HAL
archives-ouvertes.fr

The Renaming Problem in Shared Memory Systems: an Introduction

Armando Castañeda, Sergio Rajsbaum, Michel Raynal

▶ To cite this version:

Armando Castañeda, Sergio Rajsbaum, Michel Raynal. The Renaming Problem in Shared Memory Systems: an Introduction. [Research Report] PI-1960, 2010, pp.29. inria-00537914

HAL Id: inria-00537914
https://hal.inria.fr/inria-00537914

Submitted on 19 Nov 2010

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

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

Publications Internes de l'IRISA
ISSN : 2102-6327
PI 1960 – Novembre 2010

The Renaming Problem in Shared Memory Systems: an Introduction

Armando Castañeda[*] Sergio Rajsbaum[**] Michel Raynal[***]

Abstract: Exploring the power of shared memory communication objects and models, and the limits of distributed computability are among the most exciting research areas of distributed computing. In that spirit, this paper focuses on a problem that has received considerable interest since its introduction in 1987, namely the renaming problem. It was the first non-trivial problem known to be solvable in an asynchronous distributed system despite process failures. Many algorithms for renaming and variants of renaming have been proposed, and sophisticated lower bounds have been proved, that have been a source of new ideas of general interest to distributed computing. It has consequently acquired a paradigm status in distributed fault-tolerant computing.

In the renaming problem, processes start with unique initial names taken from a large name space and decide new names such that no two processes decide the same new name and the new names are from a name space as small as possible.

This paper presents an introduction to the renaming problem in shared memory systems, for non-expert readers. It describes both algorithms and lower bounds. Also, it discusses strong connections relating renaming and other important distributed problems such as set agreement and symmetry breaking.

Key-words: Algebraic Topology, Asynchronous system, Atomic snapshot, Consensus, Crash failure, Concurrency, Lower bounds, Obstruction-freedom, Read/write shared memory system, Recursion, Renaming, Set agreement, Splitter, Symmetry breaking, Test&set, Wait-freedom, Write-snapshot.

Une introduction au problème du renommage réparti

Résumé : *Ce rapport est un introduction algorithmique au problème du renommage réparti.*

Mots clés : *Renommage réparti, accord, tolérance aux fautes.*

---

[*] Instituto de Matematicas, UNAM, Mexico City, Mexico, acastanedar@uxmcc2.iimas.unam.mx
[**] Instituto de Matematicas, UNAM, Mexico City, Mexico, rajsbaum@math.unam.mx. Partly supported by PAPIME and PAPIIT UNAM Projects.
[***] IUF and Projet ASAP: équipe commune avec l'INRIA, le CNRS, l'université Rennes 1 et l'INSA de Rennes, raynal@irisa.fr

©IRISA – Campus de Beaulieu – 35042 Rennes Cedex – France – +33 2 99 84 71 00 – www.irisa.fr

1 Introduction

Motivation The consensus problem is one of the most important problems encountered in fault-tolerant distributed computing. Assuming that each process proposes a value, it states that the processes have to agree on the very same value, that value being one of the proposed values. Consensus is a basic building block when processes have to agree. As an example the totally ordered broadcast problem requires that the processes deliver in the same order all the messages they broadcast [17]. This means that totally ordered broadcast is both a communication problem (processes have to deliver the same set of messages) and an agreement problem (the messages have to be delivered in the same order at every process) [51], which is an instance of the consensus agreement problem.

The consensus problem is trivial to solve in asynchronous reliable distributed systems, and relatively easy to solve in unreliable synchronous systems [10, 40, 52]. However, it is impossible to solve deterministically in asynchronous systems in which even a single process may fail and the failure is the mildest one, namely, a process crash [19]. This impossibility result was proved for asynchronous message-passing systems, and later extended to asynchronous read/write shared memory systems in [39]. Following the names of the three researchers (Fischer, Lynch and Paterson) who proved the original impossibility in 1985, the acronym FLP has been coined by the community to refer to the impossibility of solving consensus in asynchronous distributed systems where at least one process may crash.

The FLP impossibility publication gave rise to the feeling that any non-trivial problem that requires process coordination could not be solved when one has to cope with the combined effect of asynchrony and process failures. We know today that there are infinitely many problems that can be solved in such conditions (and infinitely many than cannot be solved) [11]. Among these problems, the renaming problem has been the first to be proposed, and solved [5]. Renaming has acquired a paradigm status for fault-tolerant computing, because in addition to being the first non-trivial problem known to be solvable despite asynchrony and failures, it has turned out to be surprisingly difficult to study, and has inspired a significant number of algorithms and impossibility proof techniques.

What is the renaming problem? Intuitively, in the M-renaming problem processes start with unique initial names, i.e., integers in the interval $[1..N]$ for some large N, and have to choose unique new names in the interval $[1..M]$, where M is smaller than N. Initially a process knows only its name, and not the initial names of the other processes. In a solution to the renaming problem, processes communicate with each other through some medium, and eventually choose their new names. No two processes choose the same new name. Moreover, the processes are asynchronous and can fail at any point during their execution.

A central concern in the renaming problem is reducing the output name space as much as possible. When the size of the new name space, M, should be as small as possible, as a function of n, the number of processes, we have non-adaptive renaming. Adaptive renaming is more demanding: the size of the new name space should be as small as possible as a function of the actual number of processes participating in an execution. Most algorithms solving renaming are adaptive. But proving lower bounds for non-adaptive renaming is substantially more difficult than proving lower bounds for adaptive renaming. Indeed, it has been shown in [25] that adaptive renaming is strictly more difficult than non-adaptive renaming.

We may say that research on the renaming problem has concentrated along three lines. On the algorithmic side, many renaming algorithms have been proposed, trying to rename efficiently and with the fewest possible new names, i.e., with M as small as possible, in various models of computation, including shared memory, message passing, synchronous and asynchronous models, for renaming and its variants, e.g. [2, 3, 5, 6, 7, 13, 24, 36, 41, 42]. On the lower bounds side, the main concern has been understanding what is the smallest size of the new name space, i.e., how small can M be, e.g. [9, 15, 16, 30, 32]. Finally, researches have studied how to use renaming to solve other problems, as well as the relation between renaming and other problems, e.g. [22, 23, 25, 43, 44]. This tutorial complements previous expositions of the renaming problem in textbooks and papers e.g. [10, 26, 40] with more recent results and approaches of [14, 15, 16, 23, 24, 25, 26, 36, 44, 48].

Structure of the paper This paper, made up of 7 sections, is an introduction to the renaming problem in shared memory systems. The paper focuses on the most basic form of renaming, the one-shot version, although it discusses other variants, to give a perspective to the reader on the renaming research area.

- Section 2 defines the renaming problem and the model of computation. It discusses the main renaming variants: adaptive vs non-adaptive renaming; one-shot vs long-lived renaming; group renaming. Also, it gives an intuition of the difficulty of the renaming problem and its underlying algorithmic principles.
- Section 3 describes a few shared memory abstractions that will simplify the design of renaming algorithms: collect, snapshot, write-snapshot (also known as immediate-snapshot or block executions). All can be wait-free built on top of read/write atomic registers.
- Section 4 describes size-adaptive optimal renaming algorithms and a time-adaptive renaming algorithm.

- Section 5 situates the difficulty of the renaming problem in relation to other problems, mainly k-test&set, k-set agreement and weak symmetry breaking.
- Section 6 summarizes the lower bounds associated with renaming, and explains the difference in the lower bounds for adaptive vs non-adaptive renaming. Roughly speaking, adaptive renaming is equivalent to set agreement, and hence the lower bounds of [9, 12, 32, 54] apply, while non-adaptive renaming lower bounds require more involved algebraic topology techniques. Section 6.3 focuses on the the mathematics underlying renaming.
- Finally, Section 7 contains the conclusions for the paper.

In order not to overload the presentation, all the proofs are given in appendices. Moreover, for completeness reasons, the last appendix presents a simple renaming algorithm for message-passing systems.

2 The basics of renaming

This section starts by defining the model of computation of interest to the paper. Then it defines the renaming problem and discusses the main renaming variants. Finally, it discusses some basic results about renaming, and gives an intuition of the difficulty of the renaming problem.

2.1 Computation model

Process model The system consists of n sequential processes that we denote p_1, p_2, ..., p_n. The integer i is called the index of p_i.

Each process p_i has an initial name denoted old_name_i such that the initial names belong to the totally ordered set $[1..N]$ (hence they can be compared) with $N >> n$. A process does not know the initial names of the other processes, it only knows that no two processes have the same initial name. An initial name can be seen as a particular value defined in p_i's initial context that uniquely identifies it (e.g., its IP address).

The processes are asynchronous. This means that the relative execution speed of different processes is completely arbitrary, and there is no bound on the time it takes for a process to execute a step.

Failure model A process may crash (halt prematurely). After it has crashed, a process executes no step. A process executes correctly until it possibly crashes. A process that does not crash in a run is *correct* in that run. Otherwise it is *faulty* in that run. When any number of processes may crash, the failure model is called *wait-free* [27], because it is useless for a process to wait for events to happen related to other processes (e.g. waiting until another process writes a value to the shared memory). Thus, in a wait-free solution to the renaming problem, a process has to choose its new name in a finite number of steps, independently of the steps taken by other processes.

Let us observe that the wait-free model prevents the use of locks [49]. This is because lock-based algorithms cannot be wait-free: if a process that has locked an object crashes before releasing the lock, that object is locked forever and no other process can access the object protected by that lock. Recall that locks can be implemented from read/write atomic registers only in reliable systems [10, 40].

Communication model The processes communicate with each other by accessing atomic read/write shared registers. *Atomic* means that each read or write operation appears as if it has been executed instantaneously at some point of the time line time between its begin and end events [34, 37]. Each atomic register is a single-writer/multi-reader (1WnR) register. This means that a single process (statically determined) can write it, but every process can read it. Atomic registers are denoted with uppercase letters. The atomic registers are structured into arrays. If $X[1..n]$ is such an array, $X[i]$ denotes the register of the array that p_i is allowed to write.

A process can have local registers. Such registers are denoted with lowercase letters with the process index appearing as a subscript (e.g., $prop_i$ is a local register of p_i). The notation $\perp$ is used to denote a default value, usually assumed to be the initial value of a register, either local or shared.

This communication model provides a convenient abstraction level. More elementary communication means, such as single-writer/single-reader registers, or message passing channels, can be used to construct single-writer/multi-reader registers (although at a cost in efficiency), e.g. [10, 40, 51].

2.2 The renaming problem

One-shot renaming The M-renaming problem consists in implementing an object that provides the processes with a single operation denoted new_name() such that (a) a process can invoke it at most once and (b) the following properties are satisfied.

- Termination. The invocation of new_name() by a correct process returns it a *new name*.
- Validity. Each new name is an integer in the set $[1..M]$.
- Agreement. No two processes obtain the same new name.
- Index independence. The new name obtained by a process is independent of its index.

The index independence property states that, if in a run a process whose index is i obtains the new name v, that process could have obtained the very same new name v if its index had been j. This means that from an operational point of view the indexes define an underlying communication infrastructure, i.e., an addressing mechanism that can be used only to access entries of shared arrays. Indexes cannot be used to *compute* new names. Otherwise, a trivial solution to perfect renaming, with $M = n$, would be that p_i chooses as new name i, without any communication.

Adaptive vs non-adaptive renaming Let p be the number of processes that participate in a renaming execution, i.e., the number of processes that invoke new_name(). Let us observe that the renaming problem cannot be solved when $M < p$. There are two types of adaptive renaming algorithms.

- Size adaptive. An algorithm is size-adaptive if the size M of the new name space depends only on p, the number of participating processes. We have then $M = f(p)$ where $f(p)$ is a function on p such that $f(1) = 1$ and, for $2 \leq p \leq n$, $p - 1 \leq f(p-1) \leq f(p)$. If M depends only on n (the total number of processes) the algorithm is not size-adaptive.
- Time adaptive. An algorithm is time-adaptive if its time complexity depends only on p. If its time complexity depends only on n it is not time-adaptive.

A fundamental result An important theoretical result associated with the renaming problem in asynchronous read/write systems is the following [32]. Except for some *exceptional* values of n, the value $M = 2n - 1$ is the lower bound on the size of the new name space. For the exceptional values we have $M = 2n - 2$. These exceptional values, characterized in [15], involve sets of relatively prime integers[1].

This means that $M = 2p - 1$ is a lower bound for size-adaptive algorithms (in that case, there is no specific values of p that would allow a lower bound smaller than $2p - 1$). Consequently, the use of an optimal time-adaptive algorithm means that if "today" p' processes acquire new names, their new names belong to the interval $[1..2p'-1]$. If "tomorrow" p'' additional processes acquire new names, these processes will have their new names in the interval $[1..2p - 1]$ where $p = p' + p''$.

2.3 Renaming variants

The paper focuses on the one-shot renaming problem. Consequently the variants described below are cited only for completeness.

Long-lived renaming In the long-lived renaming problem, a process can (repeatedly) acquire a new name and then release it [41, 42]. Long-lived renaming can be useful in systems in which processes acquire and release identical resources. Each new name gives then access to a resource (e.g., its address) and the renaming algorithm control accesses to the resources.

Group renaming A generalization of the renaming problem for groups of processes has been proposed in [21] and later investigated in [2]. In this variant, each process belongs to a group and knows the original name of its group. Each process has to choose a new name for its group in such a way that two processes belonging to distinct groups choose distinct new names.

2.4 Non-triviality of the renaming problem

The aim of the discussion that follows is to give an intuition of the difficulty of the renaming problem and its underlying algorithmic principles. To that end we use a simple example. Let us consider a system with two asynchronous crash-prone processes p and q that want to acquire new names. They have to coordinate to ensure they do not choose the same new name.

To that end, each of them can write the shared memory (to communicate with the other process) and read it (to obtain information from the other process). Let us assume that a process first writes and then reads the shared memory, once. There are essentially three scenarios.

[1] More precisely, there is a $(2n-2)$-renaming algorithm for the values of n such that the integers in the set $\{\binom{n}{i} : 1 \leq i \leq \lfloor \frac{n}{2} \rfloor\}$ are relatively prime [15].

- Scenario #1. In this scenario, process p writes (e.g., its initial name) to the shared memory to inform q that it wants to acquire a new name, but when p reads the shared memory, q has not yet written it (e.g., because it is slow). Hence, p does not "see" that q is competing for a new name.

