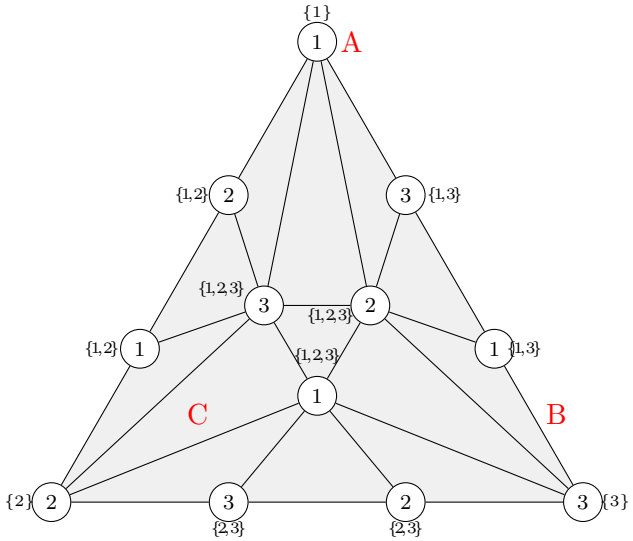


Figure 2: Simplicial complex described by the immediate snapshot protocol output views for 3 processes.

Under this premises we can, for instance, represent the set of possible executions where processes invoke $ImmSnap$ operations on a one-shot immediate snapshot object like a subdivided simplicial complex. As an example, figure 2 shows the case for 3 processes where each node is labeled by the related process' identifier (inside the circle) and by the received output view (in braces). The figure assumes that each process $i$ uses its own identifier as input to the immediate snapshot task. The vertex in the top corner of the triangle (labeled $A$) is by itself a simplex of the simplicial complex and it represents an execution in which only process 1 participates, thus obtaining a view that includes only its own identifier $\{1\}$. The edge labeled $B$ represents an execution in which process 2 doesn't take any steps and processes 1 and 3 invoke the task concurrently, thus seeing each other in their respective snapshots. Finally, the triangle labeled

$C$ represents an execution in which process 2 invokes the task first, followed by process 3 and later by process 1, therefore 2 obtains a view that only includes itself, 3 obtains a view that includes itself and the previous participants, *i.e.* $\{1, 3\}$, and process 1 gets to see a full view $\{1, 2, 3\}$ by invoking the task after everyone else.

# 3 Pre-Snapshot views

Snapshot and immediate snapshot objects have shown their vital importance in the design and verification of distributed algorithms by avoiding interference when a process attempts to obtain a global "picture" of the distributed state of the system. This interference comes from the fact that a process that attempts to obtain a picture of the system's shared variables does so by executing several partial reads of the shared memory while, in the meantime, other processes could concurrently modify the data contained in it. This kind of executions can easily render a view of the system that is inconsistent as the data it contains was never allocated in the shared memory at the same instant, thus, it isn't a representation of an actual state of the system. Nevertheless, it's interesting to observe the inherent properties of the views returned by simple scans or *collects* in an asynchronous environment. To this purpose, we recall the simple task called write/scan or WScan which abstracts a shared array with $n$ registers. Initially, every register in the array contains a default value $\perp$ different to any possible input value that a process can provide, when a process $i$ calls on this task with input value $v$, $v$ is written into the $i$-th segment and then every register is read sequentially one after the other in an arbitrary order to produce an output view which is delivered back to $i$. The output view generated is the set of values different to $\perp$ observed while doing a collect on the shared registers. For the remaining of this section, let a set of $n$ different processes with identifiers $\{1, 2, ..., n\}$ communicate through a shared object that implements WScan, each non faulty process invokes WScan once with its own identifier as input value and receives an output view in return complying to the WScan task specification. An execution in which a set of participating processes repeat this procedure generates a set of different output views, a computed view for each participating process and an empty view for each non participating process. Below, the fundamental properties that characterize these sets of views are studied.

## 3.1 Set-wise characteristics

This section starts by stating some definitions to simplify the notation and later states and proves a series of theorems that provide insight into the characteristics of the sets of views returned by WScan.

**Definition 3.1.** *Let* $\Pi = \{p_1, p_2, \ldots, p_n\}$ *denote the set of all processes and define by participating set* $\mathcal{P} \subseteq \Pi$ *the set of processes that eventually invoke the WScan task.*

**Definition 3.2.** *Each process* $p_i$ *is uniquely identified and is interchangeably referenced as* $i$ *or* $p_i$*. Its corresponding computed view is denoted by* $v_i$*.*

As we assume that every process that invokes the WScan task provides its own identifier as input, this implies that every returned view is a subset of $\mathcal{P}$.

**Definition 3.3.** *Define* $v_k = \emptyset$ *if the process* $p_k$ *never invokes the WScan task, so* $v_k = \emptyset, \forall p_k \in \Pi \setminus \mathcal{P}$*. The set of returned views is then the set of views* $v_i$ *such that* $v_i \neq \emptyset,\ i \in \mathcal{P}$*.*

The following theorem states that the set of views that are positively computed in any execution share a nonempty subset of processes.

**Theorem 3.4.** *Let* $V_1 = \{v_1, v_2, \ldots, v_m\}$ *be the set of computed views generated by an execution, then:*

$$A_1 = \bigcap_{1 \leq i \leq m} v_i \neq \emptyset$$

*Proof.* Suppose that $\bigcap v_i = \emptyset$ and let $p_k$ be the first process to write its value to the shared memory by invoking the task. As the collect starts only after the input value has been written, the collect by $p_k$ must find its own identifier written to the shared memory, and as every other process writes afterwards, any process obtaining a view will necessarily observe $p_k$.

$$\therefore p_k \in \bigcap_{1 \leq i \leq m} v_i$$

The contradiction completes the proof. □

Let the set $A_1$, like in theorem 3.4, be the nonempty subset of processes shared by all computed views, $A_1 = \bigcap v_i$ such that $v_i \in V_1$. We can then define a new set of views,

**Definition 3.5.** *Let $V_2$ be defined as the set of views obtained from $V_1$ by removing all the views generated by the processes in $A_1$. In other words, $V_2 = V_1 \setminus \{v_i | p_i \in A_1\}$.*

The next theorem states that unless the set of processes $A_1$ is equal to the total set of participating processes $\mathcal{P}$, then the intersection of the views in the set $V_2$ is also nonempty, even after removing the set $A_1$ which is evidently included in the intersection.

**Theorem 3.6.** *If $A_1 \neq \mathcal{P}$ then,*

$$A_2 = \left(\bigcap V_2\right) \setminus A_1 \neq \emptyset$$

*Proof.* Let $B = \mathcal{P} \setminus A_1$ which is nonempty and let $p_k \in B$ be the first process from $B$ to write its identifier to the shared memory. Therefore, as in the previous theorem, every single process in $B$ must see $p_k$ in their corresponding views because $p_k$ is already written to the shared memory when they start their respective collects, this implies that every view in $V_2$ must contain $p_k$,

$$\therefore p_k \in \bigcap_{v_j \in V_2} v_j \setminus A$$

□

This property of the set of views is applicable iteratively as long as there remains a nonempty subset of views $V_i$, obtained from $V_{(i-1)}$ by removing the views generated by any process which has been contained in any of the previous intersections, this is, if $\mathcal{P} \neq \bigcup A_j$ such that $j < i$. Using this property we can now define the following sets.

**Definition 3.7.** *Given a set of views $V$ obtained from a valid execution on a WScan object, we define sets $S_1, S_2, ...$ iteratively in the following manner and as long as $S_k$ remains nonempty:*

$$S_1 = \bigcap \{v_i \mid v_i \in V\}$$

$$S_n = \bigcap \{v_i \mid p_i \notin S_j\} \setminus S_j, \quad \forall j < n$$

The collection of sets $\{S_i\}$ conform a sequence of disjoint sets whose union is the total of participating processes $\mathcal{P}$. Intuitively, this family of sets preserves an order in terms of concurrent writes, so if two processes belong to the same set $S_i$ it is because there is no possible way to distinguish which of the two processes wrote earlier, we can therefore think of the related write events as concurrent. In contrast, the write by a process that belongs to a set $S_j$ occurred earlier in time than that of a process that belongs to a set $S_k$ with $j < k$. Even more, this sequence of sets lets us define another family of sets, also in an iterative manner.

