Locks Considered Harmful: A look at Non-traditional Synchronization*

Michel Raynal

IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France
raynal@irisa.fr

Abstract. This paper considers the implementation of concurrent objects in systems prone to asynchrony and process failures. It first shows that lock-based solutions have drawbacks that can make them redhibitory for systems deployed in very demanding environments, such as embedded systems. Then, considering the adaptive renaming problem as a paradigm of coordination and synchronization problems, the paper investigates wait-free implementations of an adaptive renaming object (wait-free means that these implementations do not rest on locks or waiting loops). This problem consists in assigning new distinct names (to the subset of processes that want to acquire a new name) in such a way that the new name space be as small as possible.
The best that can be done in asynchronous systems prone to process crashes, and where the processes communicate only through read/write atomic registers, is a new naming space of size $M = 2p - 1$, where p is the number of processes that want to acquire a new name (whatever the number of processes in the system). An algorithm providing such an optimal renaming space is described. Then, it is shown how the use of "additional power" such as appropriate failure detectors, or synchronization primitives stronger than read/write operations, in order to bypass the $2p - 1$ lower bound.

Keywords: Adaptive renaming, Asynchronous system, Atomic register, Concurrency, Failure detector, Fault-tolerance, Lock-free synchronization, Process crash, Shared memory system, Synchronization primitive, Wait-free computation.

1 Introduction

From mastering sequential algorithms to mastering concurrency The study of algorithms lies at the core of informatics, and participate in establishing it as a *science* with strong results on what can be computed (decidability) and what can be efficiently computed (complexity). It is consequently unanimously accepted by the community that any curriculum for undergraduate students has to include lectures on sequential algorithms. This allows the students not only to better master the basic concepts, mechanisms, techniques, difficulties and subtleties that underlie the design of algorithms, but also understand the deep nature of computer science and computer engineering.

* This work was supported by the European Network of Excellence ReSIST.

Up to now, the implementation of concurrent objects (objects that can be concurrently accessed by several processes -or threads-) is mainly based on the use of locks. More precisely, a lock is associated with each concurrent object O, and each operation accessing the internal representation of O (e.g., a $push()$ operation on a shared stack) is required to first lock the object O (thereby preventing the internal representation from being concurrently accessed) and finally to release the corresponding lock when it terminates.

Albeit (at first glance) the use of locks is relatively simple, it has several drawbacks. When locks are used only at a large grain level they can severely reduce parallelism, while using them at a fine grain level is error-prone. Moreover, whatever the grain level, lock-based solutions are deadlock prone. A more severe drawback lies in the fact that, in asynchronous systems, locks cannot cope with process crashes. This is because, when the only means processes have to communicate is read/write atomic registers, a slow process (whose slowness can be due to interrupts, swapping, etc.) cannot be distinguished from a process that has crashed. These drawbacks can make locks harmful or even irrelevant for some applications, and become redhibitory for some classes of embedded applications. A new look at synchronization concepts and techniques is hardly needed. So, the algorithmics of synchronization has to be revisited.

Adaptive renaming as a paradigm for non-traditional synchronization This paper considers the renaming problem to illustrate non-traditional synchronization concepts and mechanisms that need to be understood and mastered when one wants to implement concurrent objects while preventing the previously cited drawbacks related to the use of locks.

Let us consider a set of processes, each process having an initial name (taken from a large name space). The *adaptive renaming* problem consists in designing an algorithm that, despite asynchrony and process failures, allows processes to acquire new (distinct) names. Moreover, the new name space has to be as small as possible (in the following M denotes the size of the new name space). This means that M has to depend only on the number of processes that want to acquire a new name, and not on their total number of processes (that can be arbitrary).

