The notion of Timed Registers and its application to Indulgent Synchronization

Michel Raynal, Gadi Taubenfeld

To cite this version:

Michel Raynal, Gadi Taubenfeld. The notion of Timed Registers and its application to Indulgent Synchronization. [Research Report] PI 1820, 2006, pp.17. <inria-00108970>

HAL Id: inria-00108970
https://hal.inria.fr/inria-00108970

Submitted on 23 Oct 2006

PUBLICATION INTERNE
N° 1820

THE NOTION OF TIMED REGISTERS AND ITS APPLICATION TO INDULGENT SYNCHRONIZATION

MICHEL RAYNAL GADI TAUBENFELD

ISSN 1166-8687

The notion of Timed Registers and its application to Indulgent Synchronization

Michel Raynal[*] Gadi Taubenfeld[**]

Systèmes communicants

Publication interne n ˚ 1820 — Octobre 2006 — 17 pages

Abstract: A new type of shared object, called *timed register*, is proposed and used to design indulgent timing-based algorithms. A timed register generalizes the notion of an atomic register as follows: if a process invokes two consecutive operations on the same timed register which are a read followed by a write, then the write operation is executed only if it is invoked at most d time units after the read operation, where d is defined as part of the read operation. In this context, a timing-based algorithm is an algorithm whose correctness relies on the existence of a bound Δ such that any pair of consecutive constrained read and write operations issued by the same process on the same timed register are separated by at most Δ time units. An indulgent algorithm is an algorithm that always guarantees the safety properties, and ensures the liveness property as soon as the timing assumptions are satisfied. The usefulness of this new type of shared object is demonstrated by presenting simple and elegant indulgent timing-based algorithms that solve the mutual exclusion, ℓ-exclusion, adaptive renaming, test&set, and consensus problems. Interestingly, timed registers are universal objects in systems with process crashes and transient timing failures (i.e., they allow building any concurrent object with a sequential specification). The paper also suggests connections with schedulers and contention managers.

Key-words: Asynchronous shared memory system, Atomic register, Concurrent object, Consensus, Contention manager, Mutual exclusion, Process crash, Renaming, Simplicity, Test&Set, Timing assumption, Timing constraint, Universal object, Wait-free implementation.

(Résumé : tsvp)

[*] IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France, raynal@irisa.fr
[**] Interdisciplinary Center, P.O.Box 167, Herzliya 46150, Israel, tgadi@idc.ac.il

Le concept de registre atomique temporisé et son utilisation pour la synchronisation indulgente

Résumé : Ce rapport présente le concept de registre atomique temporisé. Il montre également son utilisation pour la synchronisation indulgente et la résolution de problèmes d'accord asynchrone.

Mots clés : Systèmes asynchrones, Tolérance aux fautes, Crash de processus, Mémoire partagée, Exclusion mutuelle, Renommage, Consensus, Test&set, Calcul sans attente, Contrainte temporelle.

1 Introduction

Context of the study Concurrent programs are made up of processes that synchronize and cooperate through shared objects (also called *concurrent* objects). These programs are executed on top of an underlying system that usually exhibits a significant degree of synchrony. This synchrony can be translated for the upper layer processes as time upper bounds in such a way that the local algorithms executed by the processes can benefit from these bounds in order to synchronize their behavior or cooperate in a correct way. Very interestingly, such synchrony assumptions allow designing algorithms with properties that could be impossible to guarantee in fully asynchronous systems.

However, program designers cannot afford to risk incorrect behavior in the periods where the synchrony assumptions are occasionally violated. An *indulgent* algorithm is an algorithm designed with (synchrony) assumptions in mind, but never violates its safety property when these assumptions are not satisfied by the underlying system [9, 11, 12, 26]. More precisely, (1) be the assumptions it relies on satisfied or not, such an algorithm never violates its safety property, and (2) it satisfies its liveness property at least when the synchrony assumptions it relies on are satisfied.

Related work (1): timing-based algorithms Lots of synchronization algorithms have been designed that assume an upper bound Δ on the time that can elapse between any two consecutive accesses to the shared memory issued by a process. The bound Δ can be a priori known or not. Such algorithms are built from base atomic registers and are usually called *timing-based* algorithms. One of the very first timing-based algorithms is Fischer's mutual exclusion algorithm. That very simple and elegant algorithm (described in [19]) assumes that Δ is known.

Fundamental results on the minimal cost (in terms of the number of atomic registers used by the algorithm) inherent to timing-based synchronization algorithms are stated and proved in [2, 3, 21]. The interested reader will find in [27] a pedagogical description of several families of timing-based algorithms. Timing-based consensus algorithms are described in [26, 27]. It is important to notice that the timing assumption used in these algorithms is on all the processes and all their shared memory accesses. Moreover, except for very few of them [26], these algorithms are not indulgent.

Content of the paper Indulgent timing-based algorithms are usually difficult to design. This comes from the fact that the safety property associated with the problem that the algorithm aims at solving [1] has to be guaranteed not only when the timing assumption are satisfied, but also when they are not.

The aim of the paper is to explore new ways for the design of efficient, elegant, and provably correct indulgent timing-based algorithms. To that end, a new kind of shared atomic object, called *timed register*, is first proposed, and a matching *timing assumption* is defined. Then, these notions are investigated and used to design two families of indulgent timing-based algorithms, one family is devoted to synchronization problems, while the other focuses on agreement problems.

The concept of timed register generalizes the classical notion of atomic register in the sense that, for each pair made up of a timed register and a process, it imposes time constraints on some write operations in order they be successful. More specifically, given a timed register Y, and a process p that invokes a read operation on Y, let us consider the first write operation on Y issued by p that follows this read (this write is called a *constrained* write, and the read is called a *constraining* read). This write is executed by the timed register (it is successful) only if it occurs at most d time units after the read, where d is a parameter defined as part of the constraining read operation. Let us notice that, not only a process can issue operations on other objects between a read and the following constrained write, but also other processes can access Y.

It is important to notice that timed registers allow restricting the timing assumptions. While classical timing-based algorithms assume a *global* timing assumption that is, for each process, on any pair of consecutive operations (even when executed on different objects, and regardless of their type), the assumption required for a constrained write operation on a timed register to be successful is *local* (in the sense it applies only to a read followed by a write issued by the same process on the same register). It is important to notice that there are no timing dependencies between operations that are invoked on different objects nor between operations other than a read followed by a write issued

[1] *Safety* properties (such as, mutual exclusion, never deciding on conflicting values, if termination claimed then the application has terminated) stipulates that "nothing bad happens", while *liveness* properties (such as, eventually entering a critical section, eventually reaching an agreement, eventually claiming termination if the application terminates) stipulates that "something eventually happens" [17].

by the same process on the same register. Moreover, as already indicated, a process is allowed to access other objects between a read and a write operations on the same timed register. It is worth noticing that, when all the read operations on a timed register Y specify a duration $d = +\infty$, all write operations on Y are successful, and the register Y boils down to a classical atomic register. This shows the way according to which the timed register notion generalizes the timed-free register notion: while the latter has a fully asynchronous sequential specification, the specification of the former is also sequential but involves an additional time duration notion.

In order for a timed register Y to be useful, not all the write operations on Y have to fail. This is captured by what we call the Δ *assumption*. This assumption states that there is a bound Δ such that any constrained write on a timed register Y issued by a process p is executed at most Δ time units after the immediately preceding read issued by p on Y. Consequently, the write is always successful when the duration specified by the read operation is $\geq \Delta$. This means that when Δ is known and the Δ assumption is permanently satisfied by the underlying system, any timed register behaves as an atomic register.

