Exploring Gafni's reduction land: from Ω^k to wait-free adaptive $(2p - \lceil \frac{p}{k} \rceil)$-renaming via k-set agreement

Achour Mostefaoui, Michel Raynal, Corentin Travers

▶ **To cite this version:**

Achour Mostefaoui, Michel Raynal, Corentin Travers. Exploring Gafni's reduction land: from Ω^k to wait-free adaptive $(2p - \lceil \frac{p}{k} \rceil)$-renaming via k-set agreement. [Research Report] PI 1794, 2006, pp.19. inria-00001189

HAL Id: inria-00001189
https://inria.hal.science/inria-00001189

Submitted on 31 Mar 2006

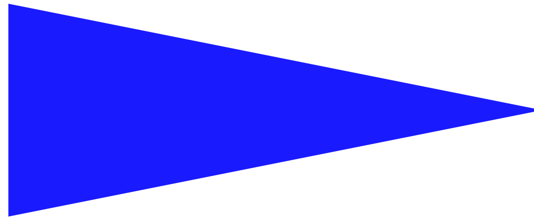
HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

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.

I R I S A

INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES

PUBLICATION
INTERNE
N° 1794



EXPLORING GAFNI'S REDUCTION LAND: FROM Ω^K TO WAIT-FREE ADAPTIVE $(2P - \lceil \frac{P}{K} \rceil)$-RENAMING VIA K-SET AGREEMENT

A. MOSTEFAOUI M. RAYNAL C. TRAVERS

ISSN 1166-8687

IRISA

CAMPUS UNIVERSITAIRE DE BEAULIEU - 35042 RENNES CEDEX - FRANCE

INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES
Campus de Beaulieu – 35042 Rennes Cedex – France
Tél. : (33) 02 99 84 71 00 – Fax : (33) 02 99 84 71 71
http://www.irisa.fr

Exploring Gafni's reduction land: from Ω^k to wait-free adaptive $(2p - \lceil \frac{p}{k} \rceil)$-renaming via k-set agreement

A. Mostefaoui* M. Raynal** C. Travers***

Systèmes communicants

Publication interne n ˚ 1794 — Avril 2006 — 19 pages

Abstract: The adaptive renaming problem consists in designing an algorithm that allows p processes (in a set of n processes) to obtain new names despite asynchrony and process crashes, in such a way that the size of the new renaming space M be as small as possible. It has been shown that $M = 2p - 1$ is a lower bound for that problem in asynchronous atomic read/write register systems.

This paper is an attempt to circumvent that lower bound. To that end, considering first that the system is provided with a k-set object, the paper presents a surprisingly simple adaptive M-renaming wait-free algorithm where $M = 2p - \lceil \frac{p}{k} \rceil$. To attain this goal, the paper visits what we call Gafni's reduction land, namely, a set of reductions from one object to another object as advocated and investigated by Gafni. Then, the paper shows how a k-set object can be implemented from a leader oracle (failure detector) of the class Ω^k. To our knowledge, this is the first time that the failure detector approach is investigated to circumvent the $M = 2p - 1$ lower bound associated with the adaptive renaming problem. In that sense, the paper establishes a connection between renaming and failure detectors.

Key-words: Adaptive algorithm, Asynchronous system, Atomic register, Consensus, Divide and conquer, Leader oracle, Renaming, Set agreement, Shared object, Wait-free algorithm.

(Résumé : tsvp)

* IRISA, Université de Rennes 1, Campus de Beaulieu, 35042 Rennes Cedex, France achour@irisa.fr
** IRISA, Université de Rennes 1, Campus de Beaulieu, 35042 Rennes Cedex, France, raynal@irisa.fr
*** IRISA, Université de Rennes 1, Campus de Beaulieu, 35042 Rennes Cedex, France, travers@irisa.fr

Centre National de la Recherche Scientifique (UMR 6074) Université de Rennes 1 – Insa de Rennes
Institut National de Recherche en Informatique et en Automatique – unité de recherche de Rennes

Renommage adaptatif sans attente
fondé sur un leader de la classe Ω^k

Résumé : Ce rapport présente un protocole qui permet de renommer les processus en présence de fautes dans un espace de $M = (2p - \lceil \frac{p}{k} \rceil)$ noms (où p est le nombre de processus qui participent à l'algorithme). Ce protocole est fondé sur un détecteur de fautes de la classe Ω^k.

Mots clés : Algorithme adaptatif, Algorithme sans attente, Registre atomique, Renommage, Consensus, Tolérance aux fautes, Accord ensembliste, Crash de processus, Détecteur de fautes, Leader, Réduction.

1 Introduction

The renaming problem The *renaming* problem is a coordination problem initially introduced in the context of asynchronous message-passing systems prone to process crashes [3]. Informally, it consists in the following. Each of the n processes that define the system has an initial name taken from a very large domain $[1..N]$ (usually, $n << N$). Initially, a process knows only its name, the value n, and the fact that no two processes have the same initial name. The processes have to cooperate to choose new names from a name space $[1..M]$ such that $M << N$ and no two processes obtain the same new name. The problem is then called *M-renaming*.

Let t denote the upper bound on the number of processes that can crash. It has been shown that $t < n/2$ is a necessary and sufficient requirement for solving the renaming problem in an asynchronous message-passing system [3]. That paper presents also a message-passing algorithm whose size of the renaming space is $M = n + t$.

The problem has then received a lot of attention in the context of asynchronous shared memory systems made up of atomic read/write registers. Numerous wait-free renaming algorithms have been designed (e.g., [2, 4, 5, 7, 9, 21]). *Wait-free* means here that a process that does not crash has to obtain a new name in a finite number of its own computation steps, regardless of the behavior of the other processes (they can be arbitrarily slow or even crash) [18]. Consequently, wait-free implies $t = n - 1$. An important result in such a context, concerns the lower bound on the new name space. It has been shown in [19] that there is no wait-free renaming algorithm when $M < 2n - 1$. As wait-free $(2n - 1)$-renaming algorithms have been designed, it follows that that $M = 2n - 1$ is a tight lower bound.

The previous discussion implicitly assumes the "worst case" scenario: all the processes participate in the renaming, and some of them crash during the algorithm execution. The net effect of crashes and asynchrony create "noise" that prevents the renaming space to be smaller than $2n - 1$. But it is not always the case that all the processes want to obtain a new name. (A simple example is when some processes crash before requiring a new name.) So, let p, $1 \leq p \leq n$, be the number of processes that actually participate in the renaming. A renaming algorithm guarantees *adaptive* name space if the size of the new name space is a function of p and not of n. Several adaptive wait-free algorithms have been proposed that are optimal as they provide $M = 2p - 1$ (e.g., [2, 4, 9]).

The question addressed in the paper Let us assume that we have a solution to the consensus problem. In that case, it easy to design an adaptive renaming algorithm where $M = p$ (the number of participating processes). The solution is as follows. From consensus objects, the processes build a concurrent queue that provides them with two operations: a classical enqueue operation and a read operation that provides its caller with the current content of the queue (without modifying the queue). Such a queue object has a sequential specification and each operation can always be executed (they are *total* operations according to the terminology of [18]), from which it follows that this queue object can be wait-free implemented from atomic registers and consensus objects [18]. Now, a process that wants to obtain a new name does the following: (1) it deposits its initial name in the queue, (2) then reads the content of the queue, and finally (3) takes as its new name its position in the sequence of initial names read from queue. It is easy to see that if p processes participate, they obtain the new names from 1 to p, which means that consensus objects are powerful enough to obtain the smallest possible new name space.

The aim of the paper is to try circumventing the lower bound $M = 2p - 1$ associated with the adaptive wait-free renaming problem, by enriching the underlying read/write register system with appropriate objects. More precisely, given M with $p \leq M \leq 2p - 1$, which objects (when added to a read/write register system) allow designing an M-renaming wait-free algorithm (without allowing designing an $(M - 1)$-renaming algorithm). The previous discussion on consensus objects suggests to investigate k-set agreement objects to attain this goal, and to study the tradeoff relating the value of k with the new renaming space. The *k-set agreement* problem is a distributed coordination problem (k defines the coordination degree it provides the processes with) that generalizes the consensus problem: each process proposes a value, and any process that does not crash must decide a value in such a way that at most k distinct values are decided and any decided value is a proposed value. The smaller the coordination degree k, the more coordination imposed on the

participating processes: $k = 1$ is the more constrained version of the problem (it is consensus), while $k = n$ means no coordination at all.

