

The x -Wait-freedom Progress Condition

Damien Imbs^{*}, Michel Raynal^{**}
damien.imbs@irisa.fr; raynal@irisa.fr

Abstract: The liveness of concurrent objects despite asynchrony and failures is a fundamental problem. To that end several progress conditions have been proposed. Wait-freedom is the strongest of these conditions: it states that any object operation must terminate if the invoking process does not crash. Obstruction-freedom is a weaker progress condition as it requires progress only when a process executes in isolation for a long enough period.

This paper explores progress conditions in n -process asynchronous read/write systems enriched with base objects with consensus number x , $1 < x \leq n$ (i.e., objects that wait-free solve consensus in a set of x processes). It is easy to solve consensus in such a system if progress is required only when one of the x processes allowed to access the underlying consensus object invokes this object and does not crash. This paper proposes and investigates a stronger progress condition that we call x -wait-freedom (n -wait-freedom is wait-freedom). While it does not need more assumptions than the previous one in order to ensure progress, that condition identifies additional scenarios in which progress is required despite the fact that none of the x processes allowed to access the underlying consensus object participates. The paper then presents and proves correct a consensus algorithm that satisfies this progress condition.

Key-words: Asynchronous system, Consensus number, Fault-Tolerance, Liveness, Obstruction-freedom, Process crash, Progress condition, Shared memory system, Wait-freedom.

Vivacité du nombre de consensus x

Résumé : *Ce rapport étudie la vivacité d'un système accédant des objets de nombre de consensus x .*

Mots clés : *Système asynchrone, Nombre de consensus, Tolérance aux défaillances, Vivacité, Sans-obstruction, Défaillance par crash, Condition de progression, Système à mémoire partagée, Sans-attente.*

^{*} Projet ASAP: équipe commune avec l'INRIA, le CNRS, l'université Rennes 1 et l'INSA de Rennes

^{**} Projet ASAP: équipe commune avec l'INRIA, le CNRS, l'université Rennes 1 et l'INSA de Rennes

1 Introduction

1.1 Context of the work

Consensus object, wait-freedom and consensus number [7] A fundamental issue of asynchronous shared memory systems prone to process crashes, consists in constructing concurrent objects that provide each process p with well-defined progress guarantees. The strongest progress condition, called *wait-freedom*, guarantees that, if any process p invokes an object operation and does not crash, it eventually returns from that invocation (i.e., it returns whatever the behavior of the other processes which can be concurrent, slow or faulty). Wait-freedom is starvation-freedom in presence of failures. The solution to implementations of concurrent objects that satisfy a non-trivial progress condition in presence of process crashes is captured by the *consensus* problem and the notion of *consensus number* of an object.

A consensus object is a concurrent object that allows each process to propose a value and guarantees that (a) every process - that proposes a value and does not crash- decides a value (termination), (b) a decided value is a proposed value (validity), and (c) no two processes decide different values (agreement). As soon as one is supplied with consensus objects, one can build a wait-free implementation of any concurrent object that has a sequential specification.

A concurrent object type μ has consensus number x , if x is the largest integer (or $+\infty$ if there is no such integer) for which a consensus object can be wait-free implemented from objects of type μ and shared read/write atomic registers in a x -process system. This means that any concurrent object can be wait-free implemented in an n -process read/write shared memory system enriched with objects whose consensus number is $x \geq n$. The *wait-free hierarchy* is an infinite hierarchy of objects such that the objects at level x are exactly the objects whose consensus number is x . Atomic read/write registers have consensus number 1, Test&Set objects have consensus number 2, etc., until Compare&Swap or LL/SC objects that have consensus number $+\infty$.

Obstruction-freedom and x -obstruction-freedom [8, 14] *Obstruction-freedom* is a progress condition strictly weaker than wait-freedom. An obstruction-free implementation of an object guarantees that a process that invokes an operation -and does not crash- returns from that invocation if it runs “long enough” in isolation [8] (“long enough” is used to capture the arbitrary duration required by that process to execute the operation). While both wait-freedom and obstruction-freedom are progress conditions whose definition is independent of the actual failure pattern, the second one guarantees progress only in “favorable” concurrency patterns. (The interested reader will find in [6] a failure detector-based approach that boosts obstruction-freedom to wait-freedom.)

x -Obstruction-freedom is a generalization of obstruction-freedom [14, 15]. It guarantees that, for every set of processes P , $|P| \leq x$, every process in P -that does not crash- returns from its operation invocation if no process outside P takes steps for “long enough”. It is easy to see that x -obstruction-freedom and wait-freedom are equivalent in an n -process system such that $n \leq x$. Differently, when $x < n$, x -obstruction-freedom depends on the concurrency pattern while wait-freedom does not.

Let us consider an n -process asynchronous crash-prone system enriched with objects that wait-free solve the consensus problem for a set of x processes, with $x < n$. It is shown in [15] that, in such a system, it is possible to design an x -obstruction-free implementation of any concurrent object (shared by the n processes) defined by a sequential specification.

x -Obstruction-freedom is a *symmetric* progress condition in the sense that it does not favor a non-faulty process with respect to another non-faulty process. All the (non-faulty) processes are equal with respect to the progress condition. Their progress guarantee depends only on the concurrency pattern, and there is no progress guarantee when more than x processes are permanently concurrent.