Unfortunately, Δ is not always known, and in a realistic setting, the underlying system does not always guarantee the Δ assumption (it can satisfy it only during "stable" periods). Assuming a bound Δ, a timing failure occurs when a read of a timed register and the following constrained write are separated by more than Δ time units. If this occurs only intermittently, the failure is *transient*. So, in our context, an indulgent timing-based algorithm has first to approximate Δ when this bound in not known. Then, because such an algorithm preserves the safety property even in presence of transient timing failures, there is no harm in the periods where such timing failures occur. As soon as the Δ assumption is again satisfied, the algorithm resumes its "normal" operation. Moreover, on a more engineering side, the fact that an indulgent algorithm tolerates timing failures, allows the system designer to estimate Δ optimistically, and ignore the possible delays resulting from page faults, memory contention and preemption.

As announced previously, the paper addresses two classes of fundamental problems, namely, synchronization problems and agreement problems. More precisely, the paper first presents simple indulgent algorithms based on timed registers that solve synchronization problems, such as mutual exclusion, ℓ-exclusion and adaptive renaming, in system where the bound Δ is known. Then it presents indulgent wait-free consensus algorithms for systems where the bound Δ is known or unknown. As timed registers allow solving, in a wait-free manner, the consensus problem in systems prone to transient timing failure and where the bound Δ is unknown, it follows that timed registers are *universal objects* in these systems, i.e., they allow building any object with an asynchronous sequential specification [13][2].

Related work (2): contention managers and wait-free algorithms A *wait-free implementation* of an object guarantees that any correct process (i.e., a process that does not crash) eventually terminates its operation on the object whatever the behavior of the other processes [13]. A weaker notion that has been proposed is the notion of *obstruction-free implementation* of an object [14]. This notion stipulates that a correct process that invokes an operation on an object, and from some point in time executes alone, must eventually complete its operation. Interestingly, the weakest failure detector class that allows boosting an obstruction-free implementation into a wait-free implementation has recently been identified [10] (it is the class of eventually perfect failure detectors [7]). The notion of *abortable* objects has also been recently introduced [1]. An abortable object behave like an ordinary object when accessed sequentially, but may return $\perp$ when accessed concurrently.

The "undesired" behavior of an obstruction-free object implementation or an abortable object (the operation does not return or returns $\perp$) can only occur in presence of concurrency. In both cases, operation liveness is guaranteed only when a single operation at a time is executed. So these notions are contention-related. They are not time-oriented.

Differently, the timed object notion as introduced in this paper is "dual" in the sense that it is related only to the speed of a process with respect to some objects, i.e., regardless of the current concurrency degree and the speed of the other processes. Moreover, when considering an indulgent timing-based algorithm, the existence of a bound Δ that makes successful the constrained write operations on a timed register, plays the role of a contention manager that guarantees the operations to always terminate.

[2]An object O is universal if any object which has sequential specification has wait-free linearizable implementation using atomic registers and objects of type O. In [13], it is proved that consensus is universal. This implies that any object that can implement wait-free consensus is also universal.

More generally, at some abstraction level, both contention managers and the existence of a bound Δ can be seen as schedulers that provide appropriate fairness rules in order for each operation issued by a correct process to terminate.

Roadmap The paper is made up of 5 sections. Section 2 presents the computational model that consists of timed and time-free atomic registers. That section also defines the notions of timing-based algorithm, timing failure and indulgent algorithm. Then, assuming the bound Δ is known, Section 3 presents indulgent algorithms for basic synchronization problems (mutual exclusion, ℓ-exclusion and adaptive renaming, respectively). The next section, Section 4, focuses on classical agreement problems (test&set and consensus) when the bound Δ is known and when it is unknown. Finally, Section 5 provides concluding remarks.

2 Computational Model

2.1 Process model

The system is made up of a finite number of processes denoted $p_1, p_2, \ldots$. Each process p_i has an index i, and no two processes have the same index. When the number of processes is fixed, n denotes the total number of processes. Moreover, when solving some classes of problems (mainly the class of agreement problems), we assume that a process may crash. A process that crashes stops its execution in a definitive manner. In addition to the usual statements, a process p_i can execute the statement $\mathsf{delay}(d)$ where d is a time duration. Its effect is to delay p_i for an arbitrary but finite period longer than d time units. All the durations are measured with the same time unit. (Notice that this does not require the processes to have synchronized clocks; local clocks with the same drift are sufficient.)

2.2 Communication model

The processes communicate by reading and writing shared registers. (Shared registers are denoted with uppercase letters. Local registers are denoted with lowercase letters with the process index appearing as a subscript.) There are two types of shared registers.

Time-free register A time-free register is nothing else than a classical atomic register which support atomic read and atomic write operations [16, 18].

The statement $\ell_i \leftarrow X$ denotes a read of the shared register X by p_i, and the value obtained is assigned to p_i's local variable ℓ_i. Similarly, $X \leftarrow v$ denotes the writing of the value v into X.

Timed register A timed register supports atomic read and atomic write operations. A read operation of a timed register Y is denoted $Y.\mathsf{read}(d)$ (where d is a time duration), and always returns the current value of Y. A write operation is denoted $Y.\mathsf{write}(v)$.

In order to define the semantics of a write operation on a timed register, let us consider the sequence of read and write operations issued by a process p_i on a timed register Y. A write operation on Y issued by p_i is *constrained* if it immediately follows a read operation by p_i on Y in that sequence (that read operation is the *associated constraining* read).

Let $\mathsf{wr}_i = Y.\mathsf{write}(v)$ be a constrained write, and $\mathsf{rd}_i = Y.\mathsf{read}(d)$ the associated constraining read. The write wr_i is *successful* if it is issued at most d time units after rd_i. In that case, v is written into Y and wr_i returns $true$. Otherwise, the write is not executed and wr_i returns $false$. Timed registers can be implemented from time-free registers and timers[3].

It follows from the previous definition, that a read operation such that $\mathsf{rd}_i = Y.\mathsf{read}(+\infty)$ imposes no constraint on the next write on Y issued by the same process. More generally, as already observed, if all the read operations on a timed register are such that $d = +\infty$, that register behaves as an atomic time-free register.

It is important to notice that, between the rd_i and wr_i operations as defined above, (1) p_i can issue read and write operations on any (time-free or timed) register different from Y, and (2) all the other processes can issue operations on any register (including Y).

[3]Instead of constraining only the first write that appears after a read issued by the same process, one could envisage to constrain all the write operations issued by the same process and that appear between two reads of the same register. As this requirement is stronger than necessary for solving the problems we are interested in, we do not consider here this stronger definition.

Remark A *sticky* bit [23] is a register initialized to a default value $\perp$ that can then contain either 0 or 1. If several processes are concurrently trying to write into a sticky bit, only one of them succeeds. A write operation returns *failure* if the value it is trying to write disagrees with the already written value, and *success* otherwise. Both timed registers and sticky bits are atomic objects that provide the processes with a write operation that returns a boolean value. But, they are objects of different types as, while a sticky bit is not time-constrained but allows the write of a single value only to win, a timed register is time-constrained but does not prevent different values from being successfully written.

