# Test&Set, Adaptive Renaming and Set Agreement: a Guided Visit to Asynchronous Computability

Eli GAFNI† Michel RAYNAL⋆ Corentin TRAVERS⋆
†Department of Computer Science, UCLA, Los Angeles, CA 90095, USA
⋆IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France
eli@cs.ucla.edu {raynal|ctravers}@irisa.fr

## Abstract

*An important issue in fault-tolerant asynchronous computing is the respective power of an object type with respect to another object type. This question has received a lot of attention, mainly in the context of the consensus problem where a major advance has been the introduction of the consensus number notion that allows ranking the synchronization power of base object types (atomic registers, queues, test&set objects, compare&swap objects, etc.) with respect to the consensus problem. This has given rise to the well-known Herlihy's hierarchy.*

*Due to its very definition, the consensus number notion is irrelevant for studying the respective power of object types that are too weak to solve consensus for an arbitrary number of processes (these objects are usually called subconsensus objects). Considering an asynchonous system made up of $n$ processes prone to crash, this paper addresses the power of such object types, namely, the $k$-test&set object type, the $k$-set agreement object type, and the adaptive $M$-renaming object type for $M = 2p - \lceil \frac{p}{k} \rceil$ and $M = \min(2p-1, p+k-1)$, where $p \leq n$ is the number of processes that want to acquire a new name. It investigates their respective power stating the necessary and sufficient conditions to build objects of any of these types from objects of any of the other types. More precisely, the paper shows that (1) these object types define a strict hierarchy when $k \neq 1, n-1$, (2) they all are equivalent when $k = n-1$, and (3) they all are equivalent except $k$-set agreement that is stronger when $k = 1 \neq n-1$ (a side effect of these results is that that the consensus number of the renaming problem is 2.)*

**Keywords***: Adaptive renaming, Asynchronous system, Atomic register, Atomic snapshot, Process crash, Reduction, Set agreement, Shared memory, Test&set, $t$-Resilience, Wait-free algorithm.*

## 1 Introduction

### 1.1 Asynchronous distributed problems

*Test&set*, *renaming* and *set agreement* are among the basic problems that lie at the core of wait-free computability in asynchronous shared memory systems prone to process crashes. *Wait-free* means that the algorithm that solves the problem must allow each process (that does not crash) to terminate all its operations 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 very slow or even crash) [13].

The renaming problem has been introduced in [3] in the context of asynchronous message-passing systems prone to process crash failures. It consists in designing an algorithm that allows processes (that do not crash) to obtain new names from a name space of size $M$ (such an algorithm is called an $M$-renaming algorithm). It has been shown that in an asynchronous system made up of $n$ processes prone to crash failures, that communicate through atomic read/write registers only, the smallest size for the new name space that a wait-free algorithm can produce is $M = 2n-1$ [15]. (This result clearly shows the additional price that has to be paided, namely $n-1$ name slots, in order to cope with the net effect of asynchrony and failures).

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 algorithm. Several adaptive renaming algorithms have been designed such that the size of the new name space is $M = 2p-1$ (e.g., [2, 4, 6, 7]). These algorithms are consequently optimal with respect to the size of the new name space when considering read/write shared memory systems.

The test&set problem is an old and well-known problem (a lot of shared memory multiprocessor machines provides

the processes with a test&set primitive that allows them to coordinate and synchronize). It consists in providing the processes with an operation that return 1 (winner) or 0 (loser) to the invoking process, in such a way that only one process obtains the value 1. The $k$-test&set problem is a simple generalization of the previous problem (that does correspond to 1-test&set): at least one and at most $k$ of the invoking processes are winners.

The $k$-set agreement problem has been introduced in [9]. It is a paradigm of coordination problems encoutered 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 [8, 15, 20] 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 $k \leq t$. 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).

The $k$-set agreement problem is on the values proposed by the processes. In order that problem be non-trivial, the number of values proposed has to be greater than $k$. That problem defines a relation linking its inputs and its outputs. Differently, the test&set problem is purely "syntactic" in the sense there are no input values. In the following, we consider that any number $p \leq n$ of processes participate in the $k$-test&set problem or the $k$-set agreement. This means that we implicilty consider their adaptive versions (as we implicitly do for the underlying atomic registers).

## 1.2 The kind of questions addressed in the paper

An important and pretty natural question is the following: While $M = 2p - 1$ is a lower bound for adaptive renaming when the processes communicate through atomic registers only, is it possible to obtain a smaller name space when the system is equipped with test&set objects, or with $k$-set agreement objects? More generally, what are the relations linking these three problems?

These questions have been partially answered in [11], [18] and [19]. A wait-free algorithm is presented in [18] that solves the renaming problem from $k$-test&set objects[1] for $M = 2p - \lceil \frac{p}{k} \rceil$. Another wait-free algorithm is presented in [11] that solves the $M$-renaming problem from $k$-set agreement objects for $M = p + k - 1$. A $t$-resilient algorithm is presented in [19] that solves the $k$-set agreement problem from any adaptive $\min(2p-1, p+k-1)$-renaming algorithm for $k = t$.

Are all these algorithms optimal? Among $M$-renaming, $k$-test&set, and $k'$-set agreement, are they values of $M$, $k$ and $k'$ for which these problems are equivalent? If yes, which ones? Which are the values for which these problems are not equivalent? Etc. This is the type of questions addressed in the paper, the aim of which is to capture the computability power of each problem with respect to the other ones. The ultimate goal is to relate all these problem instances in a single hierarchy.

