Distributed Computing Column 47
Distributed Computability

Idit Keidar
Dept. of Electrical Engineering, Technion
Haifa, 32000, Israel
idish@ee.technion.ac.il

Today's column deals with the theory of *computability* in a distributed system. It features a tutorial on this topic by Maurice Herlihy, Sergio Rajsbaum, and Michel Raynal. The tutorial focuses on a canonical asynchronous computation model, where processes communicate by writing to and reading from shared memory. It studies which distributed tasks can be solved in this model in the presence of process failures and communication delays, and which cannot.

The tutorial highlights two powerful techniques for obtaining computability results: First, the abstraction of an iterated write-snapshot model is used in order to simplify algorithms, and reduce the complexity of the solutions space one needs to explore for impossibility proofs. Second, concepts from combinatorial topology provide an understanding of the mathematical structure induced by possible executions of a protocol in this model.
Many thanks to Maurice, Sergio, and Michel for their contribution!

Call for contributions: I welcome suggestions for material to include in this column, including news, reviews, open problems, tutorials and surveys, either exposing the community to new and interesting topics, or providing new insight on well-studied topics by organizing them in new ways.

Computability in Distributed Computing: a Tutorial

Maurice Herlihy
Dept. of Computer Science
Brown University
USA
mph@cs.brown.edu

Sergio Rajsbaum
Instituto de Matemáticas
UNAM
Mexico City, D.F. 04510, Mexico
rajsbaum@math.unam.mx

Michel Raynal
IRISA, Univ. Rennes 1
35042 Rennes
France
raynal@irisa.fr

Abstract

What can and cannot be computed in a distributed system is a complex function of the system's communication model, timing model, and failure model. This tutorial surveys some important results about computability in the canonical distributed system model, where processes execute asynchronously, they communicate by reading and writing shared memory, and they fail by crashing. It explains the fundamental role that topology plays in the distributed computability theory.

Keywords: Agreement, Asynchronous system, Concurrency, Crash failure, Distributed computability, Distributed computing model, Fault-Tolerance, Iterated model, Liveness, Model equivalence, Recursion, Resilience, Shared memory system, Snapshot, Task, Topology, Wait-freedom.

1 Introduction

In sequential systems, computability is understood through the *Church-Turing Thesis*: anything that *can* be computed, can be computed by a Turing Machine. In distributed systems [8, 33, 42, 46, 48], where computations require coordination among multiple participants, computability questions have a different flavor. Here, too, there are many problems which are not computable, but these limits to computability reflect the difficulty of making decisions in the face of ambiguity, and have little to do with the inherent computational power of individual participants.

If the participants could reliably and instantaneously communicate with one another, then each one could assemble the complete system state, and the computation could proceed sequentially. In any realistic model of distributed computing, however, each participant initially knows only part of the global system state and uncertainties caused by failures and unpredictable timing limit each participant to an incomplete picture of the global system state.

Distributed computing is concerned with certain kinds of functions, called *tasks*, that may be computable even in the presence of failures and communication delays. This model yields a rich mathematical structure closely tied to notions of combinatorial topology. Specifically:

1. There are Turing-computable functions that are not computable even in the presence of a single failure. For example, solving the consensus task is trivial if there are no failures, but impossible even if only one process may crash.
2. The question whether a task is 1-resilient computable can be reduced to a question of graph connectivity, and is therefore decidable [9, 18].
3. However, the question whether a task is computable in the presence of more complex failure models is reducible to the question whether an associated geometric structure, called a simplicial complex has higher dimensional "holes," which is known to be undecidable [20, 26].
4. Finally, similar to the oracles of classic computability, there are tasks which are computable only when given access to a distributed oracle for other tasks, leading to infinite hierarchies of tasks, e.g. [19, 27, 40].

The rest of this paper is organized as follows. Section 2 discusses the range of models in distributed computing, and why we choose to focus on the wait-free read-write model. Section 3 describes this model in detail. Section 4, introduces the notion of a *task*, the basic unit of concurrent computation. The standard model, while intuitively appealing, can be cumbersome, so Section 5, describes the *iterated write-snapshot model* which provides nicer abstractions, and yet is computationally equivalent to the standard model. Section 6 introduces concepts from combinatorial topology that can be used to reveal some elegant aspects of this model. Finally, Section 7 reviews the *safe agreement* task, a problem that can be solved simply and elegantly in the iterated write-snapshot model.

2 Overview

Our model, like any such model, is an abstraction. As with Turing machines, our aim is not to try to represent faithfully in detail the way a distributed or concurrent system works. Instead, by starting with a clean, basic abstraction, we can focus on the essential properties of distributed computation.

A distributed system is a set of communicating state machines called *processes*. We need not assume that each process is a Turing machine. The inherent limits to computability are unchanged even if each process were an infinite-state automaton capable of computing non Turing-computable functions.

Processes can *fail*. Here, we consider only *crash failures*: a faulty process simply halts and takes no further steps. In other models, not considered here, processes can display Byzantine failures [38], in which they can display arbitrary behavior, including malicious behavior. For now, we will assume a *wait-free* model where any proper subset of the set of processes may crash in any execution. Later, we will consider a more general adversary model where only certain subsets of processes may fail.

There are many possible communication models for distributed computation. Here, we assume the processes communicate simply by reading and writing a shared memory. Other popular models, such as message-passing, or various networking models that limit direct process-to-process connectivity, are essentially equivalent or less powerful than shared memory. Models that make use of atomic synchronization primitives such as "compare-and-swap" are more powerful. We choose the shared-memory communication model because its simplicity highlights the fundamental properties of distributed computation. Moreover, modern multiprocessor systems usually provide some kind of shared-memory abstraction, so the shared-memory model reflects current practice, although in a highly idealized way. (Many results about other models can be obtained from this one, via reductions and simulations [13, 30].)

There are several possible timing models. In a *synchronous* model, processes take steps at exactly the same time, while in the *asynchronous* model, their relative execution speeds may be unrelated. In between, there are semi-synchronous models that place a bound on processes' relative speeds. Here, we will be concerned primarily with the asynchronous model, which is more realistic than the synchronous model, and yet easier to analyze than semi-synchronous models.

Notice that failures and asynchrony interact: if one process fails to receive a message from another, it may be that the other process has crashed, or it may be that the sender is just slow. This ambiguity lies at the heart of computability problems in distributed computing.

In sequential computing, the study of *functions* is a central concern. A Turing machine starts with a single input, computes for a finite duration, and halts with a single output. In distributed computing, the analog of a function is called a *task*. Here, the input is split among the processes: initially each process knows its own input value, but not the others'. As each process computes, it communicates with the others, and eventually it halts with its own output value. (We can think of each process as having its own tape, which initially holds its part of the input and eventually holds its output.) Collectively, the individual output values form the task's output. Unlike a function, which deterministically carries a single input value to a single output value, interesting task specifications are typically non-deterministic to accommodate the non-determinism introduced by failures and asynchrony.

See [3, 19, 32] for a more complete discussion of this model, and related models. We will now examine some components of the model in more detail.

3 Model

There are $n+1$ sequential processes, denoted $p_0, \ldots, p_n$, that communicate by reading and writing a shared memory. Each memory location, sometimes called an *atomic register* [39], is written by a single process and read by all processes. Read and write operations to a single register are linearizable [34]: each operation appears to take place instantaneously at some instant between when the operation is called and when it returns. As noted, processes can fail by crashing: the process halts and takes no further steps. A *non-faulty* process is one that does not crash. If a process crashes before taking any steps, we say that it does not *participate* in the execution. A *protocol* is a finite program in which each process starts with a private input, repeatedly communicates with the others, and halts with an output. A protocol is *wait-free* if every non-faulty process eventually completes [25].

While processes may crash, we assume that the memory itself is reliable. Others [24, 46] have considered models where it is necessary to build a reliable memory from unreliable components.

The basic unit of distributed computation is called a *task*, and fills the same role as a function is sequential computation (see Figure 1). In a task T, each of the $n+1$ processes is initially assigned an input value, and each non-faulty process must irrevocably choose an output value. The task specification determines which outputs are permitted for which inputs. Each process is initially unaware of the others' input values. Our formal definition of a task uses elementary notions taken from combinatorial topology [3, 19, 32].