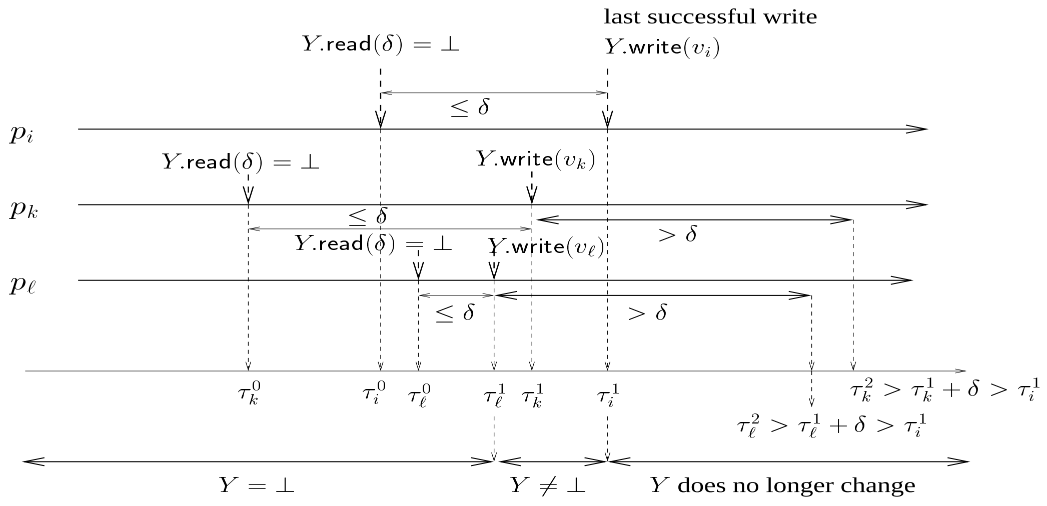
Read/write pattern associated with a timed register Let Y be a timed register initialized to a default value $\perp$, and δ a time duration. The following read/write pattern appears in a lot of algorithms based on timed registers: "**if** $(Y.\mathsf{read}(\delta) = \perp)$ **then** $Y.\mathsf{write}(v)$ **end if**"[4]. This simple pattern is very often embedded in the following more general pattern made up of a small set of statements. This pattern (where v_i is the value that p_i intends to write into Y) can be executed concurrently by several processes (it implicitly assumes that Y is not used elsewhere). As we will see, in one way or another, all our algorithms are based on this pattern.

```
while $(Y.\mathsf{read}(\delta) = \perp)$ do $Y.\mathsf{write}(v_i)$ end while; % Here $Y \neq \perp$ %
$\mathsf{delay}(\delta)$ % Here $Y$ has its definitive value %.
```

Interestingly, this pattern has the following property.

Theorem 1 *Let us assume that the timed register Y is not accessed outside the pattern, the pattern being executed by any number of processes, and $v_i \neq \perp$ for all i. (1) If a process reads Y after the repeat statement, it always obtain a non-$\perp$ value. (2) All the processes that read Y after the delay statement obtain the same non-$\perp$ value.*

Proof Item (1) follows directly from the $Y.\mathsf{read}(\delta)$ operation that allows a process to exit the loop, and the fact that no process writes $\perp$.



last successful write
$Y.\mathsf{write}(v_i)$
$Y.\mathsf{read}(\delta) = \perp$
$\leq \delta$
p_i
$Y.\mathsf{read}(\delta) = \perp$
$Y.\mathsf{write}(v_k)$
p_k
$\leq \delta$
$> \delta$
$Y.\mathsf{read}(\delta) = \perp$
$Y.\mathsf{write}(v_\ell)$
p_ℓ
$\leq \delta$
$> \delta$
τ_k^0
τ_i^0
τ_ℓ^0
τ_ℓ^1
τ_k^1
τ_i^1
$\tau_k^2 > \tau_k^1 + \delta > \tau_i^1$
$\tau_\ell^2 > \tau_\ell^1 + \delta > \tau_i^1$
$Y = \perp$
$Y \neq \perp$
Y does no longer change

Figure 1: Read/write pattern on the timed register Y

To prove item (2), let us first consider the set W of the processes that writes successfully into Y (in the **if** statement). Among them, let p_i be the last process that writes into Y. Due to the timing constraint associated with successful write operations, this set of processes is not empty (See Figure 1).

For any $p_k \in W$, let us consider the following time definitions (illustrated in Figure 1 with $W = \{i, k, \ell\}$).

- τ_k^0: time of the last reading of Y by p_k that returns $\perp$ (just before the successful write).
- τ_k^1: time of the successful write by p_k into Y.
- τ_k^2: time at which p_k issues a read operation after it has executed the $\mathsf{delay}()$ statement.

[4]This pattern shows that a pair made up of a constraining read and the corresponding constrained write operation on a timed register, can be seen as a $\mathsf{compare\&swap}()$ operation, where (1) the atomicity of which has been broken by "cutting" the $\mathsf{compare\&swap}()$ operation in two operations, and (2) such that the second operation is executed only if timing constraints are satisfied.

We have, for any $p_k \in W$:

- $\tau_k^2 > \tau_k^1 + \delta$ (this follows from the $\mathsf{delay}()$ statement).
- $\tau_k^1 > \tau_i^0$ (otherwise, p_i would have read $Y \neq \perp$), from which we have: $\tau_k^1 + \delta > \tau_i^0 + \delta$.
- $\tau_i^0 + \delta \geq \tau_i^1$ (because the write of Y by p_i in the **if** statement is successful).
- $\tau_k^2 > \tau_k^1 + \delta > \tau_i^0 + \delta \geq \tau_i^1$ (combining the previous items).

Finally, combining $\tau_k^2 > \tau_i^1$ and the fact that p_i is the last process that writes into Y, it follows that any process $p_k \in W$ reads the value written by p_i.

Let us now consider a process p_j such that $p_j \notin W$. This means that the first read of Y by p_j (in the if statement) returns a non-$\perp$ value. Let τ_j' be the time at which this read operation occurs. Let τ_ℓ^1 be the time at which the first process of W writes Y. We have $\tau_j' > \tau_\ell^1$. We have shown previously that $\tau_\ell^1 + \delta > \tau_i^1$ (recall that p_i is the last process that writes into Y), from which it follows that when p_j reads Y after the delay statement, i.e., at a time τ_j^2, we have $\tau_j^2 > \tau_j' + \delta > \tau_\ell^1 + \delta > \tau_i^1$. It follows that p_j reads the last value written into Y, i.e., the value written by p_i.
$\square_{Theorem\ 1}$

2.3 Timing assumption, timing failure and indulgent timing-based algorithm

Timing assumption A system provides a timing assumption if there is a bound Δ such that, for each timed register Y and each process p_i, the time that separates a $\mathsf{rd}_i = Y.\mathsf{read}(d)$ operation and the associated constrained write $\mathsf{wr}_i = Y.\mathsf{write}(v)$ (if any) are separated by at most Δ. That bound can be known or not known. If it is known, p_i can use it in the read operation, namely it can always invoke $\mathsf{rd}_i = Y.\mathsf{read}(\Delta)$. Otherwise, p_i has to approximate it by modifying the value of d until it becomes $\geq \Delta$. When $d \geq \Delta$, all the constrained write operations become successful in the absence of timing failures.

In our context, a timing-based algorithm is an algorithm whose liveness property relies on the existence of a such a bound Δ. Nearly all the timing-based algorithms encountered in the literature assume that the bound Δ holds permanently from the very beginning and is on any pair of consecutive operations issued by a process. Differently, in this paper, there is here no timing assumption when a process invokes operations on time-free objects, on different timed objects, or when a write follows another write on the same timed object.

Timing failure and indulgent timing-based algorithm A timing failure refers to the situation where the timing assumption the underlying system should provide the processes with, is not satisfied. Here, this means that there are periods during which there is no bound Δ such that at most Δ time units separate any constrained write from its associated constraining read operation.