## 1.3 Content of the paper

**Notation and definitions**

- $f_k$ and $g_k$ denote the integer functions $f_k(p) = 2p - \lceil \frac{p}{k} \rceil$ and $g_k(p) = \min(2p-1, p+k-1)$.

  Let us notice that $f_1 = g_1$, for $p \in [1..n]$: $f_{n-1} = g_{n-1}$ and $g_k < f_k$ when $k \in [2..n-2]$ ([2]).
- $(n, k)$-SA denotes the $k$-set agreement problem in a set of $n$ processes.
- $(n, k)$-TS denotes the $k$-test&set problem in a set of $n$ processes.
- $(n, f_k)$-AR denotes the adaptive $f_k(p)$-renaming problem in a set of $n$ processes.
- $(n, g_k)$-AR denotes the adaptive $g_k(p)$-renaming problem in a set of $n$ processes.
- Sometimes we say $(x, y)$-XX object instead of $(x, y)$-XX problem.
- $(x, y)$-XX $\succeq$ $(x', y')$-YY means that there is a wait-free algorithm that solves the $(x', y')$-YY problem from $(x, y)$-XX objects and atomic registers.
- $(x, y)$-XX $\simeq$ $(x', y')$-YY means that $(x, y)$-XX $\succeq$ $(x', y')$-YY and $(x', y')$-YY $\succeq$ $(x, y)$-XX.

**Global picture** Each instance of any of the previous problems involves two parameters. The first is the maximal number of processes ($n$) that can participate. For the adaptive renaming problem, the second is a function on the number of participating processes. That function defines the size of the actual new name space. Here we consider two function families ($f_k$ and $g_k$). For the two other problems, the

[1] Actually the algorithm presented in [18] is based on $k$-set agreement objects, but a simple observation shows that these objects can be replaced by $k$-test&set objects.

[2] $h_k < \ell_k$ means that $\forall k : 1 < k < n-1$, $\forall p : 1 \leq p \leq n$, $h_k(p) \leq \ell_k(p)$ and there is a value of $p$ such that $h_k(p) < \ell_k(p)$.

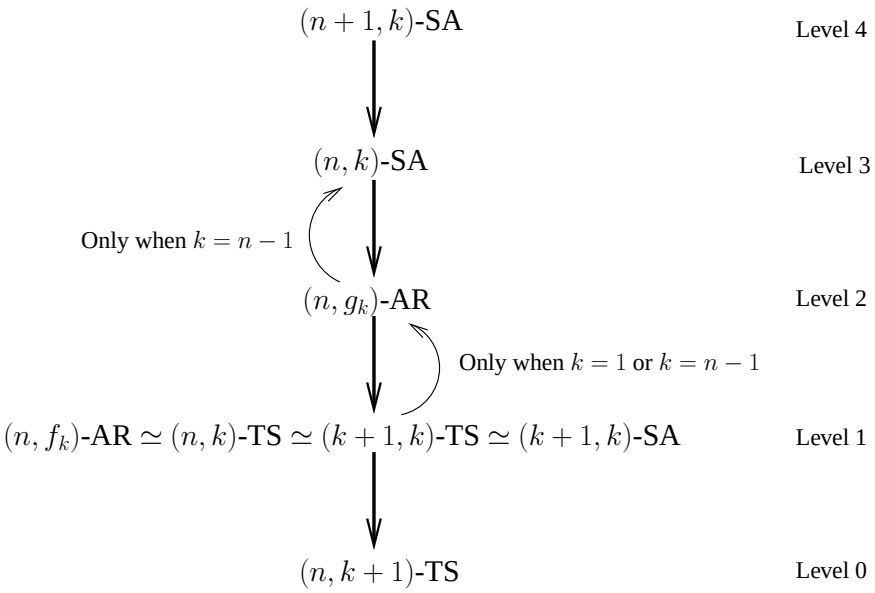


**Figure 1. A hierarchy of problems**

second parameter is the coordination degree ($k$). In both case, that parameter ($f_k$ or $g_k$ and $k$, respectively) characterizes the difficulty of the corresponding instance: the smaller that parameter is, the more difficult the problem is.

Although the renaming problem on one side, and the set agreement and test&set problems on the other side, seem to be of different nature, this paper shows that their instances can be ranked in a single hierarchy. More specifically, the results of the paper (combined with other results [10, 11, 14, 18, 19]) are depicted in Figure 1. Basically, they show that these problems can be ranked in three distinct levels (denoted 1, 2 and 3): $(n,k)$-SA is stronger than $(n,g_k)$-AR (this is denoted with a bold arrow), that in turn is stronger than$(n,f_k)$-AR, $(n,k)$-TS, $(k+1,k)$-TS, and $(k+1,k)$-SA. Moroever, these four problems are always equivalent for any value of the pair $(n,k)$.

Interestingly, it is easy to see that, when $n = k+1$, the previous hierarchy collapses, and all the problems become equivalent. It is also easy to see that, when $k = 1$ and $n > k+1$, the last two lower levels merges (as we then have $f_k = g_k$), while the $(n,1)$-SA problem remains stronger (this follows from the fact that $(n,1)$-SA is the consensus problem, while the consensus number[3] of the $(n,1)$-TS object is 2 [13]. In the other cases, the hierarchy is strict.