  Differently, when q reads the shared memory, it "sees" that p is competing for a new name.
- Scenario #2. This scenario is the same as the previous one, except that p and q are inverted. Hence, in this scenario, q does not "see" that p is competing for a new name, while p sees that q is competing for a new name.
- Scenario #3. In this scenario, both p and q write concurrently the shared memory and then each of them discovers that the other one is competing. Here, each process "sees" that the other one is competing for a new name.

The difficulty comes from the fact that in scenario #1, q does not know if p sees it or not. More explicitly, q cannot distinguish scenario #1 and scenario #3. A symmetric situation occurs for p which cannot distinguish scenario #2 and scenario #3. These indistinguishability relations are represented as a graph in Figure 1.

Remark Indistinguishability relation analysis is at the core of distributed computing, especially for proving lower bounds, since the original FLP impossibility result; graph structures arise when only one process can fail e.g. [11, 19], while higher dimensional topological structures arise when more than one process can fail, e.g. [12, 32, 54].

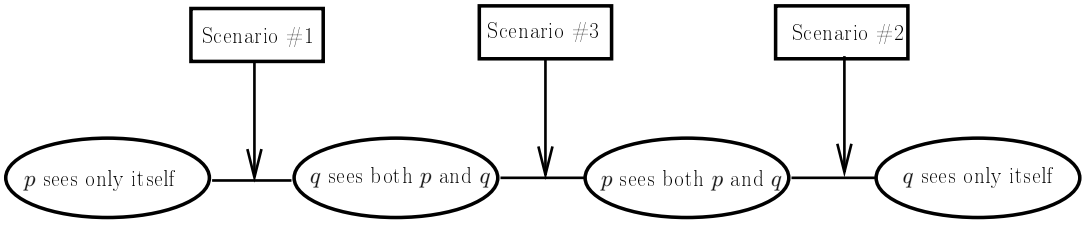

Figure 1: Uncertainties for 2 processes after one communication exchange

In order to think about the design of an algorithm, let us assume that, whenever a process does not see the other process (because it has crashed or is very slow), it chooses the new name 1. This can be done without loss of generality because the space of initial names is big, hence for every algorithm there exist two processes such that each one of them picks the same new name when it does not see the other. Consequently, p chooses the new name 1 in scenario #1 and q chooses the new name 1 in scenario #2.

Let us now look at scenario #3. Process q sees p and is aware that p may have not seen it (this is because q cannot distinguish scenario #1 and scenario #3). To avoid conflict (in case we are in scenario #1 in which case p chooses new name 1), q chooses new name 2. In that case, p (that does not know if the real scenario is scenario #2 or scenario #3) has no choice: it has to choose the new name 3 to ensure that no two processes have the same new name.

This simple algorithm solves the renaming problem for two processes with size of the new name space equal to 3. Let us observe that the scenario #4 in which no process sees the other one cannot happen. This is due to fact that processes communicate by writing and reading a shared memory made up of atomic registers, and each of them writes the shared memory before reading it.

Could it possible to solve the problem for two processes with two new names only? The previous discussion shows that the answer is "no", if each process is limited to a single communication round (during which it writes and then reads). What if processes are not restricted to one communication round? Perhaps surprisingly, the answer remains "no". This is because the two endpoints of the uncertainty graph (Figure 1) always remain connected [11]. These two endpoints represents the scenario where neither p nor q sees the other process. In these extreme cases, each has to choose the new name 1, and again it would be impossible for p and q to pick only 1 or 2 in the internal nodes, because an edge with equal new names in its endpoints would be unavoidable. Section 6.3 will discuss this issue in more detail.

3 Base shared memory abstractions

This section defines some shared memory abstractions that will simplify the design of renaming algorithms. All can be wait-free built on top of read/write atomic registers. It also defines a switch object called splitter.

3.1 The collect and snapshot abstractions

The collect abstraction The collect abstraction provides the processes with an operation denoted collect(). This operation, which is associated with an array $X[1..n]$ of atomic registers, is a simple abbreviation for an asynchronous read of each atomic register that belongs to the array. More precisely, we have the following:

operation X.collect():
  for $1 \leq j \leq n$ **do** $aa[j] \leftarrow X[j]$ **end for**;
  return($aa[1..n]$).

Let us observe that collect() is not an atomic operation. Due to the asynchrony of the reader, the values in $aa[1..n]$ may have been read at distinct times. Consequently, it is possible for the values of $aa[j1]$ and $aa[j2]$ that are returned to have never been simultaneously in the array X.

The snapshot abstraction *Snapshot* objects have been introduced in [1]. A snapshot object X abstracts an array of size n (the number of processes) that provides each process p_i with two operations denoted X.update(v) and X.snapshot(). The former assigns v to $X[i]$ (and is consequently also denoted $X[i] \leftarrow v$). Only p_i can write $X[i]$. The latter operation, X.snapshot(), returns to the invoking process p_i the current value of the whole array X. The main property of a snapshot object is that all update and snapshot operations appear as if they have been executed atomically, which means that a snapshot object is linearizable [34, 35].

These operations can be wait-free built on top of atomic read/write registers. The best implementation known so far has $O(n \log n)$ time complexity [8] (it is not known today if this bound is tight).

3.2 The write-snapshot abstraction

The one-shot write-snapshot abstraction This abstraction which, as the previous ones, also abstracts an array of atomic registers $X[1..n]$, provides the processes with a single operation denoted write_snapshot().

This operation (known as immediate snapshot [13] or block execution [54]) allows the invoking process p_i to instantaneously write a value in $X[i]$ immediately followed by a snapshot of the whole array. If several processes invoke write_snapshot() simultaneously, then their write occur concurrently followed by snapshot operations that return to the processes the same array value. The write_snapshot() operations are set-linearizable [45] (which means that they are "linearizable" with the possibility that concurrent operations are "linearized" at the same point).

We consider here the one-shot version of the write-snapshot abstraction, i.e., given an array $X[1..n]$, a process invokes X.write_snapshot() at most once. Without loss of generality, let us assume that the initial value of $X[1..n]$ is $[\bot, \ldots, \bot]$ and that the value written by p_i is old_name_i.

Let res_i denote the set that contains all non-$\bot$ values contained in the array returned by the invocation of X.write_snapshot() issued by p_i. This set is sometimes called a *view*. More formally, the write-snapshot abstraction is defined by the following properties where T denotes the set of indexes of the processes that return from their X.write_snapshot() invocations.

- Termination. Any invocation X.write_snapshot() by a correct process p_i terminates (hence $i \in T$).
- Self-inclusion. $\forall\, i \in T$: $old\_name_i \in res_i$.
- Containment. $\forall\, i, j \in T$: $(res_i \subseteq res_j) \vee (res_j \subseteq res_i)$.
- Immediacy. $\forall\, i, j \in T$: $\big((old\_name_i \subseteq res_j) \wedge (old\_name_j \subseteq res_i)\big) \Rightarrow (res_i = res_j)$.

The self-inclusion property states that a process sees its writes, 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 have obtained the same view whose size corresponds to their *concurrency level*.

Remark Let X.w_snapshot(v) be an operation that, when invoked by p_i, first writes v into $X[i]$ and then invokes once X.snapshot(). While the write and the snapshot are atomic, the operation X.w_snapshot() is not. Notice that the operation X.w_snapshot() satisfies the termination, self-inclusion and containment properties stated above. But as it does not satisfy the immediacy property, it shows the additional power power provided by the immediacy property.

A simple recursive distributed algorithm The recursive write-snapshot algorithm described in Figure 2 is from [24]. It assumes a shared array $SM[1..n]$ such that each $SM[x]$ is an array of n 1WnR atomic registers; it is initialized to $[\bot, \ldots, \bot]$. The atomic register $SM[x][i]$ can be read by all processes but written only by p_i. A process invokes SM.write_snapshot(n) (let us remember that n is the total number of processes).

Let us consider the invocation SM.write_snapshot(x) issued by p_i where x is the recursion parameter (initially equal to n). Process p_i first writes $SM[x][i]$ and reads asynchronously the array $SM[x][1..n]$ that is associated with the recursion parameter x (lines 01-02). Then, p_i computes the set of processes that have already attained the recursion level x (line 03; let us notice 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 (as

we will see) less than x processes have attained the recursion level x and p_i recursively invokes SM.write_snapshot$(x-1)$ (line 06).

```
operation SM.write_snapshot(x):
          % x (n ≥ x ≥ 1) is the recursion parameter %
(01)  SM[x][i] ← old_name_i;
(02)  aux_i ← SM[x].collect;
(03)  have_written_i ← {old_name | ∃j such that aux_i[j] = old_name};
(04)  if (|have_written_i| = x)
(05)        then res_i ← have_written_i
(06)        else  res_i ← SM.write_snapshot(x − 1)
(07)  end if;
(08)  return(res_i).
```

Figure 2: Write-snapshot algorithm (code for p_i)

The the cost of a distributed algorithm is often measured by the number of shared memory accesses, and called *step complexity*.

Theorem 1 *The algorithm described in Figure 2 is a wait-free construction of a write-snapshot abstraction. Moreover, its step complexity is $O(n(n-|res|+1))$ where res is the set returned by SM.write_snapshot(n).*

The proof is in Appendix A.

3.3 The splitter abstraction

Definition A splitter is a wait-free concurrent object that provides processes with a single operation, denoted direction(), that returns a value to the invoking process. The semantics of a splitter is defined by the following properties [38, 42].

- Validity. The value returned by direction() is *right*, *down* or *stop*.
- Solo execution. If a single process invokes direction(), only *stop* can be returned.
- Concurrent execution. If x processes invoke direction(), then:
  - At most $x-1$ processes obtain the value *right*,
  - At most $x-1$ processes obtain the value *down*,
  - At most one process obtains the value *stop*.
- Termination. If a correct process invokes direction() it obtains a value.

An implementation The very elegant and simple algorithm described in Figure 3 implements a splitter [38]. The internal state of a splitter SP is represented by two atomic multi-writer/multi-reader (nWnR) atomic registers: $LAST$ that can contain a process old name, and is initialized to any value, and a boolean $CLOSED$ initialized to *false*. A multi-writer/multi-reader atomic register can be constructed from single-writer/multi-reader registers e.g. [33, 40].

When a process p_i invokes SP.direction() it first writes its name in the atomic register $LAST$ (line 01). Then it checks if the "door" is open (line 02). If it has been closed by another process it returns *right* (line 03). Otherwise, p_i closes the door, which can be closed by several processes, (line 04) and then checks if it was the last process to invoke the operation (line 05). If this is the case it returns *stop*; otherwise it returns *down*.

```
operation SP.direction():
(01)  LAST ← old_name_i;
(02)  if (CLOSED)
(03)     then return(right)
(04)     else  CLOSED ← true;
(05)           if (LAST = old_name_i)
(06)              then return(stop)
(07)              else  return(down)
(08)           end if
(09)  end if.
```

Figure 3: A wait-free implementation of a splitter object (code for p_i) [38, 42]

Remark A process that moves right is actually a *late* process: it arrived late at the splitter and found $CLOSED = true$. Differently, a process that moves down is actually a *slow* process: it set $LAST \leftarrow true$ but was not quick enough during the period that started when it updated $LAST$ (line 01) and ended when it read $LAST$ (line 05). At most one process can be neither late not slow, it is *on time* and gets *stop*.

Theorem 2 *The algorithm described in Figure 3 implements a splitter object. Moreover, a process accesses at most four times the shared memory (multi-writer/multi-reader registers).*

The proof is in Appendix B.

4 On the algorithmic side

This section presents renaming algorithms. Section 4.1 presents three size-adaptive and optimal algorithms with respect to the value of M (i.e., $M = 2p - 1$). Section 4.2 presents a simple not size-optimal but time-adaptive algorithm. Finally, Section 4.3 presents an algorithm that considers a liveness property weaker than wait-freedom: the k-obstruction-freedom property.

4.1 Three size-adaptive algorithms

4.1.1 A simple wait-free adaptive $(2p-1)$-renaming algorithm

This section presents a simple adaptive M-renaming algorithm that provides the participating processes with an optimal new name space, i.e., $M = 2p-1$, when the processes can cooperate through read/write registers only. This algorithm, introduced in [10], is an adaptation to asynchronous read/write shared memory systems of a message-passing algorithm described in [5].

Communication medium: a snapshot object The shared memory is made up of a single snapshot object $STATE$. As we have seen this is an array of 1WnR atomic registers denoted $STATE[1..n]$ such that $STATE[i]$ can be written only by p_i and the whole array can be atomically read by p_i by invoking $STATE$.snapshot(). Each atomic register $STATE[i]$ is a pair made up of two fields: $STATE[i].old$ will contain the initial name of p_i, while $STATE[i].prop$ will contain the last proposal of p_i to acquire a new name. Each entry is initialized to $< \perp, \perp >$.

The algorithm: underlying principle and description The algorithm is described in Figure 4 (code for process p_i). The local register $prop_i$ contains p_i's current proposal for a new name. When p_i (whose initial name is old_name_i) invokes new_name(), it sets $prop_i$ to 1 (line 01), and enters a **while** loop (lines 02-12). It exits that loop when it has obtained a new name (statement return($prop_i$) issued at line 06).

```
operation new_name():
(01)  prop_i ← 1;
(02)  while true do
(03)    STATE[i] ←< old_name_i, prop_i >;
(04)    competing_i ← STATE.snapshot();
(05)    if (∀ j ≠ i : competing_i[j].prop ≠ prop_i)
(06)      then return (prop_i)
(07)      else let X    = {competing_i[j].prop | (competing_i[j].prop ≠ ⊥) ∧ (1 ≤ j ≤ n)};
(08)           let free = the increasing sequence 1, 2, ... from which
                          the integers in X have been suppressed;
(09)           let Y    = {competing_i[j].old | (competing_i[j].old ≠ ⊥) ∧ (1 ≤ j ≤ n)};
(10)           let r    = rank of old_name_i in Y;
(11)           prop_i   ← the rth integer in the increasing sequence free
(12)    end if
(13)  end while.
```

Figure 4: A simple read/write wait-free adaptive $(2p-1)$-renaming (code for p_i) [10]

The principle that underlies the algorithm is the following. A new name can be considered as a slot, and processes compete to acquire free slots in the interval of slots $[1..2p-1]$. After entering the loop, a process p_i first updates $STATE[i]$ (line 03) in order to announce to all processes its current proposal for a new name (let us notice that it also implicitly announces it is competing for a new name).