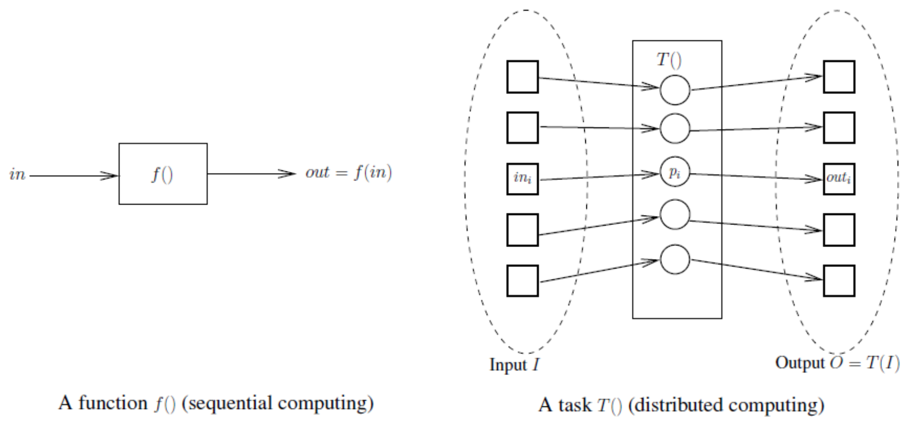
in
$f()$
$out = f(in)$
A function $f()$ (sequential computing)
$T()$
in_i
p_i
out_i
Input I
Output $O = T(I)$
A task $T()$ (distributed computing)

Figure 1: Function vs task

Consider an execution in which each process p_i has input value x_i. We can characterize the initial state of each process by a pair, (p_i, x_i), which we will call a *vertex*. If only the processes $p_{i_0}, \ldots, p_{i_k}$ participate, then the initial state of the system is characterized by the set of pairs associated with the participating processes:

$$s = \{(p_{i_0}, x_{i_0}), \ldots, (p_{i_k}, x_{i_k})\}.$$

We call such a set of pairs a *simplex*. (Note that the initial values assigned to non-participating processes are irrelevant, since they can have no effect out the values chosen by the participating processes.) We denote by names(s) the set of processes in s, and views(s) the set of its values. Note that if s is a simplex representing the start of an execution, so is any $s' \subseteq s$, because s' describes the start of an execution where fewer processes participate. A *simplicial complex* (or *complex*) $\mathcal{K}$ is a set of simplices closed under inclusion: if $s \in \mathcal{K}$ and $s' \subseteq s$, then $s' \in \mathcal{K}$. The set of all possible input value assignments for a task forms a complex. In distributed computing, each vertex v of a simplex is typically labeled with a distinct process p_i. We often say that a complex consisting of such simplexes is *colored* with those process names.

The *dimension* of a simplex s is $|s| - 1$, one less than the size of the set. The *dimension* of a complex is the largest dimension of any of is simplices, and a complex is *pure* if every maximal simplex in that complex has the same dimension. A simplex of dimension d is sometimes called a *d-simplex*, and similarly for complexes.

A simplex of dimension one is sometimes called an *edge*, dimension two a *triangle*, and dimension three a *tetrahedron*. A simplicial complex that contains only vertices and edges is called a *graph*.

A *subdivision* of a geometric complex $\mathcal{A}$ is constructed by "dividing" the simplexes of $\mathcal{A}$ into smaller simplexes. See Figure 2.

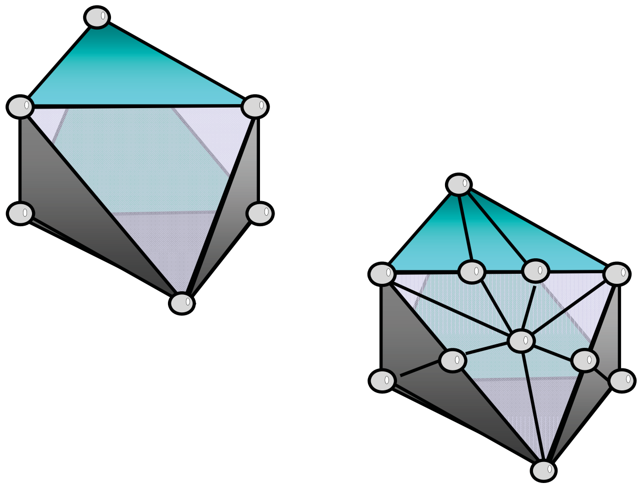

Figure 2: Complex (top) and subdivided complex (bottom)

4 Tasks

Formally, an $(n+1)$-process task is defined by a triple, $(\mathcal{I}, \mathcal{O}, \Delta)$. $\mathcal{I}$ is a pure n-dimensional complex that defines the possible input value assignments: if $\{(p_{i_0}, x_{i_1}), \ldots, (p_{i_k}, x_{i_k})\}$ is a simplex of $\mathcal{I}$, then it is possible for an execution to start with each p_{i_j} assigned input value x_{i_j}, for $0 \leq j \leq k$. Similarly, $\mathcal{O}$ is a pure n-dimensional complex that defines the possible choices of output values: if $\{(p_{i_1}, y_{i_1}), \ldots, (p_{i_k}, y_{i_k})\}$ is a simplex of $\mathcal{O}$, then it is possible for an execution to finish with each p_{i_j} having chosen output value y_{i_j}, for $0 \leq j \leq k$. Δ is a map that assigns each input simplex s in $\mathcal{I}$ a subcomplex $\Delta(s) \subseteq \mathcal{O}$, with the following interpretation: if the system begins in state $s \in \mathcal{I}$, then every execution must finish in some state $t \in \Delta(s)$.

The map Δ satisfies certain formal properties. Every process that finishes an execution must have started, so Δ must be *color-preserving*: for each $t \in \Delta(s), \text{names}(t) \subseteq \text{names}(s)$. Consider an execution in which the processes in $s \in \mathcal{I}$ participate, but a subset $s' \subset s$ finish before the rest start. The late-starting processes must be able to choose values compatible with the values already chosen by the early-starting processes. This is captured by the following *carrier map* requirement

$$\text{If } s' \subseteq s, \text{ then } \Delta(s') \subseteq \Delta(s).$$

A protocol *solves* task $(\mathcal{I}, \mathcal{O}, \Delta)$ if, for every input simplex $s \in \mathcal{I}$, and every execution of the protocol starting from s, every non-faulty process chooses an output value, and the simplex t defined by these choices is in $\Delta(s)$.

4.1 Example of Tasks

The ϵ-*agreement* task, for $\epsilon > 0$, is defined as follows [17]. Each process starts with a value from $\{0, 1\}$. Processes must decide values which are at most ϵ from each other, and if all start with the

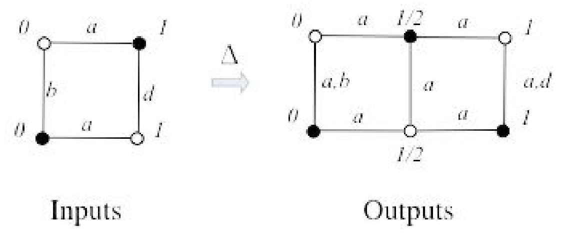

Figure 3: Approximate ϵ-agreement task for two processes and $\epsilon = 1/2$

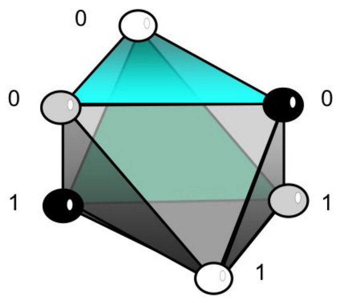

Figure 4: Binary input complex for 3 processes.

same value, they must all decide that value. We consider the case where $\epsilon = \frac{1}{2^x}$, for an integer $x > 0$. Figure 3 (from [19]) illustrates the binary approximate agreement task [17] for two processes, where $\epsilon = 1/2$. The set of possible input values is $\{0, 1\}$. A vertex $(p_i, j), i, j \in \{0, 1\}$, is represented in the figure as a circle, white for p_0 and black for p_1, labeled with value j. The input complex is on the left, and the output complex on the right. Each input value is either 0 or 1. If both start with the same value, they must both decide that value. In the figure, this requirement corresponds to an edge labeled b or d. The two edges where processes start with distinct values, are labeled a. In this case, Δ allows a process to decide any of $\{0, \frac{1}{2}, 1\}$, as long as the difference between the decided values is at most $\frac{1}{2}$. In particular, Δ of an input edge labeled a includes any output edge labeled a. Executions in which only one process participates are captured by defining $\Delta(p_i, j) = (p_i, j)$, for $i, j \in \{0, 1\}$.

The input complex for three processes is depicted in Figure 4. Each input simplex for 3 processes corresponds to a triangle, where processes start with either 0 or 1. The ϵ-agreement task has a wait-free protocol, for any $\epsilon > 0$, as described in Section 6.2.2.

