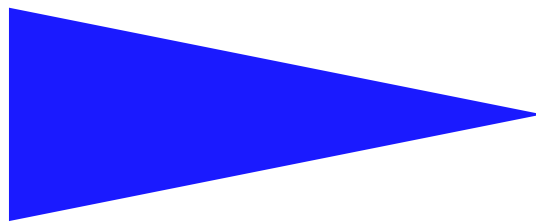


PUBLICATION
INTERNE
N° 1811



**IN SEARCH OF THE HOLY GRAIL:
LOOKING FOR THE WEAKEST FAILURE DETECTOR
FOR WAIT-FREE SET AGREEMENT**

MICHEL RAYNAL CORENTIN TRAVERS

In search of the holy grail: Looking for the weakest failure detector for wait-free set agreement

Michel Raynal* Corentin Travers**

Systèmes communicants

Publication interne n° 1811 — Juillet 2006 — 14 pages

Abstract: Asynchronous failure detector-based set agreement algorithms proposed so far assume that all the processes participate in the algorithm. This means that (at least) the processes that do not crash propose a value and consequently execute the algorithm. It follows that these algorithms can block forever (preventing the correct processes from terminating) when there are correct processes that do not participate in the algorithm. This paper investigates the wait-free set agreement problem, i.e., the case where the correct participating processes have to decide a value whatever the behavior of the other processes (i.e., the processes that crash and the processes that are correct but do not participate in the algorithm). The paper presents a wait-free set agreement algorithm. This algorithm is based on a leader failure detector class that takes into account the notion of participating processes. Interestingly, this algorithm enjoys a first class property, namely, design simplicity.

Key-words: Asynchronous algorithm, Asynchronous system, Atomic register, Consensus, Leader oracle, Participating process, Set agreement, Shared object, Wait-free algorithm.

(Résumé : *tsvp*)

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

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



A la recherche du plus faible détecteur de fautes pour résoudre l'accord ensembliste sans attente

Résumé : Ce rapport propose un détecteur de faute candidat à être le plus faible détecteur de fautes pour résoudre l'accord ensembliste asynchrone et sans attente.

Mots clés : Algorithme asynchrone, Registre atomique, Consensus, Oracle leader, Processus participant, Accord ensembliste, Object partagé, algorithme sans attente.

1 Introduction

The consensus problem Consensus is a fundamental fault-tolerant distributed computing problem. As soon as processes cooperate, they have to agree in one way or another. This is exactly what the consensus problem captures: it allows a set of processes to agree on a critical data (called value, decision, state, etc.). Consensus can be informally defined as follows. Each process proposes a value, and a process that is not faulty has to decide a value (termination), such that there is a single decided value (agreement)¹, and that value is a proposed value (validity).

It is well-known that the consensus problem cannot be solved in asynchronous systems prone to even a single process crash, be these systems read/write shared memory systems [13], or message-passing systems [5]. So, one way to circumvent this impossibility is to enrich the asynchronous system with additional objects that are strong enough to allow solving consensus.

Shared memory systems equipped with objects defined with a sequential specification and more powerful than traditional atomic read/write registers have been investigated. This line of research has produced the notion of *consensus number* that can be associated with each object type defined by a sequential specification [9]. The consensus number of a type is the maximum number of processes for which objects of that type (together with atomic registers) allows solving consensus. For example, the objects provided with a `Test&Set()` operation have consensus number 2, while the objects provided with a `Compare&Swap()` operation have consensus number $+\infty$. The consensus number notion has allowed to establish a hierarchy among the objects (with a sequential specification) according to the synchronization power of the operations they provide to the processes [9].

Another research direction has been the investigation of objects that provide processes with information on failures, namely, the objects called *failure detectors* [2]. A failure detector class² is defined by abstract properties that state which information on failure is provided to the processes. According to the quality of that information, several classes can be defined. Differently from an atomic register or a `Compare&Swap` object, a failure detector has no sequential specification.

As far as one is interested in solving the consensus problem in an asynchronous system prone to process crashes, it has been shown that Ω is the weakest failure detector class that allows solving consensus in such a context [3]. “Weakest” means that any failure detector that allows solving consensus provides information on failures that allows building a failure detector of the class Ω .

A failure detector of the class Ω provides the processes with a primitive, denoted `leader()`, that returns a process identity each time it is called, and eventually always returns the same identity that is the id of a correct process, i.e., a process that does not crash when we consider crash failures. (Examples of message-passing Ω -based consensus algorithms can be found in [7, 12, 16]. These algorithms assume a majority of correct processes, which is a necessary requirement for Ω -based message-passing consensus algorithms.)

The set agreement problem The k -set agreement problem [4] generalizes the consensus problem: it weakens the constraint on the number of decided values by permitting up to k different values to be decided (consensus is 1-set agreement). While k -set agreement can be easily solved in asynchronous systems where the number t of processes that crash is $< k$ (each of a set of k predetermined processes broadcasts its value, and a process decides the first value it receives), this problem has no solution when $k \geq t$ [1, 11, 19]. The failure detector approach to solve the k -set agreement problem in message-passing systems has been investigated in [10, 14, 15, 20].

While (as indicated before) it has been established that Ω is the weakest failure detector class for solving consensus [3], let us remind that finding the weakest failure detector class for solving k -set agreement for $k > 1$ is still an open problem.

The main question Failure detector-based consensus algorithms implicitly consider that all the processes participate in the consensus algorithm, namely, any process that does not crash is implicitly assumed to propose a value and execute the algorithm. This is also an implicit assumption in the statement that Ω is the weakest failure detector to

¹We consider here the uniform version of the consensus problem. A faulty process that decides has to decide the same value as the non-faulty processes.

²We employ the words “failure detector class” instead of “failure detector type”, as it is the word traditionally used in the literature devoted to failure detectors.

solve the consensus problem [3]. Basically, an Ω -based consensus algorithm uses the eventual leader to eventually impose the same value to all the processes. As the algorithm does not know when the leader is elected, its main work consists in guaranteeing that no two different values can be decided before the eventual leader is elected. The algorithm uses the eventual leader to *help* decide all the processes that do not crash.