**Definition 3.8.** *Given the family of sets $\{S_i\}$, we define their related sets $P_1, P_2, ...$ iteratively as follows:*

$$P_1 = \{i \mid S_1 \subseteq v_i\}$$

$$P_n = \{i \mid \bigcup_{1 \leq j \leq n} S_j \subseteq v_i\}$$

So $P_1$ is the set of processes whose views contain all the processes in $S_1$. $P_2$ is the set of processes whose views contain all the processes in $S_1$ *and* in $S_2$, and so on. The family $\{P_i\}$ defines a sequence of sets ordered by containment, this is:

$$P_1 \supseteq P_2 \supseteq \cdots \supseteq P_r \supseteq \ldots$$

**Theorem 3.9.** $p \in P_k \Rightarrow p \in P_m,\ \forall m \leq k.$

*Proof.* The proof is clear from the containments. □

**Definition 3.10.** *Given a process p, define order(p) as the maximum integer for which $p \in P_k$ is true.*

From the definitions, it is clear that if $order(p) = k$ then $p \in P_k$ and $p \notin P_{k+1}$, or in other words, we have $\bigcup_{1 \leq i \leq k} S_i \subseteq v_p$ but $S_{k+1} \not\subseteq v_p$.

**Theorem 3.11.** $p \in S_k \Rightarrow S_k \subseteq v_p$.

*Proof.* By the way of contradiction, suppose there is a process $q$ in $S_k$ that is not contained in $v_p$. This is, $\exists\ q \in S_k$ such that $q \notin v_p$. As $p \in S_k$, then it is not in any $S_j$ with $j < k$ because the family $\{S_i\}$ is disjoint. Therefore $p$'s view is intersected with others to generate the set $S_k$, but if $q$ isn't contained in $v_p$, then it's not contained in the intersection:

$$\begin{aligned} q \notin v_p \ &\Rightarrow\ q \notin \bigcap \{v_i \mid p_i \notin S_j\} \setminus S_j,\ \forall j < n \\ &\Rightarrow\ q \notin S_k \end{aligned}$$

□

**Theorem 3.12.** $p \in S_k\ \Rightarrow S_j \subseteq v_p, \forall j < k$

*Proof.* Suppose that $S_j \not\subseteq v_p$ for some $j < k$

$\Rightarrow \exists\ q \in S_j$ such that $q \notin v_p$

But as $p \in S_k\ \Rightarrow p \notin S_i$ for any $i < k$ se the sets are disjoint. Then as before:

$$\begin{aligned} q \notin v_p \ &\Rightarrow\ q \notin \bigcap \{v_i \mid p_i \notin S_j\} \setminus S_j,\ \forall j < n \\ &\Rightarrow\ q \notin S_k \end{aligned}$$

□

**Corollary 3.13.** *The next result is direct from the previous theorems:*

$$p \in S_k\ \Rightarrow p \in P_r,\ r \leq k$$

*or in other words:*

$$p \in S_k\ \Rightarrow \bigcup_{1 \leq i \leq k} S_i \subseteq v_p$$

**Theorem 3.14.** *If $order(p) = k$ then $p \in S_j$ for some $j \leq k$.*

*Proof.* We show that if $order(p) = k$ then $p \notin S_j$ with $j > k$, which, by the characteristics of the family of sets $\{S_i\}$ of being disjoint and having the total set as union, proves the theorem, since $p$ will necessarily have to be an element of some $S_r$ with $r < k$.

Suppose $p \in S_j,\ j > k$, then:

$$\Rightarrow \bigcup_{1 \leq i \leq j} S_i \subseteq v_p \ \Rightarrow p \in P_j \ \text{ con } \ j > k \ \Rightarrow \ orden(p) \geq j > k$$

Contradicting that $order(p) = k$.

□

## 3.2 Discussion

The preceding notions establish some properties that the views obtained in an execution must satisfy simply by the way in which the $WScan$ proceeds. This are not, however, sufficient conditions to describe any set of views generated by an execution. Take as an example the following sets of identifiers which intend to be a set of valid views[5] in an execution with three participating processes, namely processes 1, 2 and 3. Associate process 1 to the set $v_1 = \{1, 2, 3\}$ –which would imply that the scan by process 1 saw a full view– process 2 to the set $v_2 = \{1, 2\}$ and process 3 to the set $v_3 = \{2, 3\}$. It is easy to observe that the family of sets $\{S_i\}$ can be obtained from these sets without violating any required property. We assume for this example that the scanning of the segments is done in the natural order defined by the processes' indexes, this is, the first register to be read is the one to which process 1 writes, then the register assigned to process 2 and finally the one of process 3. The order in which scans are executed is not relevant as a counterexample one ordering is easily translatable to any other permutation of the indexes. The sets $\{S_i\}$ are then as follows:

$$S_1 = v_1 \cap v_2 \cap v_3 = \{2\}$$

[5]Where by valid we refer to a set of views that can really be generated in an execution, or in other words, an element of the set of output vectors $O$ which is related to the particular input vector by the specification of the task

$$S_2 = \{v_1 \cap v_3\} \setminus S_1 = \{3\}$$

$$S_3 = \{v_1\} \setminus \{S_1 \cup S_2\} = \{1\}$$

So none of the sets are empty, they are disjoint and their union is the total. However this sets cannot be the result of a valid execution. The sets $S_i$ imply that process 2 is the first one to write, followed by process 3 and finally by process 1. But if this is so, after writing, process 2 starts its collect, it starts by reading the first location, as its associated set $v_2$ includes the value 1 it must be that process 1 has already written when process 2 reads the register. But then process 2 should have read the value 3 as well, due to the fact that it was supposedly written before value 1 and that its register is read later.

The problem exposed by the previous counterexample arises from the observation that the previous properties are only concerned with the order in which the write events are executed. However, in order to fully grasp the characteristics proper to any set of views generated by an execution, the order in which read events occur must also be taken into account. Let the order in which the registers are read during a scan be specified by the permutation $Per : \mathcal{P} \to \{1 \dots n\}$, so the register for which $Per(p_x) = 1$ is the first to be read, followed by the register for which $Per(p_y) = 2$, and so on. The next theorem states that every time that a process reads the identifier of a process that is written later than a process whose register is still to be read, then the scanning process view must also contain the latter process' identifier. The proof is direct from the order in which the read events are ordered.

**Theorem 3.15.** *Let $p \in S_k,\ q \in S_r,\ k < r$ and $Per(q) < Per(p)$. Then $q \in v_k \Longrightarrow p \in v_k,\ \forall k$*

It has already been shown that any set of valid views necessarily complies to the properties implied by the previous theorems. This leads to the following set-wise characterization of valid views.

**Set-wise characterization of valid views:** *Let $\mathcal{P}$ be a set of processes and $\mathcal{V} = \{v_i | i \in \mathcal{P}\}$ be a set of views with $v_i \subseteq \mathcal{P}, \forall i$. Then $\mathcal{V}$ is a valid set of output views from a WScan task if and only if $\mathcal{V}$ satisfies the following two properties:*

- *The sets $\{S_i\}$ defined by $\mathcal{V}$ are non empty, disjoint and their union is the total of participating processes.*

- $p \in S_k,\ q \in S_r,\ k < r$ *and* $Per(q) < Per(p)$. *Then* $q \in v_k \Longrightarrow p \in v_k,\ \forall k$ *, where* $Per$ *is a permutation that defines the order in which the segments are read.*

## 3.3 WScan execution representations

We now present a characterization for executions over a WScan task and for the views that these generate. Consider any $(n+1) \times n$ matrix $A$ with unique elements $a_{i,j}$ from the set $\{i \mid 1 \leq i \leq n(n+1)\}$ which satisfies the following two properties[6]:

**Property 1:** $a_{n+1,i} < a_{i,j},\ \forall i, j$

**Property 2:** $a_{i,j} < a_{i,j+1},\ \forall i \leq n,\ \forall j$

The first property states that the $i$-th element from the last row (*i.e.* last row, $i$-th column) is smaller than every element in the $i$-th row. Intuitively, the elements in the matrix are timestamps of the different events occurring over the shared registers, the last row represents the writing events and the rest of the rows represent the reads. So this captures the idea that the write event of a certain process comes before any of its read events, as the $i$-th element in the last row holds the timestamp for the write event of process $i$ and the elements of the $i$-th row represent its read events. The second property captures the orderly manner in which the read events occur, because the different segments are read sequentially, so process $i$ reads segment $j$ before it reads segment $j+1$ regardless of the time it takes it to do so.

