From Renaming to Set Agreement

Achour Mostefaoui, Michel Raynal, and Corentin Travers

IRISA, Université de Rennes, 35042 Rennes, France
{achour,raynal,ctravers}@irisa.fr

Abstract. The M-renaming problem consists in providing the processes with a new name taken from a new name space of size M. A renaming algorithm is adaptive if the size M depends on the number of processes that want to acquire a new name (and not on the total number n of processes). Assuming each process proposes a value, the k-set agreement problem allows each process to decide a proposed value in such a way that at most k different values are decided. In an asynchronous system prone to up to t process crash failures, and where processes can cooperate by accessing atomic read/write registers only, the best that can be done is a renaming space of size $M = p + t$ where p is the number of processes that participate in the renaming. In the same setting, the k-set agreement problem cannot be solved for $t \geq k$.

This paper focuses on the way a solution to the renaming problem can help solving the k-set agreement problem when $k \leq t$. It has several contributions. The first is a t-resilient algorithm ($1 \leq t < n$) that solves the k-set agreement problem from any adaptive $(n + k - 1)$-renaming algorithm, when $k = t$. The second contribution is a lower bound that shows that there is no wait-free k-set algorithm based on an $(n + k - 1)$-renaming algorithm that works for any value of n, when $k < t$. This bound shows that, while a solution to the $(n + k - 1)$-renaming problem allows solving the k-set agreement problem despite $t = k$ failures, such an additional power is useless when $k < t$. In that sense, in an asynchronous system made up of atomic registers, $(n + k - 1)$-renaming allows progressing from $k > t$ to $k = t$, but does not allow bypassing that frontier. The last contribution of the paper is a wait-free algorithm that constructs an adaptive $(n + k - 1)$-renaming algorithm, for any value of the pair (t, k), from a failure detector of the class Ω_*^k (this last algorithm is a simple adaptation of an existing renaming algorithm).

1 Introduction

Asynchronous Computability. Renaming and set agreement are among the basic problems that lie at the core of computability in asynchronous systems prone to process crashes. The *renaming* problem (introduced in [3]) consists in designing an algorithm that allows processes (that do not crash) to obtain new names from a new name space that is as small as possible. In the following M denotes the size of the new name space, and a corresponding algorithm is called an M-renaming algorithm.

G. Prencipe and S. Zaks (Eds.): SIROCCO 2007, LNCS 4474, pp. 66–80, 2007.

A *wait-free* algorithm is an algorithm that allows each process that does not crash to terminate in a finite number of computation steps, whatever the behavior of the other processes (i.e., despite the fact that all the other processes are extremely slow, or even have crashed) [12]. It has been shown that, in a system of n processes that can communicate through atomic read/write registers only, the smallest new name space that a wait-free renaming algorithm can produce is lower bounded by $M = 2n - 1$ [15]. More generally, in an asynchronous system where up to t processes may crash, the smallest value of M is $n+t$ (the wait-free case corresponds to $t = n - 1$).

A renaming algorithm is *adaptive* if the size of the new name space depends only on the number of processes that ask for a new name (and not on the total number of processes). Let p be the number of processes that *participate* in the renaming. Several adaptive algorithms have been designed such that the size of the new name space is $M = 2p - 1$ (e.g., [2,5]). These adaptive algorithms are consequently optimal with respect to the size of the new name space.

Recently, with the aim of circumventing the $M = 2p - 1$ lower bound, researchers have investigated the use of base objects stronger than atomic registers in order to solve the renaming problem. Following this line of research, it has been shown in [19] that, as soon as k-test&set objects can be used, the renaming problem can be wait-free solved with a new name space the size of which is $M = 2p - \lceil \frac{p}{k} \rceil$ [1]. Among the processes that access it, a k-test&set object ensures that at least one and at most k processes obtain the value 1 (they win), while all the other processes obtain the value 0 (they lose)[2]. It has also been shown in [10] that the renaming problem can be wait-free solved with a new name space of size $M = p + k - 1$ as soon as k-set agreement objects can be used. According to the base objects they use, respectively, both algorithms are optimal with respect to the size of their new name space.

The k-set agreement problem (sometimes abbreviated k-set), has been introduced in [8]. It is a paradigm of coordination problems encountered in distributed computing and is defined as follows. Each process is assumed to propose a value. The problem consists in designing an algorithm such that (1) each process that does not crash decides a value (termination), (2) a decided value is a proposed value (validity), and (3) no more than k different values are decided (agreement). (The well-known consensus problem is nothing else than the 1-set agreement problem.) The parameter k can be seen as the coordination degree (or the difficulty) associated with the corresponding instance of the problem. The smaller k is, the more coordination among the processes is imposed: $k = 1$ means the strongest possible coordination, while $k = n$ means no coordination.

It has been shown in [6,15,22] that, in an asynchronous system made up of processes that communicate through atomic registers only, and where up to t processes may crash, there is no wait-free k-set agreement algorithm for $k \leq t$.

[1] The renaming algorithm presented in [19] is actually based on k-set agreement objects. But, as observed by E. Gafni, these objects can be trivially replaced by k-test&set objects without affecting the behavior of the renaming algorithm.

[2] The usual test&set object is a 1-test&set object.

Differently, when $k > t$ the problem can be trivially solved (a predefined set of k processes write their proposal, and a process decides the first proposal it reads).