A timing-based algorithm is indulgent with respect to transient timing failures, if it never violates the safety property of the problem it has to solve even in the presence of timing failures, and satisfies its liveness property if, after some time, there are no more timing failures.

Remark In systems which allow processes to be preempted (as almost all modern computing systems do), Δ must include the time spent preempted between two statements. Furthermore, in determining the value of Δ, possible cache misses, page faults, and delays due to memory contention should be taken into account. Thus, the value of Δ in most cases must be very large. However, because indulgent algorithms ensure correctness even when timing assumptions are violated, Δ can be estimated optimistically, ignoring possible delays resulting from page faults, memory contention and preemption. The exact value of Δ, when evaluated optimistically, should be tuned for each individual machine architecture. It can also be changed over time if the algorithm fails to make sufficient progress.

3 Indulgent Synchronization

This section focuses on the use of timed registers for the design indulgent of timing-based synchronization algorithms. A simple mutual exclusion is first presented. This algorithm is then used as a building block to design an $(\ell - 1)$-

resilient indulgent solution to the ℓ-exclusion problem. The ℓ-exclusion algorithm is in turn used as a building block to design a wait-free solution to the adaptive renaming problem, that is optimal with respect to the new name space.

3.1 An indulgent mutual exclusion algorithm with a known bound

The algorithm The *mutual exclusion* problem is well-known. We consider here its version where (1) no two processes can be in their critical section at the same time (safety property), and (2) if one or more processes want to enter their critical section, at least one of them will succeed (deadlock freedom property).

An indulgent mutual exclusion algorithm is described in Figure 2. It uses a single timed register. This algorithm is actually a simple adaptation of Fischer's non-indulgent algorithm (as described in [19]) in which the time-free atomic register is replaced by a timed register (denoted Y, and initialized to $\perp$). The statement "**await** *condition*" is used as an abbreviation for "**while** $\neg$*condition* **do** *skip*". A process p_i first waits until $Y = \perp$ ($Y \neq \perp$ mean that at least one process is already competing for the critical section). It then assigns its index i to Y. In case of a timing failure, the constrained write fails, and p_i starts all over again. Otherwise the write operation is successful. In that case, p_i delays itself for at least Δ time units. If $Y = i$ after that period, p_i enters the critical section. Otherwise, p_i repeats the procedure. Releasing the critical section is done by resetting Y to $\perp$. Let us observe that only p_i can set Y equal to i.

```
operation enter_mutex($i$):
(1) repeat
(2)     await $\big(Y.\mathsf{read}(\Delta) = \perp\big)$;
(3)     if $Y.\mathsf{write}(i)$ then $\mathsf{delay}(\Delta)$ end if
(4) until $\big(Y.\mathsf{read}(\infty) = i\big)$ end repeat

operation exit_mutex():
(5) $Y.\mathsf{write}(\perp)$
```

Figure 2: An indulgent mutual exclusion algorithm with a known bound (code for p_i)

Theorem 2 *The algorithm described in Figure 2 always satisfies the mutual exclusion safety property. It satisfies the deadlock freedom property as soon as there are no more timing failures.*

Proof Mutual exclusion property. Let us consider a set of processes that concurrently execute $\mathsf{enter\_mutex}()$. Among them, the process p_k such that $Y.\mathsf{read}(\Delta) = \perp$ at line 2, but whose $Y.\mathsf{write}(k)$ operation fails (line 3) cannot enter the critical section as $Y = k$ is a necessary requirement for p_k to enter its critical section[5]. So, let us consider the processes that succeed in writing their index into Y (line 3). They all are delayed for at least Δ time units (line 3). The execution pattern associated with these competing processes is an instance of the read/write pattern exhibited in Section 2.2. It then follows from Theorem 1 that all these processes read the same value (say i) at the end of their delay period ($Y.\mathsf{read}(\infty) = i$ at line 4). Consequently, only p_i is allowed to enter its critical section.

Deadlock freedom property. Assuming a time τ after which there are no more timing failures, let us consider a time τ', $\tau' \geq \tau$, at which one or more processes want to enter their critical section. The reasoning to show that one process eventually enters its critical section is exactly the same as for proving the mutual exclusion property, with the additional observation that now, as there are no more timing failures, all the $Y.\mathsf{write}(k)$ operations issued at line 3 are successful. The previous proof has shown that, in such a case, one process enters its critical section. $\Box_{Theorem\ 2}$

Additional properties of the algorithm The previous indulgent mutual exclusion algorithm enjoys the following additional properties (where the index i of p_i is considered as its identifier).

- It works for any number of processes (assuming the processes have unique identifiers).
- It is symmetric. The only way for distinguishing processes is by comparing their identifiers, which are unique. Identifiers can only be written, read and compared for equality (they are not used to index shared registers.)

[5]Let us observe that, if permanently there are timing failures, all $Y.\mathsf{write}(k)$ operations may fail.

- It follows from a result in [21] that any indulgent timing-based mutual exclusion algorithm for n processes, using only time-free atomic registers must use at least n atomic registers, even when the bound Δ is known. An indulgent timing-based mutual exclusion algorithm was presented in [26], which uses $O(n)$ atomic registers where n is the number of processes.

  The indulgent mutual exclusion algorithm presented in Figure 2 uses only one timed register. It is consequently optimal with respect to the number of timed registers. This shows that the timed register notion and the associated timing assumption can help circumventing the lower bound on the number of registers required for solving the mutual exclusion problem.

- As for the case where the bound is *not* known, it can easily be shown, by modifying the lower bound proof from [5], that any indulgent timing-based mutual exclusion algorithm for n processes using timed registers and atomic registers must use at least n such registers. There exist many asynchronous mutual exclusion algorithms using atomic registers only that match this $\Omega(n)$ space bound [24, 27], thus using timed registers does not seem to help in the case where the bound Δ is unknown.

3.2 An indulgent ℓ-exclusion algorithm with a known bound

The problem The *ℓ-exclusion* problem is a natural generalization of the mutual exclusion problem: it consists in designing an algorithm that guarantees that up to ℓ processes and no more may simultaneously access identical copies of the same non-sharable resource (that is, ℓ processes are permitted to be simultaneously in their critical section [6].). A solution is required to withstand the slow-down or even the crash (fail by stopping) of up to $\ell - 1$ processes (i.e., it has to be $(\ell-1)$-resilient).

A good example is that of a bank where people are waiting for a teller. Here the processes are the people, the resources are the tellers, and the parameter ℓ is to the number of tellers. We notice that the usual bank solution, where people line up in a single queue, and the person at the head of the queue goes to any free teller, does *not* solve the ℓ-exclusion problem. If $\ell \geq 2$ tellers are free, a proper solution should enable the first ℓ people in line to move simultaneously to a teller. However, the bank solution, requires them to move past the head of the queue one at a time. Moreover, if the person at the front of the line "fails", then the people behind this person wait forever.

An ℓ-exclusion algorithm must guarantees that at most ℓ processes are simultaneously in their critical sections (safety property), and, when strictly less than ℓ processes fail (or are delayed forever), if processes (at least one) are trying to enter their critical sections, then at least one process will enter it (deadlock-freedom property).