Any execution of a WScan task can be represented as a matrix holding the two properties. The converse is also true, any matrix that satisfies the properties has an associated execution which is very easy to simulate as one only needs to follow the events on the shared array as the timestamps in the matrix indicate.

Given a matrix representation of an execution, the set of views is efficiently computed by the next simple rule:

[6]The matrix elements are standard with usual matrix notation, so the element indexed as $a_{i,j}$ represents the matrix entry in the $i$-th row and $j$-th column.

**Rule 3.16.** $j \in v_i \Leftrightarrow a_{i,j} > a_{n+1,j}$

So a process $j$ is in a process $i$'s view if $j$ wrote its identifier before $i$ read the corresponding register, which makes perfect sense. With this in mind, it is clear that a particular set of views $V$ can be considered valid if and only if one can construct a matrix that satisfies properties 1 and 2 from which the proposed set of views $V$ can be obtained by following the previous rule.

As an example, lets construct a matrix from a particular execution of the WScan task with 3 processes which renders a set of view which do not comply with the snapshot specification. First process $p_1$ invokes WScan and its identifier 1 gets written (event is numbered 1 in the first matrix), it then starts its scan by reading the first and second segment (numbered 2 and 3 in the second matrix). Then it stops executing steps while process $p_2$ invokes the task, gets written and proceeds to read the whole array (events numbered 4(write event), 5, 6 and 7 (sequential read events) all shown in the third matrix).

$$\begin{array}{c} \\ p_1 \\ p_2 \\ p_3 \\ w \end{array}\!\!\begin{array}{c} \begin{array}{ccc} p_1 & p_2 & p_3 \end{array} \\ \begin{pmatrix} . & . & . \\ . & . & . \\ . & . & . \\ 1 & . & . \end{pmatrix} \end{array} \qquad \begin{array}{c} \\ p_1 \\ p_2 \\ p_3 \\ w \end{array}\!\!\begin{array}{c} \begin{array}{ccc} p_1 & p_2 & p_3 \end{array} \\ \begin{pmatrix} 2 & 3 & . \\ . & . & . \\ . & . & . \\ 1 & . & . \end{pmatrix} \end{array} \qquad \begin{array}{c} \\ p_1 \\ p_2 \\ p_3 \\ w \end{array}\!\!\begin{array}{c} \begin{array}{ccc} p_1 & p_2 & p_3 \end{array} \\ \begin{pmatrix} 2 & 3 & . \\ 5 & 6 & 7 \\ . & . & . \\ 1 & 4 & . \end{pmatrix} \end{array}$$

Later, process $p_3$ invokes the task and its write event occurs with timestamp 8, afterwards $p_1$ continues making progress and finishes its last read event (labeled 9 in the penultimate matrix). At the end, $p_3$ finishes its invocation by executing its 3 read events (numbered 10, 11 and 12 in the last matrix).

$$\begin{array}{c} \\ p_1 \\ p_2 \\ p_3 \\ w \end{array}\!\!\begin{array}{c} \begin{array}{ccc} p_1 & p_2 & p_3 \end{array} \\ \begin{pmatrix} 2 & 3 & . \\ 5 & 6 & 7 \\ . & . & . \\ 1 & 4 & 8 \end{pmatrix} \end{array} \qquad \begin{array}{c} \\ p_1 \\ p_2 \\ p_3 \\ w \end{array}\!\!\begin{array}{c} \begin{array}{ccc} p_1 & p_2 & p_3 \end{array} \\ \begin{pmatrix} 2 & 3 & 9 \\ 5 & 6 & 7 \\ . & . & . \\ 1 & 4 & 8 \end{pmatrix} \end{array} \qquad \begin{array}{c} \\ p_1 \\ p_2 \\ p_3 \\ w \end{array}\!\!\begin{array}{c} \begin{array}{ccc} p_1 & p_2 & p_3 \end{array} \\ \begin{pmatrix} 2 & 3 & 9 \\ 5 & 6 & 7 \\ 10 & 11 & 12 \\ 1 & 4 & 8 \end{pmatrix} \end{array}$$

Analyzing this last matrix we see that $v_1 = \{1, 3\}$ because $2 = a_{1,1} > a_{4,1} = 1$ and because $9 = a_{1,3} > a_{4,3} = 8$, however $\{2\} \notin v_1$ because $3 = a_{1,2} \not> a_{4,2} = 4$. Analogue observations

render the views $v_2 = \{1, 2\}$ and $v_3 = \{1, 2, 3\}$. It should be noted that this is an example of a set of views which wouldn't be obtained from a snapshot object because $v_1$ and $v_2$ cannot be ordered by containment.

Finally, we present a last matrix which renders a different execution but which is not distinguishable to any of the processes because every view is identical to the views from the previous example:

$$\begin{array}{c} \\ p_1 \\ p_2 \\ p_3 \\ w \end{array} \begin{array}{c} \begin{array}{ccc} p_1 & p_2 & p_3 \end{array} \\ \begin{pmatrix} 2 & 3 & 11 \\ 5 & 6 & 7 \\ 9 & 10 & 12 \\ 1 & 4 & 8 \end{pmatrix} \end{array}$$

We denote any representation of this kind as an *execution matrix*. The following theorem states sufficient and necessary conditions for a set of views to be considered as valid. The proof is direct from the observed relation between the execution and the matrixes, it is clear that there is a matrix representation for every possible execution of a WScan task and that every possible well constructed matrix represents a particular execution. We note again that a same set of views can be obtainable from a number of different executions.

**Theorem 3.17.** *Let $\mathcal{P}$ be a set of $n$ processes and $\mathcal{V} = \{v_1, v_2, \ldots, v_n\}$ be a set of views with $v_i \subseteq \mathcal{P}, \forall i$. Then $\mathcal{V}$ is a valid set of output views from a WScan task if and only if there is an execution matrix $M$ from which this set of views is obtainable by following rule 3.16.*

# 4 Recursive Snapshots

This section presents a series of algorithms which explore different recursive structures aiming to obtain implementations for the snapshot and immediate snapshot tasks. The first of this is the classic implementation of the participating set protocol in its recursive version.

## 4.1 Recursive participating set protocol

The *participating set protocol* presented in [13] is a wait-free algorithm that solves the Immediate Snapshot task using only single-writer/multi-reader atomic registers. Gafni and Rajsbaum adopt this algorithm in [22] and propose a recursive version of it in which different instances of a same task are invoked by the different participating processes in order to obtain a view which complies with the specifications of the problem. Both algorithms have an upper bound complexity of $O(n^2)$ although it is proven in [22] that the creation of a view of size $s$ takes $\Theta(n(n-s+1))$ steps in total. This recursive algorithm, ImmSnap, serves as motivation to the present study and is shown next as Algorithm 1.

**Algorithm 1**

ImmSnap$_n(i)$ (code for $p_i$)

```
1: write i to R_i
2: view ← scan{R_1, ..., R_n}
3: if |view| = n then
4:     return view
5: else
6:     ImmSnap_{n-1}(i)
```

In this and all following algorithms the convention is used that variables in uppercase are

shared in that instance and variables in lowercase are local to the executing process. In the algorithm, registers $R_1$ up to $R_n$ compose the shared memory of each particular instance of the immediate snapshot task. These registers are all single-writer/multi-reader atomic registers which means that only the $i$-th process can write values into register $R_i$ and that every single process can read the contents of every single register. Each independent instance of the task[7] has an independent set of $n$ shared registers known internally as $R_1, \ldots, R_n$ which are private to that particular instance; this can be seen as every instance of the task having its own clean copy of the register array. Initially every single register in every single instance is set to a default value $\perp$ which cannot be written by any process. When the algorithm is invoked by process $i$, it starts by writing $i$'s identifier into register $R_i$ in the shared memory of $\textsc{ImmSnap}_n$. This announces $i$'s participation in the protocol to any subsequent process that executes the same instance of the task, and so, any process that starts execution of line 2 after the write event by $i$, and later returns at this same instance, should include $i$'s identifier in its view. After writing its identifier, $i$ computes its own view by sequentially reading every register in the shared memory and creating a vector *view* that contains every value different to $\perp$ read by $i$. Afterwards, the cardinality of the computed view is calculated, *i.e.* the number of different identifiers it contains, and if this number is equal to the subindex of the particular instance then $i$ can return with *view* as output; otherwise, a recursive invocation is made to a different instance of the algorithm with a subindex one less than the current instance. The algorithm works as each instance of the task admits a number of participants smaller or equal to what is indicated by its subindex. Proofs of correctness and termination are given in [22].