From k-set to $(2p - \lceil \frac{p}{k} \rceil)$-renaming Assuming k-set agreement base objects, and $p \leq n$ participating processes, the paper presents an adaptive wait-free renaming algorithm providing a renaming space whose size is $M = (2p - \lceil \frac{p}{k} \rceil)$. Interestingly, when considering the two extreme cases we have the following: $k = 1$ (consensus) gives $M = p$ (the best that can be attained), while $k = n$ (no additional coordination power) gives $M = 2p - 1$, meeting the lower bound for adaptive renaming in read/write register systems.

The proposed algorithm follows Gafni's reduction style [13]. It is inspired by the adaptive renaming algorithm proposed by Borowsky and Gafni in [9]. In addition to k-set objects, it also uses simple variants of base objects introduced in [9, 10, 15, 16], namely, *strong k-set agreement* [10], *k-participating set* [9, 15, 16]. These objects can be incrementally built from base k-set objects as indicated in Figure 1 (an arrow means "used by", the reverse direction means "can be reduced to").

Figure 1: From k-set to $(2p - \lceil \frac{p}{k} \rceil)$-renaming

The renaming algorithm is surprisingly simple. It is based on a very well-known basic strategy: decompose a problem into independent subproblems, solve each subproblem separately, and finally piece together the subproblem results to produce the final result. More precisely, the algorithm proceeds as follows:

- Using a k-participating set object, the processes are partitioned into independent subsets of size at most k.
- In each partition, the processes compete in order to acquire new names from a small name space. Let h be the number of processes that belong to a given partition.They obtain new names in the space $[1..2h - 1]$.
- Finally, the name spaces of all the partitions are concatenated in order to obtain a single name space $[1..M]$.

The key of the algorithm is the way it uses a k-participating set object to partition the p participating processes in such a way that, when the new names allocated in each partition are pieced together, the new name space is upper bounded by $M = (2p - \lceil \frac{p}{k} \rceil)$ ([1]). Interestingly, the processes that belong to the same partition can use any wait-free adaptive renaming algorithm to obtain new names within their partition (distinct partitions can even use different algorithms). This noteworthy modularity property adds a generic dimension to the proposed algorithm.

From the oracle Ω^k to k-set objects Unfortunately, k-set agreement objects cannot be wait-free implemented from atomic registers [10, 19, 23]. So, the paper investigates additional equipment the asynchronous read/write register system has to be enriched with in order k-set agreement objects can be implemented. To that aim, the paper investigates the family of leader oracles denoted $(\Omega^z)_{1 \leq z \leq n}$, and presents a k-set algorithm based on read/write registers and any oracle of the class Ω^k.

So, the paper provides reductions showing that adaptive wait-free $(2p - \lceil \frac{p}{k} \rceil)$-renaming can be reduced to the Ω^k leader oracle class. To our knowledge, this is the first time that oracles (failure detectors) are proposed and used to circumvent the $2p - 1$ adaptive renaming space lower bound. Several problems remain open. The most crucial is the statement of the minimal information on process crashes that are necessary and sufficient for bypassing the lower bound $2p - 1$. This seems to be related to the open problem that consists in finding the minimal assumptions on failures that allow solving the k-set agreement problem.

[1] When we were designing that algorithm, we had in mind sequential sorting algorithms such as *quicksort, mergesort* and *heapsort*, and were thinking to possible relations linking renaming and sorting.

Roadmap The paper is made up of 6 sections. Section 2 presents the asynchronous computation model. Then, Section 3 describes the adaptive renaming algorithm. This algorithm is based on a k-participating set object. Section 4 visits Gafni's reduction land by showing how the k-participating set object can be built from a k-set object. Then, Section 5 describes an algorithm that constructs a k-set object in an asynchronous read/write system equipped with a leader oracle of the class Ω^k. Finally, Section 6 provides a few concluding remarks while presenting open problems.

2 Asynchronous system model

Process model The system consists of n processes that we denote $p_1, \ldots, p_n$. The integer i is the index of p_i. Each process p_i has an initial name id_i such that $id_i \in [1..N]$. Moreover, a process does not know the initial names of the other processes; it only knows that no two processes have the same initial name. A process can crash. Given an execution, a process that crashes is said to be *faulty*, otherwise it is *correct* in that execution. Each process progresses at its own speed, which means that the system is asynchronous.

Coordination model The processes cooperate and communicate through two types of reliable objects: atomic multi-reader/single-write registers, and k-set objects.

A k-set object KS provides the processes with a single operation denoted $\mathsf{kset_propose}_k()$. It is a one-shot object in the sense that each process can invoke $KS.\mathsf{kset_propose}_k()$ at most once. When a process p_i invokes $KS.\mathsf{kset_propose}_k(v)$, we say that it "proposes v" to the k-set object KS. If p_i does not crash during that invocation, it obtains a value v' (we then say "p_i decides v'"). A k-set object guarantees the following two properties: a decided value is a proposed value, and no more than k distinct values are decided.

Notation Identifiers with upper case letters are used to denote shared registers or shared objects. Lower case letters are used to denote local variables; in that case the process index appears as a subscript. As an example, $level_i[j]$ is a local variable of the process p_i, while $LEVEL[j]$ is an atomic register.

3 An adaptive $(2p - \lceil \frac{p}{k} \rceil)$-renaming algorithm

This section presents an adaptive wait-free $(2p - \lceil \frac{p}{k} \rceil)$-renaming algorithm (where p is the number of processes that participate in the algorithm). As announced previously, this algorithm is based on atomic registers and k-set objects.

3.1 Non-triviality

Let us observe that the trivial renaming algorithm where p_i takes i as its new name is not adaptive, as the renaming space would always be $[1..m]$, where m is the greatest index of a participating process (as an example consider the case where only p_1 and p_n are participating in the renaming). To rule out this type of ineffective solution, we consider the following requirement for a renaming algorithm [7]:

- The code executed by process p_i with initial name id is exactly the same as the code executed by process p_j with initial name id.

This constraint imposes a form of anonymity with respect to the process initial names. It also means that there is a strong distinction between the index i associated with p_i and its original name id_i. The initial name id_i can be seen as a particular value defined in p_i's initial context. Differently, the index i can be seen as a pointer to the atomic registers that can be written only by p_i. This means that the indexes define the underlying "communication infrastructure".

3.2 k-participating set object

The renaming algorithm is based on a *k-participating set* object. Such an object generalizes the *participating set* object first defined in [9]. The particular case $k = 2$ when $n = 3$ has been introduced in [15, 16].

Definition A k-participating set object PS is a one-shot object that provides the processes with a single operation denoted $\mathsf{participating_set}_k()$. A process p_i invokes that operation with its name id_i as a parameter. The invocation $PS.\mathsf{participating_set}_k(id_i)$ returns a set S_i to p_i (if p_i does not crash while executing that operation). The semantics of the object is defined by the following properties [9, 15, 16]:

- Self-membership: $\forall i$: $id_i \in S_i$.
- Comparability: $\forall i, j$: $S_i \subseteq S_j \ \vee \ S_j \subseteq S_i$.
- Immediacy: $\forall i, j$: $(id_i \in S_j) \Rightarrow (S_i \subseteq S_j)$.
- Bounded simultaneity: $\forall \ell: \ 1 \le \ell \le n$: $|\{j \ : \ |S_j| = \ell\}| \le k$.

The set S_i obtained by a process p_i can be seen as the set of processes that, from its point of view, have accessed or are accessing the k-participating set object. A process always sees itself (self-membership). Moreover, such an object guarantees that the S_i sets returned to the process invocations can be totally ordered by inclusion (comparability). Additionally, this total order is not at all arbitrary: it ensures that, if p_j sees p_i (i.e., $id_i \in S_j$) it also sees all the processes seen by p_i (Immediacy). As a consequence if $id_i \in S_j$ and $id_j \in S_i$, we have $S_i = S_j$. Finally, the object guarantees that no more than k processes see the same set of processes (Bounded simultaneity).