The algorithm Solving the ℓ-exclusion problem using atomic registers only requires, as in the case of mutual exclusion, at least n such registers. In the following, we present a simple ℓ-exclusion algorithm using only ℓ timed-registers. The algorithm (described in Figure 3) makes use of ℓ (slightly modified) copies of the mutual exclusion algorithm from the previous section (Figure 2). Those are indexed from 0 to $\ell - 1$. The c-th copy uses a single timed register denoted $Y[c]$. In its entry code, a process (say p_i) starts participating in one of these copies (say c_i). Process p_i is allowed to enter its critical section when it wins in one of the ℓ copies.

```
operation enter_ℓ-exclusion($i$):
(1) repeat
(2)     while $(Y[c_i].\mathsf{read}(\Delta) \neq \perp)$ do $c_i \leftarrow c_i + 1 \pmod{\ell}$ end while;
(3)     if $Y[c_i].\mathsf{write}(i)$ then delay$(\Delta)$ end if
(4) until $(Y[c_i].\mathsf{read}(\infty) = i)$ end repeat

operation exit_ℓ-exclusion($c_i$):
(5) $Y[c_i].\mathsf{write}(\perp)$
```

Figure 3: An indulgent ℓ-exclusion algorithm with a known bound (code for p_i)

More specifically, to enter its critical section, a process p_i executes the following statements where c_i is a local variable (the initial value of which is arbitrary):

[6]The 1-exclusion problem is exactly the mutual exclusion problem.

- p_i first looks for a free "slot" c_i, i.e., a copy $Y[c_i]$ such that $Y[c_i] = \bot$. This means that p_i moves to the next copy $c_i + 1 \pmod \ell$ if $Y[c_i] \neq 0$. (Differently from the mutual exclusion algorithm, p_i does not busy-wait on a single copy but on all the copies.)

- As in the base mutual exclusion algorithm, p_i assigns its index i to $Y[c_i]$ as soon as it finds a copy such that $Y[c_i] = \bot$. In case of a timing failure, since $Y[c_i]$ is a timed register, the write operation can fail. If this is the case, p_i restarts from the beginning.

- Otherwise the write operation succeeds, and, accordingly, p_i delays itself for a duration d, $d \geq \Delta$. If after this delay period, $Y[c_i]$ has not been changed (i.e., $Y[c_i] = i$), p_i is attributed the copy c_i, otherwise it moves to the next copy.

As before, releasing the critical section is done by setting $Y[c_i]$ to $\bot$. The reader can notice that, as the base mutual exclusion algorithm, this algorithm works for any number of processes.

The proof that the algorithm satisfies the safety and deadlock freedom properties, are similar to the proofs of the mutual exclusion algorithm. They are left as an exercise.

3.3 An indulgent adaptive renaming algorithm with a known bound

The problem For this problem, we assume that there are n processes. Moreover, an arbitrary number of these n processes can crash.

In the *long-lived M-renaming* problem, each process p_i has an initial name id_i, that is a value from a large space $[1..N]$ (i.e., such that $n << N$). No two processes have the same initial name. A process p_i can invoke two operations $\mathsf{get\_name}(id_i)$ and $\mathsf{release\_name}()$. The former operation provides p_i with a new name new_i, while the latter allows p_i to release the new name it has previously acquired. At any time, any two processes that have invoked $\mathsf{get\_name}(id_i)$ (and not yet invoked $\mathsf{release\_name}()$) have distinct new names (safety property). Moreover, let p be the number of processes that have currently invoked $\mathsf{get\_name}(id_i)$ (and not yet invoked $\mathsf{release\_name}()$). These are the processes that (1) have acquired a new name and not yet released it, plus (2) the processes that are currently competing to acquire a new name. The current new name space must be $[1..M]$ where M is a function of p (adaptivity property). Ideally, this means that, when x new names have been attributed and not yet released and a single process p_i wants to acquire a new name, it has to obtain the name $x + 1$, whatever its index i.

A solution to the renaming problem is required to be *wait-free*. This means that it should guarantee that every participating process (that does not crash) will always be able to get a new name in a finite number of steps regardless of the behavior of other processes [13]. That is, the solution has to tolerate any number of process crash failures. It is proved in [6, 15] that, when one can use only time-free atomic registers, $2p - 1$ is a lower bound on the new name space for wait-free solving the renaming problem, where p ($p \leq n$) is the number of processes that want concurrently acquire new names.

The M-renaming problem with $M = p$ is is consequently unsolvable when one can use atomic registers only. This section shows that the p-renaming problem can be solved with timed registers. Said differently, timed registers not only allow circumventing the $2p - 1$ lower bound of the adaptive renaming problem, but allow designing an optimal solution as far as the size of the new name space is concerned.

The algorithm The adaptive algorithm based on timed registers is described in Figure 4. A new name can be seen as a resource that a process can acquire and then release. It follows from this simple observation that an indulgent wait-free adaptive renaming algorithm can be designed by adapting the previous ℓ-exclusion indulgent algorithm. The only difference is the number of resources: here there are n possible new names, so, instead of an array with $\ell < n$ entries, the algorithm uses an array of n timed registers $Y[1..n]$, such the register $Y[c]$ controls the assignment of the new name c.

As n new names are possible, and there are n processes, whatever its initial name, a process that invokes $\mathsf{get\_name}(id_i)$ cannot be blocked by other processes when it wants to acquire a new name. It follows that the algorithm is trivially wait-free. The number of steps a process has to execute to obtain a new name cannot be bounded, as it depends on the occurrence of timing failures. When (1) there are no timing failures, (2) no new name is attributed, and (3) p processes concurrently want to acquire a new name, a process has to executes at most p times the **repeat** loop (lines 2-5).

```
operation get_name($id_i$):
(1)  $c_i \leftarrow 1$;
(2)  repeat
(3)      while $\big(Y[c_i].\mathsf{read}(\Delta) \neq \bot\big)$ do $c \leftarrow c_i + 1$ end while;
(4)      if $Y[c_i].\mathsf{write}(id_i)$ then $\mathsf{delay}(\Delta)$ end if
(5)  until $\big(Y[c_i].\mathsf{read}(\infty) = id_i\big)$ end repeat;
(6)  return $(c_i)$

operation release_name($c_i$):
(7)  $Y[c_i].\mathsf{write}(\bot)$
```

Figure 4: An indulgent adaptive renaming algorithm with a known bound (code for p_i)

The algorithm adaptivity is an immediate consequence of the fact that each process starts always scanning the new name space from 1. More generally, the proof that the algorithm is correct follows from the ℓ-exclusion algorithm instantiated with $\ell = n$.

4 Indulgent Agreement

This section explores the use of timed registers to design wait-free indulgent test&set and consensus algorithms. In both cases, algorithms are given for the case where the bound Δ is known and the case where it is unknown. A main result of this section, is the fact that timed registers are universal objects in the system model described in Section 2, which means that, in these systems, they allow wait-free implementation of any object with a sequential specification.

4.1 Indulgent test&set algorithms

The problem Test&set bits are well known atomic objects that are usually implemented in hardware by many computer architectures. In this section we show how to implement test&set bits from timed registers. It is important to notice that the other direction is not possible. Namely, there are problems that are solvable using timed registers which are not solvable using only test&set bits and time-free atomic registers. One such problem, the wait-free consensus problem (which is covered in the next subsections) can be solved only for two processes using test&set bits and time-free atomic registers, while using timed registers enables to solve it for any number of processes.

A *test&set* object TS is a bit with initial value 1, that supports an atomic reset operation (denoted $TS.\mathsf{reset}()$) and an atomic test&set operation (denoted $TS.\mathsf{test\&set}()$). $TS.\mathsf{reset}()$ is equivalent to writing 1 into TS. $TS.\mathsf{test\&set}()$ atomically returns the current value of TS and then sets it to 0.