Randomized or failure detector-based algorithms have been proposed to circumvent the previous impossibility result [13,17,18]. An algorithm that wait-free solves the $(n-1)$-set agreement in a system of n crash-prone asynchronous processes from $(2n-2)$-renaming objects is described in [9].

Content of the Paper. The paper has three contributions. The first is motivated by the computability power of the renaming problem with respect to the set agreement problem. More specifically, the paper considers systems made up of n processes. In such a system, an algorithm is *t-resilient* if it always preserves its safety and liveness properties when no more than t processes commit failures. (The notion of t-resilience boils down to the wait-free notion when $t = n-1$.) The first contribution investigates the t-resilience notion to solve the k-set agreement problem from renaming objects. It presents a t-resilient algorithm that solves the k-set problem from an adaptive $(n+k-1)$-renaming object when $k = t$. Interestingly, this result generalizes a previous result presented in [9] that also considers $k = t$, but only for the wait-free case (i.e., $t = n-1$). So, the algorithm presented in the paper works for any value of t. When we consider the constructions relating renaming and set agreement that are known, we obtain the transformations described in Figure 1. Interestingly, it follows from the proposed algorithm (that considers $k = t$) that, in asynchronous shared memory systems prone to a single process crash ($t = 1$), a solution to the renaming problem allows solving the consensus problem (and vice-versa).

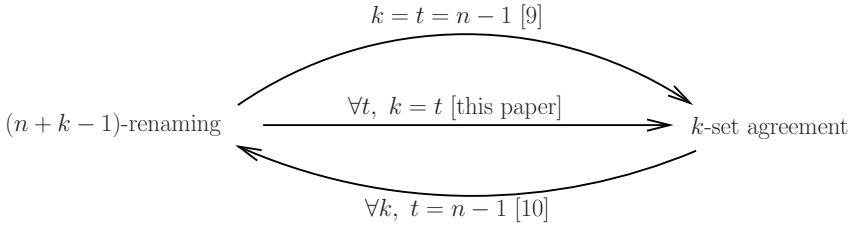

Fig. 1. Piecing together the transformations

The second contribution of the paper is a lower bound. While, in an asynchronous shared memory system made up of atomic registers only, the k-set agreement problem can be (trivially) solved when $k > t$, and is impossible to solve when $k \leq t$, the previous algorithm shows that enriching the system with an adaptive $(n+k-1)$-renaming algorithm allows progressing from $k > t$ to $k = t$. So, an important question is the following: does an $(n+k-1)$-renaming algorithm allows bypassing the $k = t$ frontier? The second contribution of the paper shows that such a renaming algorithm is not powerful enough to do it. More precisely, it shows that, in an asynchronous shared memory system made up of atomic registers and $(n+k-1)$-renaming, there are values of n for which

it is not possible to solve the k-set agreement problem when $k < t$. (Showing that this is true for any value of n remains an open problem.)

The last contribution is a wait-free algorithm that builds a $(p+k-1)$-renaming object from an oracle of the class Ω_*^k. Such an oracle class has been introduced in [21]. It generalizes the "leader" oracles (failure detectors) classes introduced in [7,11,19,20]. Basically, such an oracle provides the processes with a primitive $\mathsf{leader}()$ that always returns a set of at most k processes, and after some unknown but finite time, returns always the same set that contains at least one correct participating process. Interestingly, that algorithm is a simple generalization of an $(n + t)$-renaming algorithm described in [4] (that is in turn an adaptation to the shared memory setting of an $(n + t)$-renaming algorithm designed for message-passing systems [3]).

Roadmap. The paper is made up of 5 sections. Section 2 describes the computation model. Section 3 presents a t-resilient algorithm that solves the k-set problem from a single $(n+k-1)$-renaming object. Section 4 shows that $(n+k-1)$-renaming does not allow solving the k-set agreement problem when $k < t$, for any value of n. Then, Section 5 presents a wait-free construction from Ω_*^k to an adaptive $(p + k - 1)$-renaming object.

2 Basic Computation Model

Process Model. The system is made up of n asynchronous processes $p_1, \ldots, p_n$. The integer i is the index of p_i while its identity is kept in id_i. Π denotes the set of indexes, i.e., $\Pi = \{1, \ldots, n\}$. *Asynchronous* means that there is no bound on the time it takes for a process to execute a computation step. A process may crash (halt prematurely). After it has crashed a process executes no step. A process executes correctly its algorithm until it possibly crashes. The integer t, $0 \leq t < n$, denotes an upper bound on the number of processes that may crash; t is known by the processes. A process that does not crash in a run is *correct* in that run; otherwise, it is *faulty* in that run.

Communication Model. The processes cooperate by accessing atomic read/write registers. *Atomic* means that each read or write operation appears as if it has been executed instantaneously at some time between its begin and end events [16]. Each atomic register is a one-writer/multi-readers (1WnR) register. This means that a single process (statically determined) can write it. Moreover such a register is a write-once register (the writing process writes it at most once). Atomic registers are denoted with uppercase letters. The atomic registers are structured into arrays. $X[1..n]$ being such an array, $X[i]$ denotes the register of that array that p_i only is allowed to write. A process can have local registers. Such registers are denoted with lowercase letters with the process index appearing as a subscript (e.g., $winner_i$ is a local register of p_i).

The processes are provided with an atomic snapshot operation [1] denoted $\mathsf{snapshot}(X)$, where $X[1..n]$ is an array of atomic registers. It allows a process p_i to atomically read the whole array. This means that the execution of a $\mathsf{snapshot}()$

operation appears as it has been executed instantaneously at some point in time between its begin and end events. Such an operation can be built from 1WnR atomic registers [1].