*Remark.* A weaker version of the renaming problem (namely, non-adaptive renaming) is considered in [12] where the authors show that non-adaptive renaming is strictly less powerful than set agreement. That work is based on combinatorial topology and does not consider the test&set problem. In addition to considering the adaptive version of the renaming problem, the approach considered here is totally different. It is entirely based on reductions.

[3]The consensus number of an object defined by a sequential specification is the maximum number of processes for which that object can wait-free solve the consensus problem.

**Structure and content of the paper** To establish the relations described in Figure 1, the paper proceeds as follows. It first presents the system model in Section 2. Then, each section presents a particular point. More precisely we have the following.

1. Section 3 shows that $(n,k)$-TS $\simeq (k+1,k)$-TS.
2. Section 4 shows that $(k+1,k)$-TS $\simeq (k+1,k)$-SA.
3. Section 5 shows that $(n,f_k)$-AR $\simeq (n,k)$-TS.
4. Section 6 shows that there is no construction of an $(n,g_{k-1})$-AR object from $(n,k)$-SA objects.

**Piecing these results (and other results) to obtain the global picture** We are now in order to justify the presence and the absence of arrows in the Figure 1. (As we can see, as it aggregates new results with previous results, the paper has also a "survey flavor".)

- Equivalences. The previous items 1, 2 and 3 establish the equivalences stated in level 1.
- Bold arrows (going down).
  - From level 4 to level 3: trivial transformation from $(n+1,k)$-SA to $(n,k)$-SA.
  - From level 3 to level 2: transformation $(n,k)$-SA $\succeq (n,g_k)$-AR in [11].
  - From level 2 to level 1: from the fact that $\forall k: 1 \leq k \leq n-1$: $g_k \leq f_k$.

– From level 1 to level 0: trivial transformation from $(n, k)$-TS to $(n, k + 1)$-TS.

- Slim arrows (going up).
  - From level 1 to level 2: follows from $f_1 = g_1$ (case $k = 1$) and $f_{n-1} = g_{n-1}$ (case $k = n - 1$).
  - From level 1/2 to level 3: when $n = k+1$ $(n, k)$-SA is $(k+1, k)$-SA (and both are then equivalent to $(k + 1, g_k)$-AR).
- Impossibility.
  - From level 3 (resp., 0) to level 4 (resp., 1): proved in [14].
  - From level 2 to level 3 for $k \neq n - 1$: proved in [19].
  - From level 1 to level 2 for $k \neq 1, n - 1$: follows from $\forall k : 1 < k < n - 1$: $g_k < f_k$.
- Optimality.
  - As $f_k < f_{k+1}$, the algorithm described in [18], that builds an $(n, f_k)$-AR object from $(n, k)$-TS objects, is optimal in the sense that an $(n, f_k)$-AR object cannot be built from $(n, k+1)$-TS objects.
  - Due to item 4, The algorithm described in [11], that solves the $(n, g_k)$-AR problem from $(n, k)$-SA objects (bold arrow from level 3 to level 2) is optimal in the sense that no new name space smaller than $g_k$ can be obtained from $(n, k)$-SA objects only.

# 2 Computing model, object types, and transformation requirements

## 2.1 Process model and atomic registers

**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 $id_i$. A process does not know the initial names of the other processes, it only knows that no two processes have the same initial name. (The initial name is a particular value defined in $p_i$'s initial context.) A process can crash. Given a run, a process that crashes is said to be *faulty*, otherwise it is *correct* in that run. Each process progresses at its own speed, which means that the system is asynchronous.

While we are mainly interested in wait-free transformations, we sometimes consider a $t$-resilient transformation. Such a transformation can cope with up to $t$ process crashes, where $t$ is a system parameter known by all the processes. A wait-free transformation is nothing else than an $(n - 1)$-resilient transformation.

**Atomic registers and snapshot operation** 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, 17]. Each atomic register is a one-writer/multi-readers (1W$n$R) register. This means that a single process can write it. Atomic registers (and shared objects) are denoted with uppercase letters. The atomic registers are structured into arrays. A process can have local registers. Such registers are denoted with lowercase letters with the process index appearing as a subscript (e.g., $res_i$ is a local register of $p_i$).

The processes are provided with an atomic snapshot operation [1] denoted $X.\mathsf{snapshot}()$, where $X$ is an array of atomic registers, each entry of which can be written by at most one process. 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 event and its end event. Such an operation can be wait-free implemented from 1W$n$R atomic registers [1]. (To, our knowledge the best $\mathsf{snapshot}()$ implementation requires $O(n \log(n))$ read/write operations on base atomic registers [5].)

**Default value** 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.

## 2.2 Base objects

The objects considered here are the objects associated with the adaptive renaming, $k$-test&set and $k$-set agreement problems as discussed in the Introduction. These objects are considered in a system of $n$ processes, and are consequently accessed by at most $n$ processes.

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

- Termination. An invocation issued by a correct process terminates.
- Validity. The value returned by an invocation is 1 (winner), or 0 (loser).
- Agreement. At least one and at most $k$ processes obtain the value 1.