The adaptive renaming problem is a paradigm of resource allocation problems: the new names are the resources acquired by the processes. The fact that no two processes can acquire the same new name gives it a mutual exclusion flavor. This problem has a simple lock-based solution: namely, a shared register that contains an integer, increased by one each time it is accessed, and protected by a lock, can be used to generate new names. Interestingly, this implementation is adaptive (the value of M depends only on the number of processes that compete to acquire a new name). Moreover, this size is optimal: if p processes want to acquire a new name, M is as small as possible, i.e., we have $M = p$. Unfortunately, as indicated before, as it is lock-based, this solution cannot cope with the net effect of asynchrony and process crashes.

Wait-free adaptive renaming 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 they are extremely rapid or slow, or even have crashed) [13]. So, a wait-free implementation of an object means

absence of starvation despite asynchrony and process crashes. Trivially, a wait-free implementation cannot be lock-based.

Let p be the number of processes that require a new name. It has been shown [14] that in a system where the processes can communicate through atomic registers only, the size of the smallest new name space that can be obtained by a wait-free algorithm is $M = 2p - 1$. This shows that, due to asynchrony and failures, there is an inherent price that has to be paid in asynchronous read/write shared memory systems, namely, the size of the new name space is $M = p + (p - 1)$, i.e., p (the smallest size that can never be bypassed) plus $(p-1)$. The quantity $(p-1)$ is consequently the smallest price to be paid to master the inherent uncertainty created by the combination of asynchrony and failures in systems where processes communicate through atomic registers only.

It is important to see that adaptivity means the following. If "today" p' processes want to acquire a new name, their new names belong to the interval $[1..2p' - 1]$. If "tomorrow", p'' additional processes want to acquire a new name, their new names will be distinct from the previous ones and will belong to the interval $[1..2p - 1]$ where $p = p' + p''$.

Content of the paper In order to investigate and illustrate non-traditional synchronization (i.e., synchronization that is not based on locks) the paper considers several wait-free implementations of an *adaptive renaming object*. The paper is made up of 7 sections. Section 2 first presents the system model, and Section 3 presents the adaptive renaming problem. Then, Section 4 presents a simple basic adaptive renaming [6] that provides an optimal new name space (i.e., $M = 2p - 1$ where p is the number of processes that participate in the renaming).

In the previous algorithm, processes communicate only through atomic registers. On another side, we know that we can obtain $M = p$ when the processes can additionally use locks. So, an important question is the following: How to enrich the system in order to provide a new renaming space whose size is smaller than $2p - 1$ while ensuring that the implementation remains wait-free? The paper presents two such approaches.

- Section 5 considers that the processes are provided with an appropriate failure detector and presents a corresponding adaptive renaming algorithm such that $M = p + k - 1$, where k is parameter that capture the power of the underlying failure detector.
- Differently, Section 6 considers that the system provides the processes with synchronization primitives more powerful than the basic atomic read/write operations. It shows that, in that case, the size of the new name space directly depends on the "power" of the synchronization primitive. The paper considers three such synchronization primitives, namely, test&set, k-set agreement and compare&swap.

Finally, Section 7 concludes the paper. It is important to notice that that paper can be considered from two complementary point of views. On one side, it presents a non-traditional view for synchronization in embedded systems, and new algorithms. On another side, it has a survey flavor that tries to capture the main issues of a new emerging research topic.

2 Base 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. 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.) The processes are asynchronous. This 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. 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 [15,16]. Each atomic register is a one-writer/multi-readers (1WnR) register. This means that a single process (statically determined) can write it. 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., $prop_i$ is a local register of p_i). The notation $\perp$ is used to denote a default value.

The shared memory provides the processes with an atomic operation that is denoted $X.\mathsf{snapshot}()$, where $X[1..n]$ is an array of atomic registers [1]. That operation allows a process p_i to atomically read the whole array $X[1..n]$ (as if it was a single atomic register). This means that the execution of $X.\mathsf{snapshot}()$ operation appears as if 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]. To our knowledge the best $\mathsf{snapshot}()$ implementation proposed so far requires $O(n \log(n))$ read/write operations on base atomic registers [5].

