Chasing the Weakest Failure Detector for k-Set Agreement in Message-Passing Systems

Achour Mostefaoui, Michel Raynal, Julien Stainer

Chasing the Weakest Failure Detector for k-Set Agreement in Message-passing Systems

Achour Mostefaoui
Université de Nantes, France
achour.mostefaoui@univ-nantes.fr

Michel Raynal, Julien Stainer
Université de Rennes 1, France
{raynal|jstainer}@irisa.fr

***Abstract*—This paper continues our quest for the weakest failure detector which allows the k-set agreement problem to be solved in asynchronous message-passing systems prone to process failures. It has two main contributions which will be instrumental to complete this quest.**

The first contribution is a new failure detector (denoted $\Pi\Sigma_{x,y}$) that has several noteworthy properties. (a) It is stronger than Σ_x which has been shown to be necessary. (b) It is equivalent to the pair $\langle\Sigma,\Omega\rangle$ when $x = y = 1$ (optimal to solve consensus). (c) It is equivalent to the pair $\langle\Sigma_{n-1},\Omega_{n-1}\rangle$ when $x = y = n-1$ (optimal for $(n-1)$-set agreement). (d) It is strictly weaker than the pair $\langle\Sigma_x,\overline{\Omega}_y\rangle$ (which has been investigated in previous works). (e) It is operational: the paper presents a $\Pi\Sigma_{x,y}$-based algorithm that solves k-set agreement for $k \geq xy$ (intuitively, x refers to the maximum number of isolated groups of processes and y to the number of leaders in each of these groups).

The second contribution of the paper is a proof that, for $1 < k < n-1$, the eventual leaders failure detector Ω_k (which eventually provides each process with the same set of k process identities, this set including at least one correct process) is not necessary to solve k-set agreement problem.

***Index Terms*—Asynchronous distributed system, Eventual leader, Failure detector, Fault tolerance, Quorum.**

I. Introduction

a) The k-set agreement problem: This problem is a natural generalization of the consensus problem. It is a coordination problem (also called decision problem) introduced by S. Chaudhuri [10] to explore the relation linking the number of process failures and the minimal number of values that processes are allowed to decide. This problem can be defined as follows [10]. Each process proposes a value and every non-faulty process has to decide a value (termination), in such a way that a decided value is a proposed value (validity) and no more than k different values are decided (agreement). The problem parameter k defines the coordination degree: $k = 1$ corresponds to its most constrained instance (consensus) while $k = n-1$ corresponds to its weakest non-trivial instance (called set agreement).

Let t be the model parameter that defines the upper bound on the number of processes that may crash in a run, $0 \leq t < n$. If $t < k$, k-set agreement can be trivially solved in both synchronous and asynchronous systems: k predetermined processes broadcast (write in the shared memory) the values they propose and a process decides the first proposed value it receives (reads from the shared memory). Hence, the interesting setting is when $t \geq k$, i.e., when the number of values that can be decided is smaller or equal to the maximal number of processes that may crash in a run.

Round-based algorithms that solve the k-set agreement problem for $k \leq t < n$ in crash-prone synchronous message-passing systems are presented in [2], [18]. These algorithms are optimal in the sense that the processes decide in at most $\lfloor \frac{t}{k} \rfloor + 1$ rounds which has been shown to be a lower bound on the number of rounds for a process to decide. For asynchronous systems where the processes communicate by reading/writing a shared memory or sending/receiving messages, the situation is different, namely, when $t \geq k$, the k-set agreement problem has no solution [6], [16], [25].

b) Failure detectors: Let us observe that in an asynchronous system where the only means for processes to communicate is a read/write shared memory or send/receive message-passing network, no process is able to know if another process has crashed or is only very slow. The concept of a failure detector originates from this simple observation. A failure detector is a device (distributed oracle) that enriches a distributed system by providing alive processes with information on failed processes [8]. Several classes of failure detectors can be defined according to the type of information on failures they provide to processes.