**Set agreement object** A $(n, k)$-SA object is an object that allow processes to propose values and decide values.

To that end, it provides them with an operation denoted $\mathsf{SA\_propose_k}()$. A process invokes that operation at most once. When it invokes $\mathsf{SA\_propose_k}()$, the invoking process supplies the value $v$ it proposes (input parameter). That operation returns a value $w$ called "the value decided by the invoking process" (we also say "the process decides $w$"). The invocations on such an object satisfy the following properties:

- Termination. An invocation issued by a correct process terminates.
- Validity. A decided value is a value that has been proposed by a process.
- Agreement. At most $k$ distinct values are decided.

**Adaptive renaming** A renaming object allows the processes to obtain new names from a new name space $[1..M]$. It provides the processes with a single operation denoted $\mathsf{AR\_rename_k}()$. A renaming algorithm is adaptive with respect to the size $M$ of its new name space, if $M$ depends on the number $p$ of processes that actually participate in the renaming algorithm (the processes that invoke $\mathsf{AR\_rename_k}()$). We consider here two families of $M$-adaptive renaming objects for $1 \leq k \leq n-1$, namely, the family of $(n, f_k)$-AR objects and the family of $(n, g_k)$-AR objects, where $M = f_k(p) = 2p - \lceil \frac{p}{k} \rceil$, and $M = g_k(p) = \min(2p-1, p+k-1)$, respectively. The invocations on such an adaptive renaming object satisfy the following properties:

- Termination. An invocation issued by a correct process terminates.
- Validity. A new name belongs to the set $[1..M]$.
- Agreement. No two invocations return the same new name.

### 2.3 Transformation requirement

This paper focuses on transformations that satisfy the following property:

- Index independence. The behavior of a process is independent of its index.

This property states that, if, in a run, a process whose index is $i$ obtains a result $v$, that process would have obtained the very same result if its index was $j$. From an operational point of view, the indexes define an underlying communication infrastructure, namely, an addressing mechanism that can only be used to access entries of shared arrays.

## 3 $(n, k)$-TS and $(k+1, k)$-TS are equivalent

As $n > k$, building a $(k+1, k)$-TS object from an $(n, k)$-TS object is trivial. So, the interesting construction is the one in the other direction. This section presents such a construction.

**Principles and description of the construction** The idea of the construction is simple. It is based on the following two principles.

First, in order to satisfy the index independence property, the transformation first uses an underlying renaming object that provides the processes with new names that they can thereafter use "instead of" their indexes. Renaming algorithms that satisfy the index independence property and use only atomic registers do exist (e.g., see [6]). These algorithms provide a new renaming space whose maximal size is $M = 2n-1$. So, the new name of a process $p_i$ is an integer in the set $\{1, \ldots, 2n-1\}$ that is independent of its index $i$. This underlying base renaming object is denoted $BASE\_AR$.

Let $m = C(2n-1, k+1)$ (the number of distinct subsets of $(k+1)$ elements in a set of $2n-1$ elements). Let us order these $m$ subsets in an array $SET\_LIST[1..m]$ in such a way that $SET\_LIST[x]$ is the set of the $k+1$ processes that define the $x$th subset. Moreover, let $BASE\_TS[1..m]$ be an array of $m$ base objects with type $(k+1, k)$-TS.

The principle that underlie the second part of the construction is the following. First, the $(k+1)$ processes that define $SET\_LIST[x]$ are the only ones that can access $BASE\_TS[x]$. When a process $p_i$ invokes $\mathsf{TS\_compete_k}()$, starting from the first base object $BASE\_TS[x]$ it belongs to, it scans (one after the other and in the increasing order on their indexes) all the sets $BASE\_TS[x]$ it belongs to. If $BASE\_TS[x].\mathsf{TS\_compete_k}()$ returns 0 (loser), $p_i$ stops scanning and returns 0 as the result of its invocation $\mathsf{TS\_compete_k}()$. Otherwise, $p_i$ is a winner among the processes that access $BASE\_TS[x]$; it then proceeds to the next object of $BASE\_TS[1..m]$ to which it belongs. If there is no such object ($p_i$ has then "successfully" scanned all the subsets it belongs to), it returns 1 as the result of $\mathsf{TS\_compete_k}()$.

The construction is described in Figure 2. The local variable $pos_i$ keeps $p_i$'s current scanning position in $SET\_LIST[1..m]$. The function $\mathsf{next}(new\_name_i, pos_i)$ returns the first entry $y$ (starting from $pos_i + 1$ and in increasing order) of $SET\_LIST[1..m]$ such that $new\_name_i$ belongs to $SET\_LIST[y]$. Finally, the predicate $\mathsf{last}(new\_name_i)$ returns $true$ iff $pos_i = m$ or $new\_name_i$ belongs to no set from $SET\_LIST[pos_i + 1]$ until $SET\_LIST[m]$. The statement $\mathsf{return}(v)$ terminates $p_i$'s invocation.