The previous observation raises the following question: What does happen if the process that is the eventual leader does not participate in the consensus algorithm? It appears that the algorithm can then block forever, and consequently the termination property can no longer be guaranteed.

So, a fundamental question is the following: *What is the weakest failure detector to solve the consensus (or, more generally, the k -set agreement) problem when only a subset of the correct processes (not known a priori) propose a value and participate in the agreement problem?* This question can be reformulated as follows: What are the weakest failure detector classes to *wait-free* solve the consensus and the k -set agreement problems? Wait-free means here that a process that proposes a value and does not crash has to decide, whatever the behavior of the other processes (they can participate or not, and be correct or not). The previous observation on Ω shows that a failure detector of that class cannot be the weakest to wait-free solve the consensus problem.

Content of the paper Answering the previous question requires to investigate new failure detector classes and show that one of them allows solving k -set agreement (sufficiency part) while being the weakest (necessity part). This paper addresses the sufficiency part. (On the necessity side, although we don't have yet formal results, we currently are inclined towards thinking that the failure detector class Ω_*^k -see below- is the weakest failure detector class for wait-free solving k -set agreement.)

More precisely, the paper presents a failure detector-based algorithm for shared memory systems that wait-free solves the k -set agreement problem whatever the number p of participating processes, and their behavior, in a set of n processes³. This algorithm assumes that, in addition to single-writer/multi-readers atomic registers, the shared memory provides the processes with a failure detector object of a class that we denote Ω_*^k . That class is an extension of the failure detector class introduced in [18], and the failure detector classes recently introduced in [6] and [17].

The failure detector class Ω^k introduced in [18] extends the classical Ω class [3] by allowing a set of up to k leaders to be returned by each invocation of the $\text{leader}()$ primitive (Ω^1 boils down to Ω). The set of k leaders that is eventually returned forever includes at least one correct process. The aim of the class Ω^* introduced in [6] is to boost obstruction-free algorithms into non-blocking algorithms. That paper also shows that this failure detector class is the weakest for such a boosting. The failure detector class introduced in [17] extends Ω^k by explicitly referring to the notion of participating processes. It has been introduced to circumvent the $2p - 1$ lower bound of the renaming problem [11] (where p is the number of participating processes). Using such a failure detector, the proposed renaming algorithm provides the processes with a renaming space whose size is reduced from $2p - 1$ to $2p - \lceil \frac{p}{k} \rceil$ (where the value k comes from " k "-set agreement).

Roadmap The paper is made up of 6 sections. Section 2 presents the computation model. Section 3 introduces the failure detector class Ω_*^k . Then, Section 4 presents the Ω_*^k -based k -set algorithm. This algorithm uses an underlying object denoted KA . So, Section 5 presents an algorithm constructing a KA object from atomic read/write registers. Finally, Section 6 concludes the paper.

2 Asynchronous system model

2.1 Process and communication model

Process model The system consists of n sequential processes that we denote p_1, \dots, p_n . A process can crash. Given an execution, a process that crashes is said to be *faulty*, otherwise it is *correct* in that execution. Each process progresses at its own speed, which means that the system is asynchronous. In the following, *Correct* denoted the set of processes that are correct in the run that is considered.

³Let us remind that all the algorithms based on an object O with consensus number n allows solving consensus whatever the number ($p \leq n$) and the behavior of the participating processes, i.e., they are wait-free consensus algorithms. (Such an object O has always a sequential specification.) In some sense, this paper looks for a failure detector class that, while being as weak as possible, is as strong as the object O , i.e., a class that allows designing wait-free failure detector-based set agreement algorithms. (Failure detectors cannot be defined from a sequential specification.)

Coordination model The processes cooperate and communicate through two types of reliable objects: two arrays of single-writer/multi-reader atomic registers and a shared object that we call KA (as shown in Section 5, such an object can be built from single-writer/multi-reader atomic registers). The processes are also provided with a failure detector object of the class Ω_*^k (see below).

Identifiers with upper case letters are used to denote shared objects. Lower case letters are used to denote local variables; in that case the process index appears as a subscript. As an example, $part_i[j]$ denotes the j th entry of a local array of the process p_i , while $PART[j]$ denotes the j th entry of the shared array $PART$.

2.2 The KA object

The KA object is a variant of a round-based object introduced in [8] to capture the safety properties of Paxos-like consensus algorithms [8, 12]. This object provides the processes with an operation denoted $\text{alpha_propose}_k()$. That operation has two input parameters: the value (v_i) proposed by the invoking process p_i , and a round number (r_i). The round numbers play the role of a logical time and allows identifying the invocations. The KA object assumes that no two processes use the same round numbers, and successive invocations by the same process use increasing round numbers. Given a KA object, the invocations $\text{alpha_propose}_k()$ satisfy the following properties (\perp is a default value that cannot be proposed by a process):

- Termination (wait-free): an invocation of $\text{alpha_propose}_k()$ by a correct process always terminates (whatever the behavior of the other processes).
- Validity: the value returned by any invocation $\text{alpha_propose}_k()$ is a proposed value or \perp .
- Agreement: At most k different non- \perp values can be returned by the whole set of $\text{alpha_propose}_k()$ invocations.
- Convergence: If there is a time after which the operation $\text{alpha_propose}_k()$ is invoked infinitely often, and these invocations are issued by an (unknown but fixed) set of at most k processes, there is then a time after which none of these invocations returns \perp .

A KA object can store up to k non- \perp different values. A process invokes it with a value to store and obtains a value in return. If it is permanently accessed concurrently by more than k processes, the KA object might store anything. If there is a period during which it is accessed concurrently by at most $k' \leq k$ processes, it stores forever the corresponding k' proposed values.