It is easy to see that, until $TS.\mathsf{reset}()$ is invoked, the first invocation of $TS.\mathsf{test\&set}()$ returns 1, while the following invocations return 0. The process that obtains the value 1 is usually called the *winner*. Let us notice that the familiar *leader election* problem can be easily solved with a test&set object Y, the winner process is the elected leader.

An indulgent test&set algorithm with a known bound The algorithm described in Figure 5 uses a single timed register named Y (initialized to $\bot$) in order to wait-free implement a test&set object TS.

To implement $TS.\mathsf{test\&set}()$, a process p_i first tests Y. If $Y = \bot$, it assigns its index i to Y. As a result of a timing failure, that write operation may fail. If there is no timing failure, the write operation succeeds and p_i delays itself. This procedure repeats until $Y \neq \bot$. At this point, observing that the algorithm implementing $TS.\mathsf{test\&set}()$ follows the basic read/write pattern exhibited in Section 2.2, it follows from Theorem 1, that the value of Y does not change until it is reset. Once $Y \neq \bot$, the process p_i such that $Y = i$ becomes the winner, while the other processes lose. The reset operation is implemented as before by resetting Y to its initial value $\bot$.

The most common correctness requirement used for implementing shared objects is *linearizability*, which means that although operations of concurrent processes may overlap, each operation should appear to take effect instantaneously at some time between its first and last event [16]. In the implementation of Figure 5, the linearization point of each test&set operation, for all the processes except the winner, is exactly when the test $Y.\mathsf{read}(\Delta) = \bot$ at line 1 fails; the linearization point of the test&set operation of the winner is when it starts.

```
operation $TS$.test&set():
(1) while $(Y.\mathsf{read}(\Delta) = \bot)$ do
(2)     if $Y.\mathsf{write}(i)$ then $\mathsf{delay}(\Delta)$ end if
(3) end while;
(4) if $Y.\mathsf{read}(\infty) = i$ then $\mathsf{return}(1)$ else $\mathsf{return}(0)$ end if

operation $TS$.reset():
(5) $Y.\mathsf{write}(\bot)$
```

Figure 5: Indulgent implementation of a test&set bit with known bound (code for p_i)

An indulgent test&set algorithm with an unknown bound This section presents our first algorithm where the bound Δ is not known. The technique that is used is both simple and general enough to be employed again in the following sections to design wait-free consensus algorithms with an unknown bound. That technique, that is very classical, consists in computing successive approximations of the unknown bound.

```
operation $TS$.test&set():
(1) while $(Y.\mathsf{read}(d_i) = \bot)$ do
(2)     if $Y.\mathsf{write}(i)$ then $\mathsf{delay}(\max(\{DELAY[k]_{1\le k\le n}\}))$
(3)                        else $d_i \leftarrow d_i + 1$; $DELAY[i] \leftarrow d_i$
(4)     end if
(5) end while;
(6) % this line can be added for efficiency only: $DELAY[i] \leftarrow 1$; $d_i \leftarrow \lceil \frac{d_i}{2} \rceil$; %
(7) if $Y.\mathsf{read}(\infty) = i$ then $\mathsf{return}(1)$ else $\mathsf{return}(0)$ end if

operation $TS$.reset():
(8) $Y.\mathsf{write}(\bot)$
```

Figure 6: Indulgent implementation of a test&set object with unknown bound (code for p_i)

The algorithm is described in Figure 6. The shared array $DELAY[1..n]$ is an array of atomic time-free registers such that $DELAY[i]$ contains p_i's current estimate of the unknown bound; d_i is a local variable of process p_i initialized to 1. When we look at the algorithm of Figure 5, we observe that Δ is used only at line 2, so that line has to be appropriately modified to cope with the fact that Δ is not known. Now, a write operation may fail not only because there is a timing failure, but also because the current estimate is smaller than Δ. If the write operation fails the process increases its estimate for Δ and tries again. If it succeeds, p_i delays itself for a duration equal to the maximum estimate of Δ taken over the estimates of all the processes, i.e., $\max(\{DELAY[k]_{1\le k\le n}\})$. So, line 2 of Figure 5 is replaced by the lines 2-4 in Figure 6.

We notice that, as a result of timing failures, p_i's estimate of Δ may be much higher than its actual value. Thus, in line 6 of the algorithm, d_i is reset to $\lceil \frac{d_i}{2} \rceil$, so that p_i can better estimate Δ next time. Also, in line 6, $DELAY[i]$ is set to 1 so that in the future it will not slow down other processes when performing the delay statement in line 2. The proof of this algorithm is similar to the proof of the "corresponding" consensus algorithm presented in Section 4.3. So, only that second proof is given.

4.2 Indulgent multi-valued consensus algorithms with a known bound

The problem The *consensus* problem consists in designing an algorithm in which all the correct processes (i.e., the processes that do not crash) reach a common decision based on their initial opinions. Each process p_i proposes a value v_i (its opinion), and the processes have to decide in such a way that (1) each correct process eventually decides (termination property), (2) no two processes decide on different values (agreement) [7], and (3) the decided value is a proposed value (validity).

The consensus is *binary* when the set of values that can be proposed contains only two values, otherwise it is *multi-valued*. It is well-known that the consensus problem cannot be solved in asynchronous systems made up of atomic

[7] This property is sometimes called *uniform* agreement, as it prevents a process that crashes from deciding differently from the correct processes.

registers only, when assuming that processes may crash [8, 13, 20]. This impossibility can be circumvented by using stronger synchronization operations (such as compare&swap()) [13], unreliable failure detectors [7, 25], restricting the input vectors that the processes can collectively propose [22], or weakening the termination property [4].

A simple wait-free consensus algorithm It is interesting to note that for the known bound model, an indulgent wait-free timing-based consensus algorithm was presented in [26], which uses infinitely many time-free atomic registers. The algorithm presented in Figure 7 is surprisingly simple and uses a single atomic timed register Y (initialized to $\perp$). A process decides when it executes the return() statement at line 3. As the reader can notice this algorithm is nothing else than the basic read/write pattern exhibited in section 2.2, and consequently its correctness follows directly from Theorem 1.

```
operation consensus($v_i$):
 (1)  while ($Y$.read($\Delta$) $= \perp$) do $Y$.write($v_i$) end while;
 (2)  delay($\Delta$);
 (3)  return($Y$.read($\infty$))
```

Figure 7: An indulgent consensus algorithm with a known bound (code for p_i)

As it can be observed (and similarly to the previous agreement algorithms), this algorithm works for any number of processes. Moreover, differently from failure detector-based consensus algorithms, this algorithm does not use rounds. These two noteworthy features are common to all the agreement algorithms presented in the paper.

A simple fast wait-free consensus algorithm The previous algorithm is not *fast* in the sense that a process has always to be delayed before deciding (see line 2). A nice property of an agreement algorithm is to allow for a fast decision in "good" circumstances. Here, those are defined as being the cases where a single value is proposed.