While processes can agree on values arbitrarily close to each other, using a wait-free ϵ-agreement protocol, they cannot reach perfect agreement. When ϵ is set to zero, this problem is known as consensus (Figure 5), which has no wait-free protocol in this model [18, 41]. The consensus task can be relaxed to the *k-set agreement* task, where each process chooses an input value, but as many as k distinct values may be chosen [16]. This task can be solved if and only if the number of processes

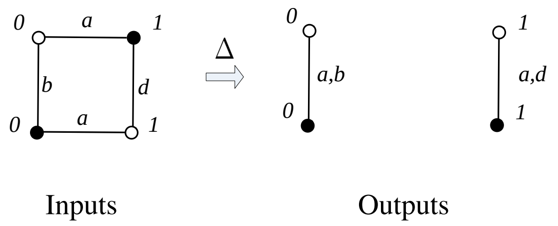

Figure 5: Consensus task for two processes

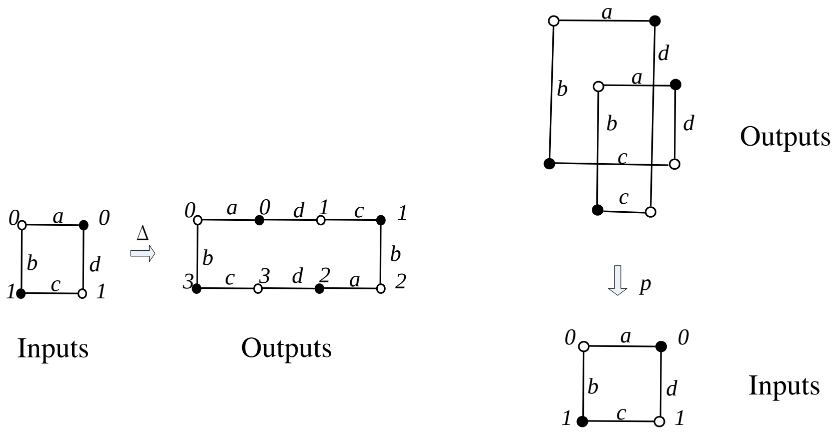

Figure 6: A two-cover task

that may crash is less than k [10, 32, 47]. In particular, consensus, which is 1-set agreement, has a protocol only if no process may crash.

Locality-preserving tasks [19] have a different nature. It is known that they do not have wait-free protocols in this model [32]. Figure 6 (from [19]) shows a two-process example. Processes start with binary inputs. On the left side of the figure, the task specification says that if both start with the same value, they must decide the same value: if they start with 0, decide either 0 or 2; if they start with 1, decide 1 or 3. If they start with different values, the valid outputs are defined in the figure, via edge labels b or d. The relation Δ also defines the possible outputs when only one process runs: for example, when the white process starts with 0, it can decide 0 or 2, while if it starts with 1, it can decide 1 or 3. Notice that $\mathcal{O}$, a cycle of length 8, locally looks like $\mathcal{I}$, a cycle of length 4, in the sense that the 1-neighborhood of each vertex v in $\mathcal{I}$ is identical to the 1-neighborhood of a corresponding vertex in $\Delta(v)$. The right side of the figure shows how $\mathcal{O}$ covers $\mathcal{I}$, by wrapping around it twice, where p identifies edges with the same label. Informally, $\mathcal{O}$ *covers* $\mathcal{I}$ means that there is a map p from $\mathcal{O}$ to $\mathcal{I}$, such that for each vertex v in $\mathcal{I}$, if one considers the vertexes w_i, with $p(w_i) = v$, the the neighborhoods around each w_i look identically to the neighborhood around v.

Other tasks include loop agreement [27] (see description below), k-simultaneous consensus [4], locality preserving tasks [19], renaming [6], (see [15] for an introductory survey), weak symmetry breaking, and generalized symmetry breaking tasks [35] (a family of tasks that that includes renaming and weak symmetry breaking).

Locality preserving tasks lead to infinite hierarchies of tasks, where tasks are computable only

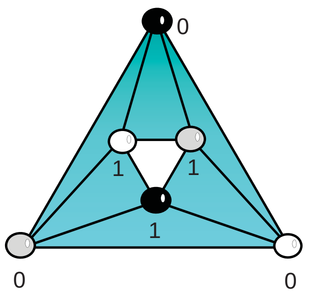

Figure 7: Weak symmetry breaking output complex for two processes

when given access to a distributed oracle for other tasks, e.g. [19, 27, 40].

4.2 Colored vs Colorless Tasks

Many of the tasks previously studied, such as consensus, set agreement, approximate agreement and loop agreement are *colorless* in the sense that they can be defined only in terms of *sets* of possible input and output values, without having to specify which process can be assigned or choose which value. In a colorless task, a process may adopt the input or output value of another process, without violating the task's specification.

One example of a task that is *not* colorless is the *renaming* task [6]. In this task, processes start with unique names taken from a large name space, and must halt with unique names taken from a smaller name space. Here, a process cannot adopt another's output name because the output names would not be unique. A similar remark holds for the weak symmetry-breaking task, which requires that if all processes participate, at least one must decide 0 and at least one must decide 1. The output complex for three-processes weak symmetry-breaking is illustrated in Figure 7 (the inner white triangle is a forbidden state where all processes choose the same output value).

A colorless task $(\tilde{\mathcal{I}}, \tilde{\mathcal{O}}, \tilde{\Delta})$ is defined in terms of complexes which are not colored with process names, and may be of any dimension, unrelated to the number of processes. Indeed, a colorless task specifies a *family* of tasks, one for each number of processes $n > 1$.

Definition 4.1. A colorless task $\tilde{T}$ is a triple $(\tilde{\mathcal{I}}, \tilde{\mathcal{O}}, \tilde{\Delta})$ where $\tilde{\mathcal{I}}$ and $\tilde{\mathcal{O}}$ are complexes and $\tilde{\Delta}$ is a carrier map from $\tilde{\mathcal{I}}$ to $\tilde{\mathcal{O}}$.

Each vertex of $\tilde{\mathcal{I}}$ and $\tilde{\mathcal{O}}$ is just a value, not a process name, value pair as in the colored case. Because $\tilde{\Delta}$ is a carrier map (as defined at the beginning of Section 4), $\tilde{\Delta}(\tilde{s'}) \subseteq \tilde{\Delta}(\tilde{s})$ for every $\tilde{s'}, \tilde{s} \in \tilde{\mathcal{I}}$, such that $\tilde{s'} \subseteq \tilde{s}$.

If $\tilde{s}$ is a simplex of $\tilde{\mathcal{I}}$, then there is an initial system state where $\tilde{s}$ is the processes' set of input values. Symmetrically, a set of output values $\tilde{t}$ is a simplex in $\tilde{\mathcal{O}}$ if there is a final system state where $\tilde{t}$ is the process set of the output values. Operationally, the processes are initially assigned (not necessarily distinct) vertices from a simplex $\tilde{s}$ in $\tilde{\mathcal{I}}$, and they halt after choosing (not necessarily

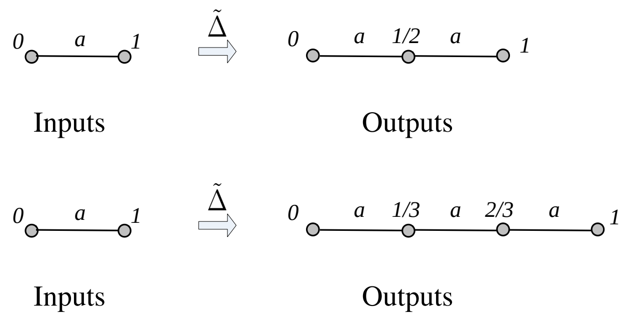

Figure 8: Two colorless ϵ-agreement tasks for two processes and $\epsilon = 1/2, 1/3$

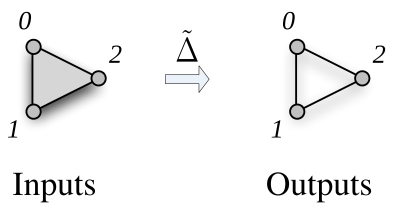

Figure 9: Colorless 2-set agreement task

distinct) vertices from a simplex $\tilde{t}$ in $\tilde{\mathcal{O}}$, where $\tilde{t} \in \tilde{\Delta}(\tilde{s})$.

Formally, for each n, a colorless task $\tilde{T} = (\tilde{\mathcal{I}}, \tilde{\mathcal{O}}, \tilde{\Delta})$ induces a task $T = (\mathcal{I}, \mathcal{O}, \Delta)$ for n processes, as follows. An input simplex s is in $\mathcal{I}$ if the names in s are distinct, and the set of values of s form a simplex $\tilde{s} \in \tilde{\mathcal{I}}$. Similarly for an output simplex t in $\mathcal{O}$, $\tilde{t} \in \tilde{O}$ with $\tilde{O} \in \Delta(\tilde{s})$. We say $\tilde{s}$ is obtained from s by projecting out the names. Now, $t \in \Delta(s)$ if (s, t) is an input-output pair, and there are projections $\tilde{s}$, $\tilde{t}$, with $\tilde{t} \in \tilde{\Delta}(\tilde{s})$.

The approximate agreement task with $\epsilon = 1/2$ of Figure 3 is more succinctly represented as a colorless task, in Figure 8, where the case of $\epsilon = 1/3$ is also depicted. Notice that tasks for any number of processes are implied. That is, processes start with input value 0 or 1. If they all start with the same value, they all decide that same value. Otherwise, they all decide values at most ϵ apart from each other: values that form an output simplex (a vertex or an edge in the output complex).