As we will see later (Section 3.2), such an object can be constructed from k-set objects. When $k = n$, the bounded simultaneity requirement is always satisfied, and can consequently be omitted (then, the definition boils down to the participating set definition introduced in [9]).

level	stopped processes	S_i sets
10	p_5,p_9	$S_5 = S_9 = \{p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8, p_9, p_{10}\}$
9		empty level
8	p_1, p_3, p_{10}	$S_1 = S_3 = S_{10} = \{p_1, p_2, p_3, p_4, p_6, p_7, p_8, p_{10}\}$
7		empty level
6		empty level
5	p_2, p_8	$S_2 = S_8 = \{p_2, p_4, p_6, p_7, p_8\}$
4		empty level
3	p_7	$S_7 = \{p_4, p_6, p_7\}$
2	p_4, p_6	$S_4 = S_6 = \{p_4, p_6\}$
1		empty level

Table 1: An example of k-participating object ($p = 10 \le n$, $k = 3$)

Notation and properties Let S_j be the set returned to p_j after it has invoked $\mathsf{participating_set}_k(id_j)$, and $\ell = |S_j|$ (notice that $0 \le \ell \le n$). The integer ℓ is called the *level* of p_j, and we say "p_j is -or stopped- at *level* ℓ". If there is a process p_j such that $|S_j| = \ell$, we say "the level ℓ is not empty", otherwise we say "the level ℓ is empty". Let $\mathcal{L}$ be the set of non-empty levels ℓ, $|\mathcal{L}| = m \le n$. Let us order the m levels of $\mathcal{L}$ according to their values, i.e., $\ell_1 < \ell_2 < \cdots < \ell_m$ (this means that the levels in $\{1, \ldots, n\} \setminus \{\ell_1, \ldots, \ell_m\}$ are empty).

$|S_j| = \ell$ (p_j stopped at level ℓ) means that, from p_j point of view, there are exactly ℓ processes that (if they do not crash) stop at the levels ℓ' such that $1 \le \ell' \le \ell$. Moreover, these processes are the processes that define S_j. (It is possible that some of them have crashed before stopping at a level, but this fact cannot be known by p_j.) We have the following properties:

- If p processes invoke $\mathsf{participating_set}_k()$, no process stops at a level higher than p.
- $(|S_i| = |S_j| = \ell) \Rightarrow (Si = S_j)$ (from the comparability property).
- Let S_i and S_j such that $|S_i| = \ell_x$ and $|S_j| = \ell_y$ with $x < y$.
  - $S_i \subset S_j$ (from $\ell_x < \ell_y$, and the comparability property).

– $|S_j \setminus S_i| = |S_j| - |S_i| = \ell_y - \ell_x$ (consequence of the set inclusion $S_i \subset S_j$).

A k-participating set object can be seen as "spreading" the $p \leq n$ participating processes on at most p levels ℓ. This spreading is such that (1) there are at most k processes per level, and (2) each process has a consistent view of the spreading (where "consistent" is defined by the self-membership, comparability and immediacy properties). As an example, let us consider Table 1 that depicts the sets S_i returned to $p = 10$ processes participating in a k-participating set object (with $k = 3$), in a failure-free run. As we can see some levels are empty. Two processes, p_2 and p_8, stopped at level 5; their sets are equal and contain exactly five processes, namely the processes that stopped at a level ≤ 5.

The following lemma captures an important property provided by a k-participating set object. Let $ST[\ell_x] = \{j \text{ such that } |S_j| = \ell_x\}$ (the processes that have stopped at the level ℓ_x). For consistency purpose, let $\ell_0 = 0$.

Lemma 1 $|ST[\ell_x]| \leq \min(k, \ell_x - \ell_{x-1})$.

Proof $|ST[\ell_x]| \leq k$ follows immediately from the bounded simultaneity property. To show $|ST[\ell_x]| \leq \ell_x - \ell_{x-1}$, let us consider two processes p_j and p_i such that p_j stops at the level ℓ_x while p_i stops at the level ℓ_{x-1}. We have:

1. $|S_j| = \ell_x$ and $|S_i| = \ell_{x-1}$ (definition of "a process stops at a level").
2. $ST[\ell_x] \subseteq S_j$ (from the self-membership and comparability properties),
3. $ST[\ell_x] \cap S_i = \emptyset$ (from $S_j \neq S_i$ and the immediacy and self-membership properties),
4. $ST[\ell_x] \subseteq S_j \setminus S_i$ (from the items 2 and 3),
5. $|S_j \setminus S_i| = \ell_x - \ell_{x-1}$ (previous discussion),
6. $|ST[\ell_x]| \leq \ell_x - \ell_{x-1}$ (from the items 4 and 5).

$\square_{Lemma\ 1}$

Considering again Table 1, let us assume that the processes p_1, p_3 and p_{10} have crashed while they are at level $\ell = 8$, and before determining their sets S_1, S_3 and S_{10}. The level $\ell = 8$ is now empty (as no process stops at that level), and the levels 10 and 5 are now consecutive non-empty levels. We have then $ST[10] = \{p_5, p_9\}$, $ST[5] = \{p_2, p_8\}$, and $|ST[10]| = 2 \leq \min(k, 10 - 5)$.

3.3 An adaptive renaming protocol

The adaptive renaming algorithm is described in Figure 2. When a process p_i wants to acquire a new name, it invokes new_name(id_i). It then obtains a new name when it executes line 05. Remind that p denotes the number of processes that participate in the algorithm.

Base objects The algorithm uses a k-participating set object denoted PS, and a size n array of adaptive renaming objects, denoted $RN[1..n]$.

Each base renaming object $RN[y]$ can be accessed by at most k processes. It provides them with an operation denoted rename(). When accessed by $h \leq k$ processes, it allows them to acquire new names within the renaming space $[1..2h - 1]$. Interestingly, such adaptive wait-free renaming objects can be built from atomic registers, e.g., [2, 4, 9] (for completeness, one of them is described in appendix A). As noticed in the introduction, this feature provides the proposed algorithm with a modularity dimension as $RN[y]$ and $RN[y']$ can be implemented differently.

The algorithm: principles and description The algorithm is based on the following (well-known) principle.

- Part 1. Divide for conquer.
  A process p_i first invokes $PS.\mathsf{participating_set}_k(id_i)$ to obtain a set S_i satisfying the self-membership, comparability, immediacy and bounded simultaneity properties (line 01). It follows from these properties that (1) at most k processes obtain the same set S (and consequently belong to the same partition), and (2) there are at most p distinct partitions.

  An easy and unambiguous way to identify the partition p_i belongs to is to consider the level at which p_i stopped in the k-participating set object, namely, the level $\ell = |S_i|$. The $h \leq k$ processes in the partition $\ell = |S_i|$ compete then among themselves to acquire a new name. This is done by p_i invoking the appropriate renaming object, i.e., $RN\big[|S_i|\big].\mathsf{rename}(id_i)$ (line 03). As indicated before, these processes obtain new names in renaming space $[1..2h-1]$.

```
operation new_name(id_i):
(01)    S_i ← PS.participating_set_k(id_i);
(02)    base_i ← (2 × |S_i| − ⌈|S_i|/k⌉);
(03)    offset_i ← RN[|S_i|].rename(id_i);
(04)    myname_i ← base_i − offset_i + 1;
(05)    return(myname_i)
```

Figure 2: Generic adaptive renaming algorithm (code for p_i)

- Part 2. Piece together the results of the subproblems.
  The final name assignment is done according to very classical *(base,offset)* rule. A base is attributed to each partition as follows. The partition $\ell = |S_i|$ is attributed the base $2 \times |S_i| - \lceil \frac{|S_i|}{k} \rceil$ (line 02). Let us notice that no two partitions are attributed the same base. Then, a process p_i in partition ℓ considers the new name obtained from $RN[\ell]$ as an offset (notice that an offset in never equal to 0). It determines its final new name from the base and offset values it has been provided with, considering the name space starting from the base and going down (line 04).

3.4 Proof of the algorithm

Lemma 2 *The algorithm described in Figure 2 ensures that no two processes obtain the same new name.*

Proof Let p_i be a process such that $|S_i| = \ell_x$. That process is one of the $|ST[\ell_x]|$ processes that stop at the level ℓ_x and consequently use the underlying renaming object $RN[\ell_x]$. Due to the property of that renaming object, p_i computes a value $offset_i$ such that $1 \leq offset_i \leq 2 \times |ST[\ell_x]| - 1$. Moreover, as $|ST[\ell_x]| \leq \min(k, \ell_x - \ell_{x-1})$ (Lemma 1), the previous relation becomes $1 \leq offset_i \leq 2 \times \min(k, \ell_x - \ell_{x-1})$.