Then, thanks to the snapshot() operation on the snapshot object $STATE$ (line 04), p_i obtains a consistent view (locally saved in the array $competing_i$) of the system global state. Let us notice that this view is consistent because it has been obtained from an atomic snapshot operation. Then the behavior of p_i depends on the consistent global state of the shared

memory it has obtained, more precisely on the value of the predicate $\forall\ j \neq i :\ competing_i[j].prop \neq prop_i$. We consider both cases.

- Case 1: the predicate is true. This means that no process p_j is competing with p_i for the new name $prop_i$. In that case, p_i considers the current value of $prop_i$ as its new name (line 06).
- Case 2: the predicate is false. This means that several processes are competing to obtain the same new name $prop_i$. So, p_i constructs a new proposal for a new name and enters again the loop. This proposal is built from the consistent global state of the system that p_i has obtained in $competing_i$.

  The set $X = \{competing_i[j].prop \mid (competing_i[j].prop \neq \bot) \wedge (1 \leq j \leq n)\}$ (line 07) contains the proposals (as seen by p_i) for new names, while the set

  $$Y = \{competing_i[j].old \mid (competing_i[j].old \neq \bot) \wedge (1 \leq j \leq n)\}$$

  (line 09) contains the initial names of the processes that p_i sees as competing for obtaining a new name.

  The determination of a new proposal by p_i is based on these two sets. First, p_i considers the increasing sequence (denoted *free*) of the integers that are "free" and can consequently be used to define new name proposals. This is the sequence of positive integers from which the proposals in X have been suppressed (line 08). Then, p_i computes its rank r among the processes that (from its point of view) wants to acquire a new name (line 09). Finally, given the sequence *free* and r, p_i defines its new name proposal as its rank in this sequence (this rank is r, i.e., its rank in the set of old names of the processes it sees as competing processes).

Discussion A proof of this algorithm can be found in [10]. The proof that no two new names are the same does not depend on the way the new names are chosen, it depends only on the fact that all the $STATE$.snapshot() operations appear as if they were executed one after the other. The fact that the new names belong to the interval $[1..2p-1]$ depends on the way the new names are chosen (lines 09-11).

It is shown in [20] that there are particular scenarios in which processes can execute an exponential (with respect to n) number of steps (shared memory accesses). Hence, the simplicity of this adaptive renaming algorithm is at the price of a set of runs that -albeit very rare, but possible- are very time-inefficient.

4.1.2 An efficient recursion-based wait-free $(2p-1)$-renaming adaptive algorithm

This section presents an adaptive wait-free renaming algorithm introduced in [48] that is a variant of a recursive algorithm presented in [24]. This algorithm is both optimal with respect to the size of the new name space (i.e., as $M = 2p-1$) and time-efficient, in the sense that its step complexity is not exponential but is $O(n^2)$.

One of the noteworthy features of this algorithm is the fact that its design is based on recursion. This allows for a concise definition of the algorithm and for an invariant-based proof of it.

Communication medium: atomic 1WnR atomic registers The processes cooperate through a three-dimensional array of size $n \times (2n-1) \times 2$ denoted $SM[n..1, 1..2n-1, \{up, down\}]$. Each element of this array is a vector of n atomic 1WnR registers. Hence, $SM[x, f, d]$ is a vector with n entries, and $SM[x, f, d][i]$ is an atomic register that can be written only by p_i but read by any process p_j. For every 4-tuple $\langle x, f, d, i\rangle$, $SM[x, f, d][i]$ is initialized to $\bot$.

As far notation is concerned, we have $up = 1 = \overline{down}$ and $down = -1 = \overline{up}$.

The algorithm: underlying principle A process invokes new_name$(x, first, dir)$ with $x = n$, $first = 1$ and $dir = up$, to acquire a new name. The parameter x is the recursion parameter and will take values n (main call), $n-1$, $n-2$, etc. until process p_i decides a new name. Its smallest possible value is 1. Hence, differently from sequential recursion where recursion is usually on the size and the structure of the data that is visited, here recursion is on the number of processes.

The value up is used to indicate that the concerned processes are renaming "from left to right" (as far as the new names are concerned) while $down$ is used to indicate that the concerned processes are renaming "from right to left". More precisely, when p_i invokes new_name(x, f, up), it considers the renaming space $[f..f+2x-2]$ to obtain a new name, while it considers the space $[f-(2x-2)..first]$ if it invokes new_name$(x, f, down)$. Hence, a process p_i considers initially the renaming space $[1..2n-1]$, and then (as far p_i is concerned) this space will shrink at each recursive call (going up or going down) until p_i obtains a new name.

The algorithm: description The algorithm is presented in Figure 5. Let us consider a process p_i that invokes new_name$(x, first, dir)$, it first writes its old name in $SM[x, first, dir][i]$ (line 01) and reads (asynchronously) the array of atomic registers $SM[x, first, dir][1..n]$, of size n. This array is used to allow the processes that invoke new_name$(x, first, dir)$ to "synchronize" to obtain new names. More precisely, all processes that compete for new names in $[first..first + 2x - 2]$ if $dir = up$, or $[first - (2x - 2)..first]$ if $dir = down$, deposit their old names in $SM[x, first, dir]$ (line 01). Then, according to the value (saved in the local array $competing_i$) that a process has read (asynchronously) from the vector $SM[x, first, dir][1..n]$ (line 02), the behavior of the set X of processes that invoke new_name$(x, first, dir)$ is the behavior of a *splitter* [42] (see below).

It is important to notice that, for each triple (x, f, d), all invocations new_name(x, f, d) coordinate their respective behavior with the help of the size n array of atomic registers $SM[x, f, d][1..n]$. The local variable $competing_i$ is an array of n local registers such that $competing_i[j]$ contains either $\perp$ or old_name_j, the initial name of p_j (line 01). The following notations are used. $|competing_i|$ denotes the number of entries that are different from $\perp$, while $\max(competing_i)$ is the greatest initial name it contains. As a process p_i deposits its initial name in $SM[x, first, dir][i]$ before reading $SM[x, first, dir][1..n]$, it follows that $competing_i$ contains at least one non-$\perp$ entry when it is read by p_i.

Let us observe that if p processes participate in the renaming, their main call new_name$(n, 1, up)$ will systematically entail the call new_name$(n - 1, 1, up)$, etc., until the call new_name$(p, 1, up)$. Then, the behavior of a participating process p_i depends on both the concurrency pattern and the failure pattern.

```
algorithm new_name(x, first, dir):
        % x (n ≥ x ≥ 1) is the recursion parameter %
(01)  SM[x, first, dir][i] ← old_name_i;
(02)  competing_i ← SM[x, first, dir].collect();
(03)  if |competing_i| = x
(04)    then last ← first + dir(2x − 2);
(05)         if old_name_i = max(competing_i)
(06)           then res_i ← last
(07)           else res_i ← new_name(x − 1, last + dir‾, dir‾)
(08)         end if
(09)    else res_i ← new_name(x − 1, first, dir)
(10)  end if;
(11)  return(res_i).
```

Figure 5: Recursive adaptive renaming algorithm (code for p_i) [48]

Considering the at most x processes that invoke new_name$(x, first, dir)$, the *splitter behavior* (adapted to renaming) is defined by the following properties.

- At most $x-1$ processes invoke new_name$(x-1, first, up)$ (line 09). Hence these processes will obtain new names (going up) in $[first..first + 2x - 2]$.
- At most $x - 1$ processes invoke new_name$(x - 1, last + \overline{dir}, \overline{dir})$ (line 07) where $last = first + dir(2x - 2)$ (line 04). Hence these processes will obtain their new names in a renaming space starting at $last + 1$ and going from left to right if $\overline{dir} = up$, or starting at $last - 1$ and going from right to left if $\overline{dir} = down$. Let us observe that the value $last \pm 1$ is considered as starting name because the slot $last$ is reserved for the new name of the process (if any) that stops during its invocation of new_name$(x, first, dir)$ (see the next item).
- At most one process "stops", i.e., defines its new name as $last = first + dir(2x - 2)$ (lines 04 and 06). Let us observe that the only process p_k that can stop is the one such that old_name_k has the greatest value in $SM[x, first, dir][1..n]$ (line 05) that contains then exactly x old names (line 03).

Theorem 3 *The algorithm described in Figure 5 is an adaptive M-renaming algorithm such that $M = 2p - 1$ (where p is the number of participating processes). Its step complexity is $O(n^2)$.*

The proof is in Appendix C.

4.1.3 A variant of the previous recursion-based renaming algorithm

Eliminate recursive calls Let us consider the previous size-adaptive algorithm described in Figure 5 when $y < n$ processes participate. It is easy to see that these processes recursively invoke (line 09) new_name$(n, 1, 1)$, new_name$(n - 1, 1, 1)$, etc., until new_name$(y, 1, 1)$. It is only from this invocation that the processes start doing "interesting" work. Hence, the question: Is it possible to eliminate (whatever the value of y) these useless invocations?

Solving this issue amounts to direct a process to directly "jump" to the invocation new_name$(y, 1, 1)$ such that we have $|competing_i| = y$ in order to execute only the lines 04-08 of Algorithm of Figure 5. Interestingly, the property we are

looking for is exactly what is provided by the write_snapshot() operation. This operation directs a process to the appropriate *concurrency level* (the word "level" refers to the terminology used in Section 3.2 and Appendix A), i.e., the number of processes that the invoking process perceives as concurrent with it.

Description of the algorithm The resulting algorithm is described in Figure 6. Interestingly, this is the algorithm introduced in [13].

A process p_i invokes new_name$(\ell, first, dir)$, where $first = dir = 1$, and ℓ is a list initialized to $\langle n \rangle$. This list is the recursion parameter. The lists generated by the recursive invocations of all participating processes define a tree that is the recursion tree associated with the whole execution; the list $\langle n \rangle$ is associated with the root of this tree. These lists are used to address the appropriate entry of the array SM. As previously, each entry $SM[\ell]$ is a size n array of a 1WnR atomic registers (only p_i can write $SM[\ell][i]$).

Let us remember that, when invoked by p_i, the operation write_snapshot() described in Figure 2 (a) writes the value old_name_i and (b) assumes that the initial value of its recursion parameter is the size (here denoted s) of the set of processes that can invoke it.

Let $s = \text{last}(\ell)$ be the last element of the list ℓ. At most s processes access $SM[\ell]$.write_snapshot(s). The elements of the list indicate the current recursion path.

```
algorithm new_name(ℓ, first, dir):
      % the increasing list ℓ is the recursion parameter %
(01)  s ← last(ℓ);
(02)  competing_i ← SM[ℓ].write_snapshot(s);
(03)  last ← first + dir(2 × |competing_i| − 2);
(04)  if old_name_i = max(competing_i)
(05)     then res_i ← last
(06)     else nextℓ ← ℓ ⊕ |competing|;
(07)          res_i ← new_name(nextℓ, last + dir̄, dir̄)
(08)  end if;
(09)  return(res_i).
```

Figure 6: Recursive adaptive renaming algorithm (code for p_i) [13]

A process p_i first invokes $SM[\ell]$.write_snapshot(s) (lines 01-02). This allows it to access its concurrency level skipping thereby all useless recursive invocations. It then executes only "useful work" (lines 03-08). When considering these lines, the only difference with respect to the algorithm of Figure 6 lies in the management of the recursion parameter needed in the case where $old\_name_i \neq \max(competing_i)$. The value of the recursion parameter (new list) used at line 07 is defined from the current recursion parameter, (the list $\ell = \langle n_1, n_2, \ldots, n_\alpha \rangle$ where $n_1 = n$), and the size of the actual concurrency set $competing_i$. The new list is $next\ell = \langle n_1, n_2, \ldots, n_\alpha, |competing_i| \rangle$. This value is computed at line 06 where $\oplus$ is used to denote concatenation.

Let us observe that the recursive invocations entailed by new_name$(\langle n \rangle, 1, 1)$ issued by a process p_i are such that $n_1 = n > n_2 > \cdots > n_\alpha > |competing_i| > 0$ (from which it is easy to prove that any invocation new_name$(\langle n \rangle, 1, 1)$ terminates.

An example of an execution of this algorithm is described in Appendix D.

Number of shared memory accesses As far as the step complexity (measured as the number of shared memory accesses) is concerned, we have the following. Let us consider that a process p_i invokes recursively k times new_name(). This means that when $|competing_i| = s_k$ we also have $old\_name_i = \max(competing_i)$, and p_i obtains its new name.

Let us consider a process p_i that invokes $SM[\langle n \rangle]$.write_snapshot(n). Let s_1 be the size of the set it obtains from its invocation $SM[\langle n \rangle]$.write_snapshot(n), s_2 be the size of the set it obtains from its invocation $SM[\langle n, s_1 \rangle]$.write_snapshot$(s_1)$, etc., until s_k be the size of the set it obtains from its invocation $SM[\langle n, s_1, \ldots, s_{k-1} \rangle]$.write_snapshot$(s_{k-1})$.

The invocation $SM[\langle n \rangle]$.write_snapshot(n) at line 02 of Figure 6 generates $n - s_1 + 1$ recursive invocations, similarly the invocation $SM[\langle n, s_1 \rangle]$.write_snapshot$(s_1)$ generates $s_1 - s_2 + 1$ recursive invocations, etc, and the invocation $SM[\langle n, s_1, \ldots, s_{k-1} \rangle]$.write_snapshot$(s_{k-1})$ generates $s_{k-1} - s_k + 1$ recursive invocations. Hence, the total number of invocations of write_snapshot() entailed by newname$(\langle n \rangle, 1, 1)$ is $(n - s_1 + 1) + (s_1 - s_2 + 1) + \cdots + (s_{k-1} - s_k + 1) = n - s_k + k$. As we have seen (Figure 2) each of these invocations issues $n + 1$ shared memory accesses. It follows that the total number of shared memory accesses due to an invocation new_name$(\langle n \rangle, 1, 1)$ is $(n+1)(n - s_k + k)$. As $1 \leq k \leq n$, the cost is $O(n^2)$.

4.2 An optimal time-adaptive algorithm

This section presents a time-adaptive renaming algorithm that is optimal, namely, when p processes participates, a process executes at most $O(p)$ shared memory accesses. This algorithm is also size-adaptive but not optimal in that respect. The size of its new name space is $M = p(p+1)/2$.