Given a system model $\mathcal{M}$ (e.g., asynchronous read/write shared memory system model or asynchronous send/receive message-passing system model) a failure detector A is stronger than a failure detector B with respect to $\mathcal{M}$ (denoted $A \succeq_{\mathcal{M}} B$ or $B \preceq_{\mathcal{M}} A$) if there is an algorithm (called *reduction*) that builds B in $\mathcal{M}$ enriched with A (we then also say that B is weaker than A). If A is stronger than B and B is stronger than A, then A and B are equivalent with respect to $\mathcal{M}$ (denoted $A \simeq_{\mathcal{M}} B$). If $A \succeq_{\mathcal{M}} B$ and $B \not\succeq_{\mathcal{M}} A$ then A is strictly stronger than B -equivalently B is strictly weaker than A- (denoted $A \succ_{\mathcal{M}} B$ or $B \prec_{\mathcal{M}} A$). If $A \not\succeq_{\mathcal{M}} B$ and $B \not\succeq_{\mathcal{M}} A$ (denoted $A \not\simeq_{\mathcal{M}} B$), A and B cannot be compared in $\mathcal{M}$.

Failure detectors, to circumvent the "$t \geq k$" impossibility result associated with the k-set agreement problem in asynchronous systems, have been investigated since 2000 [19]. (Random oracles to solve the k-set agreement problem have also been investigated [20].) The question of the weakest failure detector to solve the k-set agreement problem ($k > 1$) has been stated first in [24]. A failure detector A is the weakest failure detector that allows a problem P to be solved in a model $\mathcal{M}$ if any failure detector B that allows P to be solved

in $\mathcal{M}$ is such that $B \succeq_{\mathcal{M}} A$.

c) The weakest failure detector for k-set agreement in shared memory systems where $t = n - 1$: The *eventual leader* failure detector Ω introduced in [9] is the weakest failure detector that allows consensus (i.e., 1-set agreement) to be solved in shared memory systems where any number of processes may crash [17]. Ω ensures that there is an unknown but finite time after which all the processes have the same non-faulty leader (before that time, there is an anarchy period during which each process can have an arbitrarily changing faulty or non-faulty leader). At the other end of the spectrum ($k = n - 1$), the failure detector $\overline{\Omega}_{n-1}$ (anti-omega) has been introduced in [26] where it is shown to be the weakest failure detector that allows $(n-1)$-set agreement to be solved in these systems.

A simple generalization of Ω and $\overline{\Omega}_{n-1}$ denoted $\overline{\Omega}_k$, $1 \leq k \leq n-1$, ($\overline{\Omega}_1$ is Ω) has been introduced in [23] where it is conjectured to be the weakest failure detector class for solving k-set agreement in asynchronous read/write shared memory systems. This conjecture has been proved in [14]. A failure detector of the class $\overline{\Omega}_k$ provides each process with a (possibly always changing) set of k processes such that, after some unknown but finite time, all the sets that are output have in common the same non-faulty process. The optimality of $\overline{\Omega}_k$ to solve k-set agreement in shared memory systems seems to be related to the fact that this problem is equivalent to the k-simultaneous consensus problem [1] in which each process executes k independent consensus instances (to which it proposes the same input value) and is required to terminate in one of them. As indicated in [26], this problem has been instrumental in determining the weakest failure detector for wait-free solving the $(n-1)$-set agreement problem in asynchronous shared memory systems.

d) The cases $k = 1$ and $k = n - 1$ in message-passing systems where $t = n-1$: When $k = 1$, as already indicated k-set agreement boils down to consensus, and it is known that the failure detector denoted Ω is the weakest to solve consensus in asynchronous message-passing systems where $t < n/2$ [9]. This lower bound result is extended to any value of t in [12] where the failure detector Σ is introduced and is shown that $\Sigma \times \Omega$ is the weakest failure detector to solve consensus in message-passing systems when $t < n$. This means that Σ is the minimal additional power (as far as information on failures is concerned) required to overcome the barrier $t < n/2$ and attain $t \leq n-1$. Actually the power provided by Σ is the minimal one required to implement a shared register in a message-passing system [4], [12]. Σ provides each process with a quorum (set of process identities) such that the values of any two quorums (each taken at any time) intersect, and there is a finite time after which any quorum includes only correct processes. Fundamentally, Σ prevents partitioning. A failure detector $\Sigma \times \Omega$ outputs a pair of values, one for Σ and one for Ω.