The value $\perp$ denotes a default value that can appear only in the algorithms described in the paper. It always remains everywhere else unknown to the processes.

Notions of t-resilience and Wait-freeness. An algorithm is t-resilient if it copes with up to t process failures. In our context, this means that it satisfies its safety and liveness (termination) properties despite up to t process crashes. A wait-free algorithm is an $(n-1)$-resilient algorithm.

Notion of Adaptive Renaming. In the renaming problem, each process p_i has an initial name denoted id_i (that it is the only to know). These names are from a very large name space, i.e., $\max(id_1, \ldots, id_n) >> n$. A renaming algorithm is adaptive with respect to the size of its new name space, if that size depends on the number of processes that actually participate in the renaming algorithm. A process participates in an algorithm as soon as it has written an atomic register used by that algorithm. Let us remark that an adaptive renaming algorithm cannot systematically assign the new name i to p_i. This is because, if only p_n wants to acquire a new name, the new name space is $[1..n]$, which depends on the number of processes instead of depending on the number of participating processes (here a single process). To rule out this type of ineffective solution, the following symmetry requirement is usually considered for the renaming problem [4]: the code executed by p_i with name id is the same as the code executed by process p_j with name id. This means that the process indexes can be used only for addressing purposes.

As indicated in the introduction, if p processes participate in a renaming algorithm based on atomic registers only, the best that can be done is an adaptive name space of size $M = 2p - 1$. This means that if "today" p' processes acquire new names, their new names belong to the interval $[1..2p' - 1]$. If "tomorrow" p'' additional processes acquire new names, these processes will have their new names in the interval $[1..2p - 1]$ where $p = p' + p''$.

3 From Adaptive $(p + k - 1)$-Renaming to k-Set Agreement

Considering an asynchronous system made up of n processes, where up to t $(1 \leq t < n)$ may crash and where the processes can cooperate through 1WnR write-once atomic registers, plus an adaptive $(p + t - 1)$-renaming object (where $p \leq n$ is the number of participating processes), this section presents and proves correct an algorithm that builds a t-set agreement object.

3.1 Principles and Description of the t-Resilient Algorithm

The principle of the transformation algorithm rests on two simple ideas.

1. First, use the underlying adaptive renaming object to partition the participating processes into two groups: the processes the name of which is smaller or equal to t (the winners); and the processes the name of which is greater than t (the losers). So, there are at most t winners.
2. Then, direct a process p_i to decide a value proposed by a winner. If p_i does not see winner processes, direct it to decide the value proposed by a process that has proposed a value but not yet obtained a new name.

To make operational these ideas, the shared memory is composed of two arrays of 1WnR write-once atomic registers.

- The array $PROP[1..n]$, initialized to $[\bot, \ldots, \bot]$, is such that $PROP[i]$ will contain the value (denoted v_i) proposed by p_i to the set agreement problem. A process p_i becomes *participating* as soon as $PROP[i] \neq \bot$.
- The aim of the array $RENAMED[1..n]$, also initialized to $[\bot, \ldots, \bot]$, is to allow the processes to benefit from the renaming object. When a process p_i has obtained a new name, $RENAMED[i]$ is set to 1 if its new name is smaller or equal to t (p_i is then a winner), while $RENAMED[i]$ is set to 0 if p_i is a loser. It trivially follows that $RENAMED[i] \neq \bot$ means that p_i has acquired a new name.

The behavior of a process p_i is described in Figure 2. A process p_i invokes $\mathsf{kset\_propose}_t(v_i)$ where v_i is the value it proposes to the k-set agreement problem. It decides a value when it executes the $\mathsf{return}(v)$ statement (line 09) where v is the value it decides. The way it implements the previous design ideas can be decomposed in two stages.

1. The first stage is composed of the lines 01-04. After it has deposited its proposal (line 01), obtained a new name (line 02), and updated $RENAMED[i]$ accordingly (line 03), a process p_i atomically reads the array $RENAMED$ (using the $\mathsf{snapshot}()$ operation) until it sees that at least $n - t$ processes have acquired new names (line 04).
2. The second stage, composed of the lines 05-09, is the decision stage. It p_i sees a winner, it decides the value proposed by that winner process (lines 05, 06 and 09). If p_i sees no winner, it decides the value proposed by a process that (from its point of view) has not yet obtained a new name. The proof will show that this is a consistent rule for deciding a value.

3.2 Proof of the Algorithm

The proof considers that (1) $k = t$, i.e., the size of the new name space of the underlying adaptive renaming is $M = p+t-1$ when p processes participate, and (2) at least $(n-t)$ correct processes participate in the k-set agreement problem.