The recursive algorithm ImmSnap to solve Immediate Snapshot is simple, concise and can be easily understood and analyzed. Even more, the fact that it is recursive and uses clean copies of the shared array in each instance give the algorithm the desirable property of belonging to an iterated model of computation, which allows analyzing it from a topological viewpoint by describing the set of possible outputs as a subdivision of a simplicial complex [25]. Nevertheless, the complexity of the algorithm might be considered too high as to be considered a viable solution for real life applications.

Next, we present a series of attempts to modify the previous algorithm with the goal of

[7]Different instances of the task are recognized by their subindex, so $\textsc{ImmSnap}_k$ and $\textsc{ImmSnap}_l$ are different instances of the ImmSnap task if $k \neq l$

lowering the complexity in order to obtain a recursive implementation for the snapshot task which approaches the trivial minimum bound of $O(n)$, obtained from the fact that a process running by itself cannot obtain a snapshot if it doesn't read at least once each of the other processes' registers.

## 4.2 *Reversed* invocations (1 through $n$)

The first noticeable aspect of analyzing algorithm ImmSnap for the worst case, is that this occurs when a process $i$ executes the protocol in a *solo* execution, this is, no other process takes steps until the referenced process returns. In this scenario, every instance of the Immediate Snapshot task will obtain a view that only contains $i$'s own identifier, leading $i$ to execute the task recursively with a lesser subindex until reaching the bottom of the recursion at the instance $\text{ImmSnap}_1(i)$, where it will return the view $\{i\}$ that only contains its own identifier. Therefore, the process executes $n$ recursions making one write and $n$ reads in each, thus leading to the mentioned complexity of $O(n^2)$ steps on the shared memory.

A first approach to try to improve the complexity of this algorithm is presented next as algorithm 2 or algorithm Reverse. This version is very similar to the previous algorithm but attempts to avoid the preceding worst case scenario simply by *reversing* the execution order on which different instances are invoked. To do this, the first invocation is done upon the instance referred to as $\text{Reverse}_1$ and recursive invocations by a task instance $\text{Reverse}_k$ point to the instance $\text{Reverse}_{k+1}$. This new protocol succeeds in reducing the complexity of the preceding worst case scenario, because when a process $i$ invokes $\text{Reverse}_1(i)$ and executes solo, it will return in line 3 after a single write and $n$ reads from shared memory.

However, a quick glimpse at algorithm Reverse is all it takes to find counterexamples that breach the minimal required termination property for any algorithm, this is, that a process should finish its execution of the algorithm in finitely many steps. As an example of such an execution, think of that in which the full set of $n$ processes (where $n > 1$) invoke the algorithm concurrently, thus writing concurrently to the shared registers, this will render the same *full* view to every process and as $|view| = n > 1 = k$ every process will eventually invoke the algorithm

**Algorithm 2**

REVERSE$_k(i)$ (code for $p_i$)

```
1: write i to R_i
2: view ← scan{R_1, ..., R_n}
3: if |view| = k then
4:     return view
5: else
6:     REVERSE_{k+1}(i)
```

recursively with the next tag $k = 2$. Lets think that this style of concurrent invocations continues in the same manner throughout the different instances, with every process obtaining a full view each time, up to the bottom of the recursion. When the $n$ participating processes finally invoke the instance with subindex tag $n$, one process $p$ crashes before writing it's identifier on this last instance's shared registers, while every other process continues to execute normally. Every single process besides $p$ will obtain a view, however as non of these views includes $p$, they will all have views with cardinality lower than $n$. This will render the conditional on line 3 false and so, will lead the processes to recursively invoke REVERSE$_{n+1}$, point after which, it is clear that the recursion will go on infinitely deep, as the conditional escape clause of line 3 will never be met by any of the subsequent views. Any process captured in a scenario similar to this will not terminate its execution regardless of the number of steps it executes. Even more, it is clear that if we restrict ourselves to executions that do finish correctly, the complexity of such executions is also quadratic in the worst case, although it's evident by now that these restricted executions are not the worst case. Therefore, algorithm REVERSE is definitely not an improvement to the previous implementation of the immediate snapshot task, as it doesn't even effectively solve the task. In certain settings it could be arguable that it isn't even a proper algorithm as it doesn't satisfy the termination property.

## 4.3 Knowledge accumulation

A different approach to obtain a more efficient protocol to solve the atomic immediate snapshot task comes from the idea of *knowledge accumulation* which is borrowed from [8]. This concept is equivalent in this case to that of a full information execution in the sense that processes take their computed views from a certain instance of the protocol as inputs for the next instance.

The algorithm, named KNOWLEDGE is shown next.

**Algorithm 3**

KNOWLEDGE$_k(I_i)$ (code for $p_i$)

```
1: write I_i to R_i
2: scan R_1, ..., R_n
3: view ← ∪{R_1, ..., R_n}
4: if |view| = k then
5:     return view
6: else
7:     KNOWLEDGE_{k+1}(view)
```

Any process that invokes algorithm KNOWLEDGE, initially does so by invoking the instance with subindex tag $k = 1$ and with its own identifier as the only element of its input view $I_i$. The process then writes its input view to the shared registers of that particular instance and then performs a scan on the whole array. A new view is generated by calculating the union[8] of the views already written to the shared memory which the process was able to read. If the cardinality of the computed view equals the subindex tag of the instance being executed, the process exits with such view. In other case, it recursively invokes the next instance proposing the computed view as its new input. The only difference between algorithms KNOWLEDGE and REVERSE is therefore the accumulation of knowledge, as tags are incremental in both. If a certain process $p$ obtains information of other process' participation (by reading the latter's identifier $q$ in any of the instances' shared arrays), it never loses this information in the remaining of the execution. This is due to the fact that, after reading $q$'s identifier, $p$ creates a view which contains this identifier and then uses this view as input to the next instance, where $p$ itself will write $q$'s identifier into the shared array as part of its own input, and later read it again during the posterior scan. This has some advantages over the previous algorithm. First, this works somewhat as a helping mechanism because a process can learn of some other process' participation in an indirect manner by taking the information from a third process. In other words, even if a process $p$ never reads any information written in the shared memory by a process $q$, it can obtain evidence of $q$'s participation by reading $q$'s identifier in another process $k$'s input

[8]This union refers to the classic notion of set union, we note that as views are simply sets of process identifiers this operation is well defined.

view. Secondly, as information is never lost, a view calculated in a posterior instance can never be smaller than a view obtained in a previous instance, avoiding the problem which led to the described counterexample for algorithm REVERSE. This implies that an invocation to algorithm KNOWLEDGE from a correct process always terminates after executing a finite amount of steps. Next, we show that this last algorithm is an implementation for the atomic snapshot task.

**Theorem 4.1.** *Algorithm* KNOWLEDGE *solves the atomic snapshot task.*

*Proof.* It is clear that the algorithm satisfies the self-inclusion property, as each process's own identifier works as input for the first instance and this knowledge is never lost throughout the recursion, leading the executing process's identifier to be necessarily contained in the final view, so $i \in v_i, \forall i$. To show that the final views are order by containment, we first show that two final views of equal size must necessarily be the same view. Suppose that two processes $i$ and$j$ finish their respective executions with views $view_i$ and $view_j$, such that $|view_i| = |view_j| = k$; we prove that $view_i = view_j$. In order for $i$ and $j$ to return with a view of size $k$, both processes need to exit the recursion at the $k$-th level, which at the same time implies that in the instance with tag $k-1$, both processes obtained a view with a cardinality larger than $k-1$. Even more, the view obtained at level $k-1$ must necessarily be of size $k$ as in another case (view larger than k) this same view would become a subset of the view computed at level $k$, leading once again to a view with a cardinality larger than $k$ and contradicting the fact that both processes obtain a final view of size $k$. Without loss of generality, assume that $i$ is the first of the two processes to write its input view to the shared registers which are private to the $k$-th instance of the algorithm, then it must necessarily read its own input view while performing the scan of line 2 and as we know it will not obtain any more knowledge, its computed view must be exactly the same as the one it wrote as input, this again is clear because more knowledge would render a larger view which would in turn contradict the premises. The same is true for $j$, after writing its own $k$-sized input view, it will perform its scan, reading its own input view as well as $i$'s input together with possibly other views. However, just as it happened with $i$, it cannot obtain more knowledge than it already had in its input view, not even from $i$'s input. This implies that $i$'s and $j$'s input views were identical, and as none of them acquires any new knowledge, so are their final views. In the case where $|view_i| = k_1$ y $|view_j| = k_2$ with $k_1 < k_2$. We need to show that $view_i \subseteq view_j$ if the order by containment is to be preserved. Again, think of the level $k_1 - 1$, in which both processes obtained their respective partial views $view'_i$ and $view'_j$ of size at