A fast wait-free indulgent consensus algorithm is described in Figure 8. In addition to the timed register Y, this algorithm uses an array of time-free atomic boolean register $X[1..b]$, where b is the number of input values that can be proposed. (To simplify the presentation we assume that the values that can be proposed are the values $1, 2, \dots, b$.) For each v, the flag $X[v]$, initialized to *false*, is set to $true$ when a process proposes the value v (line 1). The array $X[1..b]$ is then used (line 3) by a process p_i to known if a value v different from its proposal v_i has been proposed. If it is the case, p_i delays itself before deciding. Otherwise it decides immediately (line 4). It is easy to see that when a single

```
operation consensus($v_i$):
 (1)  $X[v_i] \leftarrow true$;
 (2)  while($Y$.read($\Delta$) $= \perp$) do $Y$.write($v_i$) end while;
 (3)  if ($\exists v :\ v \neq v_i \wedge X[v]$) then delay($\Delta$) end if;
 (4)  return($Y$.read($\infty$))
```

Figure 8: An indulgent fast consensus algorithm with a known bound (code for p_i)

value v is proposed, the atomic register $X[v]$ only is set, and consequently, no process is delayed. Moreover, whatever the number of distinct values that are proposed, if there is no timing failure, a process decides after 2 or 3 accesses to the timed register Y, and b accesses to the array of time-free registers $X[1..b]$ (which means 4 or 5 accesses for binary consensus).

Theorem 3 *The algorithm described in Figure 8 is a wait-free indulgent consensus algorithm.*

Proof The proof of this algorithm is verbatim the proof of Theorem 4 after having replaced the δ_α durations by the constant Δ. $\square_{Theorem\ 3}$

4.3 Indulgent fast consensus algorithms with an unknown bound

This section focuses on the design of fast consensus algorithms built from timed registers with an unknown bound. Such an algorithm is described in Figure 9. It is basically the same as the previous algorithm (where the bound Δ

is known), with the addition of a local variable d_i (initialized to 1), and a shared variable $DELAY[i]$ per process p_i. These variables allow the processes to compute approximations of the unknown bound Δ.

```
operation consensus($v_i$):
(1) $X[v_i] \leftarrow true$;
(2) while $\big(Y.\mathsf{read}(d_i) = \perp\big)$ do
(3)         if $\neg\big(Y.\mathsf{write}(v_i)\big)$ then $d_i \leftarrow d_i + 1$; $DELAY[i] \leftarrow d_i$ end if
(4) end while;
(5) if $(\exists v : v \neq v_i \wedge X[v])$ then $\mathsf{delay}(\max(\{DELAY[k]_{1\leq k\leq n}\}))$ end if;
(6) $\mathsf{return}(Y.\mathsf{read}(\infty))$
```

Figure 9: An indulgent fast consensus algorithm with an unknown bound (code for p_i)

Due to the presence of the array $DELAY[1..n]$, this algorithm requires the knowledge of n. This array can be replaced by a single atomic variable (denoted $DELAY$ and initialized to 1) where this variable is provided with an atomic $\mathsf{incr}()$ operation (i.e., $DELAY.\mathsf{incr}()$ atomically increases $DELAY$ by 1). (It is important to recall that such an atomic operation does not enable solving consensus for an arbitrary number of processes [13].) Using such an operation, we obtain the fast consensus algorithm with an unknown bound described in Figure 10. Interestingly, this modified algorithm works for any number of processes. (The reader can notice that the structure of this algorithm is the same as the one of the fast consensus algorithm described in Figure 8, where the bound Δ is known.)

```
operation consensus($v_i$):
(1) $X[v_i] \leftarrow true$;
(2) while $\big(Y.\mathsf{read}(DELAY) = \perp\big)$ do
(3)     if $\neg\big(Y.\mathsf{write}(v_i)\big)$ then $DELAY.\mathsf{incr}()$ end if
(4) end while;
(5) if $(\exists v : v \neq v_i \wedge X[v])$ then $\mathsf{delay}(DELAY)$ end if;
(6) $\mathsf{return}(Y.\mathsf{read}(\infty))$
```

Figure 10: Indulgent consensus with unknown bound, for any number of processes (code for p_i)

It is easy to see that the algorithm is fast: regardless of timing failures when a single value is proposed, no $\mathsf{delay}()$ statement is executed. Moreover, a soon as $DELAY \geq \Delta$, with no timing failures, a process decides after b accesses to the array $X[1..b]$, 2 or 3 accesses to the timed register Y, and 2 accesses to the atomic register $DELAY$.

Theorem 4 *The algorithm described in Figure 10 is a wait-free indulgent consensus algorithm.*

Proof Validity property. That property follows directly from the fact that a decided value is a value that has been written into Y (line 6), and only proposed values can be written into Y (line 3).

Wait-free (termination) property. Let us first first observe, that the $DELAY$ counter is increased until a write into Y is successful (lines 2-4). Moreover, as soon as a value is written in Y, the processes are no longer prevented from deciding. So, let us assume that no value is ever written into Y. This means that $DELAY$ is increased and after some time τ_1 we have $DELAY \geq \Delta$. Let τ_2 be a time after which there are no more timing failures. Finally, let $\tau \geq \max(\tau_1, \tau_2)$. Clearly, after τ, the first constrained write into Y by a process (line 2) succeeds, and from now we have forever $Y \neq \perp$, contradicting the initial assumption.

Agreement property. We have to show that no two processes decide different values. If no process ever writes into Y, the agreement property is trivially satisfied. So, let us assume that at least one process successfully writes into Y. Let $W = \{p_x, p_y, \dots\}$ be the set of the processes that successfully write into Y. Moreover, let p_x be the first process that writes Y. Let us observe that, after p_x has written v_x into Y, the processes that execute line 2, find $Y \neq \perp$ and are consequently prevented from writing into Y. So, there is a "last" value v written into Y. (As the successful write operations on Y are atomic, and we assume an arbitrary but finite number of processes, the notions of "first" and "last" are well-defined). We show that the last value written into Y is the only value that can be decided by the processes. In the following p_α is any process of W, or any process of W, distinct from p_x, when both p_x and p_α are used. Let p_y be any process $\notin W$.

- Time instant definitions.

  1. Let τ_α^{-1} be the time at which the write operation $X[v_\alpha] \leftarrow true$ is executed by p_α (line 1).
  2. Let τ_α^0 be the time of the last reading of Y by p_α. (As noticed before, such a read returns $\perp$.)
     Let δ_α be the value of $DELAY$ used by p_α in this last read statement.
  3. Let τ_α^1 be the time of the successful write of Y by p_α. We have $\tau_\alpha^1 - \tau_\alpha^0 \leq \delta_\alpha$.
  4. Let τ_α^2 be the time at which p_α starts reading the array $X[1..b]$. We have $\tau_\alpha^{-1} < \tau_\alpha^0 < \tau_\alpha^1 < \tau_\alpha^2$. Let τ_y^2 be the time at which p_y starts reading the array $X[1..b]$.

- As p_α reads $\perp$ from Y at τ_α^0 and p_x writes it at τ_x^1, we have $\tau_\alpha^{-1} < \tau_\alpha^0 < \tau_x^1 < \tau_\alpha^1 < \tau_\alpha^2$, from which we conclude that, $\forall\, p_\alpha,\ p_\beta \in W$: $\tau_\alpha^{-1} < \tau_\alpha^0 < \tau_\beta^2$.

  Similarly, for a process p_y that does not write Y, we have $Y \neq \perp$ when p_y reads Y at line 2 for the last time, and consequently we have $\tau_x^1 < \tau_y^2$, from which we conclude that $\forall\, p_\alpha$: $\tau_\alpha^{-1} < \tau_\alpha^0 < \tau_y^2$.

  This means that, when any process p_i (member of W or not) starts evaluating the predicate $\exists v :\ v \neq v_i \wedge X[v]$ (line 5), all the processes that write into Y have previously updated the flag of the array $X[1..b]$ associated with the value they write.

  Let us consider two cases according to the value of the predicate when evaluated by p_i.