Lemma 1. *The number of values that are decided is at most t, and a decided value is a proposed value.*

```
operation kset_propose(v_i):
(1)  PROP[i] ← v_i;
(2)  new_name_i ← rename(id_i);
(3)  RENAMED[i] ← 1 if new_name_i ≤ t, 0 otherwise;
(4)  repeat renamed_i ← snapshot(RENAMED)
                until |{j : renamed_i[j] ≠ ⊥}| ≥ (n − t);
(5)  let winners_i = {j : renamed_i[j] = 1};
(6)  if winners_i ≠ ∅ then ℓ_i ← any value ∈ winners_i
(7)                  else let set_i = {j : PROP[j] ≠ ⊥ ∧ renamed_i[j] = ⊥};
(8)                       ℓ_i ← any value ∈ set_i
(9)  end if;
(10) return(PROP[ℓ_i])
```

Fig. 2. From $(n + k - 1)$-renaming to k-set, for $k = t$, $\forall t$ (code for p_i)

Proof. Let RENAMED$_i$ be the last value of $renamed_i$ when p_i exits the **repeat** loop at line 04. As a process p_x writes $RENAMED[x]$ at most once, we have RENAMED$_i[x] \neq \bot \wedge$ RENAMED$_j[x] \neq \bot \Rightarrow$ RENAMED$_i[x]$=RENAMED$_j[x]$. Let us define RENAMED$_i \leq$ RENAMED$_j$ as $\forall x$: RENAMED$_i[x] \neq \bot \Rightarrow$ RENAMED$_i[x] =$ RENAMED$_j[x]$. Due to the atomicity property of the snapshot() operation (line 04) we have $\forall i, j$: RENAMED$_i \leq$ RENAMED$_j \vee$ RENAMED$_j \leq$ RENAMED$_i$ (this is sometimes called the *containment* property provided by the snapshot() operation).

If no process ever executes line 05, the agreement and validity property are trivially satisfied. So, let us assume that at least one process executes line 05. Moreover, let RENAMED be the smallest array value obtained by a process when it exits the **repeat** loop at line 04. We consider two cases.

- $\exists x$: RENAMED$[x] = 1$.
  In that case there is at least one winner, namely, p_x. Due to the containment property, RENAMED$_i[x] = 1$ for any process p_i that decides. It follows from that observation and the lines 05-06 that any process that decides, does decide the value proposed by a winner process. As at most t processes can obtain a new name comprised between 1 and t (lines 02-03), it follows that there are at most t winners. Consequently, no more than t different values can be decided.
- $\forall x$: RENAMED$[x] \neq 1$.
  In that case, let $R = \{x$: RENAMED$[x] = 0\}$ (hence, all other entries of RENAMED are equal to $\bot$). Due to the exit condition of the **repeat** loop (line 04), we have $|R| \geq n - t$, from which it follows that $|\Pi \setminus R| \leq t$. We claim (claim $C1$) that any process p_i that decides, decides a value proposed by a process p_y such that $y \in \Pi \setminus R$. Combining this claim with $|\Pi \setminus R| \leq t$, we conclude that at most t different values can be decided.
  Proof of the claim $C1$. Let p_i be a process that decides. It decides the value in $PROP[y]$ where y has been determined at line 06 or line 08.
  - p_i selects y at line 06. In that case, p_i decides the value proposed by a process p_y such that RENAMED$_i[y] = 1$. As RENAMED $\leq$ RENAMED$_i$

(snapshot containment property), and RENAMED does not contain the value 1, we conclude that $y \notin R$, and the claim $C1$ follows.

- p_i selects y at line 08. In that case, p_i decides a value proposed by a process p_y such that $\text{RENAMED}_i[y] = \perp$. We claim (claim $C2$) that $set_i \neq \emptyset$, i.e., p_y does exist. As $\text{RENAMED}_i[y] = \perp$ and $\text{RENAMED} \leq \text{RENAMED}_i$, we conclude from the definition of R that $y \notin R$, which proves the claim $C1$.

Proof of the claim $C2$ ($set_i \neq \emptyset$). Let p_i be a process that executes line 07. That process is such that $\forall x \in \Pi$: $\text{RENAMED}_i[x] = \perp$ or 0. Let $R_i = \{x : \text{RENAMED}_i[x] = 0\}$, and $\alpha = |R_i|$. Moreover, let $r = |\{x : PROP[x] \neq \perp\}|$ where the value of $PROP[x]$ is the value read by p_i at line 07. (See Figure 3, where the time instants are such that $\tau_0 < \tau_2 < \tau_3 < \tau_4$). We show that $\alpha < r$, from which the claim follows (namely, there is a process p_y such that $PROP[y] \neq \perp \ \wedge \ \text{RENAMED}_i[y] = \perp$ when p_i executes line 07).

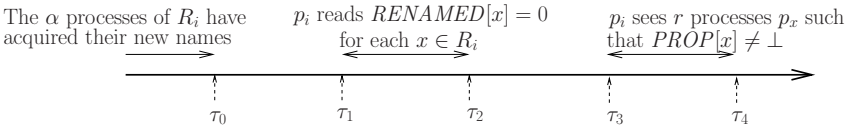

Fig. 3. Timing scenario

1. Let us first consider the processes p_x of the set R_i (i.e., the processes p_x such that $\text{RENAMED}_i[x] = 0$). These processes have obtained new names in a name space $[1..M]$ before time τ_0. We can conclude from the text of the algorithm that the new name obtained by each of these processes p_x (a loser) is such that $new\_name_x > t$ (lines 02 and 03). As there are α such processes we have $t + \alpha \leq M$.
2. Let ρ be the number of processes that started participating in the renaming before τ_0. We have seen (item 1) that M is the greatest name obtained by a process of R_i and that name has been obtained before τ_0. As the algorithm is adaptive, we have $M \leq \rho + t - 1$.
3. As the ρ processes started participating in the renaming before τ_0, they updated their entry in $PROP$ to a non-$\perp$ value before τ_0, and consequently we have $\rho \leq r$.
4. It follows from the previous items that $t + \alpha \leq M \leq \rho + t - 1 \leq r + t - 1$, from which we conclude $\alpha < r$, that terminates the proof of the claim $C2$.