3 Adaptive M-renaming

The renaming problem has been introduced in [3]. Each process p_i has an initial name id_i such that no two processes have the same initial name. These initial names are taken from a set $\{1, \ldots, N\}$ such that $n << N$. Let $\mathsf{new\_name}()$ be the (only) operation provided by an adaptive M-renaming object, i.e., an object that allows processes to obtain new distinct names belonging to the interval $[1..M]$ ([1]). The behavior of this object (i.e., the the adaptive renaming problem) is defined by the following properties. Let p denote the number of processes that invoke $\mathsf{new\_name}()$ (the set of $p$articipating processes).

- Termination. If a correct process invokes $\mathsf{new\_name}()$ it obtains a new name.

[1] Trivially, whatever the operations the processes can use, there is no M-renaming object with $M < p$.

– Agreement. No two processes obtain the same new name.
– Adaptivity. A new name belong to $[1..M]$ where M is a function of p.
– Index independence. The behavior of a process is independent of its index.

The last property states that, if, in a run, a process whose index is i obtains the new name v, that process would have obtained the very same new name if its index was j. This means that, 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. Indexes cannot be used to $compute$ new names.

4 A read/write adaptive $(2p-1)$-renaming algorithm

This section presents a simple adaptive M-renaming algorithm that provides the participating processes with an optimal new name space, i.e., $M = 2p-1$, when the processes can cooperate through atomic registers only. This algorithm (due to Attiya and Welch [6]) is an adaptation to asynchronous read/write shared memory systems of a message-passing algorithm described in [3].

The communication medium: shared memory The shared memory is made up an array of 1WnR atomic registers denoted $STATE[1..n]$. Each register $STATE[i]$ is a pair made up of two fields: $STATE[i].old$ will contain the initial name of p_i, while $STATE[i].prop$ will contain the last proposal of p_i to acquire a new name. $STATE[i]$ can be written only by p_i, and is initialized to $< \bot, \bot >$.

The algorithm: underlying principle and description The algorithm is described in Figure 1 (code for the process p_i). The local register $prop_i$ contains p_i's current proposal for a new name. When p_i invokes new_name(id_i), it sets $prop_i$ to 1 (line 1), and enters a **while** loop (lines 2-12). It exits that loop when it has obtained a new name (statement return$(prop_i)$ issued at line 6).

The principle of the algorithm is as follows. A new name can be considered as a slot, and the processes compete to acquire a free slot in the interval of slots $[1..2p-1]$. After entering the loop, a process p_i first updates $STATE[i]$ (line 3) in order to announce to all the processes its current proposal for a new name (let us notice that it also implicitly announces it is competing for a new name).

Then, thanks to the snapshot operation on the shared memory (line 4), p_i obtains a consistent view of the system global state. This view is locally kept in the array $view_i$. The behavior of p_i then depends on the the consistent global state of the shared memory it has obtained, more precisely on the value of the predicate $\forall\, j \neq i :\ view_i[j].prop \neq prop_i$. We consider both cases.

– Case 1: the predicate is true. This means that no process p_j is competing with p_i for the new name $prop_i$. In that case, p_i considers the current value of $prop_i$ as its new name (line 6).
– Case 2: the predicate is false. This means that several processes are competing to obtain the same new name $prop_i$. So, p_i construct a new proposal for a new name

```
operation new_name(id_i):
(1)  prop_i ← 1;
(2)  while true do
(3)    STATE[i] ←< id_i, prop_i >;
(4)    view_i ← STATE.snapshot();
(5)    if (∀ j ≠ i :  view_i[j].prop ≠ prop_i)
(6)      then return (prop_i)
(7)      else let X    = {view_i[j].prop | (view_i[j].prop ≠ ⊥) ∧ (1 ≤ j ≤ n)};
(8)           let free = the increasing sequence 1, 2, ... , 2p − 1 from
                         which the integers in X have been suppressed;
(9)           let Y    = {view_i[j].old | (view_i[j].old ≠ ⊥) ∧ (1 ≤ j ≤ n)};
(10)          let r    = rank of id_i in Y;
(11)          prop_i   ← the rth integer in the increasing sequence free
(12)   end if
(13) end while.
```

Fig. 1. Read/write (optimal) wait-free adaptive $(2p-1)$-renaming [6]

and enters again the loop. This proposal is built from the consistent global state of the system that p_i has obtained in $view_i$.