  - $\exists v :\ v \neq v_i \wedge X[v]$ is $false$.
    In that case a single value is ever written into Y by the processes of W (recall that the other processes do not write Y). It follows that only that value can de decided.
  - $\exists v :\ v \neq v_i \wedge X[v]$ is $true$.
    In that case, several values have been proposed by processes, and it it is consequently possible that some of them have been written in Y.

    As the predicate is satisfied, the process p_i delays itself. Moreover, as $\forall\, p_\alpha \in W$, $\tau_\alpha^0 < \tau_i^2$, we conclude that the value of the duration during which p_i delays itself (line 5) is greater than $\max(\{\delta_z\}_{p_z \in S})$. As $\forall\, p_\alpha \in W$: $\tau_\alpha^1 \leq \tau_\alpha^0 + \delta_\alpha$, it follows that, when this period terminates all the processes in W have written into Y. Finally, as no more write operation on Y can be issued, it follows that p_i reads and decides the last value written into Y. $\Box_{Theorem\ 4}$

5 Conclusion

This paper has introduced and investigated the notion of *timed register*. Such an object is an atomic register that allows imposing constraints on some write operations in order they are successful. As it has been demonstrated in the paper, timed registers allow designing synchronization and agreement algorithms that are indulgent. This means that these algorithms always guarantee the safety property of the problem (they aim at solving), and guarantee their liveness property at least when the timing assumption they rest on is satisfied. To conclude we would like to emphasize the following points.

- Differently from usual timing-based algorithms, the indulgent algorithms based on timed registers are simple to design and easy to understand (and design simplicity is a first class property!).

- All the algorithms presented in the paper (be them devoted to synchronization problems or agreement problems) are very similar. They are all variants of the same basic read/write pattern. It follows that the timed register concept can be seen as a conceptual tool that allows capturing a hidden unity among these problems.

- When the bound Δ is known and there are no timing failures, all our indulgent algorithms based on timed registers remain correct even if the timed registers are replaced with usual atomic time-free registers. This may indicate that timed registers are the right “tool” for the design of indulgent algorithms.

- The algorithms that are correct for completely asynchronous systems are trivially indulgent (as they do not require additional timing assumptions). However, such asynchronous algorithms (when they exist) are usually less space and time efficient than the corresponding timed register-based algorithms when executed in timing-based systems. This is because they use an unbounded number of atomic registers (more precisely, they are round-based and use new atomic registers at each round [26]). This shows that, even when timed registers are not necessary from a computability point of view, they still allow for more efficient solutions.

- Timed registers are universal objects in systems that eventually satisfy timing constraints on a per timed register basis. As noticed in the introduction, this means that timed registers allows preserving safety properties, while the Δ assumption plays the role of a contention manager that ensures the liveness property, despite concurrency. Interestingly, this approach is dual of more traditional approaches (such as obstruction-freeness) that require an operation to execute alone in order to terminate. These two approaches can be seen as two specifications of scheduling properties that allow designing wait-free algorithms for problems that are otherwise impossible to solve in asynchronous systems with time-free atomic register only and where processes can crash.

In addition to studying timed registers, it would be interesting to define and study other types of timed objects. Such timed objects can be defined in the obvious way by extending the definition of the corresponding asynchronous objects. Other ways to extend this research would be to solve other problems and design indulgent concurrent data structures using timed objects, and assume that a timed register itself can experience some kind of failures.

References

[1] Aguilera M.K., Frolund S., Hadzilacos V., Horn S.L. and Toueg S., Abortable Shared Objects. Brief annoucement in *Proc. 20th Symposium on Distributed Computing (DISC'06)*, Springer-Verlag LNCS #4167, pp. 534-536, 2006.

[2] Alur R., Attiya H. and Taubenfeld G., Time-adaptive Algorithms for Synchronization. *SIAM Journal of Computing*, 26(2):539-556, 1997.

[3] Alur R. and Taubenfeld G., Fast Timing-based Algorithms. *Distributed Computing*, 10(1):1-10, 1996.

[4] Ben-Or M., Another Advantage of Free Choice: Completely Asynchronous Agreement Protocols. *Proc. 2nd ACM Symposium on Principles of Distributed Computing (PODC'83)*, pp. 27-30, 1983.

[5] Burns J.N. and Lynch N.A., Bounds on shared-memory for mutual exclusion. *Information and Computation*, 107(2):171–184, December 1993.

[6] Burns J.E. and Peterson G.L., The Ambiguity of Choosing. *Proc. 8th ACM Symposium on Principles of Distributed Computing (PODC'89)*, ACM Press, pages 145-158, 1989.

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

[8] Fischer M.J., Lynch N. and Paterson M.S., Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, 32(2):374-382, 1985.

[9] Guerraoui R., Indulgent Algorithms. *Proc. 19th ACM Symposium on Principles of Distributed Computing, (PODC'00)*, ACM Press, pp. 289-298, 2000.

[10] Guerraoui R., Kapalka M. and Kouznetsov P., The Weakest failure Detectors to Boost Obstruction-Freedom. *Proc. 20th Symposium on Distributed Computing (DISC'06)*, Springer-Verlag LNCS #4167, pp. 376-390, 2006.

[11] Guerraoui R. and Lynch N., A General Characterization of Indulgence. *Proc. 8th Int'l Symposium on Self-stabilization, Safety and Security of Distributed Systems (SSS'06)*, Springer-Verlag LNCS, 2006.

[12] Guerraoui R. and Raynal M., The Information Structure of Indulgent Consensus. *IEEE Transactions on Computers*, 53(4):453-466, 2004.

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

[14] Herlihy M.P., Luchangco V. and Moir M., Obstruction-free Synchronization: Double-ended Queues as an Example. *Proc. 23th IEEE Int'l Conference on Distributed Computing Systems (ICDCS'03)*, IEEE Compurter Society Press, pp. 522-529, 2003.

[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., Proving the Correctness of multiprocess programs. *IEEE Transaction on Software Engineering*, SE-3(2):125-143, 1977.

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

[19] Lamport L., A Fast Mutual Exclusion Algorithm. *ACM Transactions on Computer Systems*, 5(1):1-11, 1987.

[20] Loui M.C. and Abu-Amara H.H., Memory Requirements for Agreement among Unreliable Asynchronous Processes. *Advances in Computing Research*, JAI Press, 4;163-183, 1987.

[21] Lynch N.A. and Shavit N., Timing-based Mutual Exclusion. In *Proceedings of the 13th IEEE Real-Time Systems Symposium (RTSS'92)*, pp. 211, December 1992.

[22] Mostefaoui A., Rajsbaum S. and Raynal M., Conditions on Input Vectors for Consensus Solvability in Asynchronous Distributed Systems. *Journal of the ACM,* 50(6):922-954, 2003.

[23] Plotkin S.A., Sticky Bits and Universality of Consensus. *8th ACM Symposium on Principles of Distributed Computing, (PODC'89)*, ACM Press, pp. 159-175, 1989.

[24] Raynal M., Algorithms for mutual exclusion. *The MIT Press*, ISBN 0-262-18119-3, 107 pages, 1986.

[25] Raynal M., A Short Introduction to Failure Detectors for Asynchronous Distributed Systems. *ACM SIGACT News, Distributed Computing Column*, 36(1):53-70, 2005.

[26] Taubenfeld G., Computing in the Presence of Timing Failures. *Proc. 26th Int'l IEEE Conference on Distributed Computing Systems (ICDCS'06)*, IEEE Computer Press, Lisbon (Portugal), 2006.

[27] Taubenfeld G., Synchronization Algorithms and Concurrent Programming. *Pearson Prentice-Hall*, ISBN 0-131-97259-6, 423 pages, 2006.