Synchronization is coming back, but is it the same?

Michel Raynal

▶ **To cite this version:**

Michel Raynal. Synchronization is coming back, but is it the same?. [Research Report] PI 1875, 2007, pp.16. inria-00194744v3

HAL Id: inria-00194744
https://inria.hal.science/inria-00194744v3

Submitted on 13 Dec 2007

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES

PUBLICATION INTERNE N° 1875

SYNCHRONIZATION IS COMING BACK, BUT IS IT THE SAME?

MICHEL RAYNAL

ISSN 1166-8687

IRISA

CAMPUS UNIVERSITAIRE DE BEAULIEU - 35042 RENNES CEDEX - FRANCE

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES
Campus de Beaulieu – 35042 Rennes Cedex – France
Tél. : (33) 02 99 84 71 00 – Fax : (33) 02 99 84 71 71
http://www.irisa.fr

Synchronization is coming back, but is it the same?

Michel Raynal*

Systèmes communicants
Projet ASAP

Publication interne n ° 1875 — Décembre 2007 — 14 pages

Abstract: This invited talk surveys notions related to synchronization in presence of asynchrony and failures. To the author knowledge, there is currently no textbook in which these notions are pieced together, unified, and presented in a homogeneous way. This talk (that pretends to be neither exhaustive, nor objective) is only a first endeavor in that direction. The notions that are presented and discussed are listed in the keyword list.

Key-words: Abortable objects, Atomic register, Concurrent object, Consensus, Contention manager, Failure detector, Graceful degradation, Lock, Non-blocking, Obstruction freedom, Shared memory, Synchronization, Timed register, t-Resilience, Universal construction, Wait-freedom.

(Résumé : tsvp)

* IRISA, Université de Rennes 1, Campus de Beaulieu, 35042 Rennes Cedex, France, raynal@irisa.fr

This technical report is the text of an invited keynote talk presented at the 22th Int'l IEEE Conference on Advanced Inforamtion Networking and Applications (AINA'2008), March 25-28 2008, Okinawa, Japan. The work corresponding to this talk is partially supported by the European Network of Excellence ReSIST. The title of this talk is inspired from the title of an invited talk given by Rachid Guerraoui at Europar 2007, the title of which was "*Transactions are coming back, but are they the same?*" Only the titles are similar. This talk and Guerraoui's talk address different topics.

Centre National de la Recherche Scientifique
(UMR 6074) Université de Rennes 1 – Insa de Rennes

Institut National de Recherche en Informatique
et en Automatique – unité de recherche de Rennes

La synchronisation est de retour, mais est-ce bien la même ?

Résumé : Ce rapport présente une nouvelle dimension que prend la synchronisation en présence de fautes dans un contexte asynchrone.

Mots clés : Registre atomique, object concurrent, consensus, gestionnaire de contention, détecteur de fautes, dégradation harmonieuse, verrou, mémoire partagée, synchronisation, regsitre temporisé, t-tolérance, cosntruction universelle, algorithme libre d'attente.

1 Introduction

Concurrent objects and traditional lock-based synchronization Synchronization problems are rarely easy to solve. To address this difficulty, a major step has been the definition of appropriate objects able to capture the current state of the synchronization problem we want to solve. The *semaphore* concept, introduced very early (in the sixties) by Dijkstra [4], is such an object. A semaphore S consists in an integer that has to always remain non-negative, and can be decremented and incremented by the well-known primitives $P()$ and $V()$. Semaphores are well-suited to implement synchronization kernels as they allow capturing the essence of low-level synchronization.

A *concurrent* object is an object that (1) has a precisely defined semantics (usually defined by a sequential specification) and (2) can be accessed concurrently by several processes. Among the most popular concurrent objects is the shared queue (whose implementation is usually described in textbooks under the name *producer/consumer* problem), and the shared file, also called disk or register (the implementation of which underlies the *readers/writers* problem). Unfortunately, from a high-level programming point of view, semaphores appear to be a too low-level mechanism to easily implement high-level concurrent objects. More precisely, providing processes with an object such as a shared queue or a shared file, only with semaphores, is an error-prone activity. Moreover, correctness proofs of semaphore-based algorithms are rarely easy. That is why synchronization-oriented high-level programming concepts such as the concept of *monitor* [13], have been later defined (in the seventies). Basically, a monitor allows the programmer to define an object in such a way that it can be accessed concurrently by several processes while providing its sequential semantics. This means, for objects that have a sequential specification (e.g., a queue or a file), that, from an external observer point of view, the operations on the object appear as if they are executed sequentially. In other words, a monitor provides the programmers with clean concepts and mechanisms to encapsulate correct implementations of concurrent objects. As a consequence, any synchronization problem can be casted as a specific concurrent object providing appropriate operations to its users. For instance, resource allocation and rendezvous coordination can be encapsulated in a resource manager object and a rendezvous object, respectively.

Net effect of asynchrony and failures When operations accessing a concurrent object overlap, a simple way to ensure that the specification of the object is never violated consists in blocking all of them but one operation during some time in order that operation can access the object without being bothered by the other operations and can consequently be able to proceed in a safe way. This is traditionally solved by associating *locks* with each concurrent object (such a lock is called a *condition* in the monitor terminology [13]). A simple way to implement a lock consists in using a semaphore. Due to their simplicity and their effectiveness, lock-based implementations are popular.