least $k_1$. From the argument used above, we also know that $view_i = view'_i$, as in another case $i$ wouldn't have returned in level $k_1$. There are two possible cases on instance KNOWLEDGE$_{k_1}$ depending on whether $j$ reads $i$'s input view or not when performing the scan at that instance. If $j$ does read the view $view_i$ written by $i$ to the shared array then $j$'s final view will necessarily contain $i$'s view as knowledge is not lost throughout the recursion and we know that $i$ will have $view_i$ as its final view, so in this case $view_i \subseteq view_j$. In the other case, if $j$ doesn't read the view written by $i$ it means that $j$ necessarily performed its write before $i$. Therefore $i$ will consequently read the input view $view'_j$ written by $j$ when performing its scan on level $k_1$. $i$ also reads its own input view, which we know is $view_i$. If it were true that $view_i \neq view'_j$ then $i$ would obtain more knowledge in level $k_1$ and, accordingly, a view of size larger than $k_1$ contradicting the fact that $i$ returns from the algorithm with view $view_i$. Therefore, it must be true that $view_i = view'_j$ and it is clear that $view'_j \subseteq view_j$ as knowledge is not lost throughout the recursion and $k_1 = |view_i| = |view'_j| < |view_j|$.

It has been shown that all final views are ordered by containment and are self-inclusive, proving that algorithm KNOWLEDGE is an implementation for the atomic snapshot task. □

Nevertheless, if this algorithm was to be proposed as an implementation to the atomic immediate snapshot task, a deeper examination will prove it faulty. As it has been proved that the algorithm satisfies the self-inclusion and containment properties, any counterexample must rise from a generated set of views that fails to embrace the last desired property: immediacy. We give an counterexample to the immediacy property using an execution with a set of three participating processes. Let $i$ and $j$ be two processes which concurrently invoke the algorithm in such a way that while performing their scan in the first instance they both read each other's identifier resulting in views $\{i, j\}$ for both processes. Later, a third process $v$ invokes the algorithm obtaining $\{i, j, v\}$ as its view for the first instance. As the three processes obtain views of size larger than 1, they all recursively invoke the next instance of the task with tag $k = 2$. In this instance, let $i$ run solo first, writing its view $\{i, j\}$ and then performing its scan. While performing the scan $i$ should only find its own input in the memory and the default value $\perp$ in everybody else's register, thus computing its same view $\{i, j\}$ of size 2 as output, which by rendering the conditional in line 4 true, also lets $i$ exit and return with that precise view. Then let $v$ run solo. It will run up to a third recursive call after seeing three identifiers in the second iteration's register array, namely $\{i, j, v\}$. In the third recursive instance, with tag $k = 3$,

process $v$ will again see the whole participating set of size 3 and will return with view $\{i, j, v\}$. Finally, let $j$ run again. After writing its input view in the registers of the instance with tag $k = 2$ it will perform its scan and read $v$'s input view obtaining thus a complete view of the three participating process. Therefore, just as process $v$, it will end up in a third instance with the same exit view as process $v$. The exit views produced by processes $i$ and $j$, $\{i, j\}$ and $\{i, j, v\}$ respectively, don't satisfy the immediacy property as we see that $j \in view_i$ but $view_j \not\subseteq view_i$.

As was mentioned before, even though algorithm KNOWLEDGE doesn't satisfy the immediacy property to correctly implement immediate snapshots, it is nevertheless a correct implementation for the more general atomic snapshot. Be that as it may, it is still quadratic in complexity so it doesn't provide a better implementation to the task than previous solutions. The quadratic scenario happens when a process has to recursively invoke $n$ different instances of the algorithm as it always sees a larger view than what is specified by the instance's subindex tag, for example in an execution where all $n$ processes invoke algorithm KNOWLEDGE concurrently. In this execution processes will get all the way to the bottom of the recursion invoking the instance with subindex tag $n$ and as there are no larger views possible than a complete view, *i.e.* one that includes every process, they will all return with that exact view. Even though all $n$ processes compute a complete view since the first instance, they nonetheless have to traverse the whole recursion before exiting with the exact same view that they all had calculated at the first instance. There is no real gain by adding a conditional clause that lets a process return when in possession of a full view, as a very similar problem which also renders quadratic complexity would happen if $n - 1$ processes executed concurrently.

## 4.4 "Jumping" recursion

The use of linear recursion seems to present an impassable problem if complexity is to be reduced; for it isn't clear how to reduce the worst case complexity if there are processes that need to invoke $n$ instances of the algorithm, regardless of the algorithm being a correct implementation. A new probable solution comes into mind, where we allow processes to jump recursion levels by recursively calling new instances that depend on the number of processes included in the partial views. The following pseudocode, called JUMPS and labeled algorithm 4, is an implementation for this approximation. We recall that the input view for the first invocation is the process'

identifier.

**Algorithm 4**

JUMPS$_k(I_i)$ (code for $p_i$)

```
1: write I_i to R_i
2: scan R_1, ..., R_n
3: view ← ∪{R_1, ..., R_n}
4: if |view| = k then
5:     return view
6: else
7:     JUMPS_{|view|}(view)
```

We note that this algorithm also uses knowledge accumulation by providing previously calculated views as inputs to subsequent invocations. The self-inclusion property for this algorithm is proven in the same way as it was done for the previous algorithm KNOWLEDGE, simply by observing the fact that knowledge is never lost and that a process's own identifier is the first piece of knowledge it acquires by issuing it as its private input. On the other hand, by analyzing the possible partial views that different participating processes can calculate, it is easy to see that the containment guarantees that were offered by algorithm KNOWLEDGE are lost when we allow process to jump recursion levels. This creates executions which work as counterexamples even to the containment property for atomic snapshots, not to mention the immediacy property to satisfy immediate snapshots. Next, a counterexample in which the final resulting views aren't order by containment is displayed. First, we note that in the first instance of the algorithm with tag $k = 1$, lines 1, 2 and 3 are exactly equivalent to an invocation on a WScan object with the process' identifier as input. We can therefore use the matrix representation presented in section 3.3 to represent the partial execution over the first instance of the task in our counterexample. The following matrix represents an execution with 4 participating processes:

$$\begin{array}{c} \\ p_1 \\ p_2 \\ p_3 \\ p_4 \\ w \end{array} \begin{array}{c} \begin{array}{cccc} p_1 & p_2 & p_3 & p_4 \end{array} \\ \begin{pmatrix} 2 & 3 & 4 & 20 \\ 7 & 9 & 11 & 13 \\ 8 & 10 & 12 & 14 \\ 16 & 17 & 18 & 19 \\ 1 & 5 & 6 & 15 \end{pmatrix} \end{array}$$

In the execution, $p_1$ is the first to write its identifier and later scans the first three registers before stopping. Then processes $p_2$ and $p_3$ execute concurrently writing and scanning the array completely. Later, a fourth process $p_4$ writes and scans, finally $p_1$ finishes its scan by reading $p_4$'s identifier in the last register. The views obtained are then $v_1 = \{1, 4\}$, $v_2 = v_3 = \{1, 2, 3\}$ and $v_4 = \{1, 2, 3, 4\}$ with respective cardinalities of 2, 3, 3 and 4 elements, all larger than one. The computed views then make process $p_1$ invoke recursively the instance with $k = 2$, $p_2$ and $p_3$ invoke the instance with $k = 3$ and $p_4$ the one with tag $k = 4$. In these recursively invoked instances, the four processes will find themselves computing right-sized views that will allow them to return, but these views are the same that were obtained in the first instance. We have that $p_1 \in v_2$, and $v_1 \not\subseteq v_2 \ \wedge \ v_2 \not\subseteq v_1$ [9], proving that neither the containment property nor the immediacy property are satisfied by algorithm JUMPS.