$\square_{Lemma\ 1}$

Lemma 2. *Each correct process decides a value.*

Proof. As there are at least $n - t$ correct process that participate in the set agreement problem, no process can block forever at line 04. Moreover, as the set set_i of a process p_i that executes line 07 is not empty (see the claim $C2$ in the

proof of Lemma 1), the entry ℓ_i from which p_i decides is well-defined (it does exist). It follows that each correct process decides. $\square_{Lemma\ 2}$

Theorem 1. *The algorithm described in Figure 2 is a t-resilient t-set agreement algorithm.*

Proof. The proof follows directly from Lemma 1 and Lemma 2. $\square_{Theorem\ 1}$

3.3 From k-Test&Set to k-Set

In the k-test&set problem, the processes invoke an operation k_test&set() and obtains the value 1 (winner), or the value 0 (loser). The values returned to the processes satisfy the following property: there are at least one and at most k winners.

In a very interesting way, the algorithm described in Figure 2 allows solving the k-set problem from any solution to the k-test&set problem, when $k = t$, $\forall t$. The only "modification" consists in replacing the lines 02-03 by the following statement: $RENAMED[i] \leftarrow$ k_test&set().

Both 1-test&set and n-renaming have consensus number 2 [10,19]. The transformation described in Figure 2 exhibits another strong connection linking k-test&set and k-set.

4 An Impossibility Result

Theorem 2. *The k-set agreement problem cannot be solved in asynchronous systems made up of atomic registers and a solution to the adaptive $(n + k - 1)$-renaming problem, for any value of n, $k < t$ and $t = n - 1$.*

Proof. The proof uses the following notations:

- f_k: the function $p \rightarrow 2p - \lceil \frac{p}{k} \rceil$.
- g_k: the function $p \rightarrow p + k - 1$.
- (n, k)-TS: the k-tes&set problem with up to n processes. (At least one and most k processes are winners.)
- (n, k)-SA: the k-set agreement problem with up to n processes.
- (n, f_k)-AR: the adaptive M-renaming problem with $M = f_k(p)$ (where $p \leq n$ is the number of processes that participate in the renaming).
- (n, g_k)-AR: the adaptive M-renaming problem with $M = g_k(p)$ (where $p \leq n$ is the number of processes that participate in the renaming).
- Any solution to the (n, ℓ)-XX problem (where XX is TS, SA, or AR, and ℓ is k, f_k or g_k) defines a corresponding (n, ℓ)-XX object.

Let us first observe that $\forall p$, $\forall k$, we have $f_1(p) = g_1(p) \leq g_k(p)$. This means that any solution to (n, f_1)-AR is a solution to (n, g_k)-AR.

The proof consists in showing the following: $\forall k$, $\forall n \geq 2k + 1$: there is no algorithm that solves (n, k)-SA from (n, g_k)-AR. The proof is by contradiction. Let us assume that there is an algorithm $\mathcal{A}$ that, for $t = n - 1$, solves (n, k)-SA from (n, g_k)-AR with $n \geq 2k + 1$. The $(2, 1)$-SA problem plays a key role in proving the contradiction.

1. On one side.
   - The $(2,1)$-TS problem and the $(2,1)$-SA problem are equivalent [9].
   - There is a wait-free construction of (n,k)-TS from $(2,1)$-TS objects [9].
   - The (n,f_1)-AR problem can be wait-free solved from $(n,1)$-TS objects [19].
   - For any $k \geq 1$, the (n,g_k)-AR problem can be wait-free solved from (n,f_1)-AR objects (previous observation).
   - Due to the assumption, the algorithm $\mathcal{A}$ solves the (n,k)-SA problem from (n,g_k)-AR objects with $n \geq 2k+1$, when $t = n-1$.
   - It follows that, when $t = n-1$, it is possible to solve the (n,k)-SA problem from $(2,1)$-SA objects for $n \geq 2k+1$.
2. On the other side.
   - It is shown in [14] that $k \geq j\lfloor\frac{t+1}{m}\rfloor + \min\big(j,(t+1) \bmod m\big)$ is a necessary requirement for having a t-resilient k-set agreement algorithm for n processes, when these processes share atomic registers and (m,j)-SA objects (objects that allow solving j-set agreement among m processes).
   - Let us consider the case where the (m,j)-SA objects are $(2,1)$-SA objects. Let us recall $t = n-1$. We have then: $k \geq \lfloor\frac{t+1}{2}\rfloor + \min\big(1,(t+1) \bmod 2\big)$, from which we obtain the necessary requirement $k \geq \lfloor\frac{n}{2}\rfloor$.
   - It follows that, for $t = n-1$, $k \geq \lfloor\frac{n}{2}\rfloor$ (i.e., $2k \geq n$) is a necessary requirement for solving the (n,k)-SA problem from $(2,1)$-SA objects and atomic registers.
3. The previous items 1 and 2 contradict each other. It follows that the initial assumption $\mathcal{A}$ cannot hold, which proves the theorem. $\square_{Theorem\ 2}$

5 From Ω^k_* to $(p+k-1)$-Renaming

This section enriches the picture by proposing a wait-free algorithm that solves the adaptive M-renaming problem with $M = \min(2p-1, p+k-1)$, p being the number of processes that participate in the algorithm. In addition to 1WnR atomic registers, this algorithm uses an oracle of the class Ω^k_*. Interestingly, when all the correct processes participate and the oracle has no additional power (i.e., $k \geq t+1$), this algorithm boils down to a t-resilient algorithm described in [4] that solves the $(n+t)$-renaming problem.