*Remark.* If a $(2n-1)$-renaming algorithm was not initially used (line 01), we would have to use $\mathsf{next}(i, pos_i)$ at line 04 (and $\mathsf{last}(i)$ at line 07). It would follow that the base objects $BASE\_TS[pos_i]$ that are accessed by a process $p_i$ at line 05 would depend on the index $i$. Consequently, the results provided to $p_i$ by these objects would depend on that index, thereby making the transformation not index indepen-

dent. As we can see, using an underlying renaming algorithm (that satisfy index independence) allows solving that issue.

```
operation TS_compete_k():
(01)  new_name_i ← BASE_AR.rename();
(02)  pos_i ← 0;
(03)  while (true) do
(04)    pos_i ← next(new_name_i, pos_i);
(05)    res_i ← BASE_TS[pos_i].TS_compete_k();
(06)    if (res_i = 0) then return (0)
(07)                   else if (pos_i = last(new_name_i))
(08)                                then return (1) end if
(09)    end if
(10)  end while
```

**Figure 2. From $(k+1,k)$-TS objects to an $(n,k)$-TS object (code for $p_i$)**

**Theorem 1** *The algorithm described in Figure 2 is a wait-free construction of an $(n,k)$-TS object from a bounded number of atomic registers and $(k+1,k)$-TS objects.*

**Proof** The validity property of the $(n,k)$-TS object is trivial: the only values that can be returned are 0 and 1 (lines 06 and 07). As far as the the termination property is concerned, let $p_i$ a correct process that invokes $\mathsf{TS\_compete_k}()$. If $p_i$ executes $\mathsf{return}(0)$ at line 06, it terminates. So, let us assume that $p_i$ never executes $\mathsf{return}(0)$ at line 06. It follows that it is a winner in each base object $BASE\_TS[y]$ it accesses. These objects define a list that has a last element $BASE\_TS[z]$. When $p_i$ accesses that base object, we have both $pos_i = z$ and $\mathsf{last}(new\_name_i) = z$, from which it follows that $p_i$ executes $\mathsf{return}(1)$ at line 07.

The proof of the agreement property is decomposed in two steps.

- At most $k$ processes are winners.
  The proof is by contradiction. Let us assume that $(k+1)$ processes are winners. Let $S$ be the set of the new names of these $(k+1)$ processes. There is a set $SET\_LIST[y]$ such that $SET\_LIST[y] = S$. Due to the code of the construction, a process $p_j$ that is a winner in the high level object $CONST$, has to be a winner in all the base $(k+1,k)$-TS objects $BASE\_TS[x]$ such that $new\_name_j \in SET\_LIST[x]$. It follows from that observation that all the processes of $S$ invoke $BASE\_TS[y].\mathsf{TS\_compete_k}()$ and obtain 1 from that base object. On another side, it follows from the agreement property of that underlying $(k+1,k)$-TS base object, that at most $k$ of these processes return the value 1. A contradiction.

- At least one process is a winner.
  Let $BASE\_TS[y]$ be the "last" $(k+1,k)$-TS base object accessed during a run ("last" means here that this base object is the one with the largest index $y$ that is accessed during a run). Due to the agreement property of the base objects, there is at least one process $p_j$ that is a winner with respect to the base object. As $p_j$ does not access other $(k+1,k)$-TS objects, it follows that it returns the value 1 as the result of its invocation $\mathsf{TS\_compete_k}()$.

$\square_{Theorem\ 1}$

# 4 $(k+1,k)$-TS and $(k+1,k)$-SA are equivalent

To show $(k+1,k)$-TS $\simeq$ $(k+1,k)$-SA, we proceed in two steps. A wait-free transformation is first presented that builds an $(n,k)$-TS object from an $(n,k)$-SA object. Then, a $t$-resilient transformation is described that builds an $(n,t)$-SA object from an $(n,t)$-TS object. When instantiated with $t = k = n-1$, that transformation becomes a wait-free transformation from $(k+1,k)$-TS to $(k+1,k)$-SA. The two transformations imply $(k+1,k)$-TS $\simeq$ $(k+1,k)$-SA.

## 4.1 From $(k+1,k)$-SA to $(k+1,k)$-TS

This section presents a simple wait-free transformation that constructs an $(n,k)$-TS object from an $(n,k)$-SA object.

**Principles and description of the construction** Its underlying idea is the following: a process that decides the value it has proposed is a winner. But, it is possible that no process decides the value it has proposed. So, the transformation consists in forcing at least one process to decide its value. To attain this goal, the processes uses a shared array, with one entry per process, that they can atomically read using a snapshot operation. This construction (initially introduced in [8]) is described in Figure 3. Its code is self-explanatory. $REG[1..n]$ is an array of atomic registers, initialized to $[\bot,\ldots,\bot]$. $KS$ denotes the underlying $(n,k)$-set object. In order that at least one and at most $k$ processes be winners, the processes are required to propose different values to the underlying $(n,k)$-set object. A simple way do that, without violating the index independence property, consists in each process $p_i$ proposing its initial identity $id_i$.

```
operation TS_compete_k():
(01)  REG[i] ← KS.SA_propose_k(id_i);
(02)  reg_i[1..n] ← REG.snapshot();
(03)  if (∃x : reg_i[x] = id_i) then res_i ← 1 else res_i ← 0 end if;
(04)  return (res_i)
```

**Figure 3. From an $(n,k)$-SA object to an $(n,k)$-TS object (code for $p_i$)**

**Theorem 2** *The algorithm described in Figure 3 is a wait-free construction of an $(n, k)$-TS object from an $(n, k)$-SA object.*

**Proof** The algorithm is trivially wait-free, which proves the termination property. The validity property of the $(n, k)$-TS object is also trivial as the only values that can be returned are 0 and 1 (line 03). For the the agreement property, we have to show that at least one and at most $k$ processes are winners.