Underlying principles The idea consists in using a half-grid made up of $n(n+1)/2$ splitters [42]. Figure 7 depicts such a grid for $n = 5$. A process p_i first enters the left corner of the grid; i.e., the splitter numbered 1. Then, it moves along the grid according to the values (*down* or *right*) it obtains from the splitters it visits until it obtains the value *stop*. Finally, it takes as its new name the name statically assigned to the splitter at which it stops. The property attached to each splitter ensures that no two processes stop at the same splitter.

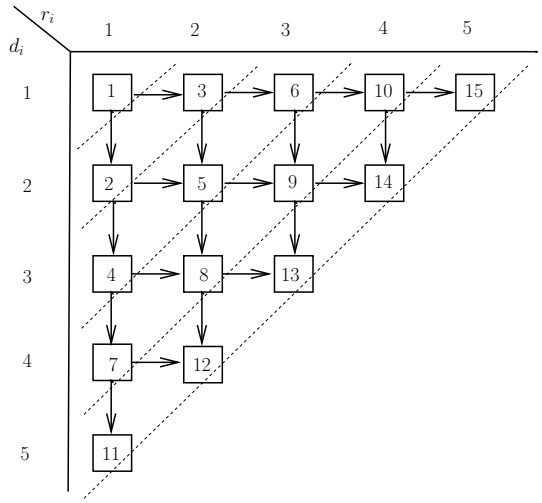

Figure 7: A grid of splitters

The algorithm The resulting algorithm is described in Figure 8. Let $SP[1..n, 1..n]$ be the grid of splitters. The shared atomic registers $LAST[d_i, r_i]$ and $CLOSED[d_i, r_i]$ (initialized to *false*) are used to implement the splitter $SP[d_i, r_i]$. Moreover, each process manages two local variables d_i and r_i for going down or right, respectively.

A process p_i invokes first $SP[1, 1]$.direction() and then moves in the grid according to the values it obtains from the splitters it visits. It follows from the properties of the splitters that, if p processes invoke $SP[[1, 1]$.direction() each visits at most p splitters before stopping. An assertional proof of this time-adaptive renaming algorithm can be found in [42].

```
operation new_name():
    d_i ← 1; r_i ← 1; move_i ← down;
    while (move_i ≠ stop) do
          move_i ← SP[d_i, r_i].direction();
          case (move_i = right) then r_i ← r_i + 1
               (move_i = down)  then d_i ← d_i + 1
               (move_i = stop)  then exit(while loop)
          end case
    end while;
    let res_i ← (d_i + r_i − 1)(d_i + r_i − 2)/2 + r_i;
    % res_i is the number associated with the splitter SP[d_i, r_i] %
     return(res_i).
```

Figure 8: Optimal time-adaptive renaming (code for p_i) [42]

.

Step complexity It follows from the previous observation that a process issues at most $4 \times p$ shared memory accesses, of multi-writer/multi-reader registers, where p is the number of participating processes. Hence the time complexity is $O(p)$ and the algorithm is consequently time-optimal.

Other algorithms More involved size-adaptive or time-adaptive renaming algorithms have been designed. Paper [3] describes a size-optimal algorithm (i.e., $M = 2p-1$) whose step complexity is $O(n^2)$. Paper [7] introduces a new shared data structure called *reflector* from which an optimal size-adaptive renaming algorithm is built. The algorithm described in [6] is size and time-adaptive for the long-lived renaming problem. Let p be the number of processes that are currently participating (those that have invoked new_name() and not yet invokes release_name(). The algorithm is such that $M = 2p - 1$ and its step complexity is $O(p^4)$.

4.3 From wait-freedom to k-obstruction freedom

This section presents an adaptive M-renaming algorithm, introduced in [36], where $M = \min(p+k-1, 2p-1)$. It guarantees that the correct processes that invoke new_name() always return from their invocation if there are long enough periods during which at most k correct processes execute in isolation. This algorithm can be seen as an "appropriate modification" of the algorithm described in Section 4.1.1 that aims at replacing the wait-freedom requirement by the k-obstruction-freedom requirement.

The k-obstruction-freedom liveness property The k-obstruction-freedom property [55] generalizes the obstruction-freedom property [28] (which corresponds to the case $k = 1$).

For $k < n$, the k-obstruction-freedom progress condition is weaker than wait-freedom (it is the same for $k = n$). It states that progress is required only when at most k processes keep on executing new_name(). Let us observe that it is possible that no process ever terminates if no set P of at most k processes execute in isolation for a long enough period. The termination property (progress condition) becomes the following.

- Termination. For any subset P of correct processes, with $|P| \leq k$, if the processes of P execute new_name() in isolation[2], each process of P eventually terminates its invocation (and obtains a new name).

This means that if the concurrency degree becomes smaller or equal to k for "long enough" periods, then all invocations of new_name() by correct processes terminate. Let us observe that n-obstruction-freedom is nothing else than wait-freedom.

Communication medium The processes cooperate through two arrays of atomic 1WnR registers denoted $OLDNAMES[1..n]$, and $LEVEL[1..n]$, and a snapshot object denoted $NAMES$.

- Register $OLDNAMES[i]$, that is initialized to $\perp$, is used by p_i to store its identity old_name_i. Hence $OLDNAMES[i] \neq \perp$ means that p_i participates in the renaming.
- Register $LEVEL[i]$ is initialized to 0. In order to obtain a new name, the processes progress asynchronously from a level (starting from 1) to the next one. $LEVEL[i]$ contains the current level attained by process p_i. As we will see, if during a long enough period at most k processes take steps, they will stabilize at the same level and obtain new names.
- $NAMES[1..n]$ is a snapshot object initialized to $[\perp, \ldots, \perp]$. $NAMES[i]$ contains the new name that p_i tries to acquire. When p_i returns from new_name(), $NAMES[i]$ contains its new name.

The algorithm: underlying principle and description The algorithm defined in Figure 9 describes the behavior of a process p_i. When it invokes new_name(old_name_i), p_i deposits old_name_i in $OLDNAMES[i]$ and proceeds from level 0 to level 1. The local variable $prop_i$ contains p_i's current proposal for a new name. Its initial value is $\perp$. Then, p_i enters a loop (lines 03-21) that it will exit at line 06 with its new name.

Each time it starts a new execution of the loop body, p_i first posts its current name proposal in $NAMES[i]$ and reads (with a $NAMES$.snapshot() invocation) the values of all current proposals (line 04). If its current proposal $prop_i$ is not $\perp$ and no other process has proposed the same new name (line 05), p_i defines its new name as the value of $prop_i$ and exits the loop (line 06). Otherwise, there is a conflict: several processes are trying to acquire the same new name $prop_i$. In that case, p_i enters lines 08-19 to solve this conflict. These lines constitute the core of the algorithm.

In case of conflict, p_i first reads asynchronously all entries of $LEVEL[1..n]$ and computes the highest level attained (line 07) $highest\_level_i$. If its current level is smaller than $highest\_level_i$, p_i jumps to that level, indicates it by writing $LEVEL[i]$ (lines 08-09) and proceeds to the next loop iteration.

If its current level is equal to $highest\_level_i$, p_i computes the set of processes it is competing with in order to acquire a new name, namely the set $competing_i$ (lines 10-11). Those are the processes whose level is equal to $highest\_level_i$. Then, the behavior of p_i depends on the size of the set $competing_i$ (predicate at line 12).

- If $|competing_i| > k$, there are too many processes competing when we consider k-obstruction-freedom. Process p_i progresses then to the next level and proceeds to the next loop iteration (line 13).
- If $|competing_i| \leq k$, p_i selects a new name proposal before proceeding to the next iteration. This selection is similar to what is done in the adaptive renaming algorithm described in Section 4.1.1. As defined at lines 14-15, *free* denotes the list of names that are currently available. Accordingly, p_i defines its new name proposal as the rth value in the list *free* where r is its rank in the set of (at most k) competing processes (hence, $1 \leq r \leq k$).

Discussion A proof of this algorithm is given in [36] where it is also shown that this algorithm is optimal with respect to the new name space, i.e., there is no k-obstruction-free adaptive M-renaming algorithm with $M < \min(p+k-1, 2p-1)$.

[2]Let us observe that this does not prevent $k' > k$ correct processes to have started executing new_name(), as long as $k' - k$ of them, whatever their progress in the code of new_name(), stop executing during a "long enough" period.

```
operation new_name():
(01) prop_i ← ⊥; my_level_i ← 1;
(02) OLDNAMES[i] ← old_name_i; LEVEL[i] ← my_level_i;
(03) repeat forever
(04)   NAMES[i] ← prop_i; names_i ← NAMES.snapshot();
(05)   if ((prop_i ≠ ⊥) ∧ (∀j ≠ i : names_i[j] ≠ prop_i))
(06)     then return(prop_i)
(07)     else levels_i ← LEVEL.collect(); highest_level_i ← max({levels_i[j]});
(08)          if (my_level_i < highest_level_i)
(09)            then my_level_i ← highest_level_i; LEVEL[i] ← my_level_i
(10)            else oldnames_i ← OLDNAMES.collect();
(11)                 competing_i ← {oldnames_i[j] | levels_i[j] = highest_level_i};
(12)                 if (|competing_i| > k)
(13)                   then my_level_i ← highest_level_i + 1; LEVEL[i] ← my_level_i
(14)                   else let X    = {names_i[j] | names_i[j] ≠ ⊥};
(15)                        let free = the increasing sequence 1, 2, 3, ... from which
                                       the integers in X have been suppressed;
(16)                        let r    = rank of old_name_i in competing_i;
(17)                        prop_i   ← the rth integer in the increasing sequence free
(18)                 end if
(19)          end if
(20)   end if
(21) end repeat.
```

Figure 9: k-Obstruction-free adaptive M-renaming with $M = \min(2p-1, p+k-1)$ [36]

5 On the computability side

This section addresses computability issues related to the renaming problem. Section 5.1 presents other problems (tasks) and Section 5.2 shows how there problems are related to renaming. Then, Section 5.3 is focused on the consensus number of renaming.

5.1 Three one-shot problems

5.1.1 k-Test&set objects

A one-shot k-Test&set object provides the processes with a single operation denoted $\mathsf{TS\_compete_k}()$. "One shot" means that, given such an object, a process can invoke that operation at most once (there is no reset operation). The invocations of $\mathsf{TS\_compete_k}()$ issued by the processes on such an object satisfy the following properties:

- Termination. An invocation of $\mathsf{TS\_compete_k}()$ by a correct process terminates.
- Validity. The value returned by an invocation of $\mathsf{TS\_compete_k}()$ is 1 (winner) or 0 (loser).
- Agreement. At least one and at most k processes obtain the value 1.

The instance $k = 1$ does correspond to the usual Test&set object.

The power of k-test&set objects when solving renaming A wait-free adaptive algorithm, based on read/write atomic registers and k-test&set objects, that provides a renaming space of size $M = 2p - \lceil \frac{p}{k} \rceil$ is described in [43]. This algorithm results from an incremental construction (k-test&set objects are used to build intermediate k-participating set objects, that are in turn used to build the renaming algorithm). Due to space limitations, this construction is not described here.

It is shown in [26] that $M = 2p - \lceil \frac{p}{k} \rceil$ is the smallest new name space size that can be obtained when one can use atomic registers and k-Test&set objects only. It follows that the algorithm described in [43] is optimal as far as the size of the renaming space is concerned.

5.1.2 k-Set agreement

The k-set agreement problem is a simple generalization of the consensus problem [18]. A k-set agreement object allows processes to propose values and decide values. To that end such an object provides the processes with an operation denoted $\mathsf{SA\_propose_k}()$. A process invokes that operation at most once on an object. When it invokes $\mathsf{SA\_propose_k}()$, the invoking process supplies the value v it proposes (input parameter). That operation returns a value w (called the value "decided by the invoking process"; we also say that the process "decides w"). The invocations on such an object satisfy the following properties:

- Termination. An invocation of $\mathsf{SA\_propose_k}()$ by a correct process terminates.

- Validity. A decided value is a proposed value.
- Agreement. At most k distinct values are decided.

The power of k-set agreement objects when solving renaming A renaming algorithm, based on atomic registers and k-set agreement objects is described in [22]. The size of the new name space is $M = p + k - 1$ which has been shown to be optimal in [22, 26].

5.1.3 Weak symmetry breaking problem

A weak symmetry breaking object offers an operation weak_sym_breaking() that can be invoked at most once by each process, and that satisfies the following properties.

- Termination. An invocation of weak_sym_breaking() by a correct process terminates.
- Validity. weak_sym_breaking() can return only 0 or 1.
- Agreement. In the executions in which all processes terminate, not all values returned are 0 and not all are 1.
- Index independence. The value returned to a process is independent of its index.

The index independence property evades the trivial solution in which processes with even index return 0 and processes with odd index return 1. Therefore, the processes must use their initial names to compute the outputs.

Remark Let us observe that $(n-1)$-Test&set is, in some sense, weak symmetry breaking with the constraint that, in every execution, at least one process obtains value 1. In that sense weak symmetry breaking is a weak version of $(n-1)$-Test&set.

5.2 A few reductions

5.2.1 A global picture

The following result is shown in [23, 43, 44]: in a system made up of n processes, $(n-1)$-set agreement, $(n-1)$-Test&set and $(2p - \lceil \frac{p}{n-1} \rceil)$-renaming are equivalent. This means given an algorithm solving any of these problems, it is possible to build an algorithm solving any of the other problems. These equivalences are represented in the top line of Figure 10 where $\simeq$ is used to denote "equivalent to".

Notice that, as $(n-1)$-set agreement is not wait-free solvable in the base read/write shared memory model [12, 32, 54], $(n-1)$-Test&set and $(2p - \lceil \frac{p}{n-1} \rceil)$-renaming both are also not wait-free solvable in that model.