5.1 The Class of Oracles Ω^k_*

This class has been defined in [21]. An oracle of the class Ω^k_* provides the processes with an operation denoted leader(). (As indicated in the introduction, this definition is based on the leader oracle classes introduced in [11,19,20].) When a process p_i invokes that operation, it provides it with an input parameter, namely a set X of processes, and obtains a set of process identities as a result[3].

[3] The definition of Ω^k_* is not expressed in the framework introduced by Chandra and Toueg to define failure detector classes. More precisely, in their framework, the failure detector operation that a process can issue has no input parameter. It would be possible to express Ω^k_* in their framework. We don't do it in order to keep the presentation simpler.

The semantics of Ω^k_* is based on a notion of time, whose domain is the set of integers. It is important to notice that this notion of time is not accessible to the processes. An invocation of leader(X) by a process p_i is *meaningful* if $i \in X$. If $i \notin X$, it is *meaningless*. The primitive leader() is defined by the following properties where L_X denotes the set of processes returned by an invocation leader(X).

- Termination (wait-free). Any invocation of leader() by a correct process always terminates (whatever the behavior of the other processes).
- Bounded size leadership. Whatever X, the set L_X returned by a leader(X) invocation is such that $|L_X| \leq k$.
- Triviality. A meaningless invocation can return any set (of size k) of processes.
- Eventual multi-leadership for each input set X: For any $X \subseteq \Pi$, such that $X \cap \mathit{Correct} \neq \emptyset$, there is a time τ_X such that, $\forall \tau \geq \tau_X$, all the meaningful leader(X) invocations (that terminate) return the same set L_X and this set is such that $L_X \cap X \cap \mathit{Correct} \neq \emptyset$.

The intuition that underlies this definition is the following. The set X passed as input parameter by the invoking process p_i is the set of all the processes that p_i considers as being currently *participating* in the computation. (This also motivates the notion of meaningful and meaningless invocations: an invoking process is trivially participating).

Given a set X of participating processes that invoke leader(X), the eventual multi-leadership property states that there is a time after which these processes obtain the same set L_X of at most k leaders, and at least one of them is a correct process of X. Let us observe that the (at most $k-1$) other processes of L_X can be any subset of processes (correct or not, participating or not).

It is important to notice that the time τ_X from which this property occurs is not known by the processes. Moreover, before that time, there is an anarchy period during which each process, as far as its leader(X) invocations are concerned, can obtain different sets of any number of leaders. Let us also observe that if a process p_i issues two meaningful invocations leader($X1$) and leader($X2$) with $X1 \neq X2$, there is no relation linking L_{X1} and L_{X2}, whatever the values of $X1$ and $X2$ (e.g., the fact that $X1 \subset X2$ imposes no particular constraint on L_{X1} and L_{X2}).

Let us consider an execution in which all the invocations leader(X) are such that $X = \Pi$ (the whole set of processes are always considered as participating). In that case, Ω^k_* boils down to the failure detector class denoted Ω^k introduced in [20]. If additionally, $k = 1$, we obtain the classical leader failure detector Ω introduced in [7].

When $X \subseteq \Pi$ and $k = 1$, Ω^k_* boils down to the failure detector class introduced in [11]. It is shown in [11] that Ω is weaker than Ω^1_* that in turn is weaker than $\diamond\mathcal{P}$ (the class of eventually perfect failure detectors: after some finite but unknown time, an eventually perfect failure detector suspects all the crashed processes and only them).

5.2 An Adaptive $\min(2p-1, p+k-1)$-Renaming Algorithm

As previously mentioned, the adaptive renaming algorithm that is now presented is inspired from a t-resilient renaming algorithm designed for read/write registers only, described in [4].

Atomic Registers. The algorithm uses an array of 1WnR atomic registers, denoted $STATE[1..n]$. Each register $STATE[i]$ contains three fields. The first field, denoted $STATE[i].old$, is for the initial name of p_i. The second field, denoted $STATE[i].prop$, is for the new name that p_i is currently trying to acquire. Finally, the third field, denoted $STATE[i].done$, is set to true once p_i has obtained a new name ($STATE[i].prop$ contains then the new name of p_i). Initially, each atomic register $STATE[i]$ is initialized to $< \bot, \bot, false >$.

Process Behavior. A process starts the renaming algorithm by setting a local flag denoted $done_i$ to *false*, and its current proposal for a new name to $\bot$ (line 01). Then, it enters a **repeat** loop and leaves it only when it has acquired a new name (line 15).

In the loop body, a process p_i first writes its current state in $STATE[i]$ to inform the other processes about its current progress, and then atomically reads $STATE$ (using the snapshot() operation) to obtain a consistent view of the global state. If it has not yet determined a name proposal or there is another process that has chosen the same name proposal (line 05), p_i enters the lines 06-11 to determine another name proposal. Differently, if its current name proposal is not proposed by another process (the test of line 05 is then negative), p_i commits its last proposal that becomes its new name (line 12), informs the other processes (line 13), and decides that new name (line 15).

To determine a name proposal, a process p_i proceeds as follows. It first determines the processes that are competing to have a new name. Those are the processes p_j that, from p_i's point of view, are participating in the renaming (namely, the processes p_j such that $state_i[j].old \neq \bot$) and have not yet obtained a new name (i.e., such that $\neg(state_i[j].done)$). Before starting the next execution of the loop body, some processes have to change their new name proposal (otherwise, it could be possible that they loop forever). So, a process p_i does the following.