On another side, the renaming space attributed to the processes p_i of $ST[\ell_x]$ starts at the base $2\ell_x - \lceil \frac{\ell_x}{k} \rceil$ (included) and goes down until $2\ell_{x-1} - \lceil \frac{\ell_{x-1}}{k} \rceil$ (excluded). Hence the size of this renaming space is

$$2(\ell_x - \ell_{x-1}) - \big(\lceil \frac{\ell_x}{k} \rceil - \lceil \frac{\ell_{x-1}}{k} \rceil\big).$$

It follows from these observations that a sufficient condition for preventing conflict in name assignment is to have

$$2 \times \min(k, \ell_x - \ell_{x-1}) - 1 \leq 2(\ell_x - \ell_{x-1}) - \big(\lceil \frac{\ell_x}{k} \rceil - \lceil \frac{\ell_{x-1}}{k} \rceil\big).$$

We prove that the algorithm satisfies the previous relation by considering two cases according to the minimum between k and $\ell_x - \ell_{x-1}$. Let

$$\ell_x = q_x \ k + r_x \text{ with } 0 \leq r_x < k \quad (\text{i.e., } \lceil \frac{r_x}{k} \rceil \in \{0,1\})), \quad \text{and}$$

$$\ell_{x-1} = q_{x-1} \ k + r_{x-1} \text{ with } 0 \leq r_{x-1} < k \quad (\text{i.e., } \lceil \frac{r_{x-1}}{k} \rceil \in \{0,1\}), \quad \text{from which we have}$$

$$\ell_x - \ell_{x-1} = (q_x - q_{x-1}) \ k + (r_x - r_{x-1}).$$

- Case $\ell_x - \ell_{x-1} \leq k$.
  In that case, the relation to prove simplifies and becomes $\lceil \frac{\ell_x}{k} \rceil - \lceil \frac{\ell_{x-1}}{k} \rceil \leq 1$, i.e., $(q_x + \lceil \frac{r_x}{k} \rceil) - (q_{x-1} + \lceil \frac{r_{x-1}}{k} \rceil) \leq 1$, that can be rewritten as $(q_x - q_{x-1}) + (\lceil \frac{r_x}{k} \rceil - \lceil \frac{r_{x-1}}{k} \rceil) \leq 1$.
  Moreover, from $\ell_x - \ell_{x-1} = (q_x - q_{x-1})\ k + (r_x - r_{x-1})$ and $\ell_x - \ell_{x-1} \leq k$, we have $(q_x - q_{x-1})\ k + (r_x - r_{x-1}) \leq k$ from which we can extract two subcases:

  – Case $q_x - q_{x-1} = 1$ and $r_x = r_{x-1}$.
    In that case, it trivially follows from the previous formulas that $(q_x - q_{x-1}) + (\lceil \frac{r_x}{k} \rceil - \lceil \frac{r_{x-1}}{k} \rceil) \leq 1$, which proves the lemma for that case.

  – Case $q_x = q_{x-1}$ and $0 \leq r_x - r_{x-1} \leq k$.
    In that case we have to prove $\lceil \frac{r_x}{k} \rceil - \lceil \frac{r_{x-1}}{k} \rceil \leq 1$. As $\lceil \frac{r_x}{k} \rceil, \lceil \frac{r_{x-1}}{k} \rceil \in \{0, 1\}$, we have $\lceil \frac{r_x}{k} \rceil - \lceil \frac{r_{x-1}}{k} \rceil \leq 1$, which proves the lemma for that case.

- Case $k < \ell_x - \ell_{x-1}$.
  After simple algebraic manipulations, the formula to prove becomes:

  $$(2k-1)(q_x - q_{x-1} - 1) + 2(r_x - r_{x-1}) - \left(\lceil \frac{r_x}{k} \rceil - \lceil \frac{r_{x-1}}{k} \rceil\right) \geq 0.$$

  Moreover, we have now $\ell_x - \ell_{x-1} = (q_x - q_{x-1})\ k + (r_x - r_{x-1}) > k$, from which, as $0 \leq r_x, r_{x-1} < k$, we can conclude $q_x - q_{x-1} \geq 1$. We consider two cases.

  – $q_x - q_{x-1} = 1$.
    The formula to prove becomes $2(r_x - r_{x-1}) \geq \lceil \frac{r_x}{k} \rceil - \lceil \frac{r_{x-1}}{k} \rceil$.
    From $\ell_x - \ell_{x-1} > k$ we have:

    * $r_x > r_{x-1}$, from which (as r_x and r_{x-1} are integers) we conclude $2(r_x - r_{x-1}) \geq 2$.
    * $1 \geq \lceil \frac{r_x}{k} \rceil \geq \lceil \frac{r_{x-1}}{k} \rceil \geq 0$, from which we conclude $\lceil \frac{r_x}{k} \rceil - \lceil \frac{r_{x-1}}{k} \rceil \leq 1$.

    By combining the previous relations we obtain $2 \geq 1$ which proves the lemma for that case.

  – $q_x - q_{x-1} > 1$. Let $q_x - q_{x-1} = 1 + \alpha$ (where α is an integer ≥ 1).
    The formula to prove becomes

    $$(2k-1)\alpha + 2(r_x - r_{x-1}) - \left(\lceil \frac{r_x}{k} \rceil - \lceil \frac{r_{x-1}}{k} \rceil\right) \geq 0.$$

    As $0 \leq r_x, r_{x-1} < k$, the smallest value of $r_x - r_{x-1}$ is $-(k-1)$. Similarly, the greatest value of $\lceil \frac{r_x}{k} \rceil - \lceil \frac{r_{x-1}}{k} \rceil$ is 1.
    It follows that, the smallest value of the left side of the formula is $(2k-1)\alpha - 2(k-1) - 1 = 2k\alpha - (2k + \alpha) + 1 = (2k-1)(\alpha - 1)$. As $k \geq 1$ and $\alpha \geq 1$, it follows that the left side is never negative, which proves the lemma for that case.

$\square_{Lemma\ 2}$

Theorem 1 *The algorithm described in Figure 2 is a wait-free adaptive* $(2p - \lceil \frac{p}{k} \rceil)$*-renaming algorithm (where* $p \leq n$ *is the number of processes that participate in the algorithm).*

Proof The fact that the algorithm is wait-free is an immediate consequence of the fact that base k-set participating set object and the base renaming objects are wait-free. The fact that no two processes obtain the same new name is established in Lemma 2.

If p processes participate in the algorithm, the highest level at which a process stops is p (this follows from the properties of the k-set participating set object). Consequently, the largest base that is used (line 02) is $2p - \lceil \frac{p}{k} \rceil$, which establishes the upper bound on the renaming space. $\square_{Theorem\ 1}$

4 Visiting Gafni's land: From k-set to k-participating set

This section presents a wait-free transformation from a k-set agreement object to a k-participating set object. It can be seen as a guided visit to Gafni's reduction land [9, 10, 15, 16].

Let us remind that a k-set object provides each process with an operation $\mathsf{kset_propose}_k()$ that allows it to propose and decide a value in such a way that at most k different values are decided and any decided value is a value that has been proposed by a process. The construction proceeds in two steps: first from k-set agreement to strong k-set agreement, and then from strong k-set agreement to k-participating set.

4.1 From set agreement to strong set agreement

Let us observe that, given a k-set object, it is possible that no process decides the value it has proposed. This feature is the "added value" provided by a *strong k-set agreement* object: it is a k-set object such that at least one process decides the value it has proposed [10]. The corresponding operation is denoted $\mathsf{strong_kset_propose}_k()$.

In addition to a k-set object KS, the processes cooperate by accessing an array $DEC[1..n]$ of one-writer/multi-reader atomic registers. That array is initialized to $[\bot, \ldots, \bot]$. $DEC[i]$ can be written only by p_i. The array is provided with a $\mathsf{snapshot}()$ operation. Such an operation returns a value of the whole array as if that value has been obtained by atomically reading the whole array [1]. Let us remind that such an operation can be wait-free implemented on top of atomic read/write base registers (the best snapshot algorithm known so far costs $O(n \log n)$ atomic register accesses [6]). This means that the base write operations (on each array entry) and the snapshot operations are linearizable [18].

```
operation strong_kset_propose_k(id_i) :
(01)   DEC[i] ← KS.kset_propose_k(id_i);
(02)   dec_i[1..n] ← snapshot(DEC[1..n]);
(03)   if (∃h : dec_i[h] = id_i) then decision_i ← id_i else decision_i ← dec_i[i] end_if;
(04)   return(decision_i)
```

Figure 3: Strong k-set agreement algorithm (code for p_i)

The construction (introduced in [10]) is described in Figure 3. A process p_i first proposes its original name to the underlying k-set object KS, and writes the value it obtains (an original name) into $DEC[i]$ (line 01). Then, p_i atomically reads the whole array (line 02). Finally, if it observes that some process has decided its original name id_i, p_i also decides id_i, otherwise p_i decides the original name it has been provided with by the k-set object (lines 03-04).