Unfortunately, in asynchronous systems (i.e., the class of systems where no assumption on the speed of the processes is possible), the lock-based approach presents intrinsic major drawbacks. If a slow process holds a lock during a long period of time, it can delay faster processes from accessing (some parts of) the object. More severely, the lock-based approach does not prevent by itself deadlock scenarios from occurring. Preventing them requires additional mechanisms or strategies that can give rise to long waiting periods that degrade the whole system efficiency. The situation becomes even more critical in presence of failures. When a process holding a lock crashes, as the system is asynchronous, there is no way to know whether this process has crashed or is only very slow. It follows that such a crash can block the system during an arbitrarily long period (i.e., until an appropriate recovery action is started, either manually, or after the operating system becomes aware of the crash of the process).

Obstruction freedom, non-blocking and wait-freedom To cope with the previous drawbacks due to the use of lock-based solutions when one has to implement concurrent objects, several properties related to synchronization and implementation of concurrent objects in presence of asynchrony and process crashes have been recently introduced. Considering a concurrent object, these properties are the following ones, starting from the weaker and going to the stronger.

- Obstruction-freedom. An implementation of a concurrent object is obstruction-free, if each of its operations is guaranteed to terminate when it is executed without concurrency (assuming that the invoking process does not crash) [12].

  An operation executes "without concurrency" if the only process p that is currently accessing the internal representation of the object is the process that invoked that operation. This does not prevent other processes to

have started and not yet finished operations on the same object[1]. The important point here is that, if other processes have started executing operations on the same object, whatever their progress in their operations, they have "momentarily" stopped accessing the internal representation of the object, allowing thereby p to execute its operation in a concurrency-free context.

It is important to notice that an obstruction-free implementation of an object has the following noteworthy property. If several processes execute concurrently operations on that object, the semantics of the object is always ensured (i.e., if an operation terminates, it returns a correct result), but it is possible that no operation at all terminates in the presence of concurrent accesses to the internal representation of the object. Termination is not guaranteed when several operations access concurrently the internal representation of the object.

So, despite asynchrony and failures, an obstruction-free implementation of an object guarantees (1) always the safety property (consistency) of each of operation, and (2) the liveness property of each operation when there are no concurrent accesses to the internal representation of the object.

- Non-blocking. This property states that, despite asynchrony and process crashes, if several processes execute operations on the same object and do not crash, at least one of them terminates its operation [11].

  So, non-blocking means deadlock-freedom despite asynchrony and crashes. It is a property strictly stronger than obstruction-freedom: it is obstruction-freedom + deadlock-freedom.

- Wait-freedom. This property states that, despite asynchrony and process crashes, any process that executes an operation on the object (and does not crash), terminates its operation [11].

  So, wait-freedom means starvation-freedom despite asynchrony and crashes. It is a property strictly stronger than non-blocking: it is obstruction-freedom + starvation-freedom.

Content of the paper The paper investigates the notions of obstruction-freedom, non-blocking, and wait-freedom (see Figure 1). To that end, it first present (Section 3) an algorithm that, given a concurrent object defined by a sequential specification (e.g., a shared stack, or a shared double-ended queue), constructs a wait-free implementation of that object. Such an algorithm is called a *universal construction* [11].

Then, considering that we are given an obstruction-free implementation of an object (without knowing its specification), the paper presents algorithms that transforms that implementation into a non-blocking or a wait-free implementation. Two approaches are considered, for such a transformation. One is based on the use of a synchronization operation plus an additional assumption on the system behavior (Section 4.1), while the second one assumes that the underlying system is enriched with a failure detector (Section 4.2).

As shown in Figure 1, an interesting problem remains open. To date, no paper presents an algorithm that builds an obstruction-free implementation of an object from its sequential specification. Such an implementation should allow several operations that access independent parts of the internal representation of the object to proceed concurrently (this is not allowed by a universal construction).

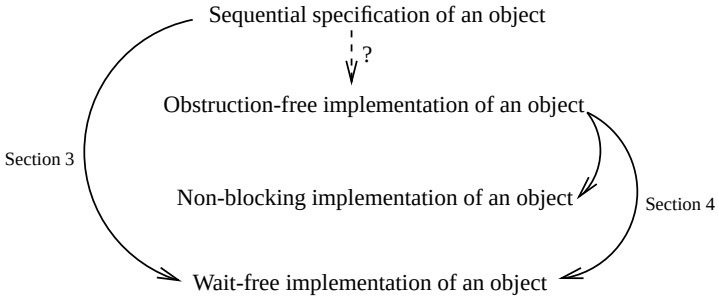

Figure 1: Global picture

[1] This is for example the case of a process that crashed in the middle of an operation on the object.

Then, the paper addresses other notions related to synchronization in presence of failures, namely, the notion of t-resilience associated with an object implementation, and the notion of gracefully degrading implementation (Section 5). So, the paper surveys important topics related to asynchronous computing in presence of failures. Its aim is to introduce the reader to synchronization notions that will become more and more familiar as multi-core architectures will develop.