- According to the set of processes perceived as competing with it, p_i computes a current set of leaders (line 07).
- If it does not appear in the set of leaders, p_i starts directly another execution of the loop body. Let us notice that, in that case, p_i's new name proposal is not modified.
- Differently, if it appears in the set of leaders (line 08), p_i determines a new name proposal before starting another execution of the loop body. This determination (done exactly as in [4]) consists for p_i in first computing its rank within the leader set, and then taking as its new name proposal the first integer not yet used by the other processes (lines 09-10).

5.3 Proof of the Algorithm

Lemma 3. *Let p be the number of processes that participate in the renaming. The size of the new name space is $M = \min(2p-1, p+k-1)$.*

Proof. Let us consider a run in which p processes participate. Let p_i be a process that returns a new name (line 15). The new name obtained by p_i is the last name it has proposed (at line 10 during the previous iteration). When p_i defined its last name proposal, at most $p-1$ other processes have previously defined a name proposal, i.e., $|\{j \ : \ (j \neq i) \wedge (state_i[j].prop \neq \perp)\}| \leq p-1$ (O1). Moreover, due to the definition of Ω^k_*, when it defines its last name proposal, the rank of p_i in $leaders_i$ is at most $\min(p,k)$ (O2). It follows from (O1) and (O2) that the last name proposal computed by p_i is upper bounded by $(p-1)+\min(p,k)$, i.e., $M = \min(2p-1, p-1+k)$. $\square_{Lemma\ 3}$

Lemma 4. *No two processes decide the same new name.*

Proof. [Preliminary Remark. This proof is verbatim the same as the corresponding proof in [4]. We give it only for completeness purpose. As noticed in [4], this follows from the fact that this proof does not depend on the way the new names are chosen. It is based only on the structure of the algorithm and the containment property of the the snapshot() operation.]

The proof is by contradiction. Let us assume that p_i and p_j obtain the same new name a. Let STATE_i (resp., STATE_j) be the last snapshot value obtained by p_i (resp., p_j) before returning its new name a. Due to the sequence of the lines 10, 02 and 04 executed by p_i (resp., p_j) before deciding its new name, we have $\text{STATE}_i[i].prop = a$ (resp., $\text{STATE}_j[j] = a$). Moreover, after having written its last new name proposal, a process does not change its entry of $STATE.prop$.

Due to the containment property of the snapshot($STATE$) operation, we have $\text{STATE}_i \leq \text{STATE}_j$ or $\text{STATE}_j \leq \text{STATE}_i$. Let us assume without loss of generality that $\text{STATE}_i \leq \text{STATE}_j$. It follows from the containment property that $\text{STATE}_j[i].prop = \text{STATE}_i[i].prop = a$. According to the test of line 05, p_j proceeds to lines 06-11 to select a new name proposal distinct from $\text{STATE}_j[i].prop = a$, which proves the lemma. $\square_{Lemma\ 4}$

Lemma 5. *Each correct process that participates obtains a new name.*

Proof. As in [4], the proof is by contradiction. Let us assume that a process takes infinitely many steps without obtaining a new name. Let $CORRECT$ be the set of correct processes, and NT the subset of correct processes that do not terminate. Let τ be a time such that:

1. Each (correct or not) participating process p_j has written its initial name id_j in $STATE[j].old$ before $\tau_1 < \tau$.
2. Each (correct or not) process p_j that decides, has set $STATE[j].done$ to *true* before $\tau_2 < \tau$.

3. Each process $p_i \in NT$ has taken at least one snapshot of $STATE$ between $\max(\tau_1, \tau_2)$ and τ.
   Due to the containment property provided by the snapshot() primitive, it follows that, after τ, each process $p_i \in NT$ sees the same set of participating processes and the same set of processes that have decided.
4. Let $\tau_3 < \tau$ be the time from which the multi-leadership property of Ω^k_* remains forever satisfied.

Let $contending_x[\tau']$ be the value, at time τ', of the set $\{j : (state_x[j].old \neq \bot) \wedge \neg(state_x[j].done)\}$. Let p_i be a process of NT, and $CTD = contending_i[\tau]$. Let us observe that, at any time $\tau' \geq \tau$, and for each process $p_j \in NT$, we have $contending_j[\tau'] = CTD$. Moreover, $NT \subseteq CTD$. It follows from the properties of Ω^k_*, that there is a set $leaders$ such that, after τ, each time a process $p_j \in NT$ invokes leader(CTD), it obtains $leaders$. Since $CTD \setminus NT$ contains only faulty processes, and (due to the definition of τ) $leaders \cap CTD \cap CORRECT \neq \emptyset$, the set $leaders \cap CTD$ is not empty and contains at least one correct process.