3 The failure detector class Ω_*^k

3.1 Definition

A failure detector of the class Ω_*^k provides the processes with an operation denoted $\text{leader}()$. (As indicated in the introduction, this definition is inspired by the leader failure detector classes introduced in [6, 17, 18].) When a process p_i invokes that operation, it provides it with an input parameter, namely a set X of processes, and obtains a set of process identities as a result⁴.

The semantics of Ω_*^k is based on a notion of time, whose domain is the set of integers. It is important to notice that this notion of time is not accessible to the processes. An invocation of $\text{leader}(X)$ by a process p_i is *meaningful* if $i \in X$. If $i \notin X$, it is *meaningless*. The primitive $\text{leader}()$ is defined by the following properties:

- Termination (wait-free): Any invocation of $\text{leader}()$ by a correct process always terminates (whatever the behavior of the other processes).
- Triviality: A meaningless invocation returns any set of processes.
- Eventual multi-leadership for each input set X : For any $X \subseteq \Pi$, such that $X \cap \text{Correct} \neq \emptyset$, there is a time τ_X such that, $\forall \tau \geq \tau_X$, all the meaningful $\text{leader}(X)$ invocations (that terminate) return the same set L_X and this set is such that:

⁴The definition of Ω_*^k is not expressed in the framework introduced by Chandra and Toueg to define failure detector classes [2]. More precisely, in their framework, the failure detector operation that a process can issue has no input parameter. It would be possible to express Ω_*^k in their framework. We don't do it in order to make the presentation simpler.

- $|L_X| \leq k$.
- $L_X \cap X \cap \text{Correct} \neq \emptyset$.

The intuition that underlies this definition is the following. The set X passed as input parameter by the invoking process p_i is the set of all the processes that p_i considers as being currently *participating* in the computation. (This also motivates the notion of meaningful and meaningless invocations: an invoking process is trivially participating).

Given a set X of participating processes that invoke $\text{leader}(X)$, the eventual multi-leadership property states that there is a time after which these processes obtain the same set L_X of at most k leaders, and at least one of them is a correct process of X . Let us observe that the (at most $k - 1$) other processes of L_X can be any subset of processes (correct or not, participating or not).

It is important to notice that the time τ_X from which this 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 $\text{leader}(X)$ invocations are concerned, can obtain different sets of any number of leaders. Let us also observe that if a process p_i issues two meaningful invocations $\text{leader}(X1)$ and $\text{leader}(X2)$ with $X1 \neq X2$, there is no relation linking L_{X1} and L_{X2} , whatever the values of $X1$ and $X2$ (e.g., the fact that $X1 \subset X2$ imposes no particular constraint on L_{X1} and L_{X2}).

Let us consider an execution in which all the invocations $\text{leader}(X)$ are such that $X = \Pi$ (the whole set of processes are always considered as participating). In that case, Ω_*^k boils down to the failure detector class denoted Ω^k introduced in [18]. If additionally, $k = 1$, we obtain the classical leader failure detector Ω introduced in [3]. When $k = 1$, Ω_*^k boils down to the failure detector class introduced in [6]. It is shown in [6] that Ω is weaker than Ω_*^1 that in turn is weaker than $\diamond\mathcal{P}$ (the class of eventually perfect failure detectors: after some finite but unknown time, these failure detectors suspect all the crashed processes and only them [2]).

3.2 The family $\{\Omega_*^k\}_{1 \leq k \leq n}$

It follows from the definition of Ω_*^k , that the failure detector class family $\{\Omega_*^k\}_{1 \leq k \leq n}$ is such that $\Omega_*^k \subseteq \Omega_*^{k+1}$.

Moreover, as just indicated, when all the $\text{leader}(X)$ invocations are such that $X = \Pi$, Ω_*^k boils down to Ω^k (as defined in [18]), from which it follows that we have $\Omega^k \subseteq \Omega_*^k$. More generally, the failure detector classes Ω^k and Ω_*^k are related as indicated in Figure 1 where $A \rightarrow B$ means that any failure detector of the class A can be used to build a failure detector of the class B , and $A \cdots > B$ means that it is not possible to build a failure detector of the class B from a failure detector of the class A .

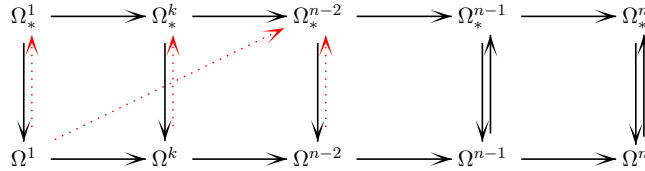


Figure 1: Wait-free (ir)reducibility results between the families $(\Omega_*^x)_{1 \leq x \leq n}$ and $(\Omega^y)_{1 \leq y \leq n}$

- The fact that $\Omega^k \subseteq \Omega_*^k$ (top-down plain arrows) follows from the definitions of the failure detector classes.
- The fact that Ω^n and Ω_*^n are the same class (top-down and bottom-up arrows at the right) follows directly from their definitions.
- The fact that $\forall k : 1 \leq k < n - 1, \forall k' : 1 \leq k' \leq n$, it is not possible to build a failure detector of the class Ω_*^k from a failure detector of the class $\Omega^{k'}$ (dotted arrows) is established in Theorem 1.
- The fact that it is possible to construct a failure detector of the class Ω_*^{n-1} from any failure detector of the class Ω^{n-1} is established in Theorem 2.

Theorem 1 $\forall k : 1 \leq k < n - 1, \forall k' : 1 \leq k' \leq n$, it is not possible to build a failure detector of the class Ω_*^k from a failure detector of the class $\Omega^{k'}$.

