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, and Corentin Travers

IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France
{achour, raynal, ctravers}@irisa.fr

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 a class denoted 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.

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 $M = n + t$.

S. Dolev (Ed.): DISC 2006, LNCS 4167, pp. 1–15, 2006.

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., [2,4,5,6]). *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) [12]. 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 [13] 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., [2,4,6]).

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 [12]), from which it follows that this queue object can be wait-free implemented from atomic registers and consensus objects [12]. 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 [9]. It is inspired by the adaptive renaming algorithm proposed by Borowsky and Gafni [6]. In addition to k-set objects, it also uses simple variants of base objects introduced in [6,7,10], namely, *strong k-set agreement* [7], *k-participating set* [6,10]. These objects can be incrementally built as follows: (1) base k-set objects are used to build k-participating set objects, and then (2) k-participating set objects, are used to solve $(2p - \lceil \frac{p}{k} \rceil)$-renaming.

The renaming algorithm that we obtain 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:

- (1) Using a k-participating set object, the processes are partitioned into independent subsets of size at most k.
- (2) 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 $[1..2h - 1]$.
- (3) 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 $M = (2p - \lceil \frac{p}{k} \rceil)$ 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 [7,13,17]. 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 a family of leader oracles (denoted here $(\Omega^z)_{1\leq z\leq n}$), and presents a k-set algorithm based on read/write registers and any oracle of such a 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 $2p-1$.

Roadmap. The paper is made up of 5 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.

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.

In the following, given a run of an algorithm, p denotes the number of processes that *participate* in that run, $1 \leq p \leq n$. (To participate, a process has to execute at least one operation on a shared object.)

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

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). To rule out this type of ineffective solution, we consider the following requirement for a renaming algorithm [5]:

- 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 defined in [6].

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 [6,10]:

- 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: $\forall \ell : \ 1 \leq \ell \leq n$: $|\{j \ : \ |S_j| = \ell\}| \leq k$.

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.

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

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

Notation and properties. Let S_j be the set returned to p_j after it has invoked participating_set$_k(id_j)$, and $\ell = |S_j|$ (notice that $1 \leq \ell \leq n$). 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, $|\mathcal{L}| = m \leq n$. 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 $1 \leq \ell' \leq \ell$. 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 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 $\ell_x < \ell_y$.
  - $S_i \subset S_j$ (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 next lemma captures an important property provided by a k-participating set object. Let $ST[\ell_x] = \{j$ 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 1. 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., [2,4,6]). 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.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., $RN\big[|S_i|\big]$.rename(id_i) (line 03). As indicated before, these processes obtain new names in renaming space $[1..2h-1]$.

```
operation new_name(id_i):
(1)   S_i ← PS.participating_set_k(id_i);
(2)   base_i ← (2 × |S_i| − ⌈|S_i|/k⌉);
(3)   offset_i ← RN[|S_i|].rename(id_i);
(4)   myname_i ← base_i − offset_i + 1;
(5)   return(myname_i)
```

Fig. 1. 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 1 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

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

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

$$\text{from which we have } \ \ell_x - \ell_{x-1} = (q_x - q_{x-1})\ k + (r_x - r_{x-1}).$$

- 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:

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

  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

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

  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 1 is a wait-free adaptive* $(2p - \lceil \frac{p}{k} \rceil)$*-renaming algorithm (where* $p \leq n$ *is the number of participating processes).*

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 [6,7,10]. Let us recall that a k-set object provides the processes with an operation $\mathsf{kset\_propose}_k()$.

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 (i.e., at most k different values are decided) such that at least one process decides the value it has proposed [7]. 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 construction (introduced in [7]) is described in Figure 2. 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).

operation $\mathsf{strong\_kset\_propose}_k(id_i)$:
(1) $DEC[i] \leftarrow KS.\mathsf{kset\_propose}_k(id_i)$;
(2) $dec_i[1..n] \leftarrow \mathsf{snapshot}(DEC[1..n])$;
(3) **if** $(\exists h : dec_i[h] = id_i)$ **then** $decision_i \leftarrow id_i$ **else** $decision_i \leftarrow dec_i[i]$ **endif**;
(4) $\mathsf{return}(decision_i)$

Fig. 2. Strong k-set agreement algorithm (code for p_i)

4.2 From Strong Set Agreement to k-Participating Set

The specification of a k-participating set object has been defined in Section 3.2. The present section shows how such an object PS can be wait-free implemented from an array of strong k-set agreement objects; this array is denoted $SKS[1..n]$. (This construction generalizes the construction proposed in [10] that considers $n = 3$ and $k = 2$.) In addition to the array $SKS[1..N]$ of strong k-set agreement objects, 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 [6]. 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 [6]:

repeat $LEVEL[i] \leftarrow LEVEL[i] - 1$;
  for $j \in \{1, \ldots, n\}$ **do** $level_i[j] \leftarrow LEVEL[j]$ **end_do**;
  $view_i \leftarrow \{j : level_i[j] \leq LEVEL[i]\}$;
until $(|view_i| \geq LEVEL[i])$ **end_repeat**;
let $S_i = view_i$; $\mathsf{return}(S_i)$.