Theorem 2 [10] *The algorithm described in Figure 3 wait-free implements a strong k-set agreement object.*

Proof Let us first observe that it trivially follows from the algorithm text that no process returns a name that has not been decided by the k-set object KS. So, only names of participating processes are decided. It follows that the values decided from the strong k-set object SKS satisfy the k-set agreement properties.

If a process p_i, whose original name is one of the names decided by the k-set object, crashes before returning at line 04, it is always possible to consider that p_i would have returned its name at line 04 and crashed just after, which proves the theorem. So, let us consider that none of the processes, whose original name has been decided by the k-set object KS, crashes. If one of these processes p_i is such that the predicate $(\exists h : dec_i[h] = id_i)$ is true when p_i evaluates it, the theorem follows.

So, let us suppose that no process p_i (whose original name is decided by the k-set object) crashes or finds the predicate $(\exists h : dec_i[h] = id_i)$ satisfied when it evaluates it. There is consequently a cycle $j_1, j_2, \ldots, j_x, j_1$ on a subset of these processes defined as follows: $id_{j_2} = DEC[j_1]$, $id_{j_3} = DEC[j_2], \ldots,$ $id_{j_1} = DEC[j_x]$. Among the processes of this cycle, let us consider the process p_{j_α} that is the last to update its entry $DEC[j_\alpha]$, thereby creating the cycle. (Let us observe that, as the write and snapshot operations that access the array DEC are linearizable, such a "last" process p_{j_α} does exist.) But then, when p_{j_α} executes line 03, the predicate $(\exists h : dec_{j_\alpha}[h] = id_{j_\alpha})$ is necessarily true (as p_{j_α} completes the cycle and -due to the snapshot

operation- sees that cycle). It follows that p_{j_α} decides its own original name at line 03-04, which proves the theorem. $\Box_{Theorem\ 2}$

4.2 From strong set agreement to k-participating set

The specification of the k-participating set object has been defined in Section 3.2. The present section shows how it can be wait-free implemented from a strong k-set agreement object SKS. The construction generalizes the construction proposed in [15, 16] that considers $n = 3$ and $k = 2$. In addition to the SKS object, the construction uses an array of one-writer/multi-reader atomic registers denoted $LEVEL[1..n]$. As before only p_i can write $LEVEL[i]$. The array is initialized to $[n+1, \ldots, n+1]$.

The algorithm is based on what we call *Borowski-Gafni's ladder*, a wait-free object introduced in [9]. It combines such a ladder object with a k-set agreement object in order to guarantee that no more than k processes, that do not crash, stop at the same step of the ladder.

Borowsky-Gafni's ladder Let us consider the array $LEVEL[1..n]$ as a ladder. Initially, a process is at the top of the ladder, namely, at level $n+1$. Then it descends the ladder, one step after the other, according to predefined rules until it stops at some level (or crashes). While descending the ladder, a process p_i registers its current position in the ladder in the atomic register $LEVEL[i]$.

After it has stepped down from one ladder level to the next one, a process p_i computes a local view (denoted $view_i$) of the progress of the other processes in their descent of the ladder. That view contains the processes p_j seen by p_i at the same or a lower ladder level (i.e., such that $level_i[j] \leq LEVEL[i]$). Then, if the current level ℓ of p_i is such that p_i sees at least ℓ processes in its view (i.e., processes that are at its level or a lower level) it stops at the level ℓ of the ladder. This behavior is described by the following algorithm [9]:

```
repeat LEVEL[i] ← LEVEL[i] − 1;
          for j ∈ {1, ... , n} do level_i[j] ← LEVEL[j] end_do;
          view_i ← {j : level_i[j] ≤ LEVEL[i]};
until (|view_i| ≥ LEVEL[i]) end_repeat;
let S_i = view_i; return(S_i).
```

This very elegant algorithm satisfies the following properties [9]. The sets S_i of the processes that terminate the algorithm, satisfy the self-membership, comparability and immediacy properties of the k-participating set object. Moreover, if $|S_i| = \ell$, then p_i stopped at the level ℓ, and there are ℓ processes whose current level is $\leq \ell$.

From a ladder to a k-participating set object The construction, described in Figure 4, is nearly the same as the construction given in [15, 16]. It uses the previous ladder algorithm as a skeleton to implement a k-participating set object. When it invokes participating_set$_k$(), a process p_i provides its original name as input parameter. This name will be used by the underlying strong k-participating set object. The array $INIT_NAME[1..n]$ is initialized to $[\bot, \ldots, \bot]$. $INIT_NAME[i]$ can be written only by p_i.

```
operation participating_set_k(id_i)
(01)  INIT_NAME[i] ← id_i;
(02)  repeat LEVEL[i] ← LEVEL[i] − 1;
(03)     for j ∈ {1, ... , n} do level_i[j] ← LEVEL[j] end_do;
(04)     view_i ← {j : level_i[j] ≤ LEVEL[i]};
(05)     if (LEVEL[i] > k) ∧ (|view_i| = LEVEL[i])
(06)                        then ans_i ← SKS.strong_kset_propose_k(id_i);
(07)                             ok_i ← (ans_i = id_i)
(08)                        else ok_i ← true
(09)     end_if
(10)  until (|view_i| ≥ LEVEL[i]) ∧ ok_i end_repeat;
(11)  let S_i = {id | ∃j ∈ view_i such that INIT_NAME[j] = id};
(12)  return(S_i)
```

Figure 4: k-participating set algorithm (code for p_i)

If, in the original Borowski-Gafni's ladder, a process p_i stops at a ladder level $\ell \leq k$, it can also stop at the same level in the k-set participating object. This follows from the fact that, as $|view_i| = \ell \leq k$ when p_i stops descending, we know from the ladder properties that at most $\ell \leq k$ processes are at the level ℓ (or at a lower level). So, when $LEVEL[i] \leq k$ (line 05), p_i sets ok_i to *true* (line 05). It consequently exits the repeat loop (line 10) and we can affirm that no more than k processes do the same, thereby satisfying the bounded simultaneity property.

So, the main issue of the algorithm is to satisfy the bounded simultaneity property when the level at which p_i should stop in the original Borowski-Gafni's ladder is higher than k. In that case, p_i uses the underlying strong k-set agreement object *SKS* to know if it can stop at that level (lines 06-07). The k-participating set object ensures that at least one (and at most k) among the participating processes that should stop at that level in the original Borowski-Gafni's ladder, do actually stop. If a process p_i is not allowed to stop (we have then $ok_i = false$ at line 07), it is required to descend to the next step of the ladder (lines 10 and 01). When a process stops at a level ℓ, there are exactly ℓ processes at the levels $\ell' \leq \ell$. This property is maintained when a process steps down from ℓ to $\ell - 1$ (this follows from the fact that when a process is required to step down from $\ell > k$ to $\ell - 1$ because $\ell > k$, at least one process remains at the level ℓ due to the k-set agreement object *SKS*).

5 From Ω^k to k-set

This section shows that a k-set object can be built in a single-writer/multi-reader atomic register system, equipped with an oracle (failure detector) of the class Ω^k.

5.1 The oracle class Ω^k

The family of oracle classes $(\Omega^z)_{1 \leq z \leq n}$ has been introduced in [22]. An oracle of the class Ω^z provides the processes with an operation denoted leader() that, each time it is invoked, provides the invoking process with a set of at most z process identities (e.g., $\{id_{x_1}, \ldots, id_{x_z}\}$). That operation satisfies the following property:

- Eventual multiple leadership: There is a time after which all the leader() invocations return forever the same set. Moreover, this set includes at least one correct participating process (if any).

Ω^1 is nothing else than the leader failure detector denoted Ω introduced in [12], where it is shown that it is the weakest failure detector for solving the consensus problem in asynchronous systems where all the correct processes are assumed to participate. It is important to notice that during an arbitrary long period, the processes can see different sets of leaders. Moreover, no process knows when this "anarchy" period is over. It is also possible that some of the processes that are eventually elected as permanent leaders, are faulty.