The set $X = \{view_i[j].prop \mid (view_i[j].prop \neq \perp) \wedge (1 \leq j \leq n)\}$ (line 7) contains the proposals (as seen by p_i) for a new name, while the set $Y = \{view_i[j].old \mid (view_i[j].old \neq \perp) \wedge (1 \leq j \leq n)\}$ (line 9) contains the initial names of the processes that p_i sees as competing for obtaining a new name.

The determination of a new proposal by p_i is based on these two sets. First, p_i considers the sequence (denoted *free*) of the integers that are "free" and can consequently be used to define a new name proposal (the sequence *free* contains at least p empty slots). This sequence is the sequence of positive integers from which the proposals in X have been suppressed (line 8). Then, p_i computes its rank r among the processes that (from its point of view) want to acquire a new name (line 9). Finally, given the sequence *free* and r, p_i defines its new proposal as its rank in this sequence (this rank is r, i.e., its rank in the set of processes it sees as competing processes).

A proof of this algorithm can be found in [6]. The proof that no two new names are the same does not depend on the way the new names are chosen, it depends only on the fact that all the $STATE$.snapshot() operations appear as if they were executed one after the other. The fact that the new names belong to the interval $[1..2p-1]$ depends on the way the new names are chosen (lines 9-11).

5 Enriching the system with a failure detector

Considering a system where the processes can communicate through 1WnR atomic registers (as before), this section shows that it is possible to bypass the $(2p-1)$ lower

bound when the processes are additionally provided with a failure detector of an appropriate class. The main point is that the implementation remains wait-free. A failure detector is a device that provides the processes with information on failures [8]. That information can be more or less accurate according to the class (type) of the failure detector.

To that end, a new class of failure detectors (denoted Ω^k_*) is first introduced. The parameter k can be seen as measuring the strength of the failure detector. Then, a wait-free adaptive renaming algorithm (introduced in [19]) that provides the processes with a name space whose size is $M = \min(2p-1, p+k-1)$ is presented. Interestingly, this algorithm can be seen as generalization of the algorithm presented in the previous section.

5.1 The class Ω^k_* of failure detectors

The class Ω^k_* of failure detectors has been introduced in [20]. A failure detector of that class provides the processes with a primitive denoted $\mathsf{leader}()$. When a process invokes that primitive, it provides a set X of processes as input parameter and obtains a non-empty set of at least one and at most k processes. Let Π be the set of all the processes, and $Correct$ the set of processes that are correct in the considered run. The class Ω^k_* is made up of all the failure detectors that satisfy the following property for each set X such that $X \cap Correct \neq \emptyset$:

– Eventual multi-leadership for each set X. There is a time after which all the invocations $\mathsf{leader}(X)$ issued by correct processes return the same set L_X and this set is such that $X \cap Correct \cap L_X \neq \emptyset$.

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. Given a set X of participating processes that invoke $\mathsf{leader}(X)$, 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).[2]

5.2 An Ω^k_*-based adaptive M-renaming algorithm with $M = \min(2p-1, p+k-1)$

Shared memory As before (algorithm described in Figure 1), the shared memory is made up of an array $STATE[1..n]$. The only difference is that now each atomic register $STATE[i]$ is made up of three fields, $STATE[i].old$ and $STATE[i].prop$ (whose content and meaning are as before), plus a boolean $STATE[i].done$. That field, initialized to $false$, is set to true by p_i when it obtains its new name.