Proof To prove the theorem, it suffices to show that it is not possible to build a failure detector of the class Ω_*^{n-2} (the weakest class in the family $(\Omega_*^k)_{1 \leq k \leq n-2}$) from a failure detector of the class Ω^1 (the strongest class in the family $(\Omega^k)_{1 \leq k \leq n}$). The proof is by contradiction. Let us assume that there is an algorithm \mathcal{A} that builds a failure detector of the class Ω_*^{n-2} from a failure detector of the class Ω^1 ⁽⁵⁾. We construct an infinite run in which at least one of the failure detectors Ω^1 or Ω_*^{n-2} fails to meet its specification. The construction uses the following claim. Let $\text{LEADER}()$ denote the leader primitive of Ω^1 . Recall that $\text{leader}(X)$ is the leader primitive of Ω_*^{n-2} .

Claim C. Let R be an arbitrary run in which each process is correct. Moreover, in run R , each process p_i periodically invokes $\text{leader}(X)$ for each X such that $i \in X \wedge |X| = n - 1$. Let τ be a time at which the leadership properties of both Ω^1 and Ω_*^{n-2} are satisfied (i.e., all the invocations of $\text{LEADER}()$ return the same process id, and, for any set X , all the invocations $\text{leader}(X)$ return the same set). We claim that there is a run R_1 of the algorithm \mathcal{A} such that

- R_1 is indistinguishable from R up to time τ .
- In R_1 , there is a process p_x and a time $\tau_1 > \tau$ such that (1) the outputs of Ω^1 at τ and τ_1 are different, or (2) the outputs of Ω_*^{n-2} (for some set X) at τ and τ_1 are different.
- No process crashes in R_1 .

Proof of the claim. By the claim assumption, the leadership properties of both Ω^1 and Ω_*^{n-2} are satisfied at time τ in R . In particular, we have:

1. Ω^1 outputs at each process the same leader identity ℓ .
2. Let $X = \Pi - \{\ell\}$. At time τ , $\exists L$ such that $|L| \leq n - 2$, and, for each process $p_i \in X$, $\text{leader}(X) = L$. Let $L' = L \cap X$. We have $|L'| \leq n - 2$. As $|X| = n - 1$, there is a process p_x such that $x \in X - L'$. Moreover, $x \neq \ell$ (because $X = \Pi - \{\ell\}$).

We show that, from the previous observations, we can build a run R_1 , identical to R up to time τ , such that there exists a time $\tau' > \tau$ at which we have:

- Ω^1 outputs at some process a leader $\ell_1 \neq \ell$ or,
- At process p_x , $\text{leader}(X) \neq L$.

Let us consider the run R' defined as follows. R' is the same as R up to time τ . At time $\tau + 1$, every process in L' crashes. Moreover, from τ , all the invocations $\text{LEADER}()$ of Ω^1 output ℓ in R' . Due to the eventual multi-leadership property of Ω_*^{n-2} , there is a time $\tau_1 > \tau + 1$ such that the invocations of $\text{leader}(X)$ at process p_x return $L_1 \neq L$. This is because the set that is returned (namely, L_1) has to be such that $L_1 \cap X \cap \text{Correct} \neq \emptyset$, and, as the processes of L' have crashed, we have $L \cap X \cap \text{Correct} = \emptyset$ (recall that $L' = L \cap X$).

Let us now consider the run R_1 identical to R up to time $\tau + 1$. During the interval $[\tau + 1, \tau_1]$ the processes in L' do not take any step as far the algorithm \mathcal{A} is concerned. The other processes behave exactly as in R' . If Ω^1 outputs $\ell' \neq \ell$ at some process p_y , the claim follows. Otherwise, let us observe that, for any process p_y (such that $y \notin L'$), R_1 cannot be distinguished from R' . In particular, the algorithm \mathcal{A} outputs, at process p_x , $L_1 \neq L$ at time τ_1 when p_x issues $\text{leader}(X)$. *End of the proof of the claim.*

Let us consider an arbitrary run R_0 in which each process is correct. There is a time τ_1 (1) at which the leadership properties of both failure detectors Ω^1 and Ω_*^{n-2} are satisfied, and (2) each process has taken at least one step. By Claim C, we can build a run R_1 identical to run R up to time τ_1 and such that the output of Ω^1 or Ω_*^{n-2} (for some input parameter X) at some process has changed at time $\tau'_1 > \tau_1$.

In run R_1 , we can find a time $\tau_2 > \tau'_1$ such that each process has taken at least one step between τ'_1 and τ_2 and the leadership properties of Ω^1 and Ω_*^{n-2} are satisfied at time τ_2 . By applying Claim C, we can build a run R_2 identical to R_1 up to time τ_2 , etc. By iterating this process, we obtain an infinite run R and an infinite sequence of increasing times $(\tau_1, \tau'_1, \tau_2, \tau'_2, \dots)$ such that $\forall i > 0, \exists p_{x(i)}$ such that, at $p_{x(i)}$ the output of Ω^1 or Ω_*^{n-2} (for some parameter X) is not the same at time τ_i and τ'_i . Due to the eventual leadership property of Ω , there is a time after which the output

⁵Let us recall that the output of Ω^1 at a given process p_i is *local*. This means that for the output of Ω^1 at p_i be known by the thread implementing the algorithm \mathcal{A} at p_j , it is necessary that that output be written in the shared memory.

of Ω^1 does not change at each process. Consequently, it follows that in run R algorithm \mathcal{A} fails to implement Ω_*^{n-2} . $\square_{Theorem 1}$

Theorem 2 *Given any failure detector of the class Ω^{n-1} , it is possible to build a failure detector of the class Ω_*^{n-1} .*

Proof The proof is constructive. Let us consider any failure detector of the class Ω^{n-1} , and let $LEADER()$ be its leader primitive. Let us consider the operation $leader(X)$ defined as follows:

operation $leader(X)$: **if** $X = \Pi$ **then return** ($LEADER()$) **else return** (X) **end_if**.

We show that $leader(X)$ satisfies the properties of the class Ω_*^{n-1} .