## 4.5 Branching recursion and related observations

The last counterexample shows that the proposed "jumping" recursion isn't the solution either. We can then think of a branching recursive structure in which recursion invocations look like a general tree. An example would be a double recursion in which some processes invoke recursively a "left" instance while others invoke a "right" one, thus the recursive structure would look as a binary tree. This kind of recursive structure is used in algorithms such as Gafni and Rajsbaum's recursive implementation of the *renaming* task proposed in [22]. The main idea would be to have a structure like the one shown in figure 3. The figure shows a structure in which each node of the binary tree is an instance of some task $T$ with $O(n)$ complexity, like WScan to give an example, referenced by its subindex, and the tree's height is $logn$.

If such a structure was able to solve the immediate snapshot task then its complexity would be only $O(nlogn)$. However, the tricky aspect is that each instance of the base algorithm, represented in the figure as the tree's nodes has to be able to split the processes that invoke it so that some invoke the left successor in their next recursive call while others invoke the remaining right successor, otherwise we could find executions in which every process traverses the same recursion path, which would mean that a linear recursion solution is available in $O(nlogn)$ complexity. Even if we assume that each node of the binary tree can satisfy this

[9] analogously $p_1 \in v_3$ and $v_1 \not\subseteq v_3 \ \wedge \ v_3 \not\subseteq v_1$

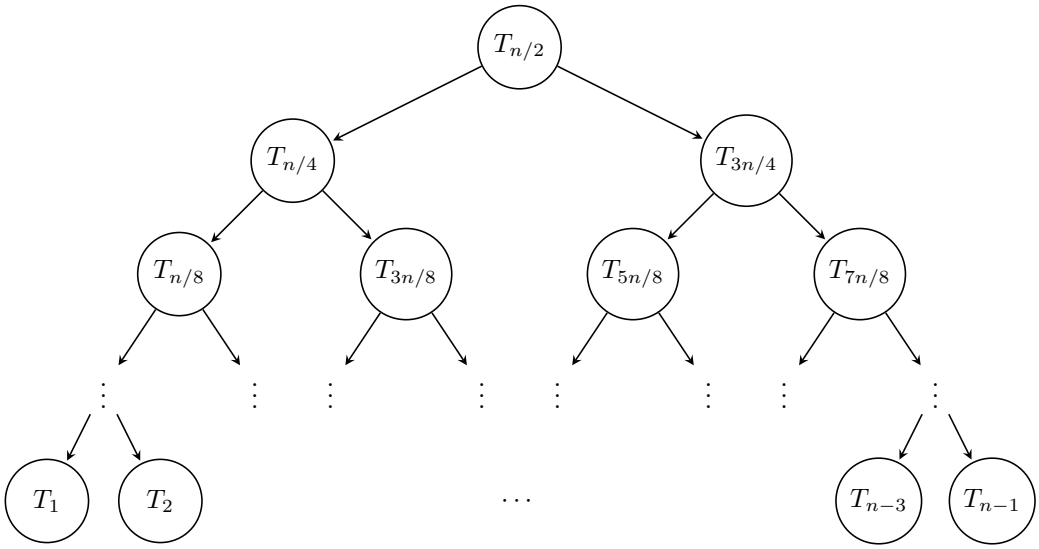


Figure 3: Branching recursion tree in which the different instances are referenced by subindex tags, each tag specifies the instances' position in the tree when labeled by an *in-order* search.

division property there are still many complications to the implementation. For instance, if the protocol doesn't rely on full information executions and process carry only their private information to a new node/instance, then two processes that end up at different leaves in the recursion tree will have trouble returning views that satisfy containment, as they will not know what the other process' view is like. Even more, as the protocol must be wait-free, a process in a leave must decide whether to include or not in its view a process that is still in the tree's inner nodes, a problem that brings up many complications even if the deciding process knows what that other process' current view is like. Extensive analysis on this regard has shown that the containment property seems impossible to satisfy in a recursive manner and thus the obtention of an $O(nlogn)$ implementation. This however, implies not only that immediate snapshots task cannot be implemented in such a complexity in an iterated model, but it implies an equivalent sentence on plain atomic snapshots as well. This gives rise to the main result of this work which is explored in the next section.

# 5 Snapshot impossibility in $O(nlogn)$ over the iterated model

This section starts by resuming the main observations obtained after analyzing the previous recursive algorithms in section 5.1. Later we state a conjecture in section 5.2 that defies the equivalence between the iterated read/write model and its non iterated version in terms of shared access complexity. We then provide analysis on the base cases for this conjecture and an elegant proof by connectivity for the case with three processors in section 5.3 and 5.4, respectively.

## 5.1 Initial observations

The previous section explored a series of recursive algorithms which differed on the structural recursive approach or in the way that knowledge was administered by the invoking processes. The goal was to find new recursive implementations for the snapshot and immediate snapshot tasks and with the goal in mind of trying to reduce the least known complexity bound of $O(n^2)$ steps on the shared memory for both tasks in an iterated setting, particularly in an attempt to bring the bound down to $O(nlogn)$ steps. Although a different implementation for the snapshot task was certainly devised, it didn't improve on the known complexity bound. The question of whether it is possible to obtain an algorithm with smaller complexity is not easily solvable. In the usual *write/read* non iterated model, where a unique shared array is used and processes can access it over and over again to read or write the different MRSW registers, there are implementations for the atomic snapshot task in $O(nlogn)$ [8]. The iterated model, although having been proved equivalent to its non iterated version, has no implementation of the snapshot task in any complexity below $O(n^2)$ where $n$ is the total number of processes. Therefore algorithms ImmSnap and Knowledge are both as good an implementation as there

is in the iterated write/read model for the snapshot task, even though the snapshot task is a weakening of the immediate snapshot task which algorithm ImmSnap also solves.

## 5.2 Postulation

The truly interesting result of the different implementations in the previous section was a set of observations which seemed to imply that the atomic snapshot task in not solvable in $O(nlogn)$ steps in the iterated setting. This final section conjectures this fact and establishes the basis for a generalized proof in this respect. We start by properly formulating the conjecture.

**Conjecture:** *There is no possible implementation for the atomic snapshot task in a read/write iterated model of distributed computing with a low bound complexity of O(nlogn) for any number n of invoking processes.*

As has been usual in this text, in the remaining of this section we assume a read/write iterated setting even when it's not explicitly stated. Also, on each round, processes make one invocation to a WScan task in order to announce their participation in that instance and obtain a scan of the shared memory. When trying to comply to the snapshot specification, a process that obtains a certain view $v_i$ from the WScan can then decide on any subset of $v_i$ as its final view, evidently attempting to comply with the self-inclusion and containment properties.

## 5.3 Base cases

The minimal distributed scenario occurs with two processes, so lets begin by analyzing the case with two processes and one iteration or round. In this setup, the atomic snapshot task is easily solvable. It is clear that when two processes, $p_1$ and $p_2$, participate and invoke a WScan task, there are only three possible output vectors. One in which both processes see both identifiers, one in which $p_1$ sees only itself while $p_2$ sees both, and the converse of this last output. It is easy to prove that there is no way in which both processes see only themselves, by observing the first process to execute its write event, and noting that both processes will necessarily observe this write. It is clear that the three possible output vectors satisfy the self-inclusion and containment properties, so a trivial implementation of the snapshot task lets each process return its computed

view regardless of what the other process' view contains.

Recall the protocol complex for the immediate snapshots task with 3 processes represented graphically in figure 2 in section 2.5. Here we present the complex for the WScan, also for three processes, as figure 4. It should be noted that both complexes are very similar, there are only three more simplices on the WScan complex than on the immediate snapshot complex. The previous paragraph implied that for 2 processes the complexes are identical, while the figures show that for three processes there are three possible sets of views which do not comply with the snapshot specifications. The number of simplices in the WScan complex that do not comply to the snapshot specifications grow exponentially to the dimension of the complex which is determined by the number of participating processes.