A 2-set agreement task is succinctly represented as a colorless task, in Figure 9. Any number of processes, n, start with values from the set $\{0, 1, 2\}$ and decide values from the same set. The values decided must belong to the values proposed in the execution. At most two different values can be decided. Thus, the input complex is induced by a colorless solid triangle, while the output complex is induced by a colorless triangle without the interior. Formally, $\tilde{\Delta}$ is defined:

- if $\tilde{s} = \{v\}$ then $\tilde{\Delta}(\tilde{s}) = \{v\}$;
- if $\tilde{s} = \{u, v\}$ then $\tilde{\Delta}(\tilde{s}) = \{\{u, v\}, \{u\}, \{v\}\}$;
- if $\tilde{s} = \{u, v, w\}$ then $\tilde{\Delta}(\tilde{s}) = \{\{u, v\}, \{u, w\}, \{v, w\}, \{u\}, \{v\}, \{w\}\}$.

Intuitively, solving a colorless task should be easier, because the algorithm can be *anonymous*. Since the task is defined by possible sets of input and output values without specifying process names, a process solving a colorless task should not have to use its id.

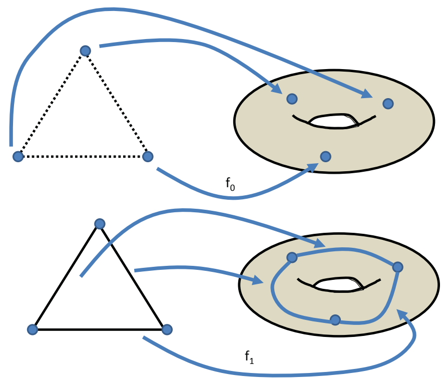

Figure 10: A loop agreement task

4.3 Colorless Tasks as Convergence Problems

One can think of colorless task $(\tilde{\mathcal{I}}, \tilde{\mathcal{O}}, \tilde{\Delta})$ as a kind of convergence task, where processes must rendez-vous on a simplex in $\tilde{\mathcal{O}}$.

For example, consider the following loop agreement task [27], illustrated in Figure 10. A *edge loop* $K = \vec{k}_0, \ldots \vec{k}_k$ is a sequence of vertices such that each $\vec{k}_i$ and $\vec{k}_{i+1}$ form an edge of $\mathcal{K}$, and all vertices are distinct except that $\vec{k}_0 = \vec{k}_k$. Let $\mathcal{K}$ be a finite (uncolored) 2-dimensional simplicial complex, K a simple loop of $\mathcal{K}$, and k_0, k_1, k_2 three distinguished vertexes in $\mathcal{K}$. For distinct i, j, and k, let K_{ij} be the sub-path of K linking k_i to k_j without passing through k_k. Each of $n+1$ processes has an input value in $\{0, 1, 2\}$. For each input simplex s, define $\Delta(s)$ by:

views(s)	$\Delta(s)$,
$\{i\}$	all decide $\vec{k}_i$,
$\{i, j\}$	vertexes span simplex in K_{ij},
$\{0, 1, 2\}$	vertexes span simplex in $\mathcal{K}$.

In other words, the processes converge on a simplex in $\mathcal{K}$. If all processes have the same input value, they converge on the corresponding vertex. If they have only two distinct input vertexes, they converge on some simplex along the path linking their vertexes. Finally, if the processes have all three input vertexes, they converge to any simplex of $\mathcal{K}$.

For instance, 2-set agreement is specified by the loop agreement task where $\mathcal{K}$ consists of the three edges of the triangle with vertices k_0, k_1, k_2, which is "hollow", namely with no 2-dimensional simplexes. In this version of set agreement, processes decide on at most two of the values k_0, k_1, k_2.

5 The Iterated Write-Snapshot Model

The base read-write model is natural, in the sense that it roughly captures the behavior of modern multiprocessors, but it can be difficult to work with directly. Here, we show how this model can be recast into a more manageable form, by defining a write-snapshot abstraction on top of the read-write model. The two models are equivalent, in the sense that any task that is computable in one is computable in the other. We do not consider complexity issues here, and indeed the iterated model may be less efficient.

It is often convenient to structure asynchronous computations as round-based executions [14, 31, 43]. In each round, processes communicate through a shared memory that can be accessed only in that round. In the simplest case the shared memory for each round consists of an array of single/writer multi/reader registers. In each round each process writes to its register and then reads one by one all the registers. A more structured iterated model is obtained if the shared memory at each round provides a snapshot operation.

5.1 The Snapshot Abstraction

A snapshot object provides the programmer with a convenient, high-level shared memory abstraction, but provides no additional computational power. In Section 6.2.1 we will show how one can construct and use a high-level snapshot object from read-write memory, e.g. [5]. This has been done in [1], but our construction will be recursive.

A snapshot transforms an array X of individually readable and writable memory locations, with one entry per process, into an array that provides an additional operation: X.snapshot(). The X.snapshot() operation returns the current value of the entire array X. A snapshot object is *linearizable*: each write or snapshot operation appears as if it has been executed instantaneously between when it is called and when it returns. The most efficient implementations of wait-free snapshots on top of linearizable read/write known so far has $O(n \log n)$ time complexity [2]. Constructions for the case of partial snapshots have been proposed in [5, 7, 36].

5.2 The Iterated Write-Snapshot Model

A one-shot write-snapshot object[11] is an array $WS[0..n]$, initialized to $[\bot, \dots, \bot]$, which can be accessed by a single write_snapshot() operation that each process invokes at most once. That operation pieces together the write() and snapshot() operations. Intuitively, when a process p_i invokes write_snapshot(v), it is as if it instantaneously executes a write $WS[i] \leftarrow v$ operation followed by a WS.snapshot() operation. If several WS.write_snapshot() operations are executed simultaneously, then their corresponding writes are executed concurrently, and then their corresponding snapshots are also executed concurrently. Each concurrent snapshot sees the values written by the concurrent writes. These operations are set-linearizable [44], meaning that each operation appears as if it has been executed instantaneously between its start event and its end event, and if operations appear to happen at the same time, each sees the value written by the others.

A write_snapshot() operation invoked by p_i behaves as follows. Let v_i be the value written by p_i, and sm_i the value (or view) p_i gets back from the operation. A view sm_i is a set of pairs (k, v_k), where v_k corresponds to the value in p_k's entry of the array. If $WS[k] = \bot$, the pair $(k, \bot)$ is not placed in sm_i. Moreover, we assume that $sm_i = \bot$ if p_i never invokes WS.write_snapshot(). Every invocation of WS.write_snapshot() by a non-faulty process returns with a value.

Definition 5.1. A *write-snapshot* object is a one-shot object whose single operation write_snapshot() is such that, if a process invokes write_snapshot(v_i), it obtains a sm_i satisfying the following properties:

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

- Immediacy. $\forall i, j : [(i, v_i) \in sm_j \wedge (j, v_j) \in sm_i] \Rightarrow (sm_i = sm_j)$.

The self-inclusion property states that a process sees its write, while the containment properties states that the views obtained by processes are totally ordered. Finally, the immediacy property states that if two processes "see each other", they obtain the same view (the size of which corresponds to the *concurrency* degree of the corresponding write_snapshot() invocations), and extends snapshots to *immediate snapshots* also called *block executions* [10, 47]. (Implementations are described in [11, 15, 46].)

The Iterated write-snapshot model The iterated write-snapshot model (IWS) is made up of an unbounded number of one-shot write-snapshot objects. These objects, denoted $WS[1], WS[2], \ldots$, are accessed sequentially and asynchronously by each process, according to the round-based pattern illustrated in Figure 11.

```
$v_i \leftarrow input$; $r_i \leftarrow 0$;
loop forever $r_i \leftarrow r_i + 1$;
             $sm_i \leftarrow WS[r_i]$.write_snapshot($v_i$);
             $v_i \leftarrow sm_i$;
end loop.
```

Figure 11: Generic algorithm for the iterated write-snapshot model

Initially, p_i stores its input in its local variable v_i. In each round r_i, p_i writes its current value v_i in the $r_i -$ th write-snapshot object, and stores the result in sm_i. This is the value v_i (called its *view*) that it will write in the next iteration. A process thus writes everything "it knows" in each iteration, because we are interested in computability results. If efficiency is a concern, p_i could write in the next round a value computed from sm_i.

Solving Tasks To solve a task $T = (\mathcal{I}, \mathcal{O}, \Delta)$ in the IWS model, we split the algorithm into two parts. In the first, each process repeatedly writes its view to a shared memory and then constructs a new view by reading the memory, as in the generic algorithm. This part is generic, in the sense that it could be part of any algorithm for any task. In the second part, however, each process decides how many iterations to execute, and then applies a task-specific decision map to its final view to determine its (irrevocable) decision value. The number of iterations, and the decision map depend on the task being solved. For the task T, the initial views of the processes form an input simplex $s \in \mathcal{I}$, and the decision values form an output simplex t, such that $t \in \mathcal{O}$.