As $|leaders| \leq k$, all the correct processes in $leaders \cap CTD$ select a new name proposal when they execute the lines 09-11, and these new name proposals are all different (this follows from the fact that they select their rank from the same set $leaders$). It follows that they decide their new name. A contradiction with the assumption that the processes of NT do not terminate. $\square_{Lemma\ 5}$

```
operation rename(id_i):
(1)  prop_name_i ← ⊥; done_i ← false;
(2)  repeat
(3)    STATE[i] ←< id_i, prop_name_i, done_i >;
(4)    state_i ← snapshot(STATE);
(5)    if (prop_name_i = ⊥) ∨ (∃j : (j ≠ i) ∧ (state_i[j].prop = prop_name_i))
(6)      then contending_i ← {j : (state_i[j].old ≠ ⊥) ∧ ¬(state_i[j].done)};
(7)           leaders_i ← leader(contending_i);
(8)           if id_i ∈ leaders_i then
(9)             let r_i = rank of id_i in leaders_i;
(10)            prop_name_i ← r_i-th integer ∉ X where
(11)              X = {state_i[j].prop : (j ≠ i) ∧ (state_i[j].prop ≠ ⊥)} end if
(12)     else new_name_i ← prop_name_i; done_i ← true;
(13)          STATE[i] ←< id_i, prop_name_i, done_i > end if
(14) until done_i;
(15) return(prop_name_i)
```

Fig. 4. From Ω^k_* to adaptive M-renaming with $M = \min(2p-1, p+k-1)$ (p_i's code)

Theorem 3. *The algorithm described in Figure 4 is an adaptive wait-free M-renaming algorithm with $M = \min(2p-1, p+k-1)$.*

Proof. The theorem follows from Lemma 3, Lemma 4 and Lemma 5. $\square_{Theorem\ 3}$

References

1. Afek, Y., Attiya, H., Dolev, D., Gafni, E., Merritt, M., Shavit, N.: Atomic Snapshots of Shared Memory. Journal of the ACM 40(4), 873–890 (1993)
2. Afek, Y., Merritt, M.: Fast, Wait-Free $(2k-1)$-Renaming. 18th ACM Symposium on Principles of Distributed Computing (PODC'99), pp. 105–112 (1999)
3. Attiya, H., Bar-Noy, A., Dolev, D., Peleg, D., Reischuk, R.: Renaming in an Asynchronous Environment. Journal of the ACM 37(3), 524–548 (1990)
4. Attiya, H., Welch, J.P.: Distributed Computing: Fundamentals, Simulations and Advanced Topics, 2nd edn. p. 414. Wiley-Interscience, New York (2004)
5. Borowsky, E., Gafni, E.: Immediate Atomic Snapshots and Fast Renaming. 12th ACM Symp on Principles of Distributed Computing (PODC'93), pp. 41–51 (1993)
6. Borowsky, E., Gafni, E.: Generalized FLP Impossibility Results for t-Resilient Asynchronous Computations. 25th ACM Symposium on Theory of Distributed Computing (STOC'93), pp. 91–100 (1993)
7. Chandra, T., Hadzilacos, V., Toueg, S.: The Weakest Failure Detector for Solving Consensus. Journal of the ACM 43(4), 685–722 (1996)
8. Chaudhuri, S.: More Choices Allow More Faults: Set Consensus Problems in Totally Asynchronous Systems. Information and Computation 105, 132–158 (1993)
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.: Renaming with k-set Consensus: an Optimal Algorithm in $n+k-1$ Slots. In: Shvartsman, A.A. (ed.) OPODIS 2006. LNCS, vol. 4305, pp. 36–44. Springer, Heidelberg (2006)
11. Guerraoui, R., Kapałka, M., Kouznetsov, P.: The Weakest Failure Detectors to Boost Obstruction-Freedom. In: Dolev, S. (ed.) DISC 2006. LNCS, vol. 4167, pp. 376–390. Springer, Heidelberg (2006)
12. Herlihy, M.P.: Wait-Free Synchronization. ACM Transactions on Programming Languages and Systems 13(1), 124–149 (1991)
13. Herlihy, M.P., Penso, L.D.: Tight Bounds for k-Set Agreement with Limited Scope Accuracy Failure Detectors. Distributed Computing 18(2), 157–166 (2005)
14. Herlihy, M.P., Rajsbaum, S.: Algebraic Spans. Mathematical Structures in Computer Science 10(4), 549–573 (2000)
15. Herlihy, M.P., Shavit, N.: The Topological Structure of Asynchronous Computability. Journal of the ACM 46(6), 858–923 (1999)
16. Herlihy, M.P., Wing, J.M.: Linearizability: a Correctness Condition for Concurrent Objects. ACM TOPLAS 12(3), 463–492 (1990)
17. Mostéfaoui, A., Raynal, M.: k-Set Agreement with Limited Accuracy Failure Detectors. 19th ACM Symp. on Principles of Distr. Comp. pp. 143–152 (2000)
18. Mostéfaoui, A., Raynal, M.: Randomized Set Agreement. 13th ACM Symposium on Parallel Algorithms and Architectures (SPAA'01), pp. 291–297 (2001)
19. Mostéfaoui, A., Raynal, M., Travers, C.: Exploring Gafni's reduction land: from Ω^k to wait-free adaptive (2p-[p/k])-renaming via k-set agreement. In: Dolev, S. (ed.) DISC 2006. LNCS, vol. 4167, Springer, Heidelberg (2006)
20. Neiger, G.: Failure Detectors and the Wait-free Hierarchy. In: Proc. 14th ACM Symposium on Principles of Distributed Computing (PODC'95), pp. 100–109 (1995)
21. Raynal, M., Travers, C.: In search of the holy grail: looking for the weakest failure detector for wait-free set agreement. In: Shvartsman, A.A. (ed.) OPODIS 2006. LNCS, vol. 4305, Springer, Heidelberg (2006)
22. Saks, M., Zaharoglou, F.: Wait-Free k-Set Agreement is Impossible: The Topology of Public Knowledge. SIAM Journal on Computing 29(5), 1449–1483 (2000)