Think next on a scenario with three processes and two rounds, devising an implementation for the atomic snapshot task is also simple in this scenario. We can see from figure 4 that any execution where the three processes participate (*i.e.* a triangle) has at least one of its vertexes in the central simplex, which represents the obtention of a full view $\{1, 2, 3\}$ by the different processes. This implies that in any execution where all processes participate, at least one of them will compute a full view[10]. This allows us to give a simple implementation for atomic snapshots for three processes and two rounds. If a process in the first round sees a full view, it returns with that view. A process that doesn't see a full view in the first round invokes a second instance, in this second round a process returns whatever view it obtains. The proof proceeds by noting that not all three processes can invoke the second instance as is if this was the case, then all processes participated but none of them obtained a full view. We know this is impossible. Therefore at most two processes proceed to a second round and the problem is thus reduced to the case of two processes and 1 round that was earlier solved trivially by returning whichever view was computed. The proof ends by noting that the views from the second round are ordered by containment (as it was shown earlier) and that the full view, obtained by any process that doesn't proceed to the second round, trivially contains any of the partial views computed in the second round.

[10]This fact becomes clear if one thinks of the last process to invoke WScan, when this process performs its scan the other processes have already written their identifiers to their corresponding registers.

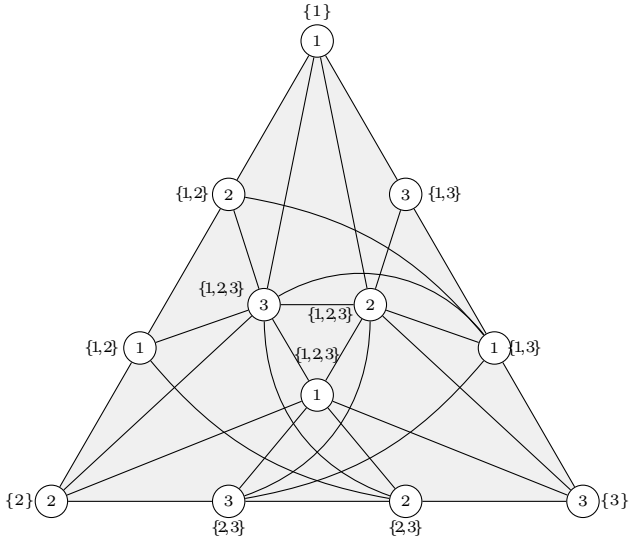


Figure 4: Simplicial complex representing all the possible write/scan views for 3 processes.

## 5.4 Proof for $n = 3$

The next case, involving three processes but just one round, is much more interesting and seems to set a precedent towards the generalized proof of the conjecture. The following theorem states that no algorithm in the write/read iterated model can solve snapshot for three processes in only one round.

**Theorem 5.1.** *Three processes cannot solve snapshot in 1 round in the read/write iterated model.*

*Proof.*

The proof makes use of the fact that the views returned by the $WScan()$ task in the iterated model do not avoid interference during a scan from other updating processes. The figure that represent all the possible sets of views as a simplicial complex is reproduced in figure 4. Assume there is an implementation $\Psi$ that solves the atomic snapshot task in the iterated read/write model for three processes in one round. Let $p_1$, $p_2$ and $p_3$ be three processes and consider the following executions. Execution $e_1$, blue-colored simplex in figure 5, is that in which process $p_1$ start solo, writes to the shared memory first and scans, thus seeing only itself, therefore it must return $\{1\}$ as its view to satisfy self-inclusion; later, process $p_2$ writes and scans, obtaining the view $\{1, 2\}$ which must be precisely the view it returns in order to satisfy the properties

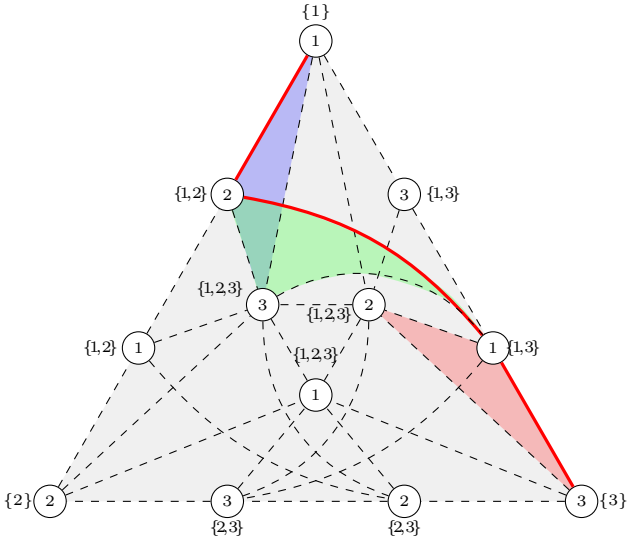


Figure 5: Path (*in red*) proving the impossibility to solve snapshot by the iterated read/write model with three processes and one round.

of the snapshot task, returning any other subset, in presence of $p_1$'s view, would breach one of the snapshot properties; finally $p_3$ reads and scans obtaining view $\{1, 2, 3\}$. Execution $e_2$, pink simplex in figure 5, is similar to $e_1$ but replacing letting process $p_2$ run first, followed by $p_3$ and finally $p_1$; therefore, $p_3$ sees only itself and must return $\{3\}$ as its view, while $p_1$ will see $\{1, 3\}$ and must return this same view in order to satisfy containment. Finally consider the following execution $e_3$, green simplex in the figure, in which $p_1$ is the first to write and then starts its scan, stopping after reading $p_2$'s register but before reading $p_3$'s register. Up to that point, $p_1$ has only seen itself written to memory. Then $p_2$ writes to memory and finishes its scan, obtaining the view $\{1, 2\}$. Later $p_3$ writes into memory and $p_1$ continues its scan, reading $p_3$'s register and obtaining $\{1, 3\}$ as a view. The executions are represented next by their matrix representations:

$$e_1: \begin{array}{c} \\ p_1 \\ p_2 \\ p_3 \\ w \end{array} \begin{array}{c} \begin{array}{ccc} p_1 & p_2 & p_3 \end{array} \\ \begin{pmatrix} 2 & 3 & 4 \\ 6 & 7 & 8 \\ 10 & 11 & 12 \\ 1 & 5 & 9 \end{pmatrix} \end{array} \qquad e_2: \begin{array}{c} \\ p_1 \\ p_2 \\ p_3 \\ w \end{array} \begin{array}{c} \begin{array}{ccc} p_1 & p_2 & p_3 \end{array} \\ \begin{pmatrix} 6 & 7 & 8 \\ 10 & 11 & 12 \\ 2 & 3 & 4 \\ 5 & 9 & 1 \end{pmatrix} \end{array} \qquad e_3: \begin{array}{c} \\ p_1 \\ p_2 \\ p_3 \\ w \end{array} \begin{array}{c} \begin{array}{ccc} p_1 & p_2 & p_3 \end{array} \\ \begin{pmatrix} 2 & 3 & 9 \\ 5 & 6 & 7 \\ 10 & 11 & 12 \\ 1 & 4 & 8 \end{pmatrix} \end{array}$$

By execution $e_1$, process $p_2$ is forced to return $\{1, 2\}$ as its final view when obtaining this same view from the WScan. Similarly, by execution $e_2$, process $p_1$ is forced to return $\{1, 3\}$ as its final view when obtaining this same view from the WScan. Returning any other subset

would render non-compliance to some snapshot property in those executions. So when running protocol $\Psi$ these two decisions are forced. The crux of the proof comes from the fact that process $p_2$ cannot distinguish execution $e_1$ from execution $e_3$ and process $p_1$ cannot distinguish between executions $e_2$ and $e_3$. This happens because the processes obtain the same views from the WScan in each pair of undistinguishable scenarios. Therefore, when running protocol $\Psi$, $p_2$ must necessarily decide $\{1, 2\}$ upon observing $\{1, 2\}$ to avoid a counterexample in execution $e_1$, and analogously, $p_1$ must return $\{1, 3\}$ upon obtaining the view $\{1, 3\}$ from the WScan. These *forced decisions* can be connected by execution $e_3$, thus rendering incompatible views which cannot be ordered by containment. In this execution, $p_1$ must decide $v_1 = \{1, 3\}$ as it cannot distinguish from execution $e_2$, while $p_2$ must decide $v_2 = \{1, 2\}$ as it cannot distinguish from execution $e_1$. We note that $(v_1 \not\subseteq v_2) \wedge (v_2 \not\subseteq v_1)$. This breach on the containment property of the atomic snapshot task is represented below as a chain of *forced decisions* and as a red path on figure 5.

$$< 1, \{1\} > \overset{e_1}{\longleftrightarrow} < 2, \{1, 2\} > \overset{e_3}{\longleftrightarrow} < 1, \{1, 3\} > \overset{e_2}{\longleftrightarrow} < 3, \{3\} >$$

□

# 6 Conclusion

Recursion offers a very useful tool for the devise and analysis of distributed algorithms. Models which use a round by round iterative perspective have proved themselves fundamental in many impossibility and lower bound results. Many algorithms over these models can be though of recursively to simplify their analysis, leading to generally simple inductive proofs over the number of recursive instances of the particular task.