Let us first consider the case where $X = \Pi$. Due to the properties of Ω^{n-1} , there is a time after which $LEADER()$ always returns the same set L_X such that $|L_X| = n - 1$ and $L_X \cap Correct \neq \emptyset$. It trivially follows that $X \cap L_X \cap Correct \neq \emptyset$. Consequently, the eventual multi-leadership property is satisfied for the invocations $leader(\Pi)$.

Given any set X such that $X \neq \Pi$, let us now consider the case of the invocations $leader(X)$. The definition of $leader(X)$ indicates that the set $L_X = X$ is then returned by these invocations, and we have $|X| = |L_X| \leq n - 1$. If X contains at least one correct process, we have $X \cap L_X \cap Correct \neq \emptyset$, and the eventual multi-leadership property is satisfied for the invocations $leader(X)$. If X contains no correct process, the set returned by $leader(X)$ can be arbitrary. $\square_{Theorem 2}$

4 Ω_*^k -based k -set agreement

4.1 Wait-free k -set agreement

The k -set agreement has been informally stated in the introduction. It has been defined in [4]. The parameter k of the set agreement can be seen as the degree of coordination associated with the corresponding instance of the problem. The smaller k , the more coordination among the processes: $k = 1$ means the strongest possible coordination (this is the consensus problem), while $k = n$ means no coordination at all. More precisely, in a set of n processes, each of a subset of $p \geq 1$ processes proposes a value. These processes are the *participating* processes. The wait-free k -set agreement is defined by the following properties:

- Termination (wait-free): a correct process that proposes a value decides a value (whatever the behavior of the other processes).
- Agreement: no more than k different values are decided.
- Validity: a decided value is a proposed value.

4.2 Principles and description of the algorithm

The k -set agreement algorithm is described in Figure 2. A process p_i that participates in the k -set agreement invokes the operation $kset_propose_k(v_i)$ where v_i is the value it proposes. If it does not crash, it terminates that operation when it executes the statement $return(decided_i)$ (line 11) where $decided_i$ is the value it decides.

Shared objects The processes share three objects:

- A KA object. A process p_i accesses it by invoking $KA.alpha_propose_k(r_i, v_i)$ where r_i is a round number and the value v_i proposed by p_i (line 07). Due to the properties of the KA object, the value returned by such an invocation is a proposed value or \perp .
- An array of atomic single-writer/multi-reader boolean registers, $PART[1..n]$. The register $PART[i]$ can be updated only by p_i ; it can read by all the processes. Each entry $PART[i]$ is initialized to *false*. $PART[i]$ is switched to *true* to indicate that p_i is now participating in the k -set agreement (line 01). $PART[i]$ is updated at most once.

- An array of atomic single-writer/multi-reader registers denoted $DEC[1..n]$. $DEC[i]$ can be written only by p_i . It can read by all the processes. Each entry $DEC[i]$ is initialized to \perp (a value that cannot be proposed by the processes). When it is updated to a non- \perp value v , that value v can be decided by any process. It is updated (to such a value v or \perp) each time p_i invokes $KA.alpha_propose_k(r_i, v_i)$ to store the value returned by that invocation (line 07).

The algorithm The behavior of a process is pretty simple. As in Paxos, it decouples the safety part from the wait-free/termination part. The safety is ensured thanks to the KA object, while the liveness rests on Ω_*^k .

After it has registered its participation (line 01), a process p_i executes a **while** loop (lines 03-09) until it finds a non- \perp entry in the $DEC[1..n]$ array. When this occurs, p_i decides such a value (lines 03 and 10-11).

Each time it executes the **while** loop, p_i first computes its local view (denoted $part_i$) of the set of the processes it perceives as being the participating processes (line 04). It then uses this participating set to invoke $Leader_i()$ (line 05). If it does not belong to the set returned by $Leader_i(part_i)$, p_i continues looping. Otherwise (it then belongs to set of leaders), p_i invokes the KA object (line 07) to try to obtain a non- \perp value from that object. The local variable r_i is used by p_i to define the round number it uses when it invokes the KA object. It is easy to see from the management of r_i at line 02 and line 06 that each process uses increasing round numbers, and that no two processes use the same round numbers (a necessary requirement for using the KA object). The properties of KA ensure that no more than k values are decided, while the properties of Ω_*^k ensure that all the correct participating processes do terminate.

```

operation kset_proposek(vi):
(1)  PART[i] ← true;
(2)  ri ← (i - n);
(3)  while (∃j : DEC[j] = ⊥) do
(4)    parti ← {j : PART[j] ≠ ⊥};
(5)    leadersi ← Leaderi(parti);
(6)    if (i ∈ leadersi) then ri ← ri + n;
(7)                                DEC[i] ← KA.alpha_proposek(ri, vi)
(8)    end_if
(9)  end_while;
(10) let decidedi = any DEC[j] ≠ ⊥;
(11) return(decidedi)

```

Figure 2: An Ω_*^k -based k -set agreement algorithm (code for p_i)

4.3 Proof of the algorithm

Theorem 3 *The algorithm described in Figure 2 wait-free solves the k -set agreement problem whatever the number p of participating processes in a set of n processes (this number p being a priori unknown).*

Proof

Validity The validity property follows from the following observations:

- The value \perp cannot be decided (lines 03 and 10).
- The $DEC[1..n]$ array can contain only \perp or values that have been proposed to the KA object (line 07).
- Any value v_i proposed to the KA object is a value proposed to the k -set agreement.

Agreement The agreement property follows directly from the agreement property of the KA object (that states that at most k non- \perp values can be returned from that object).

Termination (wait-free) If an entry of $DEC[1..n]$ is eventually set to a non- \perp value, it follows from the test of line 03 that any correct participating process terminates. So, let us assume by contradiction that no entry of $DEC[1..n]$ is ever set to a non- \perp value. Let us first observe that all the $leader()$ invocations issued by the processes are meaningful.

If no correct participating process decides, there is a time τ_0 after which we have the following:

- All the participating processes have entered the algorithm, and consequently the array $PART[1..n]$ determines the whole set of participating processes. Let X be this set of processes.
- all the $leader()$ invocations have X as input parameter.