5.3 On the Power of the Iterated Model

Let us observe that the IWS model requires each non-faulty process to execute an infinite number of rounds (although a decision is taken in a finite number of rounds). However, it is possible that a non-faulty process p_1 is unable to receive information from another non-faulty process p_2. Consider a execution where both execute an infinite number of rounds, but p_1 is scheduled before p_2 in every round. Thus, p_1 never reads a value written to a write-snapshot object by p_2. Of course, in the usual (non-iterated read/write shared memory) asynchronous model, two non-faulty processes can always eventually communicate with each other. Thus one might think that the base read/write model and the IWS model have different computability power. The fundamental result associated with the IWS model is captured by the following theorem, which shows they are, in fact, equivalent.

Definition 5.2. A task is *bounded* if its set of input vectors $\mathcal{I}$ is finite.

The following theorem appeared first in [12]. (A new simulation from the snapshot model to the iterated model is described in [22], which can be extended to work with stronger models.)

Theorem 5.3. A bounded colorless task can be wait-free solved in the *base read-write* model model if and only if it can be wait-free solved in the IWS model.

The appeal of the IWS model comes from its simple round-by-round iterative structure. Its executions have an elegant recursive structure: the structure of the global state after $r + 1$ rounds is obtained from the structure of the global state after r rounds, which eases the analysis of wait-free asynchronous computations to prove impossibility results [28, 29]. The recursive of execution structure also facilitates the design and analysis of algorithms [15, 21].

6 The Protocol Complex

6.1 The Protocol Complex and its Recursive Structure

Consider the generic algorithm of Figure 11, and the possible views v_i, at every iteration, r, $r \geq 0$. The view v_i of p_i at r is denoted $view_i^r$. Characterizing the possible views yields a characterization of which tasks can be computed in the iterated write-snapshot model.

Assume the generic algorithm is used to solve a task T defined by the triple $(\mathcal{I}, \mathcal{O}, \Delta)$. Recall that each input simplex I in $\mathcal{I}$, is a set of pairs

$$I = \{(p_i, x_i)\}$$

for some subset of the processes, $\{p_i\}$, each one with input value x_i. Therefore, each input simplex corresponds to a possible set of initial views, where $view_i^0 = x_i$. The set of all possible initial views is thus equal to the input complex $\mathcal{I}$.

In general, the set of views at round r, is always a complex, $\mathcal{K}^r$, called the *protocol complex*, because if s is a set of views of some processes at round r, then any subset of s is also a set of views at r. More precisely, a simplex $s \in \mathcal{K}^r$ consists of a set of pairs $\{(p_i, v_i)\}$, corresponding to a subset of the processes, and an execution of the generic algorithms, where each v_i is a view that p_i can obtain at iteration r of the execution.

What is the complex of views at round $r = 1$? It is defined by the write-snapshot properties of Definition 5.1: Self-inclusion, Containment, and Immediacy. All views possible at the first round form a subdivision of the input simplex. In Figure 12 the case where $\mathcal{I}$ is the binary input complex of consensus or approximate agreement is depicted. Each vertex of a process p_i is labeled with its view v_i. Notice that the 4 vertices at the corners of the square, correspond to executions where a process sees only itself. The other vertices correspond to views where a process sees both its own value and the value of the other process.

We can think of the algorithm itself as a carrier map $\mathcal{P}(\cdot)$ that carries each simplex of the input complex to a subcomplex of the protocol complex. Each input vertex goes to the solo execution in which that process alone participates, and each input simplex goes to the subcomplex of all possible executions starting with those inputs.

The protocol complex is related to the output complex by the decision map $\delta(\cdot)$, a map that sends each vertex v in the protocol complex to a vertex w in the output complex labeled with the same process name, and such that $\delta(s)$ is a simplex in $\mathcal{O}$. The value attached to w is the value

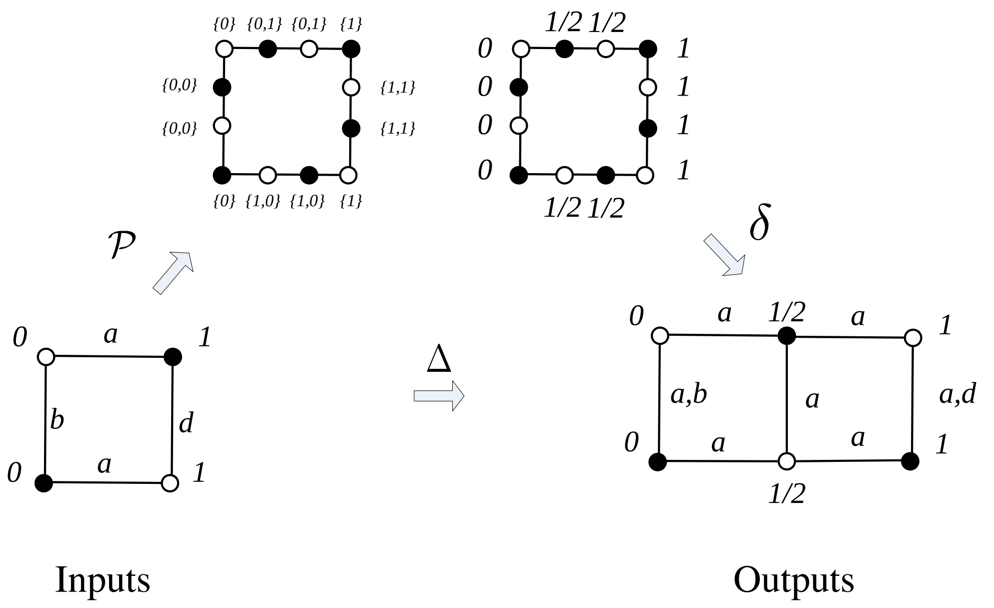

Figure 12: Approximate agreement task for two processes is below. The protocol complex after one round is on top (views on the left, decisions on the right).

which process $id(v)$ takes as its output value when its final view in the execution is $view(v)$. The map δ is *simplicial* because it preserves simplexes: if vertices v_i form a simplex in the protocol complex, then the vertices $\delta(v_i)$ form a simplex in the output complex.

The algorithm is correct if the composition of the decision map δ with the carrier map $\mathcal{P}$ is a carrier map F such that, for any $s \in \mathcal{I}$, $F(s) \subseteq \Delta(s)$.

Recursive structure The iterated model has the advantage that each round is independent of previous rounds. The outputs of round r become the inputs to round $r+1$. This is why one can reason recursively in this model, and the complex at round $r+1$ is obtained by replacing each simplex by the same complex.

For the case of two processes, let us consider in Figure 12. Each edge in the input complex (complex at round 0) is replaced by a path of three edges in the complex at round 1. Notice that whenever there is a vertex contained in two edges, in round 1, there is a situation in which the process corresponding to that vertex, does not distinguish between two executions, the ones represented by the two edges.

In general, the complex at round $r+1$ is obtained by replacing each edge of the complex at round r by a path of three edges in the complex at round r. Think of the algorithm as "stretching" each input edge to a path by inserting new vertices between the endpoints. The case of three processes, represented in Figure 13, will be explained below.

Next, think of the decision map as carrying this stretched path to the matching path in the output graph, in a way that respects Δ. This idea is used to prove the following result. It is a special case of a more general *manifold* property that is preserved from round to round, e.g. [23], and even the more general property of higher dimensional connectivity e.g. [28].

Lemma 6.1. Let r be any round. If the input complex $\mathcal{I}$ $(= \mathcal{K}^0)$ for the generic iterated write-snapshot algorithm is connected, then the complex made up of the views at round r, is also connected.

As a corollary of this lemma, we obtain that the consensus task is not solvable by the generic iterated write-snapshot algorithm. Let us consider the case of two processes, for concreteness, but the general argument is the same. Assume for contradiction that consensus can be solved, say after r rounds. Consider the corresponding complex of views $\mathcal{K}^r$. It is a subdivision of the input simplex, and hence it consists of a graph of four paths joined at the corners of a square. This graph is presented at the top left part of Figure 12, for $r = 1$. These four corners correspond to executions where a process sees only itself, in every iteration. In this complex, each process decides either 0 or 1. The decision colors each vertex with a binary value. The graph with binary values is at the top right part of Figure 12 for approximate agreement; in the case of consensus decisions of 1/2 would be replaced by either 0 or 1.

Finally, we use Sperner's Lemma to complete the proof. Consider the path at at the top of the square $\mathcal{K}^r$, the one that connects the corner vertex where the white process starts with 0 and the vertex where the black process starts with 1. The decision of the white process in the corner is 0 while the decision of the black process in the corner is 1. There must be an edge in the path whose vertices are colored with different binary values, because the path is connected and its endpoints have different colors. This edge corresponds to an execution where two different values are decided, contradicting the agreement requirement of the consensus task.

The case of three processes is represented in Figure 13 (from [45]). The highlighted 2-simplex on the left represents an execution where p_1 and p_3 access the object concurrently, each sees the other, but not p_2, who accesses the object later, and observes all 3 values. But p_2 cannot tell the order in which p_1 and p_3 accessed the object. The execution in which p_1 saw only itself, and the execution in which p_3 saw only itself are indistinguishable to p_2. These two executions are represented by the 2-simplexes at the bottom corners of the left picture. Thus, the vertices at the corners of the complex represent the executions where only one process p_i accesses the object, and the edges connecting two vertices on a border of an input simplex represent executions where only two processes access the object. The triangle in the center of the complex represents the execution where all three processes access the object concurrently, and get back the same view.