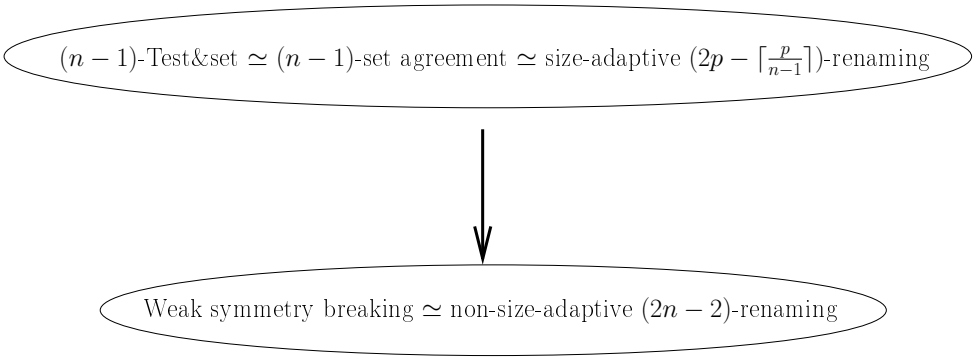

Figure 10: Hierarchy of sub-consensus problems

When we consider the size-adaptive $(2p - \lceil \frac{p}{n-1} \rceil)$-renaming we have the following. If $p = n$ (all processes participate) the size of the new name space is $2n - 2$, while it is $2p - 1$ if not all processes participate. [25].

5.2.2 Weak symmetry breaking and (non-adaptive) $(2n-2)$-renaming are equivalent

The section shows that weak symmetry breaking and non-adaptive $(2n-2)$-renaming are equivalent (bottom line of Figure 10).

From $(2n-2)$-renaming to weak symmetry breaking The processes first obtain new names in the range $[1..2n-2]$ with the help of the $(2n-2)$-renaming algorithm. Let us observe that, as the new name range is $[1..2n-2]$, not all n processes (if they all participate) can have even new names only or odd new names only. At least two distinct processes are such that one has an even name while the other has an odd name. Then a process decides the parity value (0 or 1) of its new name. It trivially follows that, when all participate, at least one process obtains 1 and at least one obtains 0. This simple algorithm is described in Figure 11 (where $NSARO$ denotes the underlying non-size-adaptive renaming object).

```
operation weak_sym_breaking():
(01) new_name_i ← NSARO.new_name();
(02) res_i ← (new_name_i mod 2);
(03) return(res_i).
```

Figure 11: From non-adaptive $(2n-2)$-renaming to weak symmetry breaking

Theorem 4 *The algorithm described in Figure 11 solves in a wait-free manner the weak symmetry breaking problem from any solution to the non-size-adaptive $(2n-2)$-renaming problem.*

From weak symmetry breaking to $(2n-2)$-renaming Let WSB be a weak symmetry breaking object. An algorithm solving the non-adaptive $(2n-2)$-renaming problem is described in Figure 12. A process p_i first invokes WSB.weak_sym_breaking() from which it obtains the value 0 or 1 that is locally saved in $side_i$.

Then, according to the value of $side_i$, p_i invokes an appropriate adaptive renaming object, namely $ARO[0]$ if $side_i = 0$, or $ARO[1]$ if $side_i = 1$. Let us notice that it is always possible to use underlying adaptive renaming objects (e.g., one of the algorithms described in Section 4.1).

Let p be the number of processes that participate in $ARO[0]$.new_name() and p' be the number of processes that participate in $ARO[1]$.new_name(). We have $p + p' \leq n$. As the underlying renaming objects are size-adaptive, the p processes that access $ARO[0]$ obtains new names in $[1..2p-1]$. and the p' processes that access $ARO[1]$ obtain new names in $[1..2p'-1]$.

```
operation new_name():
(01) side_i ← WSB.weak_sym_breaking();
(02) if (side_i = 0) then res_i ← ARO[0].new_name()
(03)                 else  res_i ← (2n − 1) − ARO[1].new_name()
(04) end if;
(05) return(res_i).
```

Figure 12: From weak symmetry breaking to non-adaptive $(2n-2)$-renaming

The algorithm requires that the p processes that access $ARO[0]$ take their new name going up from 1 to $2p-1$, and the p' processes that access $ARO[1]$ take their new name going down from $(2n-1)-1 = 2n-2$ to $(2n-1)-(2p'-1) = 2n-2p'$. As $p+p' \leq n$, i.e., $n-p' \geq p$, we have $2n-2p' > 2p-1$. Consequently, the new name space going up (defined from $ARO[0]$) and the new name space going down (defined from $ARO[1]$) do not intersect, which proves that no two processes obtain the same new name.

Theorem 5 *The algorithm described in Figure 12 solves in a wait-free manner the non-size-adaptive $(2n-2)$-renaming problem from any solution to the weak symmetry problem.*

5.2.3 From $(n-1)$-set agreement to weak symmetry breaking

This section shows that it is possible to solve the weak symmetry problem from the $(n-1)$-set agreement. Hence, it establishes the top-to-bottom arrow of Figure 10.

Shared objects The algorithm uses the following shared objects.

- $SA[1..2]$ are two $(n-1)$-set agreement objects.
- RN is a $(2n-1)$-renaming object (which can be size-adaptive or not).
- $M1[1..n]$ and $M2[n+1..2n-1]$ are two arrays of integers of size n and $n-1$, respectively. The indexes used to address their entries are the new names obtained by the processes from the renaming object R. All entries of $M1[1..n]$ and $M2[n+1..2n-1]$ are initialized to 0 (which means that they do not contain new names).

The algorithm is described in Figure 13. It works as follows. A process p_i first invokes RN.new_name() to obtain a new name that belongs to the range $[1..2n-1]$ (line 01). Then its behavior depends on the value of new_name_i. This guarantees the algorithm holds the index independence property.

- If $new\_name_i \in [1..n]$, it invokes the $(n-1)$-set agreement object $SA[1]$ to which it proposes its new name (invocation $SA[1].\mathsf{SA\_propose}_{n-1}(new\_name_i)$ at line 03) and deposits in "its" entry $M1[new\_name_i]$ the value it has decided from $SA[1]$. Then, if its new_name_i has been deposited in $M1[1..n]$, p_i outputs the value 1 (line 04), i.e., it is a winner. Otherwise, it outputs 0 (line 05), i.e., it is a loser.
- If $new\_name_i \in [n+1..2n-1]$, p_i has a similar behavior except that now $SA[1]$ and $M1$ are replaced by $SA[2]$ and $M2$, respectively, and the outputs 0 and 1 are inverted (lines 07-10).

```
operation weak_symmetry_breaking():
(01) new_name_i ← RN.new_name();
(02) if (new_name_i ≤ n)
(03)    then M1[new_name_i] ← SA[1].SA_propose_{n-1}(new_name_i);
(04)         if (∃ j | new_name_i = M1[j]) then res_i ← 1
(05)                                       else res_i ← 0
(06)         end if
(07)    else M2[new_name_i] ← SA[2].SA_propose_{n-1}(new_name_i);
(08)         if (∃ j | new_name_i = M2[j]) then res_i ← 0
(09)                                       else res_i ← 1
(10)         end if
(11) end if;
(12) return(res_i).
```

Figure 13: From $(n-1)$-set agreement to weak symmetry breaking

Theorem 6 *The algorithm described in Figure 13 solves in a wait-free manner the weak symmetry breaking problem from any solution to the $(n-1)$-set agreement problem.* (Proof in Appendix E.)

It is proved in [25] that non-adaptive $(2n-2)$-renaming is strictly weaker than $(n-1)$-set agreement, which means that there is no wait-free implementation of $(n-1)$-set agreement from non-adaptive $(2n-2)$-renaming. The proof is actually for n odd. The question for n even remains an open problem.

5.3 Consensus numbers of the renaming problems

Consensus numbers and the wait-free hierarchy The *consensus number* notion is a powerful concept that has been introduced by M. Herlihy in [27]. As indicated by its name it is strongly related to the consensus problem (see the introduction for a definition). The consensus number of an object is the greatest integer n (or $+\infty$ if there no such integer) such that these objects together with read/write registers allows consensus to be solved for n processes in a wait-free manner (which means that no process can be blocked forever by other processes).

This notion has given rise to the *wait-free hierarchy* that is an infinite hierarchy of objects such that the objects at level x are exactly the objects with consensus number x. As examples, read/write registers have consensus number 1, while Test&set objects, queues and stacks have consensus number 2 (i.e., they can solve consensus among 2 processes but not 3). In contrast, Compare&swap objects have a consensus number equal to $+\infty$. As it is possible to build any concurrent object (defined from a sequential specification) shared by n processes as soon as one has objects with consensus number $x \geq n$, these objects are said to be *universal* in systems made up of n processes [27].

Consensus number of renaming As the M-renaming problems with $M = 2p-1$ or $M = 2n-1$ can be solved from read/write registers only their consensus number is trivially 1.

Strong adaptive renaming is M-renaming with $M = [1..p]$, i.e., it is the best renaming that is possible. While it cannot be attained in a wait-free manner with read/write registers only, it can be when one has objects with consensus number 2 (e.g., Test&set objects). The simple algorithm in Figure 14 presents such a construction. The processes use an array $ONE\_TS[1..n]$ of 1-Test&set objects that they access sequentially and in the very same order. It is easy to see that, at each iteration x, exactly one process is winning and that process obtains the new name x. It follows that if p processes participates at most p iterations will be executed and we have $M = p$.

In the other direction, strong adaptive renaming allows Test&set to be trivially solved. The process that obtains the new name 1 (i.e., the smallest possible new name) is the winner. We can then conclude that the consensus number of strong adaptive renaming is 2.

```
operation new_name():
(01) for x from 1 to n do
(02)    res_i ← ONE_TS[x].TS_compete_1();
(03)    if (res_i = 1) then return(res_i) end if
(04) end for.
```

Figure 14: From 1-Test&set to optimal renaming

5.4 Renaming and failure detectors

The concept of failure detector has been introduced in [17] in order to overcome impossibility results. A failure detector is a device that provides each process with information on failures. According to the quality of this information and the type of problems they allow solving, different types (also called classes) of failure detectors have been defined. It is important to notice that an asynchronous system enriched with a non-trivial failure detector is no longer purely asynchronous [50].

The relation between failure detector and the renaming problem has been studied in [43] where a failure detector of the class Ω^k is used to implement a $(2p - \lceil \frac{p}{k} \rceil)$-renaming algorithm. This class has been proposed in [46] and generalized in [53]. A failure detector of the class Ω^k provides the processes with sets of k process identities such that eventually the processes are provided with the same set and this set contains at least one correct process.

Failure detectors suited to systems where the size of the name space is greater than n are investigated in [4] where is presented a failure detector strong enough for solving weak symmetry breaking but too weak for solving $(n-1)$-set agreement.

6 Lower bounds

This section presents two lower bounds for renaming. Section 6.1 presents a lower bound for adaptive renaming, while Section 6.2 focuses on non-adaptive renaming. Section 6.3 briefly explains the underlying mathematics for these lower bounds.

6.1 Adaptive renaming

As explained in Section 5.2, in a system made up of n processes, $(n-1)$-set agreement, $(n-1)$-Test&set and $(2p - \lceil \frac{p}{n-1} \rceil)$-renaming are equivalent [23, 43, 44]. Since $(n-1)$-set agreement is not wait-free solvable in the asynchronous read/write shared memory model [12, 32, 54], we get $(2p - \lceil \frac{p}{n-1} \rceil)$-renaming is not wait-free solvable in that model. We can think of this lower bound in this way: if we ask $(2p-1)$-renaming to save one name only when all processes participate, the task becomes unsolvable. Therefore, the adaptive $(2p-1)$-renaming algorithms presented in Section 4.1 are optimal, concerning the output name space.

Remark Interestingly, $(2p - \lceil \frac{p}{n-1} \rceil)$-renaming is impossible even if we drop the index independence requirement for renaming. This requirement is not needed to prove that $(n-1)$-Test&set can be solved from any solution for $(2p - \lceil \frac{p}{n-1} \rceil)$-renaming. Because $(n-1)$-set agreement can be solved from $(n-1)$-Test&set, it follows that $(2p - \lceil \frac{p}{n-1} \rceil)$-renaming remains impossible. This means that what makes the task unsolvable is the fact that the output name space must gradually grow as the number of participating processes grows.

6.2 Non-adaptive renaming

The lower bound for adaptive renaming may suggest that $M = 2n-1$ is a lower bound for non-adaptive renaming. In fact it is, for infinitely many values of n [9, 15, 16, 32, 30], however, there are specific values of n (infinitely many also), for which there is a $(2n-2)$-renaming algorithm [15]. More precisely, if the binomial coefficients $\binom{n}{1}$, ..., $\binom{n}{n-1}$ are not relatively prime, i.e., their greatest common divisor is not 1, then there is no wait-free M-renaming algorithm with $M < 2n-1$, in the asynchronous read/write shared memory model, otherwise there exists a wait-free $(2n-2)$-renaming algorithm in that model. For example, such an algorithm exists for $n = 6, 10, 12, 14, 15$ processes and does not exist for other values of n smaller than 14. There is no lower bound for the "exceptional" values of n.

Remark While the index independence requirement is not important in the lower bound for adaptive renaming, it is crucial in the lower bound for non-adaptive renaming. Roughly speaking, it imposes a "symmetry" on the new names obtained by the processes that precludes achieving a $(2n-2)$-renaming algorithm for some values of n. This is explained in more detail in Section 6.3.

6.3 The underlying mathematics

A deep connection between distributed computing and topology was independently discovered in [12, 32, 54]. Here we briefly recall this connection, and explain how it is used to prove lower and upper bounds for renaming.

6.3.1 The topology connection

The papers [12, 32, 54] show, roughly speaking, that the executions of any wait-free algorithm in the asynchronous read/write shared memory model with n processes, starting on one input configuration, can be represented by an $(n-1)$-dimensional solid object with no holes. Furthermore, this representation implies a topological characterization of the problems that can be solved in a wait-free manner [32]. By now there are many papers extending this characterization to other models, or using it to derive algorithms and to prove impossibility results. There are also a few tutorials such as [29, 31, 47] that can help getting into the area. We recall some basic notions next (see the tutorials for a more detailed and precise exposition).

Simplexes and complexes A discrete geometric object can be represented by a generalization of a graph, known in topology as a complex. Recall that a graph consists of a base set of elements called vertices, and two-element sets of vertices called edges. More generally, a *k-simplex* is a set of vertices, of size $k+1$. Thus, we may think of a vertex as a 0-simplex, and an edge as a 1-simplex. A *complex* is a set of simplexes closed under containment. As with a graph, it is often convenient to embed a complex in Euclidean space. Then, 1-dimensional simplexes are represented as lines, 2-dimensional simplexes as solid triangles and 3-dimensional simplexes as solid tetrahedrons. Figure 16 depicts a 1-dimensional complex and a 2-dimensional complex.