It follows from the eventual multi-leadership property associated with X , that there is a time $\tau_X \geq \tau_0$ such that, for all the times $\tau \geq \tau_X$, all the invocations of $leader(X)$ return the same set L_X of at most k processes, and this set includes at least one correct participating process.

As no process decides (assumption) and each $\alpha_propose_k()$ invocation issued by a correct process returns (termination property of the KA object), the correct participating processes of the set X execute $KA.\alpha_propose_k()$ infinitely often (lines 06-07). It then follows from the convergence property of the KA object that these processes obtain non- \perp values, and deposit these values in the array $DEC[1..n]$. A contradiction. \square *Theorem 3*

5 Building a KA object from registers

This section presents an implementation of a KA object from single-writer/multi-readers atomic registers. As already indicated, this algorithm is inspired from Paxos-like algorithms [8, 12].

5.1 Implementing KA

An algorithm constructing a KA object is described in Figure 3. It uses an array of single-writer/multi-reader atomic registers REG . As previously, $REG[i]$ can be written only by p_i . A register $REG[i]$ is made up of three fields $REG[i].lre$, $REG[i].lrww$ and $REG[i].val$ whose meaning is the following ($REG[i]$ is initialized to $\langle 0, 0, \perp \rangle$):

- $REG[i].lre$ stores the number of the last round entered by p_i . It can be seen as the logical date of the last invocation issued by p_i .
- $REG[i].lrww$ and $REG[i].val$ constitute a pair of related values: $REG[i].lrww$ stores the number of the last round with a write of a value in the field $REG[i].val$. So, $REG[i].lrww$ is the logical date of the last write in $REG[i].val$.

(To simplify the writing of the algorithm, we consider that each field of a register can be written separately. This poses no problem as each register is single writer. A writer can consequently keep a copy of the last value it has written in each register field and rewrite it when that value is not modified.)

```

operation  $\alpha\_propose_k(r, v)$ :
(1)   $REG[i].lre \leftarrow r$ ;
(2)  for  $j \in \{1, \dots, n\}$  do  $reg_i[j] \leftarrow REG[j]$  end\_do;
(3)  let  $value_i$  be  $reg_i[j].val$  where  $j$  is such that  $\forall x : reg_i[j].lrww \geq reg_i[x].lrww$ ;
(4)  if ( $value_i = \perp$ ) then  $value_i \leftarrow v$  end\_if;
(5)   $REG[i].(lrww, v) \leftarrow (r, value_i)$ ;
(6)  for  $j \in \{1, \dots, n\}$  do  $reg_i[j] \leftarrow REG[j]$  end\_do;
(7)  if ( $|\{j | reg_i[j].lre \geq r\}| > k$ ) then  $\text{return}(\perp)$ 
(8)  else  $\text{return}(value_i)$  end\_if

```

Figure 3: A KA object algorithm (code for p_i)

The principle that underlies the algorithm is very simple: it consists in using a logical time frame (represented here by the round numbers) to timestamp the invocations, and answering \perp when the timestamp of the corresponding invocation does not lie within the k highest dates (registered in $REG[1..n].lre$). To that end, the algorithm proceeds as follows:

- Step 1 (lines 01-02): Access the shared registers.
 - When a process p_i invokes $\text{alpha_propose}_k(r, v)$, it first informs the other processes that the KA object has attained (at least) the date r (line 01). Then p_i reads all the registers in any order (line 02) to know the last values (if any) written by the other processes.
- Step 2 (lines 03-05): Determination and writing of a value.

Then, p_i determines a value. In order not to violate the agreement property, it selects the last value (“last” according to the round numbers/logical dates) that has been deposited in a register $REG[j]$. If there is no such value it considers its own value v . After this determination, p_i writes in $REG[i]$ the value it has determined, together with its round number (line 05).
- Step 3 (lines 06-08): Commit or abort.
 - p_i reads again the shared registers to know the progress of the other processes (measured by their round numbers), line 07. If it discovers it is “late”, p_i aborts returning \perp . (Let us observe that this preserves the agreement property.) “To be late” means that the current date r of p_i does not lie within the window defined by the k highest dates (round numbers) currently entered by the processes (these round numbers/dates are registered in the field lre of each entry of the array $REG[1..n]$).
 - Otherwise, p_i is not late. It then returns (“commits”) $value_i$ (line 08). Let us observe that, as the notion of “being late” is defined with respect to a window of k dates (round numbers), it is possible that up to k processes are not late and return concurrently up to k distinct non- \perp values.

It directly follows from the code that the algorithm is wait-free. Moreover, in order to expedite the $\text{alpha_propose}_k()$ operation, it is possible to insert the statement

if ($|\{j | reg_i[j].lre \geq r\}| > k$) **then** $\text{return}(\perp)$ **end if**

between the line 02 and the line 03. This allows the invoking process to return \perp when, just after entering the $\text{alpha_propose}_k()$ operation, it discovers it is late.

5.2 Proof of the KA object

Theorem 4 *The algorithm described in Figure 3 wait-free implements a KA object.*

Proof

Termination (wait-free) This property follows directly from the code of the algorithm (the only loops are **for** loops that trivially terminate when the invoking process is correct).

Validity Let us observe that if a value v is written in $REG[i].val$, that value has been previously passed as a parameter in an $\text{alpha_propose}_k()$ invocation. The validity property follows from this observation and the fact that only \perp or a value written in a register $REG[i]$ can be returned from an $\text{alpha_propose}_k()$ invocation.

Convergence Let τ be a time after which there is a set of $k' \leq k$ processes such that each of them invokes $\text{alpha_propose}_k()$ infinitely often. This means that, from τ , the values of $n - k'$ registers $REG[x]$ are no longer modified. Consequently, as the k' processes p_j repeatedly invoke $\text{alpha_propose}_k()$, there is a time $\tau' \geq \tau$ after which each $REG[j].lre$ becomes greater than any $REG[x].lre$ that is no longer modified. There is consequently a time $\tau'' \geq \tau'$ after which the k' processes are such that their registers $REG[j].lre$ contain forever the k greatest timestamp values. It follows from the test done at line 07 that, after τ'' , no $\text{alpha_propose}_k()$ invocation by one of these k' processes can be aborted. Consequently, each of them returns a non- \perp value at line 08.