5.2 From Ω^k to k-set agreement

In addition to an oracle of the class Ω^k, the proposed k-set agreement algorithm is based on a variant, denoted *KA*, of a round-based object introduced in [17] to capture the safety properties of Paxos-like consensus algorithms [14, 20]. The leader oracle is used to ensure the liveness of the algorithm. *KA* is used to abstract away the safety properties of the k-set problem, namely, at most k values are decided, and the decided values are have been proposed.

The *KA* object This object provides the processes with an operation denoted $\mathsf{alpha_propose}_k(v_i)$. That operation has two input parameters: the value v_i proposed by the invoking process p_i (here its name id_i), and a round number (that allows identifying the invocations). The *KA* object assumes that no two processes use the same round numbers, and successive invocations by the same process use increasing round numbers. Given a *KA* object, the invocations $\mathsf{alpha_propose}_k()$ satisfy the following properties:

- Validity: the value returned by any invocation $\mathsf{alpha_propose}_k()$ is a proposed value or $\perp$.

- Agreement: At most k different non-$\perp$ values can be returned by the whole set of $\mathsf{alpha_propose}_k()$ invocations.

- Convergence: If there is a time after which the operation $\mathsf{alpha_propose}_k()$ is invoked infinitely often, and these invocations are issued by an (unknown but fixed) set of at most k processes, then there is a time after which none of these invocations returns $\perp$.

The k-set algorithm The algorithm constructing a k-set object KS is described in Figure 5. As in previous algorithms, it uses an array $DEC[1..n]$ of one-writer/multi-reader atomic registers. Only p_i can write $DEC[i]$. The array is initialized to $[\perp, \dots, \perp]$. The algorithm is very simple. If a value has already been decided ($\exists j : DEC[j] \neq \perp$), p_i decides it. Otherwise, p_i looks if it is a leader. If it is not, it loops. If it is a leader ($id_i \in \mathsf{leader}()$), p_i invokes $\mathsf{alpha_propose}_k(r_i, v_i)$ and writes in $DEC[i]$ the value it obtains (it follows from the specification of KA that that value it writes is $\perp$ or a proposed value).

```
operation kset_propose_k(v_i):
(01)   r_i ← (i − n);
(02)   while (∀j : DEC[j] = ⊥) do
(03)       if (id_i ∈ leader()) then r_i ← r_i + n; DEC[i] ← KA.alpha_propose_k(r_i, v_i) end_if
(04)   end_while;
(05)   let decided_i = any DEC[j] ≠ ⊥;
(06)   return(decided_i)
```

Figure 5: An Ω^k-based k-set algorithm (code for p_i)

It is easy to see that no two processes use the same round numbers, and each process uses increasing round numbers. It follows directly from the agreement property of the KA object, that at any time, the array $DEC[1..n]$ contains at most k values different from $\perp$. Moreover, due the validity property of KA, these values have been proposed.

It is easy to see that, as soon as a process has written a non-$\perp$ value in $DEC[1..n]$, any $\mathsf{kset_propose}(v_i)$ invocation issued by a correct process terminates. So, in order to show that the algorithm is wait-free, we have to show that at least one process writes a non-$\perp$ value in $DEC[1..n]$. Let us assume that no process deposits a value in this array. Due to the eventual multiple leadership property of Ω^k, there is a time τ after which the same set of $k' \leq k$ participating processes are elected as permanent leaders, and this set includes at least one correct process. It follows from the algorithm that, after τ, at most k processes invoke $KA.\mathsf{alpha_propose}_k()$, and one of them is correct. It follows from the convergence property of the KA object, that there is a time $\tau' \geq \tau$ after which no invocation returns $\perp$. Moreover, as at least one correct process belongs to the set of elected processes, that process eventually obtains a non-$\perp$ value from an invocation, and consequently deposits that non-$\perp$ value in $DEC[1..n]$. The algorithm is consequently wait-free.

5.3 Implementing KA

An algorithm constructing a KA object is described in Figure 6. It uses an array of single-writer/multi-reader atomic registers $REG[1..n]$. As previously, $REG[i]$ can be written only by p_i. A register $REG[i]$ is made up of three fields $REG[i].lre$, $REG[i].lrww$ and $REG[i].val$ whose meaning is the following ($REG[i]$ is initialized to $< 0, 0, \perp >$):

- $REG[i].lre$ stores the number of the *l*ast *r*ound *e*ntered by p_i. It can be seen as the logical date of the last invocation issued by p_i.

- $REG[i].lrww$ and $REG[i].val$ constitute a pair of related values: $REG[i].lrww$ stores the number of the *l*ast *r*ound *w*ith a *w*rite of a value in the field $REG[i].val$. So, $REG[i].lrww$ is the logical date of the last write in $REG[i].val$.

(To simplify the writing of the algorithm, we consider that each field of a register can be written separately. This poses no problem as each register is single writer. A writer can consequently keep a copy of the last value it has written in each register field and rewrite it when that value is not modified.)

```
operation alpha_propose_k(r, v):
(01)    REG[i].lre ← r;
(02)    for j ∈ {1, ..., n} do reg_i[j] ← REG[j] end_do;
(03)    let value_i be reg_i[j].val where j is such that ∀x : reg_i[j].lrww ≥ reg_i[x].lrww;
(04)    if (value_i = ⊥) then value_i ← v end_if;
(05)    REG[i].(lrww, v) ← (r, value_i);
(06)    for j ∈ {1, ..., n} do reg_i[j] ← REG[j] end_do;
(07)    if (|{j|reg_i[j].lre ≥ r}| > k) then return(⊥)
(08)                                      else return(value_i) end_if
```

Figure 6: A *KA* object algorithm (code for p_i)

The principle that underlies the algorithm is very simple: it consists in using a logical time frame (represented here by the round numbers) to timestamp the invocations, and answering $\perp$ when the timestamp of the corresponding invocation does not lie within the k highest dates (registered in $REG[1..n].lre$). To that end, the algorithm proceeds as follows:

- Step 1 (lines 01-02): Access the shared registers.
  - When a process p_i invokes $\mathsf{alpha_propose}_k(r, v)$, it first informs the other processes that the *KA* object has attained (at least) the date r (line 01). Then p_i reads all the registers in any order (line 02) to know the last values (if any) written by the other processes.

- Step 2 (lines 03-05): Determination and writing of a value.
  Then, p_i determines a value. In order not to violate the agreement property, it selects the last value ("last" according to the round numbers/logical dates) that has been deposited in a register $REG[j]$. If there is no such value it considers its own value v. After this determination, p_i writes in $REG[i]$ the value it has determined, together with its round number (line 05.

- Step 3 (lines 06-08): Commit or abort.
  - p_i reads again the shared registers to know the progress of the other processes (measured by their round numbers), line 07. If it discovers it is "late", p_i aborts returning $\perp$. (Let us observe that this preserves the agreement property.) "To be late" means that the current date r of p_i does not lie within the window defined by the k highest dates (round numbers) currently entered by the processes (these round numbers/dates are registered in the field lre of each entry of the array $REG[1..n]$).
  - Otherwise, p_i is not late. It then returns ("commits") $value_i$ (line 08). Let us observe that, as the notion of "being late" is defined with respect to a window of k dates (round numbers), it is possible that up to k processes are not late and return concurrently up to k distinct non-$\perp$ values.

It directly follows from the code that the algorithm is wait-free. Moreover, in order to expedite the $\mathsf{alpha_propose}_k()$ operation, it is possible to insert the statement

$$\textbf{if } \big(\big|\{j|reg_i[j].lre \geq r\}\big| > k\big) \textbf{ then } \mathsf{return}(\perp) \textbf{ end_if}$$

between the line 02 and the line 03. This allows the invoking process to return $\perp$ when, just after entering the $\mathsf{alpha_propose}_k()$ operation, it discovers it is late.

5.4 Proof of the *KA* object

Theorem 3 *The algorithm described in Figure 6 wait-free implements a KA object.*

Proof The wait-free property follows directly from the code of the algorithm.

Validity Let us observe that if a value v is written in $REG[i].val$, that value has been previously passed as a parameter in an $\mathsf{alpha_propose}_k()$ invocation. The validity property follows from this observation and the fact that only $\perp$ or a value written in a register $REG[i]$ can be returned from an $\mathsf{alpha_propose}_k()$ invocation.