This very elegant algorithm satisfies the following properties [6]. 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 3, is nearly the same as the construction given in [10]. 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)
(1)   INIT_NAME[i] ← id_i;
(2)   repeat LEVEL[i] ← LEVEL[i] − 1;
(3)     for j ∈ {1, ..., n} do level_i[j] ← LEVEL[j] end_do;
(4)     view_i ← {j : level_i[j] ≤ LEVEL[i]};
(5)     if (LEVEL[i] > k) ∧ (|view_i| = LEVEL[i])
(6)                   then let ℓ = LEVEL[i];
(7)                        ans_i ← SKS[ℓ].strong_kset_propose_k(id_i);
(8)                        ok_i  ← (ans_i = id_i)
(9)                   else ok_i  ← true
(10)    endif
(11)  until (|view_i| ≥ LEVEL[i]) ∧ ok_i end_repeat;
(12)  let S_i = {id | ∃j ∈ view_i such that INIT_NAME[j] = id};
(13)  return(S_i)
```

Fig. 3. 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 11) 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 ℓ 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 $SKS[\ell]$ to know if it can stop at that level (lines 07-08). This k-participating set object ensures that at least one (and at most k) among the participating processes that should stop at level ℓ 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 08), it is required to descend to the next step of the

ladder (lines 11 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 $SKS[\ell]$).

5 From Ω^k to k-Set Objects

This section shows that a k-set object can be built from single-writer/multi-reader atomic registers and 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 [16]. That definition implicitly assumes that all the processes are participating. We extend here this definition by making explicit the notion of *participating* processes. More precisely, an oracle of the class Ω^z provides the processes with an operation denoted leader() that satisfies the following properties:

- Output size: each time it is invoked, leader() provides the invoking process with a set of at most z *participating* process identities (e.g., $\{id_{x_1}, \ldots, id_{x_z}\}$).
- 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).

It is important to notice that each instance of Ω^k is defined with respect to the *context* where it is used. This context is the set of participating processes. This means that if Ω^k is used to construct a given object, say a k-set object KS, the participating processes for that failure detector instance are the processes that invoke $KS.\mathsf{kset\_propose}_k()$. Let us remark that, during an arbitrary long period, the participating processes that invoke leader() can see different sets of leaders, and no process knows when this "anarchy" period is over. Moreover, nothing prevent faulty processes to be elected as permanent leaders.

When all the processes are assumed to participate, Ω^1 is nothing else than the leader failure detector denoted Ω introduced in [8], where it is shown that it is the weakest failure detector for solving the consensus problem in asynchronous systems. (Let us notice that the lower bound proved in [8], on the power of failure detectors, assumes implicitly that all the correct processes participate in the consensus algorithm.)

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 [11] to capture the safety properties of Paxos-like consensus algorithms [14]. 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 $\mathsf{alpha\_propose}_k(r_i, v_i)$. 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 r_i (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 $\perp$.

An algorithm constructing a *KA* object from single-writer/multi-reader atomic registers is described in [15].

The k-set algorithm. The algorithm constructing a k-set object *KS* (accessed by at most n processes[1]), is described in Figure 4. 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 $[\perp, \ldots, \perp]$. 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, 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 $\mathsf{kset\_propose}_k(v_i)$:
(1) $r_i \leftarrow (i - n)$;
(2) **while** $(\forall j : DEC[j] = \perp)$ **do**
(3) **if** $id_i \in \mathsf{leader}()$ **then** $r_i \leftarrow r_i + n$; $DEC[i] \leftarrow KA.\mathsf{alpha\_propose}_k(r_i, v_i)$ **endif**
(4) **end_while**;
(5) **let** $decided_i =$ any $DEC[j] \neq \perp$;
(6) $\mathsf{return}(decided_i)$

Fig. 4. 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.

[1] Let us remind that the construction of each $SKS[\ell]$ object used in Figure 3 is based on an underlying k-set object *KS* object.

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.

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 Symp. on Principles of Dist. Comp. (PODC'99)*, ACM Press, pp. 105-112, 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. Symp. on Dist. Comp. (DISC'00)*, LNCS #1914, pp. 149-163, 2000.
5. Attiya H. and Welch J., *Distributed Computing: Fundamentals, Simulations and Advanced Topics,* (2d Edition), Wiley-Interscience, 414 pages, 2004.
6. Borowsky E. and Gafni E., Immediate Atomic Snapshots and Fast Renaming. *Proc. 12th ACM Symp. on Principles of Distr. Comp. (PODC'93)*, pp. 41-51, 1993.
7. Borowsky E. and Gafni E., Generalized FLP Impossibility Results for t-Resilient Asynchronous Computations. *Proc. 25th ACM STOC*, pp. 91-100, 1993.
8. Chandra T., Hadzilacos V. and Toueg S., The Weakest Failure Detector for Solving Consensus. *Journal of the ACM*, 43(4):685–722, 1996.
9. Gafni E., Read/Write Reductions. *DISC/GODEL presentation given as introduction to the 18th Int'l Symposium on Distributed Computing (DISC'04)*, 2004.
10. Gafni E., Rajsbaum R., Raynal M. and Travers C., The Committee Decision Problem. *Proc. 8th LATIN*, LNCS #3887, pp. 502-514, 2006.
11. Guerraoui R. and Raynal M., The Alpha of Asynchronous Consensus. *The Computer Journal*. To appear.
12. Herlihy M.P., Wait-Free Synchronization. *ACM TOPLAS*, 13(1):124-149, 1991.
13. Herlihy M.P. and Shavit N., The Topological Structure of Asynchronous Computability. *Journal of the ACM*, 46(6):858-923,, 1999.
14. Lamport L., The Part-Time Parliament. *ACM TOCS*, 16(2):133-169; 1998.
15. Mostefaoui 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. *Tech Report #1676*, IRISA, Université de Rennes (France), 2006.
16. Neiger G., Failure Detectors and the Wait-Free Hierarchy. *Proc. 14th Int'l ACM Symp. on Principles of Dist. Comp. (PODC'95)*, ACM Press, pp. 100-109, 1995.
17. 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.