Representing wait-free executions The wait-free executions of an algorithm for two processes that start with a specific input configuration (assignment of input values), can be represented by a 1-dimensional complex that is a subdivided line. Each edge of the line is a simplex that represents an execution of the algorithm. The two vertexes of each edge are labeled with the local states of the two processes, respectively, corresponding to the end of the execution represented by the edge. A vertex that is shared by two edges corresponds to a process that cannot distinguish between the two executions represented by the edges. An algorithm that executes more steps, will induce a line with a finer subdivision (more edges). But the line will always be connected. The endpoints of the line correspond to executions where a process runs solo, namely, it picks an output value without seeing any value written to the shared memory by the other process. For example, the graph in Figure 1 is the topological representation of the renaming algorithm for two processes described in Section 2.4. Observe that each vertex of the subdivision is labeled with an index of a process, and the vertexes of each edge have distinct indexes. Such a subdivision is called *chromatic*.

Similarly, for the case of three processes that start with a specific input configuration, the executions are represented by a complex that is a subdivided triangle. Wait-free algorithms that run longer, induce a finer subdivision, but one that is always connected and with no holes. Figure 15 shows the complex of an algorithm for three processes, P, Q and R, that execute one round of communication. (Actually the real complex is not a subdivision, however, a "subcomplex" of it, is a subdivision [9, 12, 54].) The corners of the subdivision correspond to solo executions, and the edges in the boundary correspond to executions in which two processes pick a value without seeing any value written by the other process. For example, the edges in the segment of the boundary linking the corner P and the corner Q (labeled $P-Q$), correspond to executions in which P and Q decide without seeing any value written by R. The triangles inside the subdivision correspond to executions in which at least one process sees a value written to the shared memory by the three processes. In particular, the triangle at the center represents the execution in which all processes see each other. The subdivision of the triangle is *chromatic* in a sense that the corners are labeled with distinct process' indexes, an edge in the segment of the boundary linking the corner X and corner Y, is labeled $X-Y$, and all triangles are labeled $P-Q-R$.

In general, a chromatic subdivision of an $(n-1)$-dimensional simplex is used to represent the executions of an algorithm for n processes.

6.3.2 Using topology to study renaming

The lower bound for non-adaptive renaming is proved via WSB. As explained in Section 5.2.2, WSB and (non-adaptive) $(2n-2)$-renaming are equivalent.

Recall that in the WSB task not all processes obtain 0 and not all processes obtain 1, in the executions in which all processes terminate. Assume we have a wait-free WSB algorithm for n processes. As explained above, the complex of the algorithm is a chromatic subdivision of an $(n-1)$-dimensional simplex. Each vertex of the subdivision can be labeled with the output value decided by the process corresponding to the vertex. This gives a binary coloring to the subdivision. Observe that the vertexes of an $(n-1)$-dimensional simplex inside the subdivision cannot be colored with the same color because

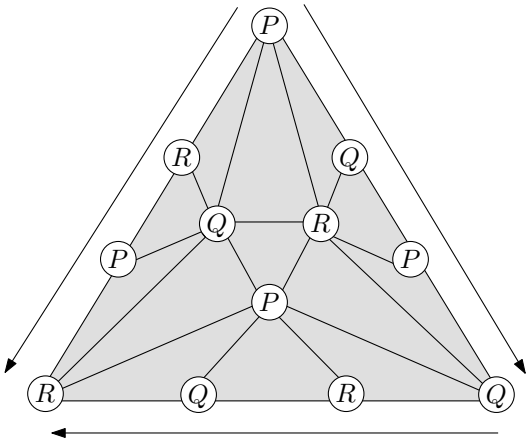

Figure 15: Complex of an algorithm for three processes

these simplexes would correspond to execution in which all processes obtain the same value, violating the task specification. We call these simplexes *monochromatic*.

The index independence property of WSB implies that the binary coloring of the subdivision, is symmetric on the boundary. Roughly speaking, this means that if we consider executions where only a subset of the processes take steps, these executions would look the same if we replaced the indexes of the processes by a different subset of processes. A more precise definition is in [9, 14, 16, 32]; for the aim of this survey it is enough to say that the coloring is *symmetric*. For example, in the subdivision in Figure 15, the corners of the subdivision have the same binary color, and the binary colors one reads on the boundary from the the corner corresponding to the solo execution of P to the one of Q, as the arrow in Figure 15 shows, are the same one reads from P to R and from Q to R, respectively.

The impossibility of WSB consists in proving that whenever n is such that the integers $\binom{n}{1}$, ..., $\binom{n}{n-1}$ are not relatively prime, then any chromatic subdivision of an $(n-1)$-dimensional simplex with a binary coloring that is symmetric, has at least one monochromatic simplex. Roughly speaking, for these values of n, the symmetry of the binary coloring forces any such subdivision to contain at least one monochromatic simplex. In other words, for these values of n, any WSB algorithm fails in at least one execution (in which all processes decide an output value). Figure 16 contains two chromatic subdivisions with a binary coloring that is symmetric; both complexes have monochromatic simplexes ($n = 6$ is the smallest value such that the binomial coefficients are relatively prime). Therefore, for these values of n, there is no wait-free WSB algorithm for n-processes, hence neither there is a wait-free $(2n-2)$-renaming algorithm, since WSB and $(2n-2)$-renaming are equivalent (see Section 5.2.2). This gives the lower bound for renaming.

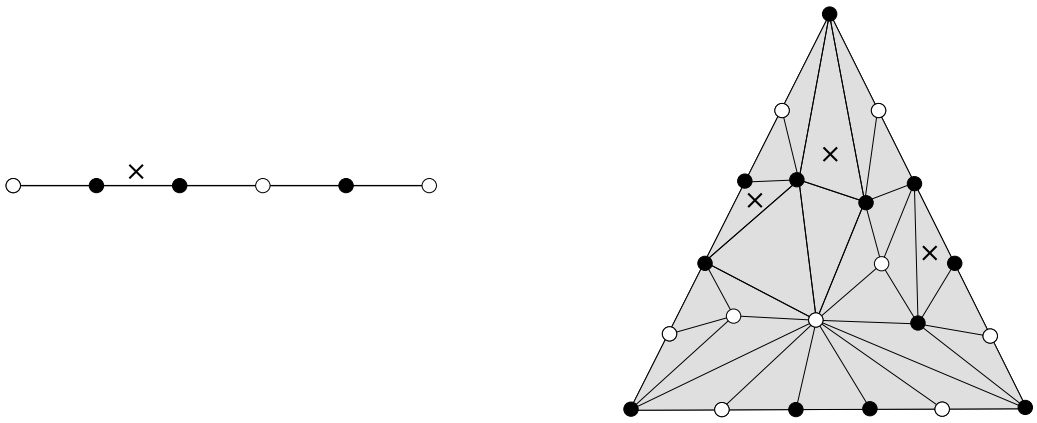

Figure 16: Two subdivided simplexes

However, for the other values of n, where the integers $\binom{n}{1}$, ..., $\binom{n}{n-1}$ are relatively prime, there exist chromatic subdivisions of an $(n-1)$-dimensional simplex, with a binary coloring that is symmetric on the boundary, and without monochromatic simplexes [14, 15]. A fundamental result in [32] implies that these subdivisions imply that there exists a wait-free WSB algorithm. Very roughly, in that paper it is shown that the subdivisions can be used to construct a WSB algorithm.

Therefore, if $\binom{n}{1}$, ..., $\binom{n}{n-1}$ are relatively prime, then there is a wait-free $(2n-2)$-renaming algorithm. More details can be found in [14, 15].

Discussion This "non-uniform" behavior of the lower bound for non-adaptive renaming is due to the fact that a topological object exists in some dimensions, while it does not exist in others. In contrast, the lower bound for adaptive renaming can be proved using the relation to set agreement (Section 5.2), and in turn, the impossibility of wait-free solving $(n-1)$-set agreement comes from Sperner's Lemma, a classic result in combinatorial topology, that is true in every dimension.

7 Conclusion

This paper has presented an overview of recent research results about renaming and its relation to other problems, in shared memory systems. While several renaming variants have been described the main focus of the paper has been placed on the most basic form of renaming, namely the one-shot version where processes start with initial names taken from a large space of names, and decide once one new names, from a name space as small as possible.

A main effort of the paper has been to give an intuition of the difficulty of the renaming problem, the techniques used in this area, and the underlying algorithmic principles, to motivate the reader to look at the literature of this rich research area. The paper started by describing the model and the renaming problem, as well as the shared memory abstractions that simplify the design of renaming algorithms: collect, snapshot, write-snapshot.

The main body of the paper has been organized in three topics. First, some of the more significant renaming algorithms have been discussed– size-adaptive optimal renaming algorithms and a time-adaptive renaming algorithm. Second, the paper has discussed the difficulty of the renaming problem in relation to other distributed computability problems, mainly k-test&set, k-set agreement and weak symmetry breaking. Third, the paper has summarized the lower bounds associated with renaming, and explained the difference in the lower bounds for adaptive vs non-adaptive renaming. Roughly speaking, adaptive renaming is equivalent to set agreement, while non-adaptive renaming lower bounds require more involved algebraic topology techniques. A brief overview of the the mathematics underlying renaming has also been presented.

References

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

[2] Afek Y., Gamzu I., Levy I., Merritt M. and Taubenfeld G., Group Renaming. *Proc. 12th Int'l Conference On Principles Of Distributed Systems (OPODIS'08)*, Springer Verlag LNCS #5401, pp.58-72, 2006.

[3] Afek Y. and Merritt M., Fast, Wait-Free $(2k-1)$-Renaming. *Proc. 18th ACM Symposium on Principles of Distributed Computing (PODC'99)*, ACM Press, pp. 105-112, 1999.

[4] Afek Y. and Nir I., Failure Detectors in Loosely Named Systems. *Proc. 27th ACM Symposium on Principles of Distributed Computing (PODC'08)*, ACM Press, pp. 67-74, 2008.

[5] 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.

[6] Attiya H. and Fouren A., Polynomial and Adaptive Long-lived $(2p-1)$-Renaming. *Proc. 14th Int'l Symposium on Distributed Computing (DISC'00)*, Springer Verlag LNCS #1914 , pp.149-163, 2000.

[7] Attiya H. and Fouren A., Adaptive and Efficient Algorithms for Lattice Agreement and Renaming. *SIAM Journal of Computing*, 31(2):642-664, 2001.

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

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

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

[11] Biran O., Moran S. and Zaks S., A Combinatorial Characterization of the Distributed 1-Solvable Tasks. *Journal of Algorithms*, 11(3):420-440, 1990.

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

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

[14] Castañeda A., A Study of the Solvability of Weak Symmetry Breaking and Renaming, *PhD thesis, Posgrado en Ciencia e Ingeniería de la Computación*, Universidad Nacional Autónoma de México (UNAM), December 2010.

[15] Castañeda A. and Rajsbaum S., New Combinatorial Topology Upper and Lower Bounds for Renaming. *Proc. 27th ACM Symposium on Principles of Distributed Computing (PODC'08)*, ACM Press, pp. 295-304, 2008.

[16] Castañeda A. and Rajsbaum S., New Combinatorial Topology Upper and Lower Bounds for Renaming: The Lower Bound. *Distributed Computing*, 22(5):287-301, 2010.

[17] Chandra T. and Toueg S., Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM*, 43(2):225-267, 1996.

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

[19] 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.

[20] Fouren A., Exponential Examples of Two Renaming Algorithms. *Technion Tech Report*, 1999. *Available at* www.cs.technion.ac.il/ hagit/pubs/expo.ps.gz.

[21] Gafni E., Group Solvability. *Proc. 18th Int'l Symposium on Distributed Computing (DISC'04)*. Springer Verlag LNCS #3274, pp. 30-40, 2004.

[22] Gafni E., Renaming with k-set Consensus: an Optimal Algorithm in $n+k-1$ Slots. *Proc. 10th Int'l Conference On Principles Of Distributed Systems (OPODIS'06)*, Springer Verlag LNCS #4305, pp. 36-44, 2006.

[23] Gafni G., Mostéfaoui A., Raynal M. and Travers C., From Adaptive Renaming to Set Agreement. *Theoretical Computer Science*, 410(14-15): 1328-1335, 2009.

[24] 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.

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

[26] Gafni G., Raynal M. and Travers C., Test&set, Adaptive Renaming and Set Agreement: a Guided Visit to Asynchronous Computability. *26th IEEE Symposium on Reliable Distributed Systems (SRDS'07)*, IEEE Computer Society Press, pp. 93-102, Beijing (China), 2007.

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

[28] Herlihy M.P., Luchangco V. and Moir M., Obstruction-free Synchronization: Double-ended Queues as an Example. *Proc. 23th Int'l IEEE Conference on Distributed Computing Systems (ICDCS'03)*, pp. 522-529, 2003.

[29] Herlihy M. and Rajsbaum S., New Perspectives in Distributed Computing. *Proc. 24th Int'l Symposium on Mathematical Foundations of Computer Science (MFCS'99)*, Springer Verlag LNCS #1672, pp.170-186, 2006.

[30] Herlihy M. and Rajsbaum S., Algebraic Spans. *Mathematical Structures in Computer Science*, 10(4):549-573, 2000.

[31] Herlihy M., Rajsbaum S., and Tuttle M., An Overview of Synchronous Message-Passing and Topology. *Electronic Notes in Theoretical Computer Science*, 39(2):1-17, 2001.

[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 Kaufmann Pub.*, San Francisco (CA), 508 pages, 2008.

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

[35] Imbs D. and Raynal M., Help when Needed, but no More: Efficient Read/Write Partial Snapshot. *Proc. 23th Int'l Symposium on Distributed Computing (DISC'09)*, Springer-Verlag LNCS #5805, pp. 142-156, 2009

[36] Imbs D. and Raynal M., On Adaptive Renaming Under Eventually Limited Contention. *12th Int'l Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS'10)*, Springer Verlag LNCS #6366, pp. 377-387, 2010.

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

[38] Lamport L., A Fast Mutual Exclusion Algorithm. *ACM Transactions on Computer Systems*, 5(1):1-11, 1987.

[39] Loui M. and Abu-Amara H., Memory Requirements for for agreement among Unreliable Asynchronous processes. *Advances in Computing Research*, 4:163-183, JAI Press Inc., 1987.

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

[41] Moir M., Fast, Long-Lived Renaming Improved and Simplified. *Science of Computer Programming*, 30:287-308, 1998.