Convergence Let τ be a time after which there is a set of $k' \leq k$ processes such that each of them invokes $\mathsf{alpha_propose}_k()$ infinitely often. This means that, from τ, the values of $n-k'$ registers $REG[x]$ are no longer modified. Consequently, as the k' processes p_j repeatedly invoke $\mathsf{alpha_propose}_k()$, there is a time $\tau' \geq \tau$ after which each $REG[j].lre$ becomes greater than any $REG[x].lre$ that is no longer modified. There is consequently a time $\tau'' \geq \tau'$ after which the k' processes are such that their registers $REG[j].lre$ contain forever the k greatest timestamp values. It follows from the test done at line 07 that, after τ'', no $\mathsf{alpha_propose}_k()$ invocation by one of these k' processes can be aborted. Consequently, each of them returns a non-$\perp$ value at line 08.

Agreement If all invocations returns $\perp$, the agreement property is trivially satisfied. So, let us consider an execution in which at least one $\mathsf{alpha_propose}_k()$ invocation returns a non-$\perp$ value. To prove the agreement property we show that:

- Before the first non-$\perp$ value is returned by an invocation, there is a time at which the algorithm has determined a set V of at least one and at most k non-$\perp$ values[2].
- Any value $v \neq \perp$ returned by an invocation is a value of V.

To simplify the reasoning, and without loss of generality, we assume that a process that repeatedly invokes $\mathsf{alpha_propose}_k()$, stops invoking that operation as soon as it returns a non-$\perp$ value at line 08.

1. Invariants. $\forall j \in \{1, \ldots, n\}$:
   - $REG[j].lre$ is increasing (assumption on the successive round numbers used by p_j).
   - $REG[j].lrww \leq REG[j].lre$ (because p_j executes line 05 after line 01).
2. Among all the invocations that execute the test of line 07, let $\mathcal{I}$ be the subset of invocations for which the predicate $\big|\{j | reg_i[j].lre \geq r\}\big| \leq k$ is true. (This means that any invocation of $\mathcal{I}$ either returns a non-$\perp$ value -at line 08-, or crashes after it has evaluated the predicate at line 07, and before it executes line 08.) Among the invocations of $\mathcal{I}$, let I be the invocation with the smallest round number. Let p_{j_1} be the process that invoked I and r the corresponding round number.
3. Time instants (see Figure 7).
   - Let τ be the time at which I executes line 05 (statement $REG[j_1] \leftarrow < r, r, v >$).
   - Let τ' be the time just after I has finished reading the array $REG[1..n]$. Without loss of generality, we consider that this is the time at which I locally evaluates the predicate of line 07.
   - Let $\tau[j]$ be the time at which I reads $REG[j]$ at line 06. We have $\tau < \tau[j] < \tau'$.

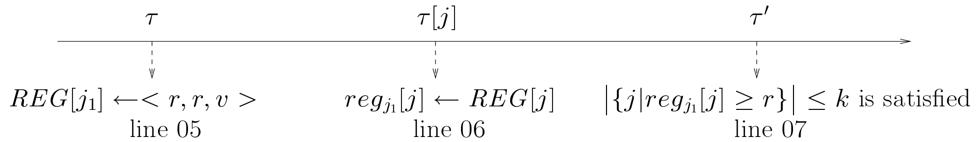


Figure 7: Time instants with respect to accesses to the registers $REG[1..n]$

4. From $\tau[j] < \tau'$, the fact that predicate $\big|\{j | reg_{j_1}[j].lre \geq r\}\big| \leq k$ is true at τ', and the monotonicity of $REG[j].lre$, we can conclude that a necessary requirement for the predicate $REG[j].lre \geq r$ to be true at τ is that it is true at τ'.

   Let $L = \{j_1, \ldots, j_x, \ldots, j_\ell\}$ be the set of processes p_j such that $REG[j].lre \geq r$ is true at τ. As the predicate $\big|\{j | reg_i[j].lre \geq r\}\big| \leq k$ is true at τ', we have $1 \leq \ell = |L| \leq k$.

[2] According to the terminology introduced in [11], the set V defines the values that are *locked*. This means that from now on the set of non-$\perp$ values that can be returned is fixed forever: no value outside V can ever be returned.

5. From the previous item, we conclude that there are at least $n - \ell \geq n - k$ entries j of the array $REG[1..n]$ such that $REG[j].lre < r$ at time τ. Let $\overline{L}$ denote this set of processes (L and $\overline{L}$ define a partition of the whole set of processes).

6. Let the τ-*time* invocation of p_j be the invocation issued by p_j whose round number is the value of $REG[j].lre$ at time τ (assuming a fictitious initial invocation if needed).

7. The τ-time invocations of the processes p_j in L define a set, denoted V, including at most $\ell \leq k$ values, such that these values are written in $REG[1..n]$ with a write timestamp (value of the field $REG[j].lrww$) that is $\geq r$. This claim follows from the following observation.

   - The τ-time invocation by p_{j_1} (namely I) writes a value and the round number r in $REG[j_1]$.
   - Let $p_{j_x} \in L$, $p_{j_x} \neq p_{j_1}$. From the definition of L, it follows that the round number of the τ-time invocation issued by p_{j_x} is $REG[j_x].lre = r' > r$. When it executes that invocation, p_{j_x} atomically executes $REG[j_x] \leftarrow < r', r', v' >$ (if it does not crash before executing the line 05).
   - It is possible that, on one side, no process in L crashes before executing line 05, and, on another side, all the values that are written are different. It consequently follows that up to $\ell \leq k$ different values (with a write timestamp $lrww \geq r$) can be written in $REG[1..n]$. Hence, V can contain up to k values.
   - Moreover, it is also possible that each process in L returns at line 08 the value it has selected at line 05 (this depends on the value of the predicate evaluated at line 07). Consequently each value of V can potentially be returned.

8. Given an execution, the previous item has extracted a non-empty set V of at most k non-$\perp$ values that can be returned. We now show that (1) from τ, only values of V can be written in $REG[1..n]$ with a timestamp field ($lrww$) greater than r, and (2) a non-$\perp$ value returned by an invocation is necessarily a value of V.

   (a) The τ-time invocation issued by a process $p_j \in \overline{L}$ has a round number $REG[j].lre$ that is smaller than $REG[j_1].lre = r$ (this follows from the definition of $\overline{L}$). As by definition, r is the smallest round number during which a process finds true the predicate of line 07, it follows that any process in $\overline{L}$ needs to issue an invocation with a round number greater than r to have a chance to return a non-$\perp$ value.

   (b) Let $\mathcal{I}'$ be the set of all the invocations that have a round number greater than r. They are issued by the processes of $\overline{L}$ or the processes of L whose τ-time invocation has returned $\perp$ at line 07. Let us observe that any invocation of $\mathcal{I}'$ starts after τ.

   Let I' be the first invocation of $\mathcal{I}'$ that executes 05. I' (issued by some process p_j) selects (at line 03) a value $value_j$ from a register $REG[y]$ such that $REG[y].lrww \geq REG[j_1].lrww = r$. As up to now, only processes of L have written values in $REG[1..n]$ with a write timestamp ($lrww$) $\geq r$, it follows that I' selects a value from V [3]. Consequently, this invocation does not add a new value to V.

   Let I'' be the invocation of $\mathcal{I}'$ that is the second to execute line 05. The same reasoning (including now I') applies. Etc. It follows from this induction that a value written at line 05 by an invocation of $\mathcal{I}'$ is a value of V, which proves that only values of V can be written in the array $REG[1..n]$ with a write timestamp greater than r.

   (c) Finally, an invocation that returns a value at line 08, returns the value it has written at line 05. Due to the definition of r, its round number r' is $\geq r$. It follows that the non-$\perp$ value that is returned is a value of V.

$\square_{Theorem\ 3}$

[3] It is possible that, when I' reads the array $REG[1..n]$ at line 02, not all the values of V have yet been written in that array. The important points are here that (1) at least one value of V has already been written in the array (namely, $REG[j_1].val$ with the timestamp $REG[j_1].lrww = r$), and (2) any register $REG[x]$ that currently contains a value not in V, is such that $REG[x].lrww < r$.

6 Concluding remarks

What was the paper on This paper has presented a wait-free adaptive renaming algorithm whose renaming space is $M = (2p - \lceil \frac{p}{k} \rceil)$, where p is the number of participating processes. This algorithm relies on an underlying k-set agreement object. It has also been shown how such an object can be built from atomic read/write registers and a leader oracle of the class Ω^k. The construction is based on the reduction style advocated by Gafni [13]. It uses several intermediate objects introduced in [9, 10, 15, 16].