- Due to the agreement property of the underlying $(n, k)$-SA object, there are at most $k$ processes that obtain their index from that object. It follows that the shared array $REG[1..n]$ contains at most $k$ different non-$\perp$ values. Consequently, the predicate $(\exists x : reg_i[x] = i)$ (line 03) can be true for at most $k$ processes $p_i$. It follows that at most $k$ processes can return the value 1 at line 03.
- Let us now prove that at least one process is a winner. If there is a process $p_i$ that obtains its own index from the invocation $KS.\mathsf{SA\_propose_k}(i)$, that process is a winner. So, let us assume that no process $p_i$ obtains its own index from its invocation $KS.\mathsf{SA\_propose_k}(i)$. There is consequently a cycle $j_1, j_2, \ldots, j_x, j_1$ on a subset of processes defined as follows: $j_2 = REG[j_1]$, $j_3 = REG[j_2], \ldots, j_1 = REG[j_x]$. Among the processes of this cycle, let us consider the process $p_j$ that is the last to update its entry $REG[j]$, thereby creating the cycle. (Let us observe that, as the write and snapshot operations that access the array $REG$ are linearizable, such a "last" process $p_j$ does exist.) But then, when $p_j$ executes line 03, the predicate $(\exists x : reg_j[x] = j)$ is necessarily true (as $p_j$ completes the cycle and -due to the snapshot operation- sees that cycle). It follows that $p_j$ returns 1, which completes the proof of the theorem.

$\square_{Theorem\ 2}$

### 4.2 From $(k+1, k)$-TS to $(k+1, k)$-SA

This section presents a $t$-resilient construction of an $(n, t)$-SA object from an $(n, t)$-TS object. Taking $t = k = n - 1$ gives a wait-free construction.

**Principles and description of the construction** This construction (described in Figure 4) uses two arrays of atomic registers, denoted $REG[1..n]$ and $COMPETING[1..n]$ (both initialized to $[\perp, \ldots, \perp]$). The behavior of a process can be decomposed in two parts.

- Part 1: Write and read shared registers (lines 01-04).
  When a process $p_i$ invokes $\mathsf{SA\_propose_k}(v_i)$ it first deposits in $REG[i]$ the value it proposes, in order to make it visible to all the processes (line 01). Then, it invokes $KTS.\mathsf{TS\_compete_k}()$ (where $KTS$ is the underlying $(n, t)$-TS object) and writes 1 or 0 into $COMPETING[i]$ according to the fact it is a winner or not (line 02). Then, using the snapshot operation, $p_i$ reads the whole array $COMPETING$ until it sees that at least $n - t$ processes are competing to win (notice that, in the wait-free case, a process executes only once the loop body).
- Part 2: Determine a value (lines 05-11).
  Then, $p_i$ computes the processes it sees as winners (line 05). If it sees a winner, it decides the value proposed by that process (line 07). If $p_i$ sees no winner (lines 08-09), it decides the value proposed by a process ($p_j$) that does participate ($REG[j] \neq \perp$) but not seen as a competitor by $p_i$ ($competing_i[j] \neq \perp$). In that case, as the underlying $(n, k)$-TS object is adaptive, the $p_j$ is one of the $t$ processes that can be winners ($p_j$ is not seen winner by $p_i$ because it is slow or it has crashed).

It is easy to see that the indexes are used only as pointers, thereby guaranteeing the index independence property.

```
operation SA_propose_t(v_i);
(01) REG[i] ← v_i;
(02) COMPETING[i] ← KTS.TS_compete_t();
(03) repeat competing_i ← COMPETING.snapshot()
(04)       until (|{j : competing_i[j] ≠ ⊥}| ≥ (n − t));
(05) let winners_i = {j : competing_i[j] = 1};
(06) if winners_i ≠ ∅
(07)   then ℓ_i ← any value ∈ winners_i
(08)   else let set_i = {j : REG[j] ≠ ⊥ ∧ competing_i[j] = ⊥};
(09)        ℓ_i ← any value ∈ set_i
(10) end if;
(11) return (REG[ℓ_i])
```

**Figure 4. From an $(n, t)$-TS object to an $(n, t)$-SA object (code for $p_i$)**

**Theorem 3** *Let us assume that at least $(n - t)$ correct processes participate in the algorithm described in Figure 4. Then, that algorithm is a $t$-resilient construction of an $(n, t)$-SA object from an $(n, t)$-TS object.*

**Proof** As we are concerned by $t$-resilience, we assume that at most $t$ processes may crash, and at least $n-t$ correct processes participate in the algorithm. Let us first observe that, as the underlying $(n, t)$-TS object is wait-free and at least $n-t$ correct processes participate, the termination property is trivially satisfied.

The validity property follows from the two following observations. First, if the value returned by a process $p_i$ is determined from its set $winners_i$, it is a value proposed

by a winner, and any process (winner or loser) deposits its value (line 01) before competing to be winner (line 02). Second, if the returned value is not determined from the set $winners_i$, it follows from the definition of $set_i$ that the value $REG[j]$ decided by $p_i$ has been previously deposited by $p_j$ (the proof that $set_i$ is not empty is given below in the proof of the agreement property).