We have worked over a standard MRSW register shared memory model of distributed computing in an asynchronous wait-free setting and have based our results on constructions over a most basic task of iterated models, the write-scan or WScan task. We have characterized the set of possible output views deliverable by a WScan object, first in a set-wise formulation which specifies the necessary interactions that an arbitrary set of vectors needs to satisfy in order to be obtainable as output of a WScan execution. Next, we presented a second characterization of WScan views based in matrix representations which is easier to comprehend and use. This is due to the fact that the matrix representation that corresponds to a given set of views is deeply linked to the execution that renders such output views. This kind of representation also allows any feasible execution over a shared memory abstracted by a WScan task, as concurrent as can be, to be thought of in a sequential order of isolated read and write shared memory events.

Iterated models and their non iterated versions have been proved equivalent in a wide spectrum of circumstances. For instance, the usual snapshot model and its iterated version have been shown to be equivalent in terms of wait-free solvability by the means of several proposed simulations. This work attempts to shed some light on the question of whether iterated models and their non iterated versions are equivalent in terms of complexity. A conjecture is presented to this respect by stating that the complexity to solve the atomic snapshot task in an iterated

model is strictly higher than the complexity required to solve the same task in the non iterated setting. Some particular cases are analyzed and a proof is offered for the case of three processors and one round. We believe that this proof for a particular case may lay the foundations for a future generalized proof, by observing the connectivity of certain simplices in the associated WScan protocol complex.

# References

[1] Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic snapshots of shared memory. *J. ACM*, 40(4):873–890, 1993.

[2] S. Akhter and J. Roberts. *Multi-Core Programming.* Intel Press, 2006.

[3] James Anderson. Composite registers. In *PODC '90: Proceedings of the ninth annual ACM symposium on Principles of distributed computing*, pages 15–29, New York, NY, USA, 1990. ACM.

[4] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine Yelick. A view of the parallel computing landscape. *Commun. ACM*, 52(10):56–67, 2009.

[5] Hagit Attiya, Amotz Bar-Noy, Danny Dolev, David Peleg, and Rüdiger Reischuk. Renaming in an asynchronous environment. *J. ACM*, 37(3):524–548, 1990.

[6] Hagit Attiya, Rachid Guerraoui, and Eric Ruppert. Partial snapshot objects. In *SPAA '08: Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pages 336–343, New York, NY, USA, 2008. ACM.

[7] Hagit Attiya, Maurice Herlihy, and Ophir Rachman. Atomic snapshots using lattice agreement. *Distrib. Comput.*, 8(3):121–132, 1995.

[8] Hagit Attiya and Ophir Rachman. Atomic snapshots in $O(nlogn)$ operations. In *PODC '93: Proceedings of the twelfth annual ACM symposium on Principles of distributed computing*, pages 29–40, New York, NY, USA, 1993. ACM.

[9] Hagit Attiya and Sergio Rajsbaum. The combinatorial structure of wait-free solvable tasks. *SIAM J. Comput.*, 31(4):1286–1313, 2002.

[10] Hagit Attiya and Jennifer Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. John Wiley, 2004.

[11] P. Berman, J. A. Garay, and K. J. Perry. Towards optimal distributed consensus. In *SFCS '89: Proceedings of the 30th Annual Symposium on Foundations of Computer Science*, pages 410–415, Washington, DC, USA, 1989. IEEE Computer Society.

[12] Ofer Biran, Shlomo Moran, and Shmuel Zaks. A combinatorial characterization of the distributed tasks which are solvable in the presence of one faulty processor. *J. of Algorithms*, 11:420–440, 1990.

[13] Elizabeth Borowsky and Eli Gafni. Generalized FLP impossibility result for t-resilient asynchronous computations. In *STOC*, pages 91–100, 1993.

[14] Elizabeth Borowsky and Eli Gafni. Immediate atomic snapshots and fast renaming (extended abstract). In *PODC*, pages 41–51, 1993.

[15] Elizabeth Borowsky and Eli Gafni. A simple algorithmically reasoned characterization of wait-free computations (extended abstract). In *PODC*, pages 189–198, 1997.

[16] T. Chandra and C. Dwork. Using consensus to solve atomic snapshots. Manuscript, 1993.

[17] Peter J. Denning and Jack B. Dennis. The resurgence of parallelism. *Commun. ACM*, 53(6):30–32, 2010.

[18] Khanh Do Ba. Wait-free and obstruction-free snapshot. *Senior Honors Thesis, Dartmouth Computer Science Technical Report, June 2006.*

[19] Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.

[20] G. Florin, C.R. Gomez, and I. Lavalle. Recursive distributed programming schemes. In *ISADS93*, 1993.

[21] Eli Gafni. On the wait-free power of iterated-immediate-snapshots (extended abstract), 1998.

[22] Eli Gafni and Sergio Rajsbaum. Recursion in distributed computing. *Technical Report 875, Instituto de Matemáticas, Universidad Nacional Autónoma de México, (May 30, 2010). To appear in 12th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2010), New York, September 2010.*

[23] Eli Gafni and Sergio Rajsbaum. Distributed programming with tasks. *UCLA Computer Science Dept. Technical Report TR-100001*, Nov 5, 2009.

[24] Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991.

[25] Maurice Herlihy and Nir Shavit. The topological structure of asynchronous computability. *Journal of the ACM*, 46(6):858–923, 1999.

[26] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming.* Morgan Kaufmann, 2008.

[27] Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[28] Damien Imbs and Michel Raynal. Help when needed, but no more: Efficient read/write partial snapshot. In *DISC*, pages 142–156, 2009.

[29] Amos Israeli, Amnon Shaham, and Asaf Shirazi. Linear-time snapshot implementations in unbalanced systems. *Mathematical Systems Theory*, 28(5):469–486, 1995.

[30] Lefteris M. Kirousis, Paul G. Spirakis, and Philippas Tsigas. Simple atomic snapshots: A linear complexity solution with unbounded time-stamps. *Inf. Process. Lett.*, 58(1):47–53, 1996.

[31] Dmitry Kozlov. *Combinatorial Algebraic Topology.* Springer-Verlag, 2008.

[32] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4:382–401, 1982.

[33] Nancy Lynch. *Distributed Algorithms.* Morgan Kaufmann, 1996.

[34] James R. Munkres. *Elements Of Algebraic Topology.* Westview Press, 1996.

[35] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2):228–234, 1980.

[36] S. Pllana and F. Xhafa (Editors). *Programming Multi-core and Many-core Computing Systems. Wiley Series on Parallel and Distributed Computing, 6079.* John Wiley & Sons, Inc., 2011.

[37] Sergio Rajsbaum. Iterated shared memory models. In Alejandro López-Ortiz, editor, *LATIN*, volume 6034 of *Lecture Notes in Computer Science*, pages 407–416. Springer, 2010.

[38] Sergio Rajsbaum, Michel Raynal, and Corentin Travers. The iterated restricted immediate snapshot model. In *Xiaodong Hu, Jie Wang (Eds.): Computing and Combinatorics, 14th Annual International Conference, COCOON 2008, Dalian, China, June 27-29, 2008, Proceedings. Lecture Notes in Computer Science 5092 Springer 2008.*

[39] Brian Randell. Recursively structured distributed computing systems. In *Symposium on Reliability in Distributed Software and Database Systems*, pages 3–11, 1983.

[40] Yaron Riany, Nir Shavit, and Dan Touitou. Towards a practical snapshot algorithm. *Theor. Comput. Sci.*, 269(1-2):163–201, 2001.

[41] Michael E. Saks and Fotios Zaharoglou. Wait-free k-set agreement is impossible: The topology of public knowledge. *SIAM J. Comput.*, 29(5):1449–1483, 2000.

[42] Bala Swaminathan and Kenneth J. Goldman. Hierarchical correctness proofs for recursive distributed algorithms using dynamic process creation. *Technical Report, Washington University, April*, 1993.