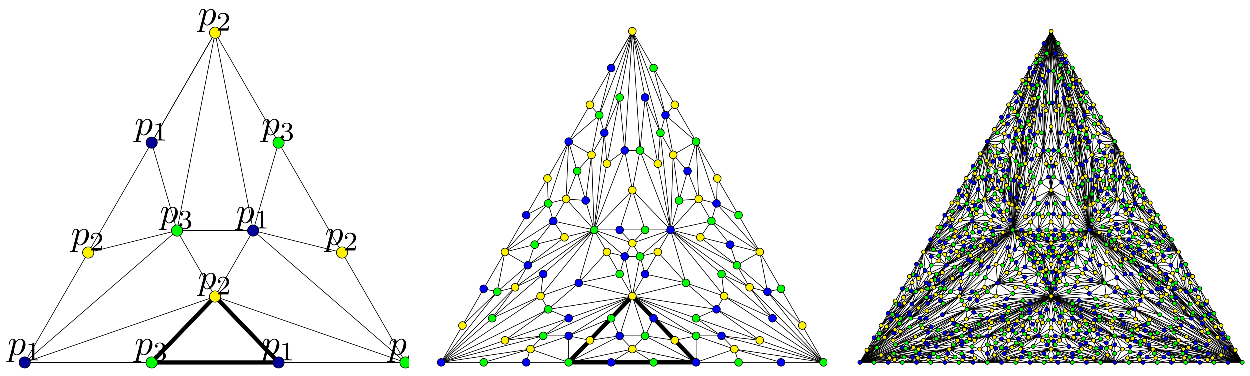

Figure 13: One, two and three rounds in the iterated write-snapshot (IWS) model

Hence, the state of an execution after the first round (with which the write-snapshot object

$WS[1]$ is associated), is represented by one of the internal triangles of the left picture (e.g., the one discussed previously that is represented by the bold triangle in the pictures). Then, the state of that execution after the second round (with which the write-snapshot object $WS[2]$ is associated), is represented by one of the small triangles inside the bold triangle in the picture in the middle, etc. More generally, as shown in Figure 13, one can see that, in the write-snapshot iterated model, at every round, a new complex is constructed recursively by replacing each simplex by a one-round complex.

6.2 Programming in the Iterated Model and Distributed Recursion

6.2.1 A Recursive Write-Snapshot Algorithm

Recall that the IWS model is an iterated model where processes communicate in each round via write_snapshot() operations. In this section we show how to implement these operations from single/writer multi/reader registers. Figure 14 presents a read/write algorithm that implements the write_snapshot() operation. Interestingly, this algorithm is recursive [15, 21] (see [15] for a proof). To allow for a recursive formulation, an additional recursion parameter is used. More precisely, in a round r, a process invokes SM.write_snapshot(n, v) where the initial value of the recursion parameter is n and SM stands for $WS[r]$.

SM is a shared array of size $n+1$ initialized to $[\perp, \ldots, \perp]$ and such that each $SM[x]$ is an array of n single-writer n-reader atomic registers. The atomic register $SM[x][i]$ can be read by all processes but written only by p_i.

Let us consider the invocation SM.write_snapshot(x, v) issued by p_i. Process p_i first writes $SM[x][i]$ and reads (not atomically) the array $SM[x][1..n]$ that is associated with the recursion parameter x (Lines 01-02). Process p_i then computes the set of processes that have already attained the recursion level x (line 03; let us note that recursion levels are decreasing from n to $n-1$, etc.). If the set of processes that have attained the recursion level x (from p_i's point of view) contains exactly x processes, p_i returns this set as a result (lines 04-05). Otherwise less than x processes have attained the recursion level x. In that case, p_i recursively invokes SM.write_snapshot$(x-1)$ (line 06) in order to attain and stop at the recursion level y attained by exactly y processes.

```
operation $SM$.write_snapshot$(x, v)$:
          % $x$ ($n \geq x \geq 1$) is the recursion parameter %
(01)  $SM[x][i] \leftarrow v$;
(02)  for $1 \leq j \leq n$ do $aux_i[j] \leftarrow SM[x][j]$ end for ;
(03)  $pairs_i \leftarrow \{(j, v') \mid \exists j$ such that $aux_i[j] = v' \neq \perp\}$;
(04)  if $(|pairs_i| = x)$
(05)        then $sm_i \leftarrow pairs_i$
(06)        else $sm_i \leftarrow SM$.write_snapshot$(x-1, v)$
(07)  end if;
(08)  return$(sm_i)$.
```

Figure 14: A recursive write-snapshot algorithm (code for p_i)

Consider two extremes examples of executions of the algorithm, a fully concurrent execution and a fully sequential execution. In the concurrent execution, all n processes invoke at the same time SM.write_snapshot(x, v), with $x = n$, then at the same time write their values v to the shared memory (Line 01), and then at the same time read the shared memory (Line 02). Thus, they all see each other, and they all have $pairs_i$ with the same value (Line 03), and they all see $|pairs_i| = x = n$ in Line 04, and end the algorithm in the next line with the same output.

In a fully sequential execution, only one process terminates in each recursive invocation of the algorithm, say p_i terminates in the i-th invocation. In the first invocation, p_1 writes its value last to the shared memory in Line 01, after all other process have read it in Line 02. Thus process p_1 sees $|pairs_i| = x = n$ in Line 04 and terminates the algorithm, while all other process see $|pairs_i| < x = n$ in Line 04 and invoke the algorithm recursively in Line 06 (with $x = n - 1$). The same pattern is repeated in the second invocation, with p_2, etc.

The cost of a shared memory distributed algorithm is usually measured by the number of shared memory accesses, called *step complexity.* The step complexity of p_i's invocation is $O(n(n - |sm_i| + 1))$.

It is interesting to observe that the iterative structure that defines the IWS model and the recursion-based formulation of the previous algorithm are closely related notions. In one case iterations are at the core of the model while in the other case recursion is only an algorithmic tool. However, the executions of a recursion-based algorithm are of an iterated nature: in each iteration only one array of registers is accessed, and the array is accessed only in this iteration.

6.2.2 A Recursive Approximate Algorithm

Figure 15 presents an algorithm that implements the x-approx_Agreement operation. Again, this algorithm is recursive. Moreover, it uses as a subroutine the recursive write_snapshot algorithm. That is, the algorithm runs in the IWS model. The idea of the algorithm is simple. In each iteration the following occurs: processes propose at most two different values, and decide on at most two different values, such that the difference between them is divided by two. Thus, if processes start with input values in $\{0, 1\}$, after x iterations, $x \geq 0$, processes decide on values which are at most $1/2^x$ apart from each other.

The processes communicate with each other their values, v, via a snapshot operation. Process p_i stores the result of the snapshot in its local variable $pairs_i$. Consider the iteration with value x. If $x = 0$, nothing will be done, and the output of each process is equal to its input. Otherwise, assume $x \geq 1$. Process p_i sees at most two different process values in $pairs_i$, let us call them p_1, p_2 (not necessarily different). Then, p_i invokes recursively the algorithm, with $x - 1$, and value v equal to $(v_1 + v_2)/2$. There are only two possibilities, either all see both values, or some see one value, say v_1, and the others see both. At least the last processes to take a snapshot sees both. It is impossible that some processes see only one and some only the other, due to the snapshot operation. Thus, in the recursive invocation will be with at most two different values, and these are half apart from each other.

```
operation approx_Agreement($x, v$):
          % $x$ is the recursion parameter ($x \geq 0$) %
(01) $pairs_i \leftarrow SM$.write_snapshot($x, v$) ;
(02) if ($x = 0$)
(03)      then $out_i \leftarrow v$
(04)      else $out_i \leftarrow$ approx_Agreement($x - 1, mid(pairs_i)$)
(05) end if;
(06) return($out_i$).
```

Figure 15: A recursive x-approximate algorithm (code for p_i)

The step complexity of the algorithm is x times the step complexity of a snapshot operation.

7 Safe Agreement

This section gives one more example of a problem solved elegantly in the iterated model, specifically in the IWS model, namely the implementation of a *safe agreement* object.

A safe agreement object provides two operations, denoted sa_propose(v) and sa_decide(), where v is a value. Each process can call each operation at most once, and must call sa_propose(v) before sa_decide(). These operations satisfy the following properties, which define a kind of consensus object with fail-free termination.

- Termination. If no process crashes while executing sa_propose(v), then any non-faulty process that invokes sa_decide() returns from that invocation.
- Agreement. At most one value is decided.
- Validity. A decided value is a proposed value.