[42] Moir M. and Anderson J., Wait-Free Algorithms for Fast, Long-Lived Renaming. *Science of Computer Programming*, 25(1):1-39, 1995.

[43] Mostéfaoui M., Raynal M., and Travers C., Exploring Gafni's Reduction Land: from Ω^k to Wait-free adaptive $(2p-\lceil \frac{p}{k} \rceil)$-renaming via k-set Agreement. *Proc. 20th Int'l Symposium on Distributed Computing (DISC'09)*, Springer Verlag LNCS #4167, pp. 1-15, 2006.

[44] Mostéfaoui A., Raynal M. and Travers C., From Renaming to k-Set Agreement. *14th Int'l Colloquium on Structural Information and Communication Complexity (SIROCCO'07)*, Springer Verlag LNCS #4474, pp. 62-76, 2007.

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

[46] Neiger G., Failure Detectors and the Wait-free Hierarchy. *14th ACM Symposium on Principles of Distributed Computing (PODC'95)*, ACM Press, pp. 100-109, 1995.

[47] Rajsbaum S., Iterated Shared Memory Models. *Proc. 9th Latin American Symposium Theoretical Informatics (LATIN'10)*, Springer Verlag LNCS #6034, pp. 407-416, 2010.

[48] Rajsbaum S. and Raynal M., A Theory-Oriented Introduction to Wait-free Synchronization Based on the Adaptive Renaming Problem. *Proc. 25th Int'l Conference on Advanced Information Networking and Applications (AINA'11)*, IEEE Press, Singapore, March 22-25, 2011.

[49] Raynal M., Locks Considered Harmful: a Look at Non-traditional Synchronization. *Proc. 6th Int'l Workshop on Software Technologies for Future Embedded and Ubiquitous Computing Systems (SEUS'08)*, Springer Verlag LNCS #5287, pp. 369-380, 2008.

[50] Raynal M., Failure Detectors for Asynchronous Distributed Systems: an Introduction. *Wiley Encyclopdia of Computer Science and Engineering*, Vol. 2, pp. 1181-1191, 2009 (ISBN 978-0-471-38393-2).

[51] Raynal M., Communication and Agreement Abstractions for Fault-Tolerant Asynchronous Distributed Systems. *Morgan & Claypool Publishers*, 251 pages, 2010 (ISBN 978-1-60845-293-4).

[52] Raynal M., Fault-Tolerant Agreement in Synchronous Message-Passing Systems. *Morgan & Claypool Publishers*, 165 pages, 2010 (ISBN 978-1-60845-525-6).

[53] Raynal M. and Travers C., In Search of the Holy Grail: Looking for the Weakest Failure Detector for Wait-free Set Agreement. *Proc. 10th Int'l Conference On Principles Of Distributed Systems (OPODIS'06)*, Springer-Verlag LNCS #4305, pp. 1-17, 2006.

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

[55] Taubenfeld G., On the Computational Power of Shared Objects. *Proc. 13th Int'l Conference On Principle Of Distributed Systems (OPODIS 2009)*, Springer Verlag LNCS #5923, pp. 270-284, 2009.

A Proof of Theorem 1 (Section 3.2)

The the cost of a distributed algorithm is often measured by the number of shared memory accesses, and called *step complexity.*

Theorem 1 The algorithm described in Figure 2 is a wait-free construction of a a write-snapshot abstraction. Moreover, the step complexity is $O(n(n-|res|+1))$ where res is the set returned by SM.write_snapshot(n).

Proof Let us first prove the termination property.
Claim C. If at most x processes invoke SM.write_snapshot(x) then at most $(x-1)$ processes invoke SM.write_snapshot$(x-1)$ and at least one process stops at line 05 of its SM.write_snapshot(x) invocation.
Let us consider a correct process p_i that invokes SM.write_snapshot(n). It follows from Claim C and the fact that at most n processes invoke SM.write_snapshot(n) that either p_i stops at that invocation or belongs to the at most $n-1$ processes that invoke SM.write_snapshot$(n-1)$. It then follows by induction from the claim that if p_i has not stopped during a previous invocation, it is the only process that invokes SM.write_snapshot(1). It then follows from the text of the algorithm that it stops at that invocation.

Proof of claim C. Assuming that at most x processes invokes SM.write_snapshot(x), let p_k the last process that writes $SM[x]$. We necessarily have $|have\_written_k| \leq x$. If p_k finds $|have\_written_k| = x$, it stops at line 05. Otherwise we have $|have\_written_k| < x$ and p_k invokes SM.write_snapshot$(x-1)$. But in that case, as p_k is the last process that wrote into $SM[x]$, we necessarily have less than x processes that have written into $SM[x]$, and consequently, at most $(x-1)$ invoke SM.write_snapshot$(x-1)$. End of the proof of claim C.

The proof of the self-inclusion property is trivial. Before stopping at a recursion level x (line 04-05), a process p_i has written old_name_i into $SM[x][i]$ (line 01), and consequently we have then $old\_name_i \in has\_written_i$ which concludes the proof of the self-inclusion.

To prove the self-containment and immediacy properties, let us consider a run in which a set X of x processes participate. The processes of X that do not crash invoke recursively SM.write_snapshot$(n-1)$, etc., until SM.write_snapshot(x). Let $X1$ be the set of processes that stop at line 05 of SM.write_snapshot(x). Let p_i and p_j two processes of $X1$. As $|have\_written_i| = |have\_written_j| = x$ when they stop, and $SM[x]$ can only increase and the read/write of $SM[x]$ are atomic, it follows that $have\_written_i = have\_written_j$, i.e., $res_i = res_j$. Hence, one with respect to the other, res_i and res_j satisfy the containment and immediacy properties. The same observation applies to the set $X2$ of processes that stops at line 05 of SM.write_snapshot$(x-1)$, etc.

Hence, considering a process p_i that stops at recursion level y and a process p_j that stops at recursion level $z < y$, we show that res_i and res_j are such that $old\_name_j \in res_i$, $old\_name_i \notin res_j$, and $res_j \subset res_i$ from which follow the containment and immediacy properties for any pair. We have:

$$res_i = \{old\_name \mid \exists k \text{ such that } have\_written_i[k] = old\_name\}, \text{ and}$$
$$res_j = \{old\_name \mid \exists k \text{ such that } have\_written_j[k] = old\_name\}.$$

where $|have\_written_i| = y > |have\_written_j| = z$,

Let us observe that a process p_x that stops at level ℓ has previously written its initial name first in $SM[n]$ then in $SM[n-1]$, etc., until $SM[\ell]$. Moreover, process p_x does not write it in $SM[\ell-1]$, etc., until $SM[1]$. Hence, $SM[1] \subseteq SM[2] \subseteq \cdots \subseteq SM[n]$. It follows from that observation, and the fact that $z < y$, that $SM[z] \subset SM[y]$. Consequently, $have\_written_j \subset have\_written_i$ (i.e., $res_j \subset res_i$). Moreover, as p_j stops at level $y < z$, $old\_name_j \in res_i$ and $old\_name_i \notin res_j$.

As far as the number of shared memory accesses is concerned we have the following. Let res be the set returned by the SM.write_snapshot(n) invocation considered. Each recursive invocation costs $n+1$ shared memory accesses (lines 01-02). Moreover, the sequence of invocations, namely, SM.write_snapshot(n), etc., until SM.write_snapshot$(|res|)$ (recursion level at which the recursion stops) contains $n-|res|+1$ invocations. It follows that the cost is $O(n(n-|res|+1))$ shared memory accesses. $\square_{Theorem\ 1}$

B Proof of Theorem 2 (Section 3.3)

Theorem 2 The algorithm described in Figure 3 implements a splitter object. Moreover, a process accesses at most four times the shared memory.

Proof The validity and solo execution properties are trivial. Moreover, as there is no loop, each invocation by a correct process trivially terminates (and consequently the algorithm is wait-free). The fact that a process issues at most 4 shared

memory accesses is also trivial.

Let us now consider the concurrent execution property. Let us assume that x processes access the splitter object. Let us first observe that, due to the initialization of *CLOSED* not all of them can get the value *right* (for a process to obtain *right*, another process has first to set *CLOSED* to *true* at line 03).

Let us now consider the last process that executes line 01. If it does not crash, due to the predicate at line 05, this process cannot get the value *down*. Hence not all processes can get the value *down*.

Finally, no two process can get the value *stop*. Let p_i be the first process that finds $LAST = old\_name_i$ at line 05 (let us notice that p_i gets the value *stop* if it does not crash). This means that no process p_j has modified X while p_i was executing the lines 01-05. It follows that any process $p_j \neq p_i$ that will modify $LAST$ (at line 01) will find $CLOSED = true$ (at line 02), and consequently will not get the value *stop*. $\square_{Theorem\ 2}$

C Proof of Theorem 3 (Section 4.1.2)

This section shows that the recursive algorithm described in Figure 5 is correct, i.e., all correct participating processes obtain a new name in the interval $[1..2p-1]$ (where p is the number of participating processes), and no two new names are identical. Moreover, the process indexes are used only as an addressing mechanism (index independence).

Notation In the following the sentence "process p_i stops in $SM[x, f, d]$" means that p_i executes line 06 during its invocation new_name(x, f, d).

Remark The proof is based on a reasoning by induction. This is a direct consequence of the recursive formulation of the algorithm. In that sense the proof provides us with a deeper insight on the way the algorithm works.

Lemma 1 *Let $\langle x, f, d\rangle$ be a triple such that $x \geq 1$ and assume that at most x processes invoke the operation* new_name(x, f, d). *When considering these processes we have the following. At most one process stops in $SM[x, f, d]$ (line 06), at most $(x-1)$ processes invoke* new_name$(x-1, f, d)$ *(line 09) and most $(x-1)$ processes invoke* new_name$(x-1, f', \overline{d})$ *(line 07).*

Proof

Let $u \leq x$ be the number of processes that invoke new_name(x, f, d). If $u < x$, it follows that the predicate at line 03 is false for these processes that consequently proceed to line 09 and invoke new_name$(x-1, f, d)$. As $u \leq x-1$, the lemma follows.

Let us now consider the case where x processes invoke new_name(x, f, d). We have then the following.

- Let y be the number of processes for which the predicate $|competing_j| = x$ (line 03) is false when they invoke new_name(x, f, d). We have $0 \leq y < x$. It follows from the text of the algorithm that these y processes invoke new_name$(x-1, f, d)$ at line 09. As $y < x$, the lemma follows for these invocations.

- Let z be the number of processes for which the predicate $|competing_j| = x$ (line 03) is true when they invoke new_name(x, f, d). We have $1 \leq z \leq x$ and $y + z = x$.

  If one of these z processes p_k is such that the predicate of line 05 is true (i.e., $old\_name_k = \max(SM[x, f, d])$), then p_k executes line 06 and stops inside $SM[x, f, d]$. Let us also notice that this is always the case if $x = 1$. If $x > 1$, it follows from $y + z = x$ that $z - 1 \leq x - 1$. Then, the other $z - 1$ processes invoke new_name$(x-1, f', \overline{d})$. As $z - 1 \leq x - 1$, the lemma follows for these invocations.

  If the test of line 05 is false for each of the z processes, it follows that the process p_k that has the greatest old name among the x processes that invoke new_name(x, f, d), is necessarily one of the y previous processes. Hence, in that case, we have $y \geq 1$. As $y + z = x$, this implies $z < x$. It then follows that at most $z \leq x - 1$ processes invoke new_name$(x-1, f', \overline{d})$, which concludes the proof of the lemma. $\square_{Lemma\ 1}$

Lemma 2 *Every correct participant decides a new name.*

Proof As there are at most n participating processes, and each starts by invoking new_name$(n, 1, 1)$, it follows from Lemma 1 that every correct process stops in some $SM[x, *, *]$ for $1 \leq x \leq n$. It then decides a new name at line 06, which proves the lemma. $\square_{Lemma\ 2}$

Lemma 3 *Let p be the number of participating processes. We have $M = [1..2p-1]$. Moreover for any pair of participating processes p_i and p_j we have $res_i \neq res_j$.*

Proof The lemma is trivially true for $p = 1$ (the single process that invokes new_name$(n, 1, 1)$ obtains the new name 1, if it does not crash). A simple case analysis shows that the new names for $p = 2$ are a pair in $[1..3]$ (each pair is actually associated with a set of concurrency and failure patterns).

The rest of the proof is by induction on the number of participating processes. The previous paragraph has established the base cases. Assuming that the lemma is true for all $p' \leq p$ (induction assumption), let us show that it is true for $p+1$ participating processes (induction assumption).

Each of the $p+1$ processes invoke new_name$(n, 1, 1)$. Each of these invocations entails the invocation of new_name$(n-1, 1, 1)$ (line 09), etc., until the invocation new_name$(p+1, f, d)$ with $f = 1$ and $d = 1$.

- Let Y be the set of processes p_j (with $|Y| = y$) such that the predicate $|contending_j| = p+1$ (line 03) is false. We have $0 \leq y < p+1$. These processes invoke new_name(p, f, d), etc., until new_name(y, f, d) and due to the induction assumption they rename (with distinct new names) in $[f..f+2y-2]$, namely, $[1..2y-1]$ since $f = 1$.
- Let Z be the set of processes p_j (with $|Z| = z$) such that the predicate $|contending_j| = p+1$ (line 03) is true. We have $1 \leq z \leq p+1$ and $y + z = p+1$. At line 04, each of these z processes obtain $last = f + 2(p+1) - 2 = f + 2p = 2p+1$.
  - If one of these z processes p_k is such that $old\_name_k = \max(SM[p+1, f, d])$ (line 05), it stops at $SM[p+1, f, d]$) obtaining the name $res = last = f + 2p = 2p+1$ (as $f = 1$).
  - If no process stops at $SM[p+1, f, d]$, we have $1 \leq z \leq p$ and $1 \leq y$ (this is because the process with the greatest old name is then necessarily a process of Y).

  Hence, the $z' = z \leq p$ or $z' = z - 1 \leq p$ processes that do not stop at $SM[p+1, f, d]$, invoke new_name$(p, last-1, \overline{d})$, etc., until new_name$(z', f+2p-1, \overline{d})$. Due to the induction assumption, these z' processes rename (with distinct new names) in the interval $[(f+2p-1)-(2z'-2)..f+2p-1] = [2p-(2z'-2)..2p]$.