2 Base computation model

Process model We consider a system made up of n sequential processes, denoted $p_1, p_2, \ldots, p_n$, such that any number of them can crash. A process is *correct in a run* if it does not crash in that run. A process that crashes in a run is said to be *faulty* in that run. A process executes correctly (i.e., according to its specification) until it possibly crashes. After it has crashed, a process executes no operation. There is no bound on the relative speed of a process with respect to another process, which means that the system is *asynchronous*.

Shared Registers A *reliable atomic register* [15] is an abstraction of shared variable. It provides the processes with two operations, usually called read and write. Atomic means that, whatever the number of processes that can concurrently access such a register, the read and write operations issued by the processes appear as if they have been executed one after the other, each one being executed instantaneously at some point of the time line between its invocation event and its response event. Reliable means that the register does not suffer failures (it always behaves according to its specification).

3 A universal construction

Let us consider an object defined by a sequential specification (e.g., a stack, or a file). This section presents an algorithm (universal construction) that builds a wait-free implementation of that object. The first such universal construction is due to Herlihy [11]. We present here a construction due to Guerraoui and Raynal introduced in [9]. That construction relies on two types of objects, atomic registers and consensus objects.

3.1 Consensus object

A *consensus* object is a concurrent atomic object that provides the processes with a single operation called $propose()$ that has one input parameter and returns one output. Given a consensus object C, we say that a process "proposes v to C", when it invokes $C.propose(v)$. When then say that "it is a participant of the consensus C". If the value returned by an invocation is v', we say that "the invoking process decides v'", or "v' is decided by the consensus object C".

Let $\perp$ denote a default value that cannot be proposed by a process. A consensus object C can be seen as maintaining an internal state variable x, initialized to $\perp$, its sequential specification being defined as follows:

operation $propose\ (v)$:
  if $(x = \perp)$ **then** $x \leftarrow v$ **endif**; **return** (x).

As we can see, a $propose()$ operation is a combination of a conditional write operation followed by a read operation. A consensus object is a one-write register that keeps forever the value proposed by the first $propose()$ operation. Any subsequent $propose()$ operation returns the first deposited value. As a consensus object is a concurrent atomic object, if a process crashes while executing a $propose()$ operation, everything appears as if that operation has been entirely or not at all executed.

The fact that the same value is returned by the $propose\ ()$ operations that terminate is called consensus *agreement* property. The fact that the decided value is a value proposed by a process is called consensus *validity* property. The fact that all the processes that invoke the $propose\ ()$ operation and do not crash are returned a value is called consensus *termination* property. So, validity rules out trivial and meaningless objects where the decided value would not be related to the proposed values. Agreement defines the coordination power of a consensus object, while termination states that a consensus object is wait-free.

3.2 The specification of the constructed object

The constructed object Y is defined by an initial state (denoted s_0), and a type that consists in a finite set of m total operations $\text{OP}_1(param_1, res_1)$, ..., $\text{OP}_m(param_m, res_m)$ and a sequential specification. In the following, $\text{OP}(param, res)$ is used to denote any of the previous operations. Each operation has a (possibly empty) set of input parameters ($param$), and returns a result (res). Sequential specification means that the behavior of Y is defined by the set of all the sequences of operations where the output res of each operation $\text{OP}()$ is appropriately determined according to the value of its input parameters ($param$) and all the operations that precede $\text{OP}()$ in the sequence. Alternatively, the sequential specification can also be defined by associating a pre-assertion and a post-assertion with each operation. The pre-assertion describes the state of the object before the operation, while, assuming no other operation is concurrently executed on the object, the post-assertion defines the result output by the operation and the new state of the object resulting from that operation execution.

For the need of the universal construction, we consider that a sequence of operations can be abstracted as an object state, and accordingly the semantics of each operation is defined by a deterministic transition function $\delta()$ (for non-deterministic objects, see [9]). More precisely, s being the current state of the object, $\delta(x,\text{OP}(param))$ returns a pair (s', res) where s' is the new state of the object and res is the the output parameter returned to the calling process.

3.3 A wait-free universal construction

The universal construction is designed incrementally. It first shows how consensus objects are used to order operations. Then it shows how atomic registers are used to ensure that no operation issued by a correct process remains pending forever. This means that the wait-free property of the construction is obtained from the atomic registers.

3.3.1 Step 1: using consensus to order

The construction is an asynchronous algorithm in which each process p_i plays two roles, namely, the role of a client when an operation is locally invoked, and the role of a server where it cooperates with the other servers to implement the object Y. The first role is implemented by appropriate statements executed each time an operation is locally invoked by p_i, while the second role is implemented by a set of background tasks, one on each process p_i.

Client role An interface with the local application layer can easily be realized as follows. First, p_i sets a local variable $result_i$ to $\perp$, and informs its server that a new operation has been locally issued (line 1, below). To that end, another local variable denoted $prop_i$ is used, that is a pair containing the description of the operation ($\text{OP}(param)$) and the identity (i) of the invoking process. Then, p_i waits until the result associated with that operation has been computed and returns then that result to the upper application layer (line 2). The hope is that the servers will cooperate to order and apply the operations on Y to each local representation s_i.

when the operation $Y.\text{OP}(param)$ **is invoked at** p_i:
1 $result_i \leftarrow \perp$; $prop_i \leftarrow \big(\text{OP}(param), i\big)$;
2 **wait until** $(result_i \neq \perp)$; *return* $(result_i)$.

Server role The idea is for each server to manage a local copy s_i of object Y (s_i is initialized to the initial value of Y). A necessary requirement for this management to be correct, is that the servers apply the operations in the same order to their local copies of Y.

To attain this goal, the server at each process p_i is implemented by a background task which is an infinite loop described below. When, $prop_i \neq \perp$, the task discovers that a new operation has locally been invoked (line 3). So, in order to both inform the other processes and guarantee that the operations will be applied in the same order on each copy, it proposes that operation to a new consensus instance (line 5). The tasks use a list of consensus objects $(CONS[k])_{k\geq 1}$, $CONS[k]$ being used to implement the kth consensus instance. As each consensus instance provides the processes with the same output, and as they invoke these instances in the same sequential order, they will be provided with the same sequence of decided values. So, each process manages a local counter k_i such that $CONS[k_i]$ denotes the next consensus object that p_i has to use (line 4).

As a value $prop_i$ proposed to a consensus instance is a pair, a decided value is also a pair made up of an operation and the identity of the process that invoked it (see line 1). So, $exec_i$, the local variable where is saved the value

decided by the last consensus instance (line 5), is a pair composed of two fields, $exec_i.op$ that contains the decided operation, and $exec_i.proc$ that contains the identity of the process that issued that operation. The server executes then "$(s_i, res) \leftarrow \delta(s_i, exec_i.op)$" to update its local copy s_i of the object Y (line 6). Moreover, if p_i is the process that invoked the operation, it locally returns the result res of that operation by writing it into the local variable $result_i$ (line 8).

```
while (true) do
   3 if (prop_i ≠ ⊥) then
   4   k_i ← k_i + 1;
   5   exec_i ← CONS[k_i].propose(prop_i);
   6   (s_i, res) ← δ(s_i, exec_i.op);
   7   let j = exec_i[r].proc;
   8   if (i = j) then prop_i ← ⊥; result_i ← res end_if
   9 end_if
end_while.
```

As each consensus instance outputs the same operation to all the processes that invoke that instance, and as the processes invoke the instances in the same order, it follows that, for any k, the processes p_i that have invoked the first k consensus instances, applied the same length k sequence of operations to their copy s_i of the object Y.

If, after it has participated in k consensus instances (s_i is then the state of Y resulting from the sequential application of the k operations output by the first k consensus instances), a process p_i does not invoke operation during some period of time, its local copy s_i is not modified. When, later, it invokes again an operation and proposes it to a consensus instance, it can have to execute consensus instances (starting from $k + 1$) to catch up a consensus instance where no value has yet been decided. While catching up, it will sequentially update its local state s_i according to the operations that have been decided after the kth instance.

3.3.2 Non-blocking vs wait-free implementation

Considering a consensus instance, exactly one process is a winner in the sense that the value proposed to that instance by that process is the decided value. In that sense the previous construction is *non-blocking*: all the processes that participate (i.e., propose a value) in a consensus instance are returned a value, and exactly one process (the winner) terminates its upper layer operation. A consensus object is *live* in the sense that a value is always decided as soon as the consensus object is proposed at least one value.

Unfortunately, that implementation is not wait-free. Indeed, it is easy to build a scenario in which, while a process p_i continuously proposes its operation to successive consensus instances, it is never a winner, because there are always processes proposing operations to these successive consensus instances and it is always a value proposed by one of these processes that is decided, and never the value proposed by p_i. In that case, the operation on Y issued by p_i is never executed, and, at the application level, p_i is prevented from progressing.

The difference between non-blocking and wait-free is fundamental. A similar difference exists in classical lock-based computing between the *deadlock-free* and *starvation-free* guarantees. While starvation-free implies deadlock-free, the opposite is not true. It is the same here where wait-free implies non-blocking, while the opposite is false.

3.3.3 Step 2: atomic registers to help

A way to go from a non-blocking construction to a wait-free construction consists in introducing a helping mechanism that allows a process to propose to a consensus instance not its own pending operation but instead all the pending operations it is aware of.

To that end, a one-writer multi-reader atomic register is associated with each process. That register allows its writer process to inform the other processes about the last operation is has issued. More explicitly, $REG[i]$ is an atomic register that can be written only by p_i, and read by all the processes. When it writes its last operation in $REG[i]$, p_i "publishes" it, and all the other processes become aware of it when they read $REG[i]$. Such a register $REG[i]$ is made up of two fields, $REG[i].op$ that contains the last operation invoked by p_i together with its input parameters (OP$(param)$), and $REG[i].sn$ that contains a sequence number ($REG[i].sn = x$, means that $REG[i].op$ is the xth operation issued by p_i). Each atomic register $REG[i]$ is initialized to $(\perp, 0)$.

In order to know whether the last operation that p_j has published in $REG[j]$ has been or not applied to its local copy of Y (i.e., s_i), each process p_i manages a local array of sequence numbers denoted $last\_seen_i[1 : n]$; $last\_seen_i[j]$

```
when the operation OP(param) is locally invoked at p_i:
(1)  result_i ← ⊥;
(2)  REG[i] ← (OP(param), last_seen_i[i] + 1);
(3)  wait until (result_i ≠ ⊥);
(4)  return (result_i)
-------------------------------------------------------------------------

Task T:   % background server task %
(5)  while (true) do
(6)       prop_i ← ϵ;  % empty list %
(7)       for 1 ≤ j ≤ n do
(8)           if (REG[j].sn > last_seen_i[j]) then append (REG[j].op, j) to prop_i end_if;
(9)       end_for;
(10)      if (prop_i ≠ ϵ) then k_i ← k_i + 1; % consensus instance number%
(11)                           exec_i ← CONS[k_i].propose(prop_i);
(12)                           for r = 1 to |exec_i| do (s_i, res) ← δ(s_i, exec_i[r].op);
(13)                                                    let j = exec_i[r].proc;
(14)                                                    last_seen_i[j] ← last_seen_i[j] + 1;
(15)                                                    if (i = j) then result_i ← res end_if
(16)                           end_for
(17)      end_if
(18) end_while
```

Figure 2: A wait-free universal construction

contains the sequence number of the last operation issued by p_j that has been applied to s_i ($\forall\ i, j$, $last\_seen_i[j]$ is initialized to 0).

The helping mechanism is realized as follows. When it invokes a consensus instance, a process p_i does not proposes its last operation only, but proposes instead all the operations that have been published by the processes in $REG[1:n]$ and that have not been applied to its local copy s_i of the object. From its point of view, those are all the operations that have not yet been executed, and consequently p_i strives to have them executed. This means that now, instead of a single operation (issued by itself), a process p_i proposes a list of operations to a consensus instance.

These design principles give rise to the universal construction described in Figure 2. As the value proposed by a process p_i to a consensus instance is a non-empty list of operations ($prop_i$), the value decided by that consensus instance is a non-empty list of operations. Consequently, the local variable $exec_i$, where is saved the value decided by the current consensus instance, is now a list of operations; $|exec_i|$ denotes its length and $exec_i[r]$, $1 \leq r \leq |exec_i|$, denotes its rth item. Let us remind that this item is a pair (namely, the pair $(exec_i[r].op,\ exec_i[r].proc)$). (see [9] for the proof.)

3.4 Consensus from a timed register

The consensus problem can not be solved in a pure asynchronous system prone to process crashes, be the underlying communication system a message-passing system [6], or a shared memory made up of atomic read/write registers [17]. The system has to be equipped with additional power in order to be able to solve consensus. Failure detectors in message-passing, and synchronization primitives (stronger than read or write) [11], are objects that provide such additional power. Here we present another approach to solve the consensus problem, namely, the notion of a *timed register*. That notion has been introduced and investigated by Raynal and Taubenfeld [18].

The notion of a timed register The shared memory is composed of classical atomic read/write registers, plus timed registers. A timed register supports atomic read and atomic write operations defined as follows. 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, but the important point here is that (as failure detector classes, or abstract data types), a timed register is defined only by behavioral properties and not by the way these properties are implemented in a particular system.

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, 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).

A process can execute a $\mathsf{delay}(d)$ statement, where d is time duration. Its effect is to delay the invoking process for an arbitrary but finite period longer than d time units.

An indulgent consensus algorithm (known bound) Such an algorithm is presented in Figure 3 [18]. It uses a single timed register Y (initialized to $\perp$) and is surprisingly simple. A process decides when it executes the $\mathsf{return}()$ statement. As it can be observed, this algorithm works for any number of processes.

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

Figure 3: An indulgent consensus algorithm with a known bound (code for p_i)

A timing failure refers to the situation where the timing assumption the underlying system should provide the processes with (i.e., the bound Δ), is not satisfied. Here, a timing failure means that there are periods during which Δ is not an upper bound on the time duration separating a 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. It is easy to see that the algorithm depicted in Figure 3 is indulgent.

4 From obstruction-freedom to a stronger property

This section considers the case where we are provided with an obstruction-free implementation of an object and we want to transform it into a non-blocking or a wait-free implementation.

It is important to notice that, in the present case, the specification of the object is not known. We have only a "black box" that provides an obstruction-free implementation of the object. Let us observe that, even if we knew the sequential specification of the object, a universal construction would not solve the problem. This is due to the following reason. A universal construction is based on consensus objects whose aim is to linearize all the operations on the object. Consequently, a universal construction eliminates concurrency among the operations in the common case (this is for example the case of all à la Paxos algorithms that are based on an eventual leader [8, 16]).

So, the problem is here to produce *efficient* transformations [5]. The transformations we are interested in have to guarantee that some operation issued by a correct process will terminate (non-blocking case), or that each operation issued by a correct process will terminate (wait-free case), without preventing operations to execute concurrently each time it is possible (e.g., two operations that access the opposite ends of a non-empty shared queue have not to be prevented to proceed in parallel).

4.1 Additional power is required

In order to go from obstruction-freedom to a stronger property, some help of the underlying system is required. This help can take different forms. One is to have new objects such as failures detectors. This approach is considered in the

next subsection.

Another approach consists in considering that the underlying system (1) provides base objects endowed with (synchronization) operations stronger than read and write, and (2) satisfies additional behavioral assumptions (these assumptions make the system no longer fully asynchronous).

Fich, Luchangco, Moir and Shavit have proposed an algorithm that transforms an obstruction-free implementation into a wait-free implementation in [5]. Their algorithm is based on a register R that provides the processes with a Fetch&Incr() operation. The invocation R.Fetch&Incr() atomically increases R by 1 and returns its previous value. This register is used to associate a unique timestamp with each operation on the object, when that operation encounters concurrency that could prevent its progress.

As previously suggested, this operation alone is not sufficient for obtaining the wait-freedom property. The underlying system has also to be eventually synchronous in the following sense: There is a finite r such that, in all runs, the ratio of the speeds of the quickest process and the slowest (non crashed) process is bounded by r. The algorithm presented in [5] assumes such a bound does exist but does not use it explicitly (i.e., that bound is used only in the proof of the algorithm, not in its text)[2].

Taubenfeld presents in [20] an algorithm that transforms an obstruction-free implementation into a non-blocking implementation. His algorithm requires the same assumption as before on the bound r, and uses a register that provides the processes with a Compare&Swap() operation. R.Compare&Swap(old, new) executes atomically the following. If $R = old$, it is updated to the value new and the operation returns $true$. Otherwise R is not modified, and the operation returns $false$.

The main difference between both previous transformations lies in their cost. The transformation described in [5] requires n atomic single writer registers, n atomic multi-writer registers, and one fetch&add register. Moreover, in presence of concurency conflict, the waiting time for an operation to execute can be exponential wrt n. Differently, the transformation described in [20] (that provides non-blocking instead of the stronger wait-freedom property) requires n atomic single writer registers, and one compare&swap register. It has a $O(1)$ time complexity.

4.2 Using appropriate failure detectors to boost obstruction-freedom

The failure detector-based approach to go from an obstruction-free implementation to an implementation satisfying a stronger property in presence of concurrency has been introduced by Guerraoui, Kapalka and Kouznetsov [7] who have investigated the weakest failure detectors to boost obstruction-freedom.

When compared to the previous approach (addition of both an operation stronger than read or write, and a behavioral assumption for the underlying system), the use of a failure detector has the advantage to encapsulate in a single object all the needed requirements.

Principle The principle of this approach is the following. Just after a process p_i starts executing an operation, and also later (even several times before it completes the operation), it invokes the primitive contender(i) to signal possible contention. An invocation of contender(i) can block the calling process until some progress condition is verified. When it returns, the calling process tries to execute its operation. If after some time, it has not yet finished it, it invokes again contender(i) to benefit from the new concurrency context. Finally, when p_i terminates its operation it invokes the primitive finished(i) to indicate it is no longer accessing the internal representation of the object. Both contender() and finished() are two primitives defining what is usually called a *contention manager*.

A failure detector-based contention manager for the non-blocking property We show here how to construct a contention manager that ensures the non-blocking property. This contention manager is based on a failure detector denoted Ω_*. Such a failure detector provides the processes with a primitive, denoted leader(), that takes a set X of processes as input parameter, and returns a process identity. For each set of processes X, all the invocations leader(X)

[2] It is important to notice that this assumption is not strong enough to allow detecting the crash of a process. Differently, the explicit knowledge of r would allow detecting process crashes.

collectively satisfy the following property (eventual leadership): there is a time after which all these invocations return the same process identity that is a correct process of X [7, 19][3].

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 concurrently accessing the object. Given a set X of processes that invoke $\mathsf{leader}(X)$, the eventual property states that there is a time after which these processes have the same correct leader p_ℓ, and this correct leader is such that $\ell \in X$. This is the process that is allowed to progress to finish executing its operation. It is important to notice that the time from which the leadership property occurs is not known by the processes. Moreover, before that time, there is an anarchy period during which each process, as far as its $\mathsf{leader}(X)$ invocations are concerned, can obtain different leaders. Let us also observe that if a process p_i issues two invocations $\mathsf{leader}(X1)$ and $\mathsf{leader}(X2)$ with $X1 \neq X2$, there is no relation linking $\ell 1$ and $\ell 2$ (the eventual leaders associated with the sets $X1$ and $X2$) whatever the values of $X1$ and $X2$ (e.g., the fact that $X1 \subset X2$ imposes no particular constraint on $\ell 1$ and $\ell 2$).

The Ω_*-based contention manager is described in Figure 4. Its text is self-explanatory. $PART[1..n]$ is a shared boolean array initialized to $[false, \ldots, false]$. $PART[i]$ is a single writer multi-reader register written only by p_i to indicate it is competing with other processes in order to terminate its operation. It is shown in [7] that this Ω_* is the weakest failure detector class that allows boosting the obstruction-freedom property to the non-blocking property. ("Weakest" means that there is no failure detector-based contention manager for such a boosting if we consider only failure detectors that provide less information on process failures than the information given by Ω_*.)

```
operation contender(i):
    PART[i] ← true;
    repeat X ← {j | PART[j]}
         until (leader(X) = i) end repeat

operation finished(i): PART[i] ← false
```

Figure 4: An Ω_*-based contention manager

A failure detector-based contention manager for the wait-freedom property This contention manager is based on the failure detector $\diamond\mathcal{P}$ introduced in [2]. It is shown in [7] that it is the weakest failure detector to boost from the obstruction-freedom property to the wait-freedom property. A failure detector of the class$\diamond\mathcal{P}$ provides each process p_i with a set SUSPECTED_i such that the set SUSPECTED_i of each correct process p_i eventually contains all the faulty processes and only them. This means that during an arbitrary long (but finite) period of time, a set can contain any value, after which it behaves "perfectly" in the sense it contains only crashed processes, and eventually all of them.

The $\diamond\mathcal{P}$-based contention manager needs an additional primitive denoted $\mathsf{get\_timestamp}()$. That primitive returns a locally increasing timestamp ts (positive integer) such that, when considering all the processes, there is a finite number of timestamps ts' that have been generated with $ts' < ts$. This primitive cane easily be implemented from atomic single writer multi-reader atomic registers.

The $\diamond\mathcal{P}$-based contention manager is described in Figure 5. It uses an array $TS[1..n]$ of timestamps (initialized to $[0, \ldots, 0]$). $TS[i] \neq 0$ means that p_i is competing in order to terminate its operation. Let us observe that all the pairs $(TS[x], x)$ (such that $TS \neq 0$) refer to competing processes and are lexicographically ordered. This order is used by the contention manager to favor the process with the smallest pair and make the other competing processes to wait (until their turn arrives). Let us observe that $\diamond\mathcal{P}$ guarantees that, after some time, all the competing processes agrees on the same order to favor processes, and due to the fact that the timestamps do increase, every process will eventually execute its operation (if it does not crash before).

5 t-Resilience and graceful degradation

Up to now, we have considered that any number of processes may crash, but we have assumed that the underlying objects they use to communicate (e.g., read/write atomic registers, consensus objects) are reliable. This section considers the case where the base objects can be faulty. It is inspired mainly from the work of Jayanti, Chandra and Toueg [14].

[3]When, $X = \{p_1, \ldots, p_n\}$, Ω_* boils down to Ω the classical leader oracle introduced in [3]. Ω_* is a "single leader" version of the more general leader oracle, denoted Ω_*^k, that has been introduced to wait-free solve the k-set agreement problem.

```
operation contender(i):
 if (TS[i] = 0) then TS[i] ← get_timestamp() end if;
 repeat compet_i ← {j | (TS[j] ≠ 0 ∧ j ∉ SUSPECTED_i)};
        let (ts, j) be the smallest pair in the set
                          ∈ {(TS[x], x) | x ∈ compet_i}
 until (j = i) end repeat

operation finished(i): TS[i] ← 0
```

Figure 5: A $\diamond\mathcal{P}$-based contention manager

The aim is to build a wait-free implementation of the operations defining an object RO, that tolerates the failure of the base objects it is built from. Let us remind that wait-free means that any operation on RO issued by a correct process has to terminate (whatever the speed or the crash of the other processes).

We focus here on *self-implementation*, i.e., implement a (high level) reliable object RO of type T from unreliable base objects of the same type T.

5.1 Failure modes

We consider that a base object can exhibit the following types of failures (called *failure modes*).

- Crash. An object experiences a crash failure if after some time all its operation invocations return $\perp$ (a default value that cannot be returned by an operation when there is no failure). This means that the object satisfies its specification until it crashes, and then satisfies the property "once $\perp$, thereafter $\perp$".
- Omission. An object experiences an omission failure with respect to a process p_i, it has the crash behavior with respect to p_i. So, an object that experiences an omission failure can be seen as crashed by some processes and correct by other processes.
- Byzantine. An object experiences a Byzantine failure if its operations answer values that are not in agreement with the object specification.

5.2 Notion of object t-resilience

Definition An implementation of a concurrent object is *t-fault tolerant with respect to the failure mode $\mathcal{F}$* (crash, omission, arbitrary) if the object remains correct and its implementation remains wait-free despite the occurrence of up to t base objects that fail according to $\mathcal{F}$.

A failure mode $\mathcal{F}$ is *less severe* than the failure mode $\mathcal{G}$ denoted $\mathcal{F} \prec \mathcal{G}$ if any implementation that is t-fault tolerant with respect to the failure mode $\mathcal{G}$ is also t-fault tolerant with respect to the failure mode $\mathcal{F}$. As an example, this means that an implementation that is t-omission tolerant is also t-crash tolerant.

Example As an example, Figure 6 describes a t-resilient wait-free implementation of a one-writer one-reader atomic register RO from unreliable base registers, for the crash failure mode. This implementation (introduced in [10]) requires $m = t + 1$ base registers (i.e., at least one base register is assumed to be reliable, but we do not know which one!). As we can see, this implementation does not use sequence numbers.

The writer simply writes the new value in each base register, in increasing order, starting from $REG[1]$ until $REG[t + 1]$. The reader scans sequentially the registers in the opposite order, starting from $REG[t + 1]$. It stops just after the first read of a base register that returns a non-$\perp$ value. As at least one base register does not crash (model assumption), the reader always obtains a non-$\perp$ value. (Let us remind that, as we want to build a t-resilient object, the construction is not required to provide guarantees when more than t base objects crash.) It is important to remark that each read and write operation must follow a predefined order when it accesses the base registers. Moreover (and this crucial for correctness!), the order for reading the base registers and the order for writing them are opposite.

```
operation RO.write(v): % invoked by the writer %
    for j from 1 to t + 1 do REG[j] ← v end_do;
    return ()

operation RO.read(): % invoked by the reader %
    for j from t + 1 to 1 do
        aux ← REG[j];
        if (aux ≠ ⊥) then return (aux) end_if
    end_do
```

Figure 6: 1W1R t-resilient atomic register

5.3 Notion of graceful degradation

An implementation of a concurrent object is *gracefully degrading* if it it never fails more severely than the failure mode of the base objects, whatever the number of base objects that fail (more details on this notion can be found in [3]).

As an example, let us assume that base objects can fail by crashing. We have the following.

- If the implementation remains wait-free and correct despite the crash of any number of processes (wait-free part) and the crash of up to t base objects (fault-tolerance part), then this implementation is t-fault tolerant with respect to the crash failure mode.
- If, additionally, the implementation is wait-free and fails only by crash (if it fails) when more than t base objects crash, and despite any number of processes, it is gracefully degrading.

As shown in [3], there are objects (e.g., consensus) for which it is impossible to build a gracefully degrading implementation for the crash failure mode.

6 To conclude

Several notions related to synchronization in presence of failure have been presented. Among the notions that have not been discussed, there is the notion of abortable object [1]. Such an object behaves like an ordinary object when accessed sequentially, but may return a special default value $\perp$ when accessed concurrently. The power of these objects is investigated in [1].

Acknowledgments

I would like to thank Rachid Guerraoui, Sergio Rajsbaum and Corentin Travers for numerous discussions on asynchronous computing in presence of failures.

References

[1] Aguilera M., Frolund S., Hadzilacos V., Horn S.L. and Toueg S., Abortable and query-abortable objects and their efficient implementations. *Proc. 26th ACM Symposium on Principles of Distributed Computing (PODC'07)*, ACM Press, pp. 23-32, 2007.

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

[3] Chandra T., Hadzilacos V. and Toueg S., The Weakest Failure Detector for Solving Consensus. *Journal of the ACM*, 43(4):685–722, 1996.

[4] Dijkstra E.W.D., Hierarchical ordering of sequential processes. *Acta Informatica*, 1(1):115-138, 1971.

[5] Fich E.F., Luchangco V. and Moir M. and Shavit N., Obstruction-free algorithms can be practically wait-free. *Proc. 19th Int'l Symp. on Distr. Computing (DISC'05)*, Springer-Verlag LNCS #3724, pp. 78-92, 2005.

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

[7] Guerraoui R., Kapalka M. and Kouznetsov P., The weakest failure detector to boost obstruction freedom. *20th Int'l Symposium on Distributed Computing (DISC'06)*, Springer-Verlag LNCS #4167, pp. 376-390, 2006.

[8] Guerraoui R. and Raynal M., The Alpha of indulgent consensus. *The Computer Journal*, 50(1):53-67, 2007.

[9] Guerraoui R. and Raynal M., A Universal construction for wait-free objects. *Proc. ARES 2007 Int'l Workshop on Foundations of Fault-tolerant Distr. Comp. (FOFDC 2007)*, IEEE Press, pp. 959-966, 2007.

[10] Guerraoui R. and Raynal M., From unreliable objects to reliable objects: the case of atomic registers and consensus. *Proc. 9th Int'l Conf. on Par. Comp. Tech. (PaCT'07)*, Springer Verlag LNCS LNCS #4671, pp. 47-61, 2007.

[11] Herlihy M.P., Wait-free synchronization. *ACM TOPLAS*, 13(1):124-149, 1991.

[12] Herlihy M.P., Luchangco V. and Moir M., Obstruction-free synchonization: double-ended queues as an example. *Proc. 23th Int'l IEEE Conference on Distributed Computing Systems (ICDCS'03)*, pp. 522-529, 2003.

[13] Hoare C.A.R., Monitors: an operating system structuring concept. *Comm. of the ACM*, 17(10):549-557, 1974.

[14] Jayanti P., Chandra T.D. and Toueg S., Fault-tolerant wait-free shared objects. *JACM*, 45(3):451-500, 1998.

[15] Lamport L., On interprocess communication, Part 1: Models, Part 2: Algorirhms. *Distributed Computing*, 1(2):77-101, 1986.

[16] Lamport L., The Part-Time Parliament. *ACM Transactions on Computer Systems*, 16(2):133-169, 1998.

[17] Loui M.C., and Abu-Amara H.H., Memory Requirements for Agreement Among Unreliable Asynchronous Processes. *Par. and Distributed Computing: vol. 4 of Advances in Comp. Research*, JAI Press, 4:163-183, 1987.

[18] Raynal M. and Taubenfeld G., The notion of a timed register and its application to indulgent synchronization. *Proc. 19th ACM Symposium on Parallel Algorithms and Architectures (SPAA'07)*, pp. 200-209, 2007.

[19] Raynal M. and Travers C., In search of the holy grail: looking for the weakest failure detector for wait-free set agreement. (Invited Talk.) *Proc. 10th Int'l Conference On Principles Of Distributed Systems (OPODIS'06)*, Springer-Verlag LNCS #4305, pp. 1-17, 2006.

[20] Taubenfeld G., Efficient transformations of obstruction-free algorithms into non-blocking algorithms. *Proc. 21th Int'l Symposium on Distributed Computing (DISC'07)*, Springer-Verlag LNCS #4731, pp. 450-464, 2007.