The *Loneliness* failure detector (denoted $\mathcal{L}$) has been proposed in [13] where it is proved that it is the weakest failure detector for solving $(n-1)$-set agreement in the asynchronous message-passing model with $t = n-1$. Such a failure detector provides each process p with a boolean (that p can only read) such that the boolean of at least one process remains always false and, if all but one process crash, the boolean of the remaining process becomes and remains true forever. Let us notice that the weakest failure detector for $(n-1)$-set agreement is not the same in the read/write shared memory model (where it is $\overline{\Omega}_{n-1}$) and the send/receive message-passing model (where it is $\mathcal{L}$).

e) The quest for the weakest failure detector for k-set agreement in message-passing systems: This quest seems to be one of the most difficult research topics in the theory of fault-tolerant distributed computing. Since a few years, several new failure detectors have been proposed for solving k-set agreement in asynchronous message passing systems prone to any number of crashes, but so far finding the weakest still remains a challenge.

The interested reader will find in [21] a study on relations linking some of these failure detectors. Here we only present the failure detector Σ_x introduced in [5] because it is central to the paper. Σ_x generalizes the quorum failure detector class Σ introduced in [12] (Σ_1 is Σ). This failure detector provides each process with a set (quorum) such that at least two quorums do intersect in any set of $x+1$ quorums (whose values are taken at any times). Moreover, there is a finite (but unknown) time after which the quorum of any process includes only non-faulty processes. Two main results are proved in [5]: (a) as far as information on failures is concerned, Σ_k is a necessary requirement to solve k-set agreement; (b) Σ_{n-1} is sufficient to solve $(n-1)$-set agreement. Interestingly, a Σ_x-based algorithm is presented in [7] that solves k-set agreement for $k \geq n - \lfloor \frac{n}{x+1} \rfloor$.

f) Contributions of the paper: This paper is a new step in the quest for the weakest failure detector for k-set agreement in message-passing systems. It has two main contributions.

- The first contribution is the definition and the investigation of a new failure detector class denoted $\Pi\Sigma_{x,y}$.
  - Intuitively $\Pi\Sigma_{x,1}$ (1) prevents the system from partitioning into more than k independent subsets and (2) guarantees that the processes of at least one of these subsets agree on a common leader. $\Pi\Sigma_{x,y}$ can be seen as y independent instances of $\Pi\Sigma_{x,1}$ in which item (2) is guaranteed in only one of these instances.
  - Let $\mathcal{AMP}$ denote the asynchronous message-passing system model where up to $n-1$ process may crash. The properties of $\Pi\Sigma_{x,y}$ are the following: (a) $\Pi\Sigma_{1,y} \simeq_{\mathcal{AMP}} \langle \Sigma_1, \overline{\Omega}_y \rangle$; (b) $\Pi\Sigma_{x,n-1} \simeq_{\mathcal{AMP}} \Sigma_x$; (c) $\Pi\Sigma_{x,y} \preceq_{\mathcal{AMP}} \langle \Sigma_x, \overline{\Omega}_y \rangle$ for $1 \leq x, y \leq n$ and $\Pi\Sigma_{x,y} \prec_{\mathcal{AMP}} \langle \Sigma_x, \overline{\Omega}_y \rangle$ for $1 < y < x < n$.
    It follows from (a) and (b) that $\Pi\Sigma_{1,1}$ and $\Pi\Sigma_{n-1,n-1}$ are the weakest failure detectors to solve k-set agreement for $k = 1$ and $k = n-1$, respectively.
    For $1 \leq k \leq n-1$, we have the following. An algorithm based on the pair of failure detectors $\langle \Sigma_x, \overline{\