The agreement property requires that at least one and at most $t$ different values are decided. Due to the underlying $(n, t)$-TS object, there are at least one and at most $t$ winner processes, so at most $t$ entries of $COMPETING$ are equal to 1. Consequently, any set $winners_i$ computed at line 05 is such that $0 \leq |winners_i| \leq t$.

Let us consider the process $p_x$ that (at line 05) obtains the smallest set $winners_x$. Due to the total order on the snapshot operations issued by the processes at line 03 (linearization order due to the atomicity of these operations), we can conclude that any process $p_i$ that executes line 05 is such that $winners_x \subseteq winners_i$. We consider two cases.

- $|winners_x| \geq 1$. In that case, it follows from the previous observation ($winners_x \subseteq winners_i$) that at least one winner is seen by each processes $p_i$ that decides. As we have $1 \leq |winners_i| \leq t$, at least one and at most $t$ different values are decided.
- $|winners_x| = 0$. In that case, it follows from line 04 that $p_x$ sees at least $(n - t)$ processes that obtained 0 from the underlying $KTS$ object (loser processes). This means that, when considering the last value of the array $COMPETING[1..n]$, there are at most $t$ processes $p_j$ such that $(COMPETING[j] = 1) \vee (REG[j] \neq \perp \wedge COMPETING[j] = \perp)$. It follows that, when $|winners_x| = 0$, at most $t$ different values can be decided.

  We now show that at least one value is decided. Let $p_i$ be a process that decides.

  - $|winners_i| \geq 1$. In that case, $p_i$ decides the value of a winner process $p_j$.
  - $|winners_i| = 0$. As the underlying $(n, t)$-TS object is adaptive, we conclude that there is at least one process $p_y$ that has invoked $KTS.\mathsf{TS\_compete_t}()$ and is a winner ($p_i$ does not see $p_y$ as a winner because $p_y$ crashed before writing $COMPETING[y]$, or has not yet written 1 into $COMPETING[y]$ because it is very slow). The important point is that such a process $p_y$ has written its value into $REG[y]$ before invoking $KTS.\mathsf{TS\_compete_t}()$. It follows that, when $p_i$ computes $set_i$, that set is not empty, and $p_i$ decides a value, which completes the proof of the theorem.

$\square_{Theorem\ 3}$

The next corollary is a rephrasing of the previous theorem for $t = k = n - 1$.

**Corollary 1** *The algorithm described in Figure 4 is a wait-free construction of a $(k+1, k)$-SA object from a $(k+1, k)$-TS object.*

## 5 $(n, f_k)$-AR and $(n, k)$-TS are equivalent

A wait-free algorithm is presented in [18] that builds an $(n, f_k)$-AR object from $(n, k)$-TS objects. So, to show that $(n, f_k)$-AR $\simeq$ $(n, k)$-TS, this section presents a wait-free construction of an $(n, k)$-TS object from an $(n, f_k)$-AR object. This construction is done in two steps. A construction of a $(k+1, k)$-TS object from a $(k+1, f_k)$-AR object is first presented. Then, this base construction is used to wait-free construct an $(n, k)$-TS object from $(n, f_k)$-AR objects.

### 5.1 From $(k + 1, f_k)$-AR to $(k + 1, k)$-TS

Let $ARF$ be a $(k + 1, f_k)$-AR object. So, the maximal size of the new name space is $M_{max} = f_k(k + 1) = 2k$. The construction from $(k + 1, f_k)$-AR to $(k + 1, k)$-TS is described in Figure 5. It is very simple: a process $p_i$ first acquires a new name, and then returns 1 (winner) if and only its new name is comprised between 1 and $k$.

```
operation TS_compete_k();
(01)  new_name_i ← ARF.rename();
(02)  if (new_name_i ≤ k) then return (1) else return (0) end if
```

**Figure 5. From an $(k + 1, f_k)$-AR object to an $(k + 1, t)$-TS object (code for $p_i$)**

**Theorem 4** *The algorithm described in Figure 5 is a wait-free construction of a $(k+1, t)$-TS object from a $(k+1, f_k)$-AR object.*

**Proof** The proofs of the termination property, the validity property and the fact that there are at most $k$ processes are trivial. So, it only remain to show that at least process returns the value 1. We consider two cases according to the number of participating processes.

- $p = k + 1$ processes invoke $ARF.\mathsf{rename}()$. We then have $M = f_k(k + 1) = 2(k + 1) - \lceil \frac{k+1}{k} \rceil = 2k$. As the new name space is $[1..2k]$, it trivially follows, from the fact that no two processes obtain the same new name, that at least one of the the $(k + 1)$ participating processes has a new name smaller or equal to $k$. Consequently, there is at least one winner.
- $p \leq k$ processes invoke $ARF.\mathsf{rename}()$. We then have $M = f_k(p) = 2p - \lceil \frac{p}{k} \rceil = 2p - 1 = p + (p - 1)$.

It follows that at least one of the $p$ processes obtains a new name in the set $[1..p]$. As $p \leq k$, it follows from the algorithm that that process is a winner.

$\square_{Theorem\ 4}$

The next corollary follows from the previous theorem, Theorem 1 and Corollary 1.