Agreement If all invocations returns \perp , the agreement property is trivially satisfied. So, let us consider an execution in which at least one $\text{alpha_propose}_k()$ invocation returns a non- \perp value. To prove the agreement property we show that:

- Before the first non- \perp value is returned by an invocation, there is a time at which the algorithm has determined a set V of at least one and at most k non- \perp values⁶.
- Any value $v \neq \perp$ returned by an invocation is a value of V .

To simplify the reasoning, and without loss of generality, we assume that a process that repeatedly invokes `alpha_proposek()`, stops invoking that operation as soon as it returns a non- \perp value at line 08.

1. Invariants. $\forall j \in \{1, \dots, n\}$:

- $REG[j].lre$ is increasing (assumption on the successive round numbers used by p_j).
- $REG[j].lrww \leq REG[j].lre$ (because p_j executes line 05 after line 01).

2. Among all the invocations that execute the test of line 07, let \mathcal{I} be the subset of invocations for which the predicate $|\{j | reg_i[j].lre \geq r\}| \leq k$ is true. (This means that any invocation of \mathcal{I} either returns a non- \perp value -at line 08-, or crashes after it has evaluated the predicate at line 07, and before it executes line 08.) Among the invocations of \mathcal{I} , let I be the invocation with the smallest round number. Let p_{j_1} be the process that invoked I and r the corresponding round number.

3. Time instants (see Figure 4).

- Let τ be the time at which I executes line 05 (statement $REG[j_1] \leftarrow \langle r, r, v \rangle$).
- Let τ' be the time just after I has finished reading the array $REG[1..n]$. Without loss of generality, we consider that this is the time at which I locally evaluates the predicate of line 07.
- Let $\tau[j]$ be the time at which I reads $REG[j]$ at line 06. We have $\tau < \tau[j] < \tau'$.

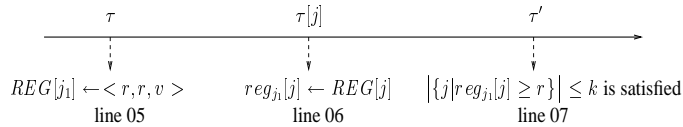


Figure 4: Time instants with respect to accesses to the registers $REG[1..n]$

4. From $\tau[j] < \tau'$, the fact that predicate $|\{j | reg_{j_1}[j].lre \geq r\}| \leq k$ is true at τ' , and the monotonicity of $REG[j].lre$, we can conclude that a necessary requirement for the predicate $REG[j].lre \geq r$ to be true at τ is that it is true at τ' .

Let $L = \{j_1, \dots, j_x, \dots, j_\ell\}$ be the set of processes p_j such that $REG[j].lre \geq r$ is true at τ . As the predicate $|\{j | reg_i[j].lre \geq r\}| \leq k$ is true at τ' , we have $1 \leq \ell = |L| \leq k$.

5. From the previous item, we conclude that there are at least $n - \ell \geq n - k$ entries j of the array $REG[1..n]$ such that $REG[j].lre < r$ at time τ . Let \bar{L} denote this set of processes (L and \bar{L} define a partition of the whole set of processes).

6. Let the τ -time invocation of p_j be the invocation issued by p_j whose round number is the value of $REG[j].lre$ at time τ (assuming a fictitious initial invocation if needed).

7. The τ -time invocations of the processes p_j in L define a set, denoted V , including at most $\ell \leq k$ values, such that these values are written in $REG[1..n]$ with a write timestamp (value of the field $REG[j].lrww$) that is $\geq r$. This claim follows from the following observation.

- The τ -time invocation by p_{j_1} (namely I) writes a value and the round number r in $REG[j_1]$.
- Let $p_{j_x} \in L$, $p_{j_x} \neq p_{j_1}$. From the definition of L , it follows that the round number of the τ -time invocation issued by p_{j_x} is $REG[j_x].lre = r' > r$. When it executes that invocation, p_{j_x} atomically executes $REG[j_x] \leftarrow \langle r', r', v' \rangle$ (if it does not crash before executing the line 05).

⁶According to the terminology introduced in [2], the set V defines the values that are *locked*. This means that from now on the set of non- \perp values that can be returned is fixed forever: no value outside V can ever be returned.

- It is possible that, on one side, no process in L crashes before executing line 05, and, on another side, all the values that are written are different. It consequently follows that up to $\ell \leq k$ different values (with a write timestamp $lrww \geq r$) can be written in $REG[1..n]$. Hence, V can contain up to k values.
 - Moreover, it is also possible that each process in L returns at line 08 the value it has selected at line 05 (this depends on the value of the predicate evaluated at line 07). Consequently each value of V can potentially be returned.
8. Given an execution, the previous item has extracted a non-empty set V of at most k non- \perp values that can be returned. We now show that (1) from τ , only values of V can be written in $REG[1..n]$ with a timestamp field ($lrww$) greater than r , and (2) a non- \perp value returned by an invocation is necessarily a value of V .
- (a) The τ -time invocation issued by any $p_j \in \bar{L}$ has a round number $REG[j].lre$ that is smaller than $REG[j_1].lre = r$ (this follows from the definition of \bar{L}). As by definition, r is the smallest round number during which a process finds true the predicate of line 07, it follows that any $p_j \in \bar{L}$ needs to issue an invocation with a round number greater than r to have a chance to return a non- \perp value.
- (b) Let \mathcal{I}' be the set of all the invocations that have a round number greater than r . They are issued by the processes of \bar{L} or the processes of L whose τ -time invocation has returned \perp at line 07. Let us observe that any invocation of \mathcal{I}' starts after τ .
- Let I' be the first invocation of \mathcal{I}' that executes 05. I' (issued by some process p_j) selects (at line 03) a value $value_j$ from a register $REG[y]$ such that $REG[y].lrww \geq REG[j_1].lrww = r$. As up to now, only processes of L have written values in $REG[1..n]$ with a write timestamp ($lrww$) $\geq r$, it follows that I' selects a value from V ⁷. Consequently, this invocation does not add a new value to V .
- Let I'' be the invocation of \mathcal{I}' that is the second to execute line 05. The same reasoning (including now I') applies. Etc. It follows from this induction that a value written at line 05 by an invocation of \mathcal{I}' is a value of V , which proves that only values of V can be written in the array $REG[1..n]$ with a write timestamp greater than r .
- (c) Finally, an invocation that returns a value at line 08, returns the value it has written at line 05. Due to the definition of r , its round number r' is $\geq r$. It follows that the non- \perp value that is returned is a value of V . \square _{Theorem 4}