  Hence, when considering the $y + z = p+1$ processes of $Y \cup Z$, the y processes of Y rename with distinct new names in $[1..2y-1]$, the z' processes of Z rename with distinct names $[2p-(2z'-2)..2p]$ and, if $z'+1 = z$, the remaining process of Z obtains the new name $2p+1$. The new name space for the whole set processes $Y \cup Z$ is consequently $[1..2p+1]$.

  It remains to show that a process of Y and a process of Z cannot obtain the same new name. To that end we have to show that the upper bound $2y-1$ of the new names of the processes of Y is smaller than the lower bound $2p-(2z'-2)$ of the new names of the processes of Z, namely, $2y-1 < 2p-(2z'-2)$, i.e., $2(y+z') < 2(p+1)+1$, which is true because $z' \leq z$ and $y + z \leq p+1$, which concludes the proof of the lemma.

□$_{Lemma\ 3}$

Theorem 3 The algorithm described in Figure 5 is an adaptive M-renaming algorithm such that $M = 2p-1$ (where p is the number of participating processes). Its step complexity is $O(n^2)$.

Proof The fact that no two new names are identical and that the new name space is $[1..2p-1]$ is proved in Lemma 3. The fact that any correct participating process decides a new name is proved in Lemma 2. Finally the index independence property follows directly from the text of the algorithm: the process indexes are used only in lines 01 and 02 where they are used to address entries of arrays.

It is easy to see that the step complexity is $O(n^2)$. This follows from the fact that writing old_name_i at line 01 costs one shared memory access, reading $SM[x, f, d][1..n]$ costs n shared memory accesses and a process executes at most n recursive calls.

□$_{Theorem\ 3}$

D An example of execution of the algorithm of Figure 6 (Section 4.1.3)

Let us consider an execution in which $p = 5$ processes participate. Hence the new name space is $[1..2p-1] = [1..9]$. In order to simplify the presentation and without loss of generality we consider the participating processes are p_1, ..., p_5 and that $old\_name_i = i$. Moreover, let us assume that none of these processes crashes.

At the beginning only p_4 and p_5 are concurrently participating, each invoking new_name$(\langle n \rangle, 1, 1)$ (the other processes p_1, p_2 and p_3 will invoke new_name$(\langle n \rangle, 1, 1)$ later). It follows that each of p_4 and p_5 invokes $SM[\langle n \rangle]$.write_snapshot(n) that returns the set $\{old\_name_4, old\_name_5\} = \{4, 5\}$ to each of them that they save in $competing_4$ and $competing_5$,

respectively. Consequently, each of p_4 and p_5 considers that it is competing for a new name with the other process (line 03). Consequently, as $|competing_4| = |competing_5| = 2$, both p_4 and p_5 compute $last = 1 + (2 \times 2 - 2) = 3$ (line 04) and both "reserve" the new name 3 the one of them with the greatest old name. Hence, p_5 executes the lines 05-06 and receives the new name 3.

In contrast, p_4 executes lines 07-08 and invokes new_name$(\langle n, 2\rangle, 2, -1)$. This invocation by p_4 entails the invocation $SM[\langle n, 2\rangle]$.write_snapshot(2) that returns it the singleton set $\{old\_name_4\} = \{4\}$ (line 03). Consequently, p_4 is such that $last = 2 - (2 \times 1 - 2) = 2$ (line 03) and as $\{old\_name_4\} = \max\{4\})$, p_4 obtains the new name $last = 2$.

It follows that, as they were early with respect to the other processes, p_4 and p_5 have obtained the new names 2 and 3, respectively. Let us observe that, if they were the only participating processes, the new name space would be [1..3]. This is due to the fact that the algorithm is size-adaptive.

Moreover, when we consider the tree associated with the execution of all participating processes (this tree is described in Figure 17), p_5 stopped at the root (whose label is $\langle n\rangle$) while p_4 stopped at its descendant labeled $\langle n, 2\rangle$.

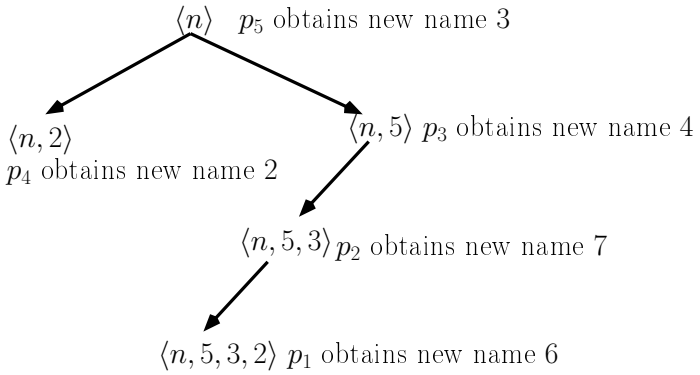

Figure 17: Tree associated with a concurrent execution of the algorithm of Figure 6

Let us now consider that, after p_4 and p_5 have returned from $SM[\langle n\rangle]$.write_snapshot(n), p_1, p_2 and p_3 concurrently invoke new_name$(\langle n\rangle, 1, 1)$. Their concurrent invocations $SM[\langle n\rangle]$.write_snapshot(n) return the set $\{1, 2, 3, 4, 5\}$ (the set of the five old names) to each of them (line 03). Hence, they all compute $last = 1 + (2 \times 5 - 2) = 9$ and reserve the new name 9 for the one of them that has the greatest old name, i.e., p_5. (This reservation is just in case , p_5 would be competing with them. let us observe that p_5 has already obtained its new name 3 but this is known neither by p_1, nor p_2 nor p_3).

It follows that each of p_1, p_2 and p_3 invokes new_name$(\langle n, 5\rangle, 8, -1)$ (where $8 = last + \overline{dir}$ with $dir = 1$) that entail their concurrent invocations of $SM[\langle n, 5\rangle]$.write_snapshot(5) that returns to each of them the set $\{1, 2, 3\}$. The three of them compute $last = 8 - (2 \times 3 - 2) = 4$ and reserve the new name 4 for the one of them with the greatest old name, namely p_3. Hence, p_3 obtains new name 4 (lines 05-06).

In contrast, p_2 and p_3 invoke new_name$(\langle n, 5, 3\rangle, 5, 1)$, where $5 = last + \overline{dir}$ with $dir = -1$ (line 08). Let us consider that both p_2 and p_3 invoke then concurrently $SM[\langle n, 5, 3\rangle]$.write_snapshot(3) from which they both obtain the set $\{2, 3\}$. Then, it is easy to see that p_3 obtains the new name $last = 5+(2\times 2-2) = 7$ (line 06). Finally p_2 invokes new_name$(\langle n, 5, 3, 2\rangle, 6, -1)$ and obtains the new name $last = 6 - (2 \times 1 - 2) = 6$.

Let us finally observe that, when a single process participates, whatever its initial name, it obtains the new name 1 from new_name$(\langle n\rangle, 1, 1)$, while it is at the root of the execution tree.

E Proof of Theorem 6 (Section 5.2.3)

Theorem 6 The algorithm described in Figure 13 solves in a wait-free manner the weak symmetry breaking problem from any solution to the $(n-1)$-set agreement problem.

Proof Let us remember that the weak symmetry breaking problem requires that not all processes decide 0 and not all processes decide 1. So, let us consider an execution in which all processes participate and decide a value.

Let us notice that, as there are n processes and the the new name space is $[1..2n-1]$, that at least one process obtains a new name in the range $[1..n]$. Let s, $1 \leq s \leq n$, be the number of processes with a new name in that range. The following observation is a direct consequence of the fact the array $M1[1..n]$ is made up of atomic registers and a process writes first $M1[1..n]$ before reading its values. It follows that, among those s processes, the process p_k whose new name new_name_k is the first written in $M1[1..n]$ is such that the predicate $(\exists\ j \mid new\_name_k = M1[j])$ is true when p_k checks it at line 04. It follows that process p_k outputs value 1 and is consequently a winner.

Let us now consider that the $s = n$. Due to the property of the $(n-1)$-set agreement object $SA[1]$ that there is at least process p_x whose new name cannot be returned from $SA[1]$. It then follows from the test of line 04 that this process decides 0 at line 05.

If $s < n$, then $n - s$ processes invokes $SA[2].\mathsf{SA\_propose}_{n-1}(new\_name_i)$ at line 07. A reasoning similar to the previous one shows that, among those $n - s$ processes, the process p_ℓ such that new_name_ℓ is the first new name written in $M2[n+1..2n-1]$, finds the predicate $(\exists\ j \mid new\_name_\ell = M2[j])$ equals to true (line 08), and consequently returns the value 0, which concludes the proof. $\square_{Theorem\ 6}$

F Renaming in message-passing systems

This appendix is motivated by "completeness": its aim is to give a glance at the renaming problem in asynchronous crash-prone message-passing systems.

Let us remember that the renaming problem has first been introduced in the context of unreliable asynchronous *message-passing* distributed systems [5]. As indicated in the introduction, the origin of this problem was motivated by the discovery of non-trivial agreement problems that can be solved in presence of faulty processes, in contrast to the consensus problem that cannot be solved in presence of even a single process crash in asynchronous systems [19]. In [5], Attiya et al. introduce the problem, analyze it, and provide several message-passing renaming algorithms. This appendix presents one of these algorithms which solves the M-renaming problem in presence of up to $t < n/2$ failures, for $M = (n - t/2)(t + 1)$. ([5] presents another algorithm that, under the same assumptions, provides $M = n + t$. Unfortunately, this algorithm is much more intricate.) Let us also observe that, as processes have to wait for messages, message-passing algorithms are not wait-free.

Message-passing algorithm: principle In this algorithm each process p_i manages a local set $view_i$ containing the initial names (old_name_j) it knows from the other processes. (Let us remember that process indexes are used for exposition, they are not know by the processes. Initially, a process knows only n and its initial name old_name_i.) Each time it learns new initial names, p_i propagate them to the other processes. To this end it uses the "broadcast $new(view_i)$" operation which is a shorthand for "send $new(view_i)$ to all processes (including itself)" (notice that a process can crash in the middle of such a statement, hence $new(view_i)$ can be propagated only to a subset of processes). When it receives a message $new(view)$, there are three cases.

- $view \subsetneq view_i$. In that case, p_i learns nothing. It simply discards the message.
- $view \setminus view_i \neq \emptyset$. In that case, p_i learns new initial names. It updates accordingly $view_i$ and consequently issues broadcast $new(view_i)$.
- $view = view_i$. It that case p_i learns that one more process knows the same set of initial names as it knows. So, p_i manages a counter ct_i to count the number of processes that know the same set $view$ as it knows.

As up to t processes can crash, p_i cannot expect to receive the same set $view$ from more than $n - t$ processes (including itself). So, when $ct_i = (n - t)$, p_i decides its new name. It is the pair $< |view_i|$, rank of old_name_i in $view_i >$. The algorithm is described in Figure 18.

Message-passing algorithm: why names are different Let a set $view$ be *stable* when a process has received $n-t$ copies of it (so, this process decides its new name from this set). A main property of the algorithm is the following: Stable sets are totally ordered (by inclusion). This follows from the fact that if $view_i$ is stable for p_i (i.e., p_i has received $new(view_i)$ from $n - t$ processes) and $view_j$ is stable for p_j (i.e., p_j has received $new(view_j)$ from $n - t$ processes), then, due the assumption $2t < n$, there is at least one process p_k from which p_i has received $new(view_i)$ and from which p_j has received $new(view_j)$. So, $view_i$ and $view_j$ are values taken by the set local variable $view_k$. As such a set $view_k$ can only increase, it follows that $view_i \subseteq view_j$ or $view_j \subseteq view_i$. This property allows to conclude that no two decided names are the same.

Message-passing algorithm: size of the new name space Let us notice that a set $view_i$ contains at most n initial names. So, a process sends its set $view_i$ at most n times. It follows that the algorithm terminates, and its message complexity is bounded by $O(n^3)$. The proof that each correct process decides follows from the fact that each set $view_i$ can only increase and its size is upper bounded by n.

As indicated above, the size of the new name space is $M = (n - t/2)(t + 1)$. This come from the following observation [5]. A new name is a pair $\langle v, r\rangle$. Due to the algorithm text, we trivially have $n - t \leq v \leq n$. Moreover, r is the rank of the deciding process p_i in the set $view_i$ containing v values. It follows that $1 \leq r \leq v$. Consequently the number of possible

```
view_i ← {old_name_i}; ct_i ← 0; decided_i ← false; broadcast new(view_i);
while (¬ decided_i) do
   wait until receive new(view);
   case (view ⊂ view_i)      then view carries old information: discard it
        (view = view_i)      then % one more process knows exactly the same %
                                  ct_i ← ct_i + 1;
                                  if (ct_i = n − t)
                                     then % view_i is stable %
                                          let v = |view_i|; r = rank of old_name_i in view_i;
                                          new name = ⟨v, r⟩; decided_i ← true
                                  end if
        (view \ view_i ≠ ∅)  then % p_i learns initial names %
                                  % Let p_j be the sender of new(view) %
                                  case  (view_i ⊂ view) then ct_i ← 1 % p_j knows view_i ∪ view %
                                       ¬(view_i ⊂ view) then ct_i ← 0 % p_j does not know view_i ∪ view %
                                  end case;
                                  view_i ← view_i ∪ view; broadcast new(view_i)
   end case
end while;
repeat forever
    wait until receive new(view); view_i ← view_i ∪ view; broadcast new(view_i)
end repeat.
```

Figure 18: A Message-Passing Renaming Algorithm [5]

decisions is $M = \Sigma_{x=n-t}^{n}\ x = (n - t/2)(t + 1)$. A fixed mapping from the $\langle v, r\rangle$ pairs to $[1..M]$ can be used to get integer names.

Message-passing vs shared memory It is important to notice that a process that decides a new name has to continue receiving and sending messages to help the other processes to decide. This help is necessary to deal with situations where some very slow processes start participating in the algorithm after some other processes have already decided. It is shown in [5] that there is no renaming algorithm if a process is required to stop just after deciding its new name. That is the price required by process coordination to solve the renaming problem in crash-prone asynchronous message-passing systems. When we look at the shared memory algorithms described in Section 4, the result of the process coordination is recorded into shared variables. As there is no such shared memory in the message-passing context, the processes have to "simulate" it by helping each other. (In a practical setting, a secondary storage -e.g., a disk- shared by the processes can be used to eliminate the second **while** loop.)