[2] If all invocations are such that $X = \Pi$ and $k = 1$, Ω^k_* boils down to the classical leader failure detector (denoted Ω) that is the weakest failure detector that allows solving the consensus problem [9]. Algorithms implementing leader failure detector in shared memory systems are described in [10].

```
operation new_name(id_i):
(1)  prop_i ← ⊥; done_i ← false;
(2)  repeat
(3)    STATE[i] ←< id_i, prop_i, done_i >;
(4)    view_i ← STATE.snapshot();
(5)    if (prop_i ≠ ⊥) ∧ (∀j ≠ i : view_i[j].prop ≠ prop_i)
(6)      then done_i ← true; STATE[i] ←< id_i, prop_i, done_i >
(7)      else contending_i ←
                  {view_i[j].old | (view_i[j].old ≠ ⊥) ∧ ¬(view_i[j].done)};
(8)           leaders_i ← leader(contending_i);
(9)           if id_i ∈ leaders_i then
(10)            let X    = {view_i[j].prop | (view_i[j].prop ≠ ⊥) ∧ (1 ≤ j ≤ n)};
(11)            let free = the increasing sequence 1, 2, ... , 2p − 1 from
                           which the integers in X have been suppressed;
(12)            let r    = rank of id_i in leaders_i;
(13)            prop_i   ← the rth integer in the increasing sequence free
(14)          end if
(15)   end if
(16) until done_i end repeat;
(17) return(prop_i).
```

Fig. 2. An Ω^k_*-based adaptive M-renaming with $M = \min(2p-1, p+k-1)$ [19]

Process behavior The algorithm executed by a process p_i is described in Figure 2. 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 $\perp$ (line 1). Then, it enters a **repeat** loop and leaves it only when it has acquired a new name (lines 6, 16 and 17).

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 p_i has already determined a new name proposal and no other process p_j has chosen the same new name proposal (the predicate of line 5 is then satisfied), p_i commits this last proposal that becomes its new name by announcing it to the other processes (write of $STATE[i]$ at line 6), and returns that proposal as its new name (line 16).

In the other case (the predicate of line 5 is not satisfied), p_i enters the lines 7-14 to determine another new name proposal. To that end, p_i 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 $my\_view_i[j].old \neq \perp$) and have not yet obtained a new name (i.e., such that $\neg(view_i[j].done)$). Before starting the next execution of the loop body, processes have to change their new name proposal (otherwise, it could be possible that they loop forever). So, p_i does the following.

– After it has determined the set of processes it perceives as currently competing with it, p_i invokes leader$(contending_i)$ to obtain a set of leaders (lines 7-8) associated with this set $contending_i$ of competing processes.

– If it does not appear in the current 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 9), p_i determines a new name proposal before starting another execution of the loop body. This determination (done similarly to the previous algorithm, Figure 1) consists for p_i in first computing its rank within the leader set, and then taking as new name proposal the first integer that, from its point of view, is not used by the other processes (lines 12-13).

Size of the new name space The most interesting part of the proof is the part showing that the size of the new name space is $\min(2p-1, p+k-1)$, where p is the number of participating processes. Let p_i be a process that returns a new name (line 17). The new name obtained by p_i is the last name it has proposed (at line 13 during the previous iteration). When p_i defined its last new name proposal, at most $p-1$ other processes have previously defined a name proposal, i.e., $|\{j \ : \ (j \neq i) \wedge (view_i[j].prop \neq \perp)\}| \leq p-1$ (O1). Moreover, due to the definition of Ω^k_*, when it defines its last new name proposal, the rank of p_i in $leaders_i$ is at most $\min(p,k)$ (O2). It follows from the observations (O1) and (O2) that the last new name proposal computed by p_i is upper bounded by $(p-1)+\min(p,k)$, i.e., $M = \min(2p-1, p-1+k)$.

The proof of the termination and agreement properties are similar to the ones of the previous algorithm. They can be found in [19].

6 Enriching the system with a synchronization primitive

This section considers the case where the asynchronous shared memory system is enriched with synchronization objects. This means that, in addition to read/write atomic registers, the shared memory provides the processes with registers that can be accessed by synchronization operations "stronger" than the base atomic read and write operations. Three types of synchronization objects are considered in the following, namely, k-test&set objects, k-set agreement objects, and compare&swap objects.