Simple implementations of the safe agreement object type are described in [13, 37]. We introduce here an alternative implementation that works in the IWS model, where the sa_propose() and sa_decide() operations use two iterations of the IWS model.

```
operation sa_propose($v_i$):
(01)    $sm_i^1 \leftarrow WS[1]$.write_snapshot($v_i$);
(02)    $sm_i^2 \leftarrow WS[2]$.write_snapshot($sm_i^1$).
```

Figure 16: Operation sa_propose(v_i) in the iterated model (code for q_i)

Figure 16 shows one way to implement sa_propose(). When a process q_i invokes sa_propose(v_i), it writes the proposed value v_i to $WS[1]$, and stores the snapshot of $WS[1]$ into a local variable sm_i^1 (Line 01). It then writes sm_i^1 to $WS[2]$, and stores the snapshot of $WS[2]$ in a local variable sm_i^2 (Line 02).

An important point here is that q_i writes atomically into $WS[2]$ (note that the snapshot of $WS[2]$ is simply discarded). Note also that sm_i^1 is a set of pairs (k, v_k) (where v_k is the value proposed by q_k) that must contain the pair (i, v_i) written by q_i. However, $WS[2]$ is a set of pairs $(x, view_x)$ where $view_x$ is the value of sm_x^1, that is, the set of pairs obtained from $WS[1]$ by q_x. Finally after q_i calls $WS[2]$.write_snapshot(), $WS[2]$ must contain the pair (i, sm_i^1).

```
operation sa_decide():
(03)    repeat $sm_i^3 \leftarrow WS[2]$.scan()
(04)         until $\big(\forall\, k : (k, -) \in sm_i^1 \Rightarrow (k, view_k) \in sm_i^3\big)$
(05)    end repeat;
(06)    $sm_i^3 \leftarrow \{(k, view_k) \in sm_i^3 \mid (k, -) \in sm_i^1\}$;
(07)    $(-, min\_view_i) \leftarrow (k, view_k) \in sm_i^3$ such that $\forall (x, view_x) \in sm_i^3 : |view_k| \leq |view_x|$;
(08)    let $dec_i = \min\{v_x \mid (x, v_x) \in min\_view_i\}$
(09)    return($dec_i$).
```

Figure 17: Operation sa_decide() in the iterated model (code for q_i)

Implementing the operation $\textsf{sa\_decide}()$ Figure 17 shows a two-part implementation of $\textsf{sa\_decide}()$.

- In the first part (Lines 03-05), q_i repeatedly reads $WS[2]$ until a closure property (explained below) is satisfied). The value read from the write-snapshot object $WS[2]$ is saved in the local variable sm_i^3. Hence, sm_i^3 is a set of pairs $(x, view_x)$ including (i, sm_i^1) (and in turn sm_i^1 includes (i, v_i)).

  Here is the closure property that allows q_i to exit the loop: for any pair $(k, v_k) \in (i, sm_i^1)$, $(k, view_k) \in sm_i^3$. From q_i's point of view, each q_k that has written to $WS[1]$ has also written to $WS[2]$ (more precisely, it has written the view sm_k^1 that it obtained from $WS[1]$).

- Process q_i examines then the value of sm_i^3 when exiting the loop (Line 06) and selects from it the view that has the smallest size: one of the views $view_k$ such that

  $$\text{for all } (x, view_x) \in sm_i^3, |view_k| \le |view_x|.$$

  As we will see in the proof, for any q_i and q_j that exit the loop, $min\_view_i = min\_view_j$. Consequently, any deterministic rule (such as min()) that extracts a decided value from such a set of pairs (x, v_x) can be used to compute the value dec_i decided by the safe agreement object (lines 08-09).

Theorem 7.1. The algorithms described in Figure 16 for the operation $\textsf{sa\_propose}()$ and in Figure 17 for the operation $\textsf{sa\_decide}()$ are a correct implementation of the safe agreement object type.

Proof Recall that if q_i calls $\textsf{sa\_decide}()$, then it has previously called $\textsf{sa\_propose}()$.

Proof of termination: we must show that, if no process crashes while executing $\textsf{sa\_propose}()$, then any non-faulty process that invokes $\textsf{sa\_decide}()$ returns from its invocation.

If no process crashes while executing $\textsf{sa\_propose}()$, then any process that executes this operation executes both $WS[1].\textsf{write\_snapshot}()$ and $WS[2].\textsf{write\_snapshot}()$. Hence, as any q_i that executes $\textsf{sa\_decide}()$ repeatedly reads $WS[2]$, eventually

$$\text{for all } (k, -) \in sm_i^1 : (k, view_k) \in sm_i^3,$$

if $(k, -) \in sm_i^1$ then p_k invoked $WS[1].\textsf{write\_snapshot}()$ and, since p_k does not crash, it eventually invokes $WS[2].\textsf{write\_snapshot}()$. Eventually, $(k, view_k) \in sm_i^3$), and q_i exits the loop. Moreover, since $(i, sm_i^1) \in sm_i^3$ and $(i, v_i) \in sm_i^1$, it follows that the minimum operations of Lines 07 and 09 terminate. Consequently q_i eventually returns from $\textsf{sa\_decide}()$.

Proof of validity: we must show that a decided value was proposed. Let v be the value decided by q_i and $view = min\_view_i$ (computed at Line 06). It follows from lines 07-08 that there exists $(x, v) \in min\_view_i = view$ and $(-, view) \in sm_i^3$, from which we conclude that some process q_j has executed $WS[2].\textsf{write\_snapshot}(sm_j^1)$ where $sm_j^1 = view$. Hence, q_j has obtained $sm_j^1 = view$ from its invocation $WS[1].\textsf{write\_snapshot}()$. It follows from the validity property of the write-snapshot object $WS[1]$ and the fact that $(x, v) \in view$, that the value v was proposed by some process.

Proof of agreement: we must show that no two processes decide different values. Let q_i and q_j be two processes that decide. We show that the sets min_view_i and min_view_j of pairs computed at Line 07 are equal (implying agreement).

First observe that, due to containment of $WS[1]$, any two pairs (x, sm_x^1) and (y, sm_y^1) written in $WS[2]$ (Line 02) are such that $sm_x^1 \subseteq sm_y^1 \vee sm_y^1 \subseteq sm_x^1$ (Observation O1). Moreover, the self-inclusion and containment properties of $WS[1]$ imply that

$$((x, v_x) \notin sm_y^1) \text{ implies } (sm_y^1 \subsetneq sm_x^1)$$

(Observation O2).

Let sm_i^3 and sm_j^3 denote the last values of the corresponding local variables obtained at line 06. As a process writes only once in both $WS[1]$ and $WS[2]$, we have $sm_i^3 = \{(k, view_k) \mid (k, -) \in sm_i^1\}$ and $sm_j^3 = \{(k', view_{k'}) \mid (k', -) \in sm_j^1\}$. If $sm_i^1 = sm_j^1$ we have $sm_i^3 = sm_j^3$ which implies $min\_view_i = min\_view_j$ and agreement follows. Hence, due to observation O1, the remaining case is $sm_i^1 \subsetneq sm_j^1$ or, equivalently, $sm_j^1 \subsetneq sm_i^1$. Without loss of generality let us assume $sm_i^1 \subsetneq sm_j^1$, from which we have $sm_i^3 \subsetneq sm_j^3$. To show $min\_view_i = min\_view_j$ when $sm_i^1 \subsetneq sm_j^1$, we show that, for each $(\ell, sm_\ell^1) \in (sm_j^3 \setminus sm_i^3)$ (i.e., a view seen by p_j but not by p_i), exists $(\ell', sm_{\ell'}^1) \in sm_j^3$ such that $sm_{\ell'}^1 \subsetneq sm_\ell^1$. Then we will have $|sm_{\ell'}^1| < |sm_\ell^1|$ and, as $sm_i^3 \subsetneq sm_j^3$, it will follow that the smallest view in sm_j^3 is the smallest view in sm_i^3.

To show that there is ℓ' such that $(\ell', sm_{\ell'}^1) \in sm_j^3$ and $sm_{\ell'}^1 \subsetneq sm_\ell^1$, let us consider $\ell' = i$. As $(\ell, sm_\ell^1) \in (sm_j^3 \setminus sm_i^3)$ we have $(\ell, v_\ell) \in sm_j^1 \setminus sm_i^1$, i.e., $(\ell, v_\ell) \notin sm_i^1$. It then follows from Observation O2 that $sm_i^1 \subsetneq sm_\ell^1$ from which we have $|sm_i^1| < |sm_\ell^1|$ and concludes the proof of the agreement property. $\Box_{Theorem\ 7.1}$

8 Conclusion

We reviewed a number of tasks in the wait-free read-write model, some solvable, some not. We showed how placing a layer of abstraction on top of this model, in the form of the iterated write-snapshot model, yields a model of equivalent computational power, but more convenient for algorithm design, and with a nicer mathematical structure.

Acknowledgment. We thank Idit Keidar for the many comments on an earlier version of this paper.

References

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

[2] Attiya H., Rachman O., Atomic Snapshots in $O(n \log n)$ Operations. *SIAM Journal of Computing*, 27(2): 319–340, 1998.

[3] Attiya H., Rajsbaum S., The Combinatorial Structure of Wait-Free Solvable Tasks. *SIAM Journal of Computing*, 31(4): 1286–1313, 2002.