**Corollary 2** *$(k+1, f_k)$-AR $\succeq$ $(n, k)$-TS and $(k+1, f_k)$-AR $\succeq$ $(k+1, k)$-SA.*

### 5.2 From $(n, f_k)$-AR to $(n, k)$-TS

As we trivially have $(n, f_k)$-AR $\succeq$ $(k+1, f_k)$-AR, we can use the wait-free transformation from a $(k+1, f_k)$-AR object to an $(k+1, k)$-TS object, to obtain a wait-free transformation from $(n, f_k)$-AR objects to a $(n, k)$-TS object. More precisely, the construction described in Figure 2 builds an $(n, k)$-TS object from $(n, f_k)$-objects when the underlying $(2n-1)$-renaming base object is replaced by an $(n, f_k)$-AR object. So now, in the transformation of Figure 2, a process first invokes the underlying $(n, f_k)$-object and obtains a new name in the interval $[1..f_k(p)]$ (let us notice that the maximal size of the new name space is then $f_k(n) \leq 2n-1$). The rest of the transformation of Figure 2 remains unchanged. We consequently have the following theorem (whose proof is the same as the one of Theorem 1).

**Theorem 5** *The algorithm described in Figure 2, in which the base renaming object is an $(n, f_k)$-AR object, is a wait-free construction of an $(n, t)$-TS object from $(n, f_k)$-AR objects.*

## 6 $(n, g_{k-1})$-AR cannot be built from $(n, k)$-SA

This section shows that an $(n, g_{k-1})$-AR object cannot be built from $(n, k)$-SA object. As $g_k(p) - g_{k-1}(p) = 1$, a corollary of this result is that the algorithm described in [11] (that wait-free builds an $(n, g_k)$-AR object from $(n, k)$-SA objects) is optimal ($M = g_k$ is the size of the smallest renaming space that can be obtained from $(n, k)$-SA objects).

**Theorem 6** *There is no wait-free construction of an $(n, g_{k-1})$-AR object from $(n, k)$-SA objects.*

**Proof** The proof is by contradiction. It considers two cases.

- $k = 1$. We have then $g_{k-1}(p) = p-1$. It is trivially impossible to rename $p$ processes in a name space smaller than $p$.
- $k > 1$. Let us assume that there is a construction $A1$ from an $(n, k)$-SA object to an $(n, g_{k-1})$-AR object (i.e., $A1$ is an adaptive $(p+k-2)$-renaming algorithm based on $(n, k)$-SA objects and atomic registers).

  The following simple construction $A2$ builds $(n, k-1)$-TS object from $A1$. $A2$ is as follows: a process $p_i$ first uses $A1$ to obtain a new name $new\_name_i$, and considers it is a winner if and only if $new\_name_i \leq k-1$ (at most $(k-1)$ processes can be winners, and, due to the adaptivity of $A1$, at least one process is a winner).

  Given the previous $(n, k-1)$-TS object, it is (trivially) possible to build a $(k, k-1)$-TS object, and from such an object to build a $(k, k-1)$-SA object (Corollary 1).

  So the previous sequence of transformations builds a $(k, k-1)$-SA object from an $(n, k)$-SA object, which has proven to be impossible [14].

$\square_{Theorem\ 6}$

## 7 Conclusion

The aim of this paper was an investigation of the relations linking the $k$-test&set problem, the $k$-set problem, and two adaptive renaming problems, namely the $(2p - \lceil \frac{p}{k} \rceil)$-renaming problem and the $\min(2p-1, p+k-1)$-renaming problem. Three main points can be learnt from that study. First, the $k$-test&set problem and the $k$-set problem are equivalent in systems of $(k+1)$ processes. Second, whatever the number $n$ of processes defining the system, the $k$-test&set problem and the $(2p - \lceil \frac{p}{k} \rceil)$-renaming problem are always equivalent. Third, in systems of $n$ processes such that $k \neq n-1$, the $k$-set problem is strictly stronger than the other two problems; if additionally $k \neq 1$, then the $\min(2p-1, p+k-1)$-renaming problem lies exactly in between $k$-set problem agreement problem (that is stronger) and the $k$-test&set problem (that is weaker). All these relations are depicted in Figure 1. So, this paper adds some unity and complements other papers that have investigated the respective computability power of each of these problems with respect to the other ones [10, 11, 12, 14, 18, 19].

Let us finally notice that the hierarchy described in Figure 1 is far from being complete. More research remains to be done in the area of problem equivalence and problem transformation. As an example, the relation linking the $(n, k)$-SA problem and the $(n-2, k-1)$-SA problem still remains an intriguing open problem.


## Acknowledgments

The authors want to acknowledge the referees for their constructive comments.


## 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, 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. 14th Symposium on Distributed Computing (DISC'00)*, LNCS #1914, pp. 149-163, 2000.

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

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

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

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

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

[10] Gafni E., Read-Write Reductions. *Proc. 8th Int'l Conference on Distributed Computing and Networking (ICDCN'06)*, Springer Verlag LNCS #4308, pp. 349-354, 2006.

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

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

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

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

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

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

[17] Lamport. L., On Interprocess Communication, Part II: Algorithms. *Distributed Computing*, 1(2):86-101, 1986.

[18] Mostéfaoui A., Raynal M. and Travers C., Exploring Gafni's reduction land: from $\Omega^k$ to wait-free adaptive $(2p - \lceil \frac{p}{k} \rceil)$-renaming via $k$-set agreement. *Proc. 20th Symposium on Distributed Computing (DISC'06)*, Springer Verlag LNCS #4167, pp. 1-15, 2006.

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

[20] 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.