6 Conclusion

Considering asynchronous systems equipped with a failure detector object, this paper focused on the set agreement problem when only a subset of the processes participate, namely, the *wait-free set agreement* problem. Wait-free means here that a correct process has to decide a value, whatever the behavior of the other processes (that can be correct or not and participate or not).

A wait-free failure detector-based k -set agreement algorithm has been presented. Its design principles follows the Paxos approach, decoupling the way the safety and the termination properties are guaranteed. The algorithm safety is based on an object denoted KA that can be built from single-writer/multi-reader atomic registers. The liveness property is based on a leader failure detector class, denoted Ω_*^k , that takes into account the participating processes. The very existence of the algorithm shows that Ω_*^k is sufficient to wait-free solve the k -set agreement problem. Showing that Ω_*^k is also necessary, or defining a class of weaker failure detectors solving the k -set agreement problem, remains one of the greatest research challenges for the fault-tolerant asynchronous computing theory community.

References

- [1] Borowsky E. and Gafni E., Generalized FLP Impossibility Results for t -Resilient Asynchronous Computations. *Proc. 25th ACM Symposium on Theory of Computation (STOC'93)*, San Diego (CA), pp. 91-100, 1993.

⁷It is possible that, when I' reads the array $REG[1..n]$ at line 02, not all the values of V have yet been written in that array. The important points are here that (1) at least one value of V has already been written in the array (namely, $REG[i].val$ with the timestamp $REG[j_1].lrww = r$), and (2) any register $REG[x]$ that currently contains a value not in V , is such that $REG[x].lrww < r$.

-
- [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] Chaudhuri S., More Choices Allow More Faults: Set Consensus Problems in Totally Asynchronous Systems. *Information and Computation*, 105:132-158, 1993.
- [5] 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.
- [6] Guerraoui R., Kapalka M. and Kouznetsov P., The Weakest Failure Detectors to Boost Obstruction-Freedom. *Proc. 20th Symposium on Distributed Computing (DISC'06)*, Springer Verlag LNCS #4167, pp. 376-390, Stockholm (Sweden), 2006.
- [7] Guerraoui R. and Raynal M., The Information Structure of Indulgent Consensus. *IEEE Transactions on Computers*, 53(4):453-466, 2004.
- [8] Guerraoui R. and Raynal M., The Alpha of Indulgent Consensus. *The Computer Journal*. To appear, 2006.
- [9] Herlihy M.P., Wait-Free Synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124-149, 1991.
- [10] Herlihy M.P. and Penso L. D., Tight Bounds for k -Set Agreement with Limited Scope Accuracy Failure Detectors. *Distributed Computing*, 18(2): 157-166, 2005.
- [11] Herlihy M.P. and Shavit N., The Topological Structure of Asynchronous Computability. *Journal of the ACM*, 46(6):858-923,, 1999.
- [12] Lamport L., The Part-Time Parliament. *ACM Transactions on Computer Systems*, 16(2):133-169; 1998.
- [13] Loui M.C., Abu-Amara H., Memory requirements for agreement among unreliable asynchronous processes. *Advances in Computing research*, JAI Press, 4:163-183, 1987.
- [14] Mostéfaoui A., Rajsbaum S., Raynal M. and Travers C., Irreducibility and Additivity of Set Agreement-oriented Failure Detector Classes. *Proc. 25th ACM Symposium on Principles of Distributed Computing PODC'06*, ACM Press, Denver (Colorado), 2006.
- [15] Mostéfaoui A. and Raynal M., k -Set Agreement with Limited Accuracy Failure Detectors. *Proc. 19th ACM Symposium on Principles of Distributed Computing (PODC'00)*, ACM Press, pp. 143-152, 2000.
- [16] Mostéfaoui A. and Raynal M., Leader-Based Consensus. *Parallel Processing Letters*, 11(1):95-107, 2001.
- [17] Mostéfaoui A., Raynal M. and Travers C., Exploring Gafni's reduction land: from Ω^k to wait-free adaptive $(2p - \lceil \frac{p}{k} \rceil)$ -renaming via k -set agreement. *Proc. 20th Symposium on Distributed Computing (DISC'06)*, Springer Verlag LNCS #4167, pp. 1-15, Stockholm (Sweden), 2006.
- [18] Neiger G., Failure Detectors and the Wait-free Hierarchy. *Proc. 14th ACM Symp. on Principles of Distributed Computing (PODC'95)*, ACM Press, pp. 100-109, 1995.
- [19] Saks M. and Zaharoglou F., Wait-Free k -Set Agreement is Impossible: The Topology of Public Knowledge. *SIAM Journal on Computing*, 29(5):1449-1483, 2000.
- [20] Yang J., Neiger G. and Gafni E., Structured Derivations of Consensus Algorithms for Failure Detectors. *Proc. 17th ACM Symp. on Principles of Distributed Computing (PODC'98)*, ACM Press, pp.297-308, 1998.