6.1 Shared memory enriched with k-test&set objects

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

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

The instance $k=1$ does correspond to the usual test&set object proposed by some processors. This object allows to elect exactly process from a set of processes. In our context, as processes can crash, it is possible that some (or even all the) winner processes are faulty.

The power of k-test&set objects when solving renaming A wait-free adaptive algorithm, based on read/write atomic registers and k-test&set objects, that provides a renaming space of size $M = 2p - \lceil \frac{p}{k} \rceil$ is described in [18]. This algorithm results from an incremental construction (k-test&set objects are used to build intermediate k-participating set objects, that are in turn used to build the renaming algorithm). Due to space limitations, this construction is not described here.

It is shown in [12] that $M = 2p - \lceil \frac{p}{k} \rceil$ is the smallest new name space size that can be obtained when one can use atomic registers and k-test&set objects only. It follows that the algorithm described in [18] is optimal as far as the size of the renaming space is concerned.

6.2 Shared memory enriched with k-set agreement objects

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

- Termination. An invocation of $\mathsf{SA\_propose_k}()$ by a correct process terminates.
- Validity. A decided value is a proposed value.
- Agreement. At most k distinct values are decided.

The power of $k$set agreement objects when solving renaming A renaming algorithm, based on atomic registers and k-set agreement objects is described in [11]. The size of the new name space is $M = p + k - 1$ which has been shown to be optimal in [11,12].

It has been shown that the synchronization power of a k-set agreement object is stronger than the one of a k-test&set object [12]. This difference in the synchronization power translates directly in the size of the new name space[3].

6.3 Shared memory enriched with compare&swap objects

Compare&swap objects In a precise sense (based on the consensus number theory [13]), a compare&swap object belongs to class of the strongest synchronization objects. Such an object CS is initialized to some value (say $\perp$) and can be accessed only through an atomic operation denoted $\mathsf{Compare\&Swap}()$ that takes two inputs parameters and returns a value. Its effect can be described by the following specification:

operation $\mathsf{Compare\&Swap}(old, new)$:
$prev \leftarrow CS$; **if** $(CS = old)$ **then** $CS \leftarrow new$ **end if**; $\mathsf{return}(prev)$.

[3] It is easy to see that (1) for $k = 1$ we have $p + k - 1 = 2p - \lceil \frac{p}{k} \rceil$, and (2) $\forall k : \ 1 < k < n - 1, \ \forall p : 1 \leq p \leq n$ we have $p + k - 1 \leq 2p - \lceil \frac{p}{k} \rceil$, and there are values of p such that $p + k - 1 < 2p - \lceil \frac{p}{k} \rceil$.

Optimal renaming space from compare&swap objects It is possible to design a very simple renaming algorithm whose renaming space is $M = p$ (i.e., M is the smallest renaming space that can be obtained whatever the synchronization power we are provided with), as soon as the processes can communicate through atomic read/write registers and compare&swap objects.

```
operation new_name(id_i):
(1) for x from 1 to n do
(2)     r ← CS[x].Compare&Swap(⊥, id_i);
(3)     if (r = ⊥) then return(x) end if
(4) end for.
```

Fig. 3. A (optimal) compare&swap-based adaptive p-renaming algorithm

Such a (very simple) wait-free adaptive renaming algorithm is described in Figure 3. It uses an array $CS[1..n]$ of underlying compare&swap objects, each initialized to $\perp$, and consists of a simple loop. During the xth iteration, the process p_i invokes $CS[x]$.Compare&Swap$(\perp, id_i)$. If it succeeds in switching $CS[x]$ from its initial value $\perp$ to its old name id_i, the process p_i adopts x as its new name and stops looping (line 3). Otherwise, the process p_i proceeds to the next iteration step and then invokes $CS[x+1]$.Compare&Swap$(\perp, id_i)$.