To our knowledge, the proposed construction is the first that uses the (possibly unreliable) information on failures provided by an oracle (failure detector) to circumvent the $2p - 1$ lower bound on the adaptive renaming space. In that sense the paper establishes a connection between Gafni's reductions and failure detectors.

Open problems If $k > t$, there are trivial algorithms for implementing a k-set object in an asynchronous read/write register systems. So, let us assume $k \leq t$. Instead of looking for a wait-free renaming algorithm, we could be interested in a t-resilient adaptive algorithm, i.e., a renaming algorithm that works when the number of crashes does not bypass the model parameter t (the wait-free case being the extreme case $t = n-1$).

We spent time looking for such an algorithm (without success until now). We nevertheless think that it should be possible to design a t-resilient adaptive M-renaming algorithm from k-set objects, where

$$M = n + (t+1) - \lceil \frac{t+1}{k} \rceil.$$

Let us notice that this formula involves the total number of processes n, the resilience bound t, and the parameter k that measures the additional power in presence of crashes -power provided by Ω^k-). When $k > t$ (i.e., when there is no additional power), we obtain $M = n + t$ (that is the lower bound in asynchronous read/write systems.

Another interesting question concerns the implementation of the $\mathsf{alpha_propose}_k()$ operation from a Borowsky-Gafni's ladder-like object. Is it possible? If the answer is "yes", it would shed a new light on the way the safety properties of à la Paxos shared memory consensus algorithms could be implemented.

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. 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, Atlanta (GA), 1999.

[3] 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.

[4] Attiya H. and Fouren A., Polynomial and Adaptive Long-lived $(2k-1)$-Renaming. *Proc. Symposium on Distributed Computing (DISC'00)*, Springer-Verlag LNCS #1914, pp. 149-163, Toledo (Spain), 2000.

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

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

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

[8] Bernstein P.A. and Goodman N., Concurrency Control in Distributed Data Base Systems. *ACM Computing Survey,* 13(2):185-221, 1981.

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

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

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

[12] Chandra T., Hadzilacos V. and Toueg S., The Weakest Failure Detector for Solving Consensus. *Journal of the ACM*, 43(4):685–722, 1996.

[13] Gafni E., Read/Write Reductions. *DISC/GODEL presentation given as introduction to the 18th Int'l Symposium on Distributed Computing (DISC'04)*, 2004. http://www.cs.ucla.edu/˜ eli/eli/godel.ppt.

[14] Gafni E. and Lamport L., Disk Paxos. *Distributed Computing*, 16(1):1-20, 2003.

[15] Gafni E. and Rajsbaum S., Musical Benches. *Proc. 19th Int'l Symposium on Distributed Computing (DISC'05)*, Springer Verlag LNCS #3724, pp. 63–77, 2005.

[16] Gafni E., Rajsbaum R., Raynal M. and Travers C., The Committee Decision Problem. *Proc. 8th Latin-American Theoretical INformatics Symposium (LATIN'06)*, Springer Verlag LNCS #3887, pp. 502-514, 2006.

[17] Guerraoui R. and Raynal M., The Alpha and Omega of Asynchronous Consensus. *Tech Report #1676*, IRISA, Université de Rennes (France), 2005. http://www.irisa.fr/bibli/publi/pi/2005/1676/1676.html, (Submitted).

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

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

[20] Lamport L., The Part-Time Parliament. *ACM Transactions on Computer Systems*, 16(2):133-169; 1998.

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

[22] Neiger G., Failure Detectors and the Wait-Free Hierarchy. *Proc. 14th Int'l ACM Symposium on Principles of Distributed Computing (PODC'95)*, ACM Press, pp. 100-109, Ottawa (Canada), August 1995.

[23] 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.

A Borowsky-Gafni's adaptive $(2h-1)$-renaming algorithm

This appendix describes an adaptive renaming algorithm that, given h participating processes in a set of H processes ($h \leq H$), provides these processes with a renaming space whose size is $M = 2h-1$. As indicated in the paper, several such algorithms have been proposed (e.g., [2, 4, 9]). We present here the algorithm proposed by Borowski and Gafni [9] as it naturally belongs to Gafni's reduction land.

Data structures The algorithm uses a set of ladder objects as defined in Section 4.2. Each ladder provides an operation denoted participating_set() that satisfies the self-membership, comparability and immediacy properties defined in Section 3.2. As we have seen, these objects can be wait-free implemented in asynchronous read/write atomic register systems.

Each ladder is identified $LADDER[tag]$ where tag specifies a ladder among several ladders. A tag is a sequence of integers, which means that the tags $(0,1)$, $(0,2)$ and $(0,1,3)$ are pointers to three different ladders. More generally, the set of ladders has a tree structure, $LADDER[(0)]$ denoting the root ladder object. The operation $\oplus$ is used to define a new tag from a previous tag (line 04). As an example, the tag $(0,1)\oplus 5$ is the sequence $(0,1,5)$. Moreover, $LADDER[(0,1,5)]$ is then a child of $LADDER[(0,1)]$.

Each process p_i manages three local variables: dir_i, $slot_i$ and tag_i; $dir_i \in \{up, down\}$ (each one being the opposite of the other); $slot_i \in [0..2H-1]$, and tag_i is a sequence of integers that allows accessing a ladder object. Initially, $dir_i = up$, $slot_i = 0$ and $tag_i = (0)$.

```
operation BG_rename(tag_i, slot_i, dir_i, id_i)
(01)  S_i ← LADDER[tag_i].participating_set(id_i);
(02)  if (dir_i = up) then slot_i ← slot_i + (2|S_i| − 1) else slot_i ← slot_i − (2|S_i| − 1) end_if;
(03)  if (id_i  =  max({id|id ∈ S_i})) then return(slot_i)
(04)                                   else  let name_i = BG_rename(tag_i ⊕ |S_i|, slot_i, ¬ dir_i, id_i);
(05)                                         return(name_i)
(06)  end_if
```

Figure 8: Borowsky-Gafni's renaming algorithm (code for p_i)

Algorithm description A process p_i invokes the operation BG_rename($tag_i, slot_i, dir_i, id_i$) described in Figure 8. Starting from the root, p_i recursively descends along the tree of ladder objects until it stops (line 03). When it enters BG_rename($tag_i, slot_i, dir_i, id_i$), p_i first invokes $LADDER[tag_i]$.participating_set(id_i) to obtain a set S_i of participating processes satisfying the self-membership, comparability and immediacy properties. Let us notice that this set can include only processes that have invoked the very same ladder object (identified by tag_i).

Considering the recursive invocations issued by p_i, let S_i^1 be the set obtained by p_i during its first invocation, S_i^2 be the set obtained by its second invocation, etc. A process p_i considers smaller and smaller renaming spaces until it obtains its final name. These renaming spaces are defined at line 02. Thanks to the direction parameter dir_i that takes alternate values, we have the following. Let $L_i^1 = 2|S_i^1| - 1$.

The first renaming space is $rs_i^1 = [1..L_i^1]$ (notice that $L_i^1 \leq 2h-1$, where h is the number of participating processes). Let $L_i^2 = 2|S_i^2| - 1$. The second renaming space used by p_i (if needed) is $rs_i^2 = [L_i^1 - L_i^2..L_i^1]$. Similarly, let $L_i^3 = 2|S_i^3| - 1$. The third renaming space used by p_i is then $rs_i^3 = [L_i^1 - L_i^2..L_i^1 - L_i^2 + L_i^3]$; etc. We have $rs_i^{x+1} \subseteq rs_i^x$. The process p_i stops descending the ladder tree when, during its xth recursive call, it obtains a set S_i^x such that id_i is the greatest identity in that set (line 03). Let us observe that, when we consider a given depth x of the ladder tree, there is at least one process p_c such that $id_c = \max(\{id \mid id \in S_c^x\})$, from which it follows that each process terminates the algorithm after at most h recursive calls. It is easy to see that the final renaming space that the processes can occupy is $[1..2h-1]$.

Let $tag[1]^x, tag[2]^x, \dots, tag[z]^x$ be the set of different tags used at the depth x of the ladder tree. The algorithm ensures the following property (from which follows the fact that no two of the $h \leq H$ processes obtain the same name). If $\alpha \neq \beta$, the renaming spaces obtained by a process p_i and a process p_j that invoke $LADDER[tag[\alpha]^x]$.participating_set(id_i) and $LADDER[tag[\beta]^x]$.participating_set(id_j), respectively, have an empty intersection. For more details on this very elegant wait-free algorithm, the reader can consult [9].