[4] Afek Y., Gafni E., Rajsbaum S., Raynal M. and Travers C., The k-Simultaneous Consensus Problem. *Distributed Computing*, 22(3):185-195, 2010.

[5] Anderson J., Multi-writer Composite Registers. *Distributed Computing*, 7(4):175-195, 1994.

[6] Attiya H., Bar-Noy A., Dolev D., Peleg D. and Reischuk R., Renaming in an Asynchronous Environment. *Journal of the ACM*, 37(3):524-548, 1990.

[7] Attiya H., Guerraoui R. and Ruppert E., Partial Snapshot Objects. *Proc. 20th ACM Symposium on Parallel Architectures and Algorithms (SPAA'08)*, ACM Press, pp. 336-343, 2008.

[8] Attiya H. and Welch J., *Distributed Computing: Fundamentals, Simulations and Advanced Topics* (2d Edition), Wiley-Interscience, 414 pages, 2004.

[9] Biran O., Moran S., and Zaks S., A combinatorial characterization of the distributed tasks which are solvable in the presence of one faulty processor. *PODC '88: Proceedings of the seventh annual ACM Symposium on Principles of distributed computing (PODC'88)*, pages 263–275, 1988.

[10] Borowsky E. and Gafni E., Generalized FLP Impossibility Result for t-Resilient Asynchronous Computations. *Proc. 25-th Annual ACM Symposium on Theory of Computing (STOC'93)*, ACM Press, pp. 91-100, 1993.

[11] Borowsky E. and Gafni E., Immediate Atomic Snapshots and Fast Renaming. *Proc. 12th ACM Symposium on Principles of Distributed Computing (PODC'93)*, ACM Press, pp. 41-51, 1993.

[12] Borowsky E. and Gafni E., A Simple Algorithmically Reasoned Characterization of Wait-free Computations. *Proc. 16th ACM Symposium on Principles of Distributed Computing (PODC'97)*, ACM Press, pp. 189-198, 1997.

[13] Borowsky E., Gafni E., Lynch N. and Rajsbaum S., The BG Distributed Simulation Algorithm. *Distributed Computing*, 14(3):127-146, 2001.

[14] Charron-Bost B. and Schiper A., The Heard-Of Model: Computing in Distributed Systems with Benign Faults. *Distributed Computing*, 22(1), 49–71, 2009.

[15] Castañeda A., Rajsbaum S. and Raynal M., The Renaming Problem in Shared Memory Systems: an Introduction. *Computer Science Review*, 5(3): 229–251, 2011.

[16] Chaudhuri S., More Choices Allow More Faults: Set Consensus Problems in Totally Asynchronous Systems. *Information and Computation*, 105(1):132-158, 1993.

[17] Dolev D., Lynch N., Pinter S., Stark E. and Weihl W., Reaching Approximate Agreement in the Presence of Faults. *Journal of the ACM* 33(3):499–516 (1986).

[18] Fischer M.J., Lynch N.A. and Paterson M.S., Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, 32(2):374-382, 1985.

[19] Fraigniaud P., Rajsbaum S., Travers C., Locality and Checkability in Wait-Free Computing. *Proc. 25th Int'l Symposium on Distributed Computing (DISC'11)*, Springer, LNCS 6950, pp. 333–347, 2011.

[20] Eli Gafni and Elias Koutsoupias. Three-Processor Tasks Are Undecidable. *SIAM J. Comput.*, 28(3):970–983, 1999.

[21] Gafni E. and Rajsbaum S., Recursion in Distributed Computing. *Proc. 12th Int'l Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS'10)*, Springer-Verlag, LNCS 6366, pp. 362-376, 2010.

[22] Gafni E. and Rajsbaum S., Distributed Programming with Tasks. *Proc. 14th Int'l Conference on Principles of Distributed Systems (OPODIS'10)*, Springer, LNCS 6490, pp. 205-218, 2010.

[23] Gafni E., Rajsbaum S., Herlihy M., Subconsensus Tasks: Renaming Is Weaker Than Set Agreement. *Proc. 20th Int'l Symposium on Distributed Computing (DISC'06)*, Springer, LNCS 4167, pp. 329-338, 2006.

[24] Guerraoui R. and Raynal M., From Unreliable Objects to Reliable Objects: the Case of atomic Registers and Consensus. *9th Int'l Conference on Parallel Computing Technologies (PaCT'07)*, Springer, LNCS 4671, pp. 47-61, 2007.

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

[26] Maurice Herlihy and Sergio Rajsbaum. The Decidability of Distributed Decision Tasks (extended abstract). *Proc. 29th annual ACM Symposium on Theory of Computing (STOC'97)*, ACM Press, pp.589–598, 1997.

[27] Herlihy M. P. and Rajsbaum S., A Classification of Wait-free Loop Agreement Tasks. *Theoretical Computer Science*, 291(1):55-77, 2003.

[28] Herlihy M. P. and Rajsbaum S., The Topology of Shared Memory Adversaries. *Proc. 29th ACM Symposium on Principles of Distributed Computing (PODC'10)*, ACM Press, pp. 105-113, 2010.

[29] Herlihy M.P. and Rajsbaum S., Concurrent Computing and Shellable Complexes. *Proc. 24th Int'l Symposium on Distributed Computing (DISC'10)*, Springer, LNCS 6343, pp. 109-123, 2010.

[30] Herlihy M. P. and Rajsbaum S., Simulations and Reductions for Colorless Tasks. *Proc. 31st ACM Symposium on Principles of Distributed Computing (PODC'12)*, ACM Press, pp. 253-260, 2012.

[31] Herlihy M.P., Rajsbaum S., and Tuttle, M., Unifying Synchronous and Asynchronous Message-Passing Models. *Proc. 17th ACM Symposium on Principles of Distributed Computing (PODC'98)*, ACM Press, pp. 133–142, 1998.

[32] Herlihy M.P. and Shavit N., The Topological Structure of Asynchronous Computability. *Journal of the ACM*, 46(6):858-923, 1999.

[33] Herlihy M.P. and Shavit N., *The Art of Multiprocessor Programming*, Morgan Kaufman Pub., San Francisco (CA), 508 pages, 2008.

[34] Herlihy M.P. and Wing J.L., Linearizability: a Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463-492, 1990.

[35] Imbs D., Rajsbaum S., Raynal M., The Universe of Symmetry Breaking Tasks. *Proc. 18th Int'l Colloquium on Structural Information and Communication Complexity (SIROCCO'11)*, Springer LNCS 6796, pp. 66–77, 2011.

[36] Imbs D. and Raynal M., Help when Needed, but no More: Efficient Read/Write Partial Snapshot. *Journal of Parallel and Distributed Computing*, 72(1): 1-12, 2012.

[37] Imbs D. and Raynal M., A Liveness Condition for Concurrent Objects: x-Wait-freedom. *Concurrency and Computation: Practice and experience*, 23(17):2154-2166, 2011.

[38] Lamport L., Shostak R., and Pease M., The Byzantine Generals Problem. *ACM Transactions Programming Languages Systems*, 4(3):382–401, 1982.

[39] Lamport. L., On Interprocess Communication, Part 1: Basic formalism, Part II: Algorithms. *Distributed Computing*, 1(2):77-101,1986.

[40] Liu X., Xu Z., and Pan J., Classifying Rendezvous Tasks of Arbitrary Dimension. *Theoretical Computer Science*, 410(21-23):2162-2173, 2009.

[41] Loui M.C., and Abu-Amara H.H., Memory Requirements for Agreement Among Unreliable Asynchronous Processes. *Parallel and Distributed Computing: vol. 4 of Advances in Computing Research,* JAI Press, 4:163-183, 1987.

[42] Lynch N.A., *Distributed Algorithms.* Morgan Kaufmann Pub., San Francisco (CA), 872 pages, 1996.

[43] Moses Y. and Rajsbaum S., A Layered Analysis of Consensus. *SIAM Journal Computing* 31(4): 989-1021, 2002.

[44] Neiger G., Set Linearizability. *Brief Announcement, Proc. 13th ACM Symposium on Principles of Distributed Computing (PODC'94)*, ACM Press, page 396, 1994.

[45] Rajsbaum S., Raynal M. and Travers C., The Iterated Restricted Immediate Snapshot (IRIS) Model. *14th Int'l Computing and Combinatorics Conference (COCOON'08)*, Springer, LNCS 5092, pp.487-496, 2008.

[46] Raynal M., *Concurrent Programming: Algorithms, Principles, and Foundations.* Springer, 450 pages, 2012 (ISBN: 978-3-642-32026-2).

[47] Saks M.E. and Zaharoglou F., Wait-Free k-Set Agreement is Impossible: The Topology of Public Knowledge. *SIAM Journal of Computing* 29(5): pp. 1449-1483, 2000.

[48] Taubenfeld G., *Synchronization Algorithms and Concurrent Programming.* Pearson Prentice-Hall, 423 pages, 2006 (ISBN 0-131-97259-6).