It is easy to see that, at each iteration step, exactly one process "wins" by writing its initial name in the compare&swap object associated with that iteration step. It follows that if p processes want to acquire a new name, at most p iterations of the loop will be executed, hence $M = p$. Let us finally notice that replacing n in the **for** loop by $+\infty$ (line 1), and assuming as many compare&swap base objects as participating processes, gives an adaptive renaming algorithm that works for any number of processes.

7 Conclusion

The aim of this paper was to show that lock-based solutions have inherent drawbacks that make them irrelevant for some applications where live synchronization in presence of asynchrony and process crashes is crucial. To cope with this problem, the wait-free approach has been presented and illustrated with a problem that is paradigm of synchronization in presence of failures and asynchrony, namely, the adaptive renaming problem. A simple solution based on read/write atomic registers has been presented. This solution provides a new name space whose size is $M = 2p - 1$ where p is the number of participating processes. The paper has then shown how this "read/write" lower bound can be circumvented when, in addition to atomic registers, one can benefit from an appropriate failure detector, or from synchronization primitives such as k-test&set, k-set agreement or compare&swap.

The renaming problem has given rise to a large literature. In addition to the papers previously cited in this text, the interested reader can have a look at the following (non-exhaustive) list of articles [2,4,7,17] that shed additional light on the problem.

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., Adaptive and Efficient Algorithms for Lattice Agreement and Renaming. *SIAM Journal of Computing*, 31(2):642-664, 2001.
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. Chandra T. and Toueg S., Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM*, 43(2):225-267, 1996.
9. Chandra T., Hadzilacos V. and Toueg S., The Weakest Failure Detector for Solving Consensus. *Journal of the ACM*, 43(4):685–722, 1996.
10. Fernandez Anta A., Jimenez E., Raynal M., Travers C., A Timing Assumption and two t-Resilient Protocols for Implementing an Eventual Leader Service in Asynchronous Shared Memory Systems. To appear in *Algorithmica*, Springer, 2008.
11. Gafni E., Renaming with k-set Consensus: an Optimal Algorithm in $n+k-1$ Slots. *10th Int'l Conference On Principles Of Distributed Systems (OPODIS'06)*, Springer Verlag LNCS #4305, pp. 36-44, 2006.
12. Gafni E., Raynal M. and Travers C., Test&set, Adaptive Renaming and Set Agreement: a Guided Visit to Asynchronous Computability. *26th IEEE Symposium on Reliable Distributed Systems (SRDS'07)*, IEEE Press, pp. 93-102, 2007.
13. Herlihy M.P., Wait-Free Synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124-149, 1991.
14. Herlihy M.P. and Shavit N., The Topological Structure of Asynchronous Computability. *Journal of the ACM*, 46(6):858-923,, 1999.
15. Herlihy M.P. and Wing J.M., Linearizability: a Correctness Condition for Concurrent Objects. *ACM TOPLAS*, 12(3):463-492, 1990.
16. Lamport. L., On Interprocess Communication, Part II: Algorithms. *Distributed Computing*, 1(2):86-101, 1986.
17. Moir M., Fast, Long-Lived Renaming Improved and Simplified. *Science of Computer Programming*, 30:287-308, 1998.
18. Mostéfaoui A., 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. *20th Symp. on Distributed Computing (DISC'06)*, Springer LNCS #4167, pp. 1-15, 2006.
19. Mostéfaoui A., Raynal M. and Travers C., From Renaming to k-Set Agreement. *14th Int'l Colloquium on Structural Information and Communication Complexity (SIROCCO'07)*, Springer Verlag LNCS #4474, pp. 62-76, 2007.
20. Raynal M. and Travers C., In Search of the Holy Grail: Looking for the Weakest Failure Detector for Wait-free Set Agreement. (Invited talk.) *10th Int'l Conference On Principles Of Distributed Systems (OPODIS'06)*, Springer LNCS #4305, pp. 1-17, 2006.