Concurrency is the adversary when one wants to benefit from x -obstruction-freedom. More precisely, x -obstruction-freedom can benefit from crashes, as it guarantees that any operation on a concurrent object issued by a non-faulty process always terminates as soon as $(n - x)$ or more processes have crashed. Crashes favor the coverage of the assumption on which x -obstruction-freedom relies to become effective [12].

1.2 Content of the paper

When we consider an asynchronous crash-prone n -process system enriched with x -process consensus objects, the x -obstruction-freedom progress condition is a face of the “progress condition” die that (according to the previous discussion) we call *symmetric*.

A “cluster-based” progress condition This paper presents and investigates another face of the “progress condition” die, namely its *asymmetric* face. This definition is motivated by the fact that, in some practical systems, all processes are not equal: due to application, geography, security, etc., there is often a cluster of processes that are more important than the other processes. Let X denote this set of processes, with $|X| = x$ (X can be defined statically or dynamically). The processes of X are called *major* while the other processes are called *minor*.

Roughly speaking, when we consider the termination property of a consensus object, this progress condition is the following. If a major participates and is not faulty, or no major participates and all minors that participate are not faulty, or a process decides, then any participating process that does not crash decides. (Actually, as we will see, the notion of faulty process is related to a vulnerability

window, namely a process can prevent other processes from deciding only if it crashes while executing some well-identified part of its code). This progress condition is called x -wait-freedom.

The paper then describes an algorithm that builds an x -wait-free n -process consensus object in an asynchronous crash-prone n -process system enriched with base x -process consensus objects. Interestingly, this consensus algorithm is not trivial: it has to solve a competition problem in presence of failures. The universal construction described in [15] (that builds x -obstruction-free concurrent objects) can then be adapted to use the proposed n -process x -wait-free consensus algorithm, in order to build x -wait-free concurrent objects.

Discussion It is easy to see that, in a system of n processes, similarly to n -obstruction-freedom, the n -wait-freedom progress condition boils down to wait-freedom. Then, for $x < n$, x -obstruction-freedom and x -wait-freedom cannot be compared. It is also easy to see that, differently from x -obstruction-freedom, x -wait-freedom does not depend on the concurrency pattern, but depends heavily on the crash failure pattern. As an extreme case, x -obstruction-freedom does not guarantee termination in runs where no process crashes but there are always more than x concurrent processes, while x -wait-freedom guarantees it as soon as only one non-faulty major participates. Differently from x -obstruction-freedom, the coverage of the assumption on which x -wait-freedom relies is pretty good in runs with few crashes (what usually occurs in practice).

On another side, the symmetry property of x -obstruction-freedom does not allow it to benefit from the fact that a given set of processes can be more relevant than another one to ensure the liveness of some object. Differently, x -wait-freedom allows distinct sets of processes X_1 and X_2 to be the majors associated with different objects O_1 and O_2 .

While we assume $x \geq 2$, it is nevertheless interesting to look at the case $x = 1$, because it is slightly different from the other values of x . Let us remember that $x = 1$ is the weakest consensus number, namely the one of read/write registers. As we will see in Section 2, when we consider the 1-wait-freedom progress condition, the underlying system is no longer a pure read/write asynchronous system. This is because, the only process in the set X has more power than the others. Differently, when $|X| \geq 2$, the majors can be defined from Test&Set objects (which have consensus number 2).

1.3 Roadmap

The paper is made up of 5 sections. The underlying system model is presented in Section 2. Section 3 defines the x -wait-free consensus problem. Then, Section 4 presents the construction of an x -wait-free consensus object, and proves it correct. It first considers the case where X is statically defined and then the case where it is dynamically defined. Finally, Section 5 concludes the paper.

2 Underlying System model

Asynchronous processes and failure model The system is made up of n asynchronous processes denoted p_1, \dots, p_n . A process executes a sequence of atomic steps as defined by its algorithm.

A process executes correctly its algorithm until it possibly crashes. After it has crashed a process executes no more steps (i.e., a crash is a premature halt). Given a run, a process that crashes is said to be *faulty* in that run, otherwise it is *correct*.

Some part of the algorithm associated with a process is called *its vulnerability window*. If a process does not crash while executing that code, it is said to be *good*. Intuitively, the crash of a process can entail the definitive blocking of other processes, only if it is not *good*. An arbitrary number of processes can crash in a run.

Communication model The processes communicate through three types of objects.

- Atomic read/write registers. These registers are multi-writer/multi-readers registers. Let us remember that such registers can be wait-free implemented on top of safe one-writer/one-reader registers [2, 10, 11, 13].
- Snapshot objects [1]. Such an object is an array with an entry per process. It provides each process p_i with two operations, namely, p_i can write its entry of the array and read the whole array (this operation is denoted `snapshot()`). Both appear as being executed atomically. This means that the operations on a snapshot object are linearizable (i.e., they can be totally ordered, and this order respects their real-time occurrence order) [9].

It is possible to build a wait-free snapshot object on top of a read/write shared memory which means that its consensus number is 1. Consequently, the great advantage of a snapshot object is not its computability power but the abstraction level it provides to its users.

The snapshot objects considered here are one-write objects. This means that, given such an object $SM[1..n]$, a process p_i first writes once $SM[i]$, and then invokes $SM.snapshot()$ (as many times as it wants). It is easy to see that a one-write snapshot object satisfies the following *containment* property. Assuming that SM is initialized to $[\perp, \dots, \perp]$, let $snap1$ and $snap2$ be

two invocations $SM.snapshot()$ that return $sm1$ and $sm2$, respectively, and are such that $snap1$ is ordered before $snap2$ in the linearization order. We have $sm1 \leq sm2$ where $(sm1 \leq sm2) \equiv (\forall i : (sm1[i] \neq \perp) \Rightarrow (sm2[i] = sm1[i]))$.

- Consensus objects for a set of x processes, with $x \geq 2$. Such an object, that can be accessed only by a predefined set X of $|X| = x$ processes, provides them with a single operation denoted $xcons.propose()$ that allows each process to propose a value and obtain a decided value. This object is wait-free: any invocation of $xcons.propose()$ issued by a correct process terminates. Let us remember that no two invocations return different values and a returned value is a value that has been proposed.

Notation All shared objects are denoted with uppercase letters. Differently, local variables are denoted with lowercase letters. Sometimes the index i of process p_i is used as a subscript for its local variables.

On the value of x As already indicated in the introduction, when $x \geq 2$, it is possible to use Test&Set objects to define which are the processes that constitute X (see Section 4.4). This is no longer possible for $x = 1$. This means that a set X of size 1 cannot be dynamically built in a pure read/write asynchronous system.

On another side, when we consider the case where the size 1 set X is statically defined, we obtain a system model stronger than an asynchronous n -process read/write shared memory system (in all runs where the process in X does not crash, consensus can be solved despite the crash of the other processes, while it cannot in a pure read/write system). This is a system with a statically predefined (possibly unreliable) leader. As it is a major, this “leader” cannot be blocked by the other processes. That is the essential difference wrt a pure asynchronous read/write system.

This shows a noteworthy property that distinguishes pure read/write systems from read/write systems enriched with x -process consensus objects such that $x > 1$.

3 n -Process x -wait-free consensus

The aim is to design on top of the previous system model a consensus object that satisfies the x -wait-freedom progress condition. This object offers the operation $xwf.decide()$ to the processes.

Vulnerability window A *vulnerability window* of an algorithm is defined in [4] as “an interval of time during the execution of the algorithm in which the delay or inaccessibility of a single process can cause the entire algorithm to wait indefinitely”.

Keeping its spirit we reformulate this definition as follows: the vulnerability windows of the algorithms that implement an object are the part of their codes such that the crash of a process while executing such a part of code can entail the permanent blocking of correct processes that invoke operations on that object¹.

A few predicates Considering a run of an agreement object that provides the processes with an operation $decide()$, let us first define the following predicates.

- $PART(i)$ is true iff p_i participates in the consensus. From an operational point of view, p_i participates from the first shared memory access entailed by $decide()$.
- $FAULTY(i)$ is true iff p_i crashes.
- $GOOD(i)$ is true iff p_i does not crash in its vulnerability window.
- $DEC(i)$ is true iff p_i returns from $decide()$.

When the vulnerability window notion is (judged) irrelevant, one can take $GOOD(i) \equiv \neg FAULTY(i)$.

x -Wait-free consensus An x -wait-free consensus differs from a classical wait-free consensus object, in its termination property. More precisely, it is defined by the following properties, where X and \bar{X} denote the corresponding sets of major and minor processes, respectively.

- Validity. A decided value is a proposed value.
- Agreement. No two processes decide different values.
- Termination. If $P1 \vee P2 \vee P3$ (as defined below), any correct participant decides.

¹The notion of vulnerability window does no longer consider an object as a “black box”, but as an “open box”. A similar “black/open box” approach is sometimes used in software engineering.

- $P1 \equiv (\exists i \in X : \text{PART}(i) \wedge \text{GOOD}(i))$,
- $P2 \equiv (\forall i \in X : \neg \text{PART}(i)) \wedge (\forall i \in \bar{X} : \text{PART}(i) \Rightarrow \text{GOOD}(i))$,
- $P3 \equiv (\exists i : \text{DEC}(i))$.

Roughly speaking, the termination property states that any correct participant decides if a correct major process participates, or no major participates but all minors that participate are correct. It is important to see that, for any correct participant, this specification does not rule out runs in which its value can be decided.

4 Building an n -process x -wait-free consensus object

This section presents a construction of an x -wait-free consensus object and proves it correct. To that end, an object type called `weak_agreement` is first introduced and proved (this type is a variant of the `safe_agreement` object type defined by Borowsky and Gafni [3]). Then the x -wait-free consensus object is incrementally built. It is first considered that the major set X is statically predetermined. Then, the construction is enriched for solving the case where X is dynamically defined.

4.1 The weak_agreement object type

Definition The `weak_agreement` object type has two operations, denoted `wa_decidei()` and `wa_terminatei()`. Its aim is to allow, under some conditions, the processes to decide a single value from the values they propose (when they invoke `wa_decidei()`). The aim of `wa_terminatei()` is to allow a process to indicate that the corresponding object has become useless (consequently, if processes are blocked inside `wa_decidei()` they become unblocked). When compared to the `safe_agreement` type, the `weak_agreement` type allows for more “termination” at the price of having several decided values.

More formally, the `weak_agreement` type is defined by the three following properties.

- **Validity.** A decided value is a proposed value.
- **Agreement.** If no process ever invokes `wa_terminatei()`, at most one value is decided.
- **Termination.** If (a) no process crashes while executing `wa_decidei()`, or (b) a process decides a value (i.e., returns from `wa_decidei()`), or (c) a process returns from `wa_terminatei()`, any invocation of `wa_decidei()` by a correct process terminates.

A weak_agreement construction This construction is a simple amendment to the construction of the `safe_agreement` type described in [3]. It uses an atomic boolean register `TERM` (initialized to `false`, and set to `true` by the `wa_terminatei()` operation), and two snapshot objects `VAL[1..n]` (initialized to $[\perp, \dots, \perp]$) and `PART[1..n]` (initialized to $[\emptyset, \dots, \emptyset]$) that are used by the processes to cooperate when they invoke `wa_proposei(v)`. `VAL[i]` is used by p_i to deposit the value it proposes, while `PART[i]` is used to store the set of processes that p_i sees as participating.

```

init: for each  $j : 1 \leq j \leq n$  do VAL[j] ← ⊥; PART[j] ← ∅ end for; TERM ← false.

operation wa_decidei(v):
(01) VAL[i] ← v;
(02) vali ← VAL.snapshot(); participantsi ← {j | vali[j] ≠ ⊥};
(03) PART[i] ← participantsi;
(04) repeat parti ← PART.snapshot() until
(05)  $[\exists j : (\text{part}_i[j] \neq \emptyset) \wedge (\forall k : [(k \in \text{part}_i[j]) \Rightarrow (\text{part}_i[k] \neq \emptyset)])] \vee \text{TERM}$ 
(06) end repeat;
(07) if  $(\neg \text{TERM})$ 
(08)   then let s_parti = smallest non-empty participant set in parti[1..n];
(09)     let m = smallest process index in s_part;
           % m is the smallest process index in the smallest participant set known by  $p_i$ 
(10)     val ← VAL.snapshot(); let res = val[m]
(11)   else let res = v
(12) end if;
(13) return(res).

operation wa_terminatei():
(14) TERM ← true.

```

Figure 1: An algorithm implementing the `weak_agreement` type (variant of [3])

The algorithms implementing `wa_decidei()` and `wa_terminatei()` are described in Figure 1. The behavior generated by `wa_decidei()` can be decomposed into several phases.

- First, p_i writes the value v it proposes into $VAL[i]$ (line 01). It then atomically reads the value of the whole array $VAL[1..n]$ to determine its view of the set of participants. Those are the processes that, from p_i 's point of view, have written $VAL[1..n]$ (line 02). Then, p_i writes this set into $PART[i]$ to inform the other processes of its view of which processes are participating (line 03).

Let us observe that if a process p_k crashes after having written $VAL[k]$ and before writing $PART[k]$, we will have forever $VAL[k] \neq \perp \wedge PART[k] = \emptyset$. Conversely, if p_k writes $VAL[k]$ and crashes (if it does) after writing $PART[k]$, we eventually have forever $VAL[k] \neq \perp \wedge PART[k] \neq \emptyset$.

- Then, p_i enters a loop in which it atomically reads the whole array $PART[1..n]$ until some predicate is satisfied. This predicate is made up of two sub-predicates. The simplest one, made of the single boolean $TERM$, is satisfied when a process has invoked $wa_terminate_i()$ (thereby indicating that the value computed by this weak_agreement object has become irrelevant).

The meaning of the other sub-predicate, namely,

$$\exists j : (part_i[j] \neq \emptyset) \wedge (\forall k : [(k \in part_i[j]) \Rightarrow (part_i[k] \neq \emptyset)])$$

is the following: there is a process p_j that (a) has made public its view of which processes participate (this is captured by $part_i[j] \neq \emptyset$), and (b) each process p_k perceived as participant by p_j has made public its view of which are the participating processes (this is captured by $part_i[k] \neq \emptyset$). If item (b) is satisfied, the processes in $PART[j]$ have not crashed between their write into VAL and their write into $PART$. It is important to notice that the local predicate used at line 05 is stable: once true it remains true forever.

- Finally, process p_i computes the value it decides from that weak_agreement object. If $TERM$ is true, p_i may return any value (hence, p_i returns the value v it proposes).

If $TERM$ remains forever false, no two processes are allowed to decide different values. To that end, p_i first computes the smallest non-empty set of processes seen as participants by a process ($s_part_i = part_i[k]$), and then the smallest process index m of this non-empty set. It finally returns the value written in $VAL[m]$. The proof will show that, if $TERM$ remains forever false, $VAL[m] \neq \perp$, and no process returns another value.

Theorem 1 *The algorithm described in Figure 1 implements the weak_agreement object type.*

Proof *Proof of the validity property.* After the boolean $TERM$ has been set to *true* (if it is ever set), a process returns its own proposed value, and the validity trivially follows. Hence, let us assume that $TERM$ is never set to *true*. We have to show that the process p_m (as determined by p_i) has previously written $VAL[m]$. It follows from the text of the algorithm that $VAL[m]$ contains either \perp or the value proposed by p_m .

As the predicate of line 05 is true and stable (because $part_i$ is no longer modified), it follows that s_part_i does exist. This means that there is a process p_j such that $s_part_i = PART[j]$. We then have $m \in PART[j]$, from which follows that p_j sees p_m as participating, and we can conclude that p_m has written $VAL[m]$, before p_j writes $PART[j]$. Consequently, $VAL[m] \neq \perp$ when it is read by p_i at line 10, which concludes the proof of the validity property.

Proof of the termination property. Let us observe that a correct process can be blocked forever only in the repeat loop (lines 04-06). So, the proof consists in showing that a correct process eventually exits from this loop. There are three cases to consider.

- A process executes $wa_terminate_i()$. In that case, no correct process can block forever in the repeat loop.
- No process crashes while executing $wa_decide_i()$. In that case, we eventually have $VAL[k] \neq \perp \wedge PART[k] \neq \emptyset$, for each process p_k that invokes $wa_decide_k()$. It follows that, when this occurs, the predicate of line 05 becomes true, which proves the termination property for that case.
- A process returns from $wa_decide_i()$. If it returns because $TERM$ is true, no process can loop forever in the repeat loop. Hence, let us consider that $TERM$ is always false.

Any process that returns from $wa_decide_i()$, has previously issued a last $PART.snapshot()$ invocation (line 04, this invocation allowed it to exit the repeat loop). These “last” invocations are totally ordered by their linearization order. Let p_i be the process whose last invocation $part_snap1 = PART.snapshot()$ is the first of these “last” invocations.

Let p_ℓ ($\ell \neq i$) be any correct process that invokes $wa_decide_\ell()$. Due to the definition of p_i and the fact that p_ℓ is correct, there is an invocation $part_snap2 = PART.snapshot()$ issued by p_ℓ after $part_snap1$. It follows from the containment property of the one-write snapshot object $PART$ that $part_snap1 \leq part_snap2$. Consequently the entry $part_i[j] = PART[j] \neq \emptyset$ that allowed p_i to exist the loop also allows p_ℓ to exit the loop because we then have $part_\ell[j] = PART[j]$, which concludes the proof of the termination property.

Proof of the agreement property. If the boolean $TERM$ is set to $true$, the agreement property is trivially satisfied. Hence, let us assume that $TERM$ remains always false.

Due to the containment property of the one-write snapshot object VAL , it follows that (a) all the sets $participants_i$ that are computed at line 02, are ordered by inclusion, and consequently, for any two sets $PART[j]$ and $PART[k]$ (which are written at line 03) we have $PART[j] \subseteq PART[k]$ or $PART[k] \subseteq PART[j]$, and (b) any two participant sets of the same size contain the same process indexes. Moreover, as for any i , we have (p_i has written $PART[i]$) $\Rightarrow (i \in PART[i])$, it follows that the corresponding sets s_part_i defined at line 08 are well-defined (they are not empty).

Any participating process p_k such that $k \notin PART[j]$ has not yet written its proposed value in $VAL[k]$ (line 01) when p_j invokes $VAL.snapshot()$ (line 02), which defines $participants_j$. Hence, p_k takes the snapshot that defines $participants_k$ later than p_j , and consequently, $PART[j] \subset PART[k]$. Taking the contrapositive of that observation, it follows that the only processes p_ℓ that can satisfy $PART[\ell] \subseteq PART[j]$ are the processes such that $\ell \in PART[j]$. Once all these processes have written their participant sets $PART[\ell]$ (that is, once the predicate at line 05 is verified), the smallest non-empty participant set present in $PART[1..n]$ will not change. This is due to the following observation. $\forall \ell \in PART[k] \setminus PART[j]$ we have $PART[j] \subset PART[\ell]$, from which follows $|last_part_k[\ell]| = |PART[\ell]| \geq |PART[j]|$ (where $last_part_k$ is the last value of $part_k$ read by p_k when it exits the repeat loop), and consequently $s_part_k = s_part_j$.

It follows that all the processes will then choose the same smallest set and the same smallest index within this set (line 09). Thus, they will all return the same value, which concludes the proof of the agreement property. $\square_{Theorem 1}$

4.2 The $xwf_decide_i()$ operation: static case

A process learns its major/minor status by invoking the operation $major(i)$ that returns $true$ iff $i \in X$. One can imagine that $major()$ is implemented by an underlying daemon. In the static case (the one addressed in this section), the daemon relies on a statically predefined assignment. In the dynamic case (considered later) the daemon uses underlying Test&Set objects.

A process p_i that wants to participate to the n -process consensus invokes the operation $xwf_decide_i(v)$ where v is the value it proposes. The algorithm implementing that operation is described in Figure 2. As already indicated, it has to solve a competition problem, namely among the majors on the one side and the minors on the other side when they compete to impose a decided value.

Base shared objects The processes cooperate by accessing the following base objects.

- A two-entry array $XCONS[0 : 1]$ of x -process consensus objects. The objects can be accessed by the processes of X only (the majors).

$XCONS[1]$ is used by the x majors to agree on a single value from the values they propose.

$XCONS[0]$ is used by the x majors to agree on a single value when considering values proposed by the minor processes and known by majors.

- A weak_agreement object WA . This object is used by the minor processes to agree on a single value from the values they propose.
- A two-entry array $PROP[0 : 1]$ initialized to $[\perp, \perp]$.

$PROP[1]$ will contain the value decided by the majors from $XCONS[1]$.

$PROP[0]$ will contain the value decided by the minors from the weak_agreement object SA .

- $WINNER$ is an atomic register, initialized to \perp , that eventually takes a value in $\{0, 1\}$.

More precisely, the majors and the minors compete to impose the value that is eventually decided. We will have $WINNER = 1$ if the majors win that competition, and $WINNER = 0$ if the minors win.

Process behavior When it invokes $xwf_decide_i()$, a process p_i first invokes $major(i)$ (line 01). Then, its behavior depends on the fact that it is a major or a minor.

- If it is a major, p_i is required to invoke $XCONS[1].xcons_propose(v)$ in order the majors that participate agree (among themselves) on a single value (line 02), and that value is made public by writing it into $PROP[1]$ (line 03).

The current value of $PROP[0]$ is \perp , a value proposed by a minor, or an arbitrary value (see below). This means that different majors can see different values $PROP[0]$. In order they agree on a single value as the value proposed by the minors, the majors use the second x -process consensus object $XCONS[0]$ (line 04). If the value decided from $XCONS[0]$ is \perp , the majors win the competition with the minors, and consequently p_i sets $WINNER$ to 1 and invokes $WA.wa_terminate_i()$ to signal the minors that the WA object is no longer useful (line 05). If majors loose the competition, p_i sets $WINNER$ to 0 (line 06).

```

init:  $PROP[0 : 1] \leftarrow [\perp, \perp]; WINNER \leftarrow \perp.$ 

operation  $xwf\_decide_i(v)$ :
(01) if (major( $i$ ))
(02)   then  $major\_dec_i \leftarrow XCONS[1].xcons\_propose(v)$ ;
(03)      $PROP[1] \leftarrow major\_dec_i$ ;
(04)      $minor\_prop_i \leftarrow XCONS[0].xcons\_propose(PROP[0])$ ;
(05)     if ( $minor\_prop_i = \perp$ ) then  $WINNER \leftarrow 1; WA.wa\_terminate_i()$ 
(06)       else  $WINNER \leftarrow 0$  end if
(07)   else  $minor\_dec_i \leftarrow WA.wa\_decide(v)$ ;
(08)      $PROP[0] \leftarrow minor\_dec_i$ ;
(09)      $major\_prop_i \leftarrow PROP[1]$ ;
(10)     if ( $major\_prop_i = \perp$ ) then  $WINNER \leftarrow 0$  else  $wait(WINNER \neq \perp)$  end if
(11)   end if;
(12)   let  $dec = PROP[WINNER]$ ;
(13)   return( $dec$ ).

```

Figure 2: An algorithm implementing the $xwf_decide_i()$ operation

- If it is a minor, p_i first invokes the underlying weak_agreement object WA (line 07). Then, it deposits the value decided from WA in $PROP[0]$ to make it public (line 08). Then, p_i reads the value (if any) decided by the majors among themselves (line 09). If the majors have not decided a value among themselves, the minors declare unilaterally that they are winners; otherwise, they wait to know which are the winners (line 10).

Finally, p_i decides the value determined by the winners (line 12-13).

Vulnerability windows A major that crashes cannot prevent another major from deciding, it can prevent only minors from deciding. This occurs when the majors crash after having written $PROP[1]$ and before giving a value to the atomic register $WINNER$. In that case, a correct minor can be blocked when it executes $wait(WINNER \neq \perp)$ (line 10). Consequently the vulnerability window of a major is made up of the lines 03-06.

A minor that crashes cannot block forever a correct participating major. On another side, a minor that crashes between line 01 and line 03 of the operation $WA.wa_decide()$ can block other minors when they execute the repeat loop inside $WA.wa_decide()$. The vulnerability window of a minor is consequently made up of line 02 of $WA.wa_propose()$ (Figure 1) that is invoked at line 07 of the $xwf_decide_i()$ operation (Figure 2).

4.3 Proof of the construction

Theorem 2 *The algorithm described in Figure 2 implements the x -wait-free consensus object type.*

Proof *Proof of the validity property.* We have to show that a decided value can be either the value present in $PROP[1]$ (a value proposed by a major process), or the value present in $PROP[0]$ (a value proposed by a minor process).

If the decided value is the value present in $PROP[1]$, then $WINNER$ has been set to 1 by a major process before any process decides. In that case, a major process has written the result of $XCONS[1].xcons_propose()$ in $PROP[1]$ before setting $WINNER$, and this value has been proposed by one of the major processes. The decided value has then been proposed by one of the major processes.

If the decided value is the value present in $PROP[0]$, then $WINNER$ has been set to 0 by either a major process or a minor process before any process decides. If $WINNER$ has been set to 0 by a major process, one of the major processes has observed $PROP[0] \neq \perp$ (lines 04-06). This value has been written to $PROP[0]$ by a minor process and is the value returned by $WA.wa_decide()$ to this minor process. Thus, the value present in $PROP[0]$ has been proposed by a minor process. If $WINNER$ has been set to 0 by a minor process, this process has written to $PROP[0]$ the value returned by $WA.wa_decide()$. The decided value has then been proposed by one of the minor processes, which concludes the proof of the validity property.

Proof of the agreement property. The proof relies on (a) the fact that a major (resp., minor) first writes $PROP[1]$ (resp., $PROP[0]$) and then reads $PROP[0]$ (resp., $PROP[1]$), and (b) the atomicity of the registers, from which follows that, if $PROP[1]$ is written before $PROP[0]$ any minor reads $PROP[1] \neq \perp$, while if $PROP[0]$ is written before $PROP[1]$ any major reads $PROP[0] \neq \perp$.

If $WINNER$ has been set to 1 by one of the major processes (line 05), then one of the major processes (not necessarily the same) has observed $PROP[0] = \perp$ and has proposed \perp to $XCONS[0]$. All the other major processes that execute line 05 will then also set $WINNER$ to 1. Before invoking the operation $XCONS[0].xcons_propose()$, the process that has observed $PROP[0] = \perp$ has previously written the value returned by $XCONS[1].xcons_propose()$ in $PROP[1]$. Thus, all the minor processes will observe $PROP[1] \neq \perp$ and will not set $WINNER$ (line 10). The value of $WINNER$ will not change from 1 to 0.

If $WINNER$ has been set to 0, then one of the major processes has observed $PROP[0] \neq \perp$ and has proposed it to $XCONS[0]$, or a minor process has observed $PROP[1] = \perp$. If a minor process has observed $PROP[1] = \perp$, it has written the value returned by $WA.wa_propose()$ in $PROP[0]$ before reading \perp from $PROP[1]$. Thus, all the major processes will observe $PROP[0] \neq \perp$ and \perp won't be returned by $XCONS[0].xcons_propose()$. All the major processes will then set $WINNER$ to 0. Because a minor process cannot set $WINNER$ to 1, the value of $WINNER$ will not change from 0 to 1. Consequently, once the value of $WINNER$ is set, it cannot change.

If $WINNER = 1$, the value decided is the value present in $PROP[1]$. This is the value returned by $XCONS[1].xcons_propose()$ to the major processes. Because this value is unique and only the major processes write to $PROP[1]$, the same value is decided by all the processes.

If $WINNER = 0$, the value decided is the value present in $PROP[0]$. This is the value returned by $WA.wa_propose()$ to the minor processes. Because a major process that does not set $WINNER$ to 1 does not execute $WA.wa_terminate()$ and because of the properties of the weak_agreement object type, this value is unique. Because only the minor processes write to $PROP[0]$, the same value is decided by all the processes, which concludes the proof of the agreement property.

Proof of the termination property. Let us remember that a good process is a process that does not crash within its vulnerability window.

A major process never executes a loop or a wait statement, and thus all correct participating major processes terminate. A minor process can be blocked during the execution of $WA.wa_propose()$ or in the wait statement at line 10 if $WINNER = \perp$.

If a good major process participates and sets $WINNER$ to 1, it will invoke $WA.wa_terminate()$ and thus, because of the properties of the weak_agreement object type, all correct participating minor processes will terminate. If a good major process participates and sets $WINNER$ to 0, one of the major processes has observed $PROP[0] \neq \perp$, meaning that a minor process has returned from its invocation of $WA.wa_decide()$ and has written $PROP[0]$. Again because of the properties of the weak_agreement object type, all correct participating minor processes will then decide.

If no major process participates and all participating minor processes are good, then the properties of the weak_agreement object type guarantee that all correct minor processes terminate their invocation of $WA.wa_decide()$. They will then observe $PROP[1] = \perp$ and will not execute the wait statement. They will then all decide.

We have already shown that if a major process decides, then all processes decide. If a minor process p_i decides, then it has returned from its invocation of $SA.wa_decide()$, and it has either set $WINNER$ to 0 or observed $WINNER = 1$ (otherwise, it would not have decided). As p_i decides, it follows from the properties of the weak_agreement object type that every correct minor process p_j will terminate its invocation of $WA.wa_decide()$. As eventually $WINNER \neq \perp$, process p_j will decide, which concludes the proof of the termination property. $\square_{Theorem\ 2}$

4.4 The $xwf_decide_i()$ operation: dynamic case

This section addresses the case where, assuming $x \geq 2$, the set X is defined dynamically. Intuitively, the majors are the x first processes that invoke the $xwf_decide_i()$ operation (Figure 2). Let us observe that, as X is determined dynamically, the underlying x -process consensus objects $XCONS[0]$ and $XCONS[1]$ used by that operation are no longer statically known. This requires to replace the invocation of the operations $XCONS[a].xcons_propose()$ at lines 02 and 04 of $xwf_decide_i()$ by invocations of an appropriately defined dynamicity-sensitive operation.

The operation $major(i)$ That operation defines as majors the x “first” processes that invoke it. (Note that any $major(i)$ operation that returns *true* to at most x processes could be used instead.) The notion of “first” is defined with the help of a size x array of Test&Set objects denoted $TS[1..n]$.

Such an object has a single operation, denoted $test\&set()$, that returns *true* to the first process that invokes it. As a Test&Set object is linearizable [9], “first” is well-defined. Moreover the fact that the consensus number of a Test&Set object is 2 is in agreement with the assumption $x \geq 2$. Finally, let us notice that it is possible to build a Test&Set object accessed by any number of processes from Test&Set objects that can be accessed by two predetermined processes only [5].

```

operation major( $i$ ):
(01)  $\ell \leftarrow 1$ ;  $major \leftarrow false$ ;
(02) while ( $\ell \leq x \wedge \neg major$ ) do
(03)    $major \leftarrow TS[\ell].test\&set(); \ell \leftarrow \ell + 1$ 
(04) end while;
(05) return( $major$ ).

```

Figure 3: An algorithm implementing the $major()$ operation

The algorithm implementing `major()` is described in Figure 3. It is easy to see that `true` is returned to the x first processes that invoke it, while `false` is returned to the others. The linearization order is defined as follows. A `major()` operation that returns `true` is linearized at the time of its invocation `TS[ℓ].test&set()` that returns it `true`, and a `major()` operation that returns `false` is linearized at the time of its invocation `TS[x].test&set()`.

The operation `dyn_xcons_proposei()` To solve the problem raised by the fact that the x majors are dynamically defined, two arrays of sets and x -process consensus objects are used. Let m be the number of subsets of size x in the set of the $n \geq x$ process indexes. elements. We have:

- `SET_LIST[1..m]` is an array containing the m subsets of size x . `SETLIST[ℓ]` contains the subset identified by ℓ .
- `XCONS[1..m]` is an array of m x -process consensus objects. `XCONS[ℓ]` is the x -process consensus object that can be accessed only by the x processes whose indexes define the subset `SETLIST[ℓ]`.

```

operation dyn_xcons_proposei(v):
(01) res ← v;
(02) for ℓ from 1 to m do
(03)   if (i ∈ SET_LIST[ℓ]) then res ← XCONS[ℓ].xcons_propose(res) end if;
(04) end for;
(05) return(res).

```

Figure 4: An algorithm implementing the `dyn_xcons_proposei()` operation

The operation `dyn_xcons_proposei()` is described in Figure 4. A major p_i scans `SET_LIST[1..m]` (all majors scan this list in the very same order). When it encounters a set `SET_LIST[ℓ]` such that $i \in SET_LIST[\ell]$, p_i invokes `XCONS[ℓ].dyn_xcons_proposei(res)`, and adopts the returned value as its current estimate of the decided value. Let us notice that, whatever is the set X , as any major that participates and does not crash scans all the list, it necessarily invokes `XCONS[ℓ].dyn_xcons_proposei()` where ℓ is such that `SET_LIST[ℓ] = X`.

Redefining the code of lines 02 and 04 of `xwf_decidei()` The array `XCONS[0 : 1]` of x -process consensus objects is replaced by the array `DXCONS[0 : 1]`, each entry containing an object whose only operation is `dyn_xcons_proposei()`.

It is important to notice that, while the implementation of both `DXCONS[0]` and `DXCONS[1]` can share the same array `SET_LIST[1..m]`, each has to use its own array of base x -process consensus objects, `XCONS[0][1..m]` and `XCONS[1][1..m]`, respectively.

5 Conclusion

Liveness of concurrent object operations is an important issue when one has to face the net effect of concurrency, failures and asynchrony. This paper has introduced a weakened form of wait-freedom that is called *x-wait-freedom*. This is a cluster-based progress condition: it relies on the assumption that, for every concurrent object, there is a set X of processes of size x (these processes are called *majors*) that is “more important” than the other processes (called *minors*). If $x = n$ (where n is the total number of processes) *x-wait-freedom* boils down to wait-freedom.

The *x-wait-freedom* progress condition assumes that the system is equipped with x -process wait-free consensus objects, $1 < x \leq n$. It states that an operation terminates at least when a correct major participates, or, if no major participates, every minor that participates is correct. An n -process consensus algorithm that satisfies the *x-wait-freedom* progress condition has been presented and proved correct. From a practical point of view, its “vulnerability window” is very small. As we have seen, while the statement of the progress condition is pretty simple, the algorithm is not trivial.

Acknowledgments

This work has benefited from the support of the French ANR project SHAMAN.

References

- [1] Afek Y., Attiya H., Dolev D., Gafni E., Merritt M. and Shavit N., Atomic Snapshots of Shared Memory. *Journal of the ACM*, 40(4):873-890, 1993.
- [2] Attiya H. and Welch J., Distributed Computing: Fundamentals, Simulations and Advanced Topics, (2d Edition), *Wiley-Interscience*, 414 pages, 2004.

- [3] Borowsky E. and Gafni E., Generalized FLP Impossibility Results for t -Resilient Asynchronous Computations. *Proc. 25th ACM Symposium on Theory of Computing (STOC'93)*, ACM Press, pp. 91-100, 1993.
- [4] 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.
- [5] Gafni E., Raynal M. and Travers C., Test&set, Adaptive Renaming and Set Agreement: a Guided Visit to Asynchronous Computability. *26th IEEE Symposium on Reliable Distributed Systems (SRDS'07)*, IEEE Computer Press, pp. 93-102, 2007.
- [6] Guerraoui R., Kapalka M. and Kouznetsov P., The Weakest Failure Detectors to Boost Obstruction-Freedom. *Distributed Computing*, 20(6):415-433, 2008.
- [7] Herlihy M.P., Wait-Free Synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124-149, 1991.
- [8] Herlihy M.P., Luchangco V. and Moir M., Obstruction-free synchronization: double-ended queues as an example. *Proc. 23th Int'l IEEE Conference on Distributed Computing Systems (ICDCS'03)*, pp. 522-529, 2003.
- [9] Herlihy M.P. and Wing J.L., Linearizability: a Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463-492, 1990.
- [10] Lamport. L., On Interprocess Communication, Part 1: Basic formalism, Part II: Algorithms. *Distributed Computing*, 1(2):77-101,1986.
- [11] Lynch N.A., Distributed Algorithms. *Morgan Kaufmann Pub.*, San Francisco (CA), 872 pages, 1996.
- [12] Powell D., Failure Mode Assumptions and Assumption Coverage. *22nd Int'l Symposium on Fault-Tolerant Computing (FTCS-22)*, IEEE Computer Society Press, pp.386-395, 1992.
- [13] Taubenfeld G., Synchronization Algorithms and Concurrent Programming. *Pearson Prentice-Hall*, ISBN 0-131-97259-6, 423 pages, 2006.
- [14] Taubenfeld G., Contention-Sensitive Data Structure and Algorithms. *Proc. 23th Int'l Symposium on Distributed Computing (DISC'09)*, Springer Verlag LNCS #5805, pp. 157-171, 2009.
- [15] Taubenfeld G., On the Computational Power of Shared Objects. *Proc. 13th Int'l Conference On Principle Of Distributed Systems (OPODIS 2009)*, Springer Verlag LNCS #5923, pp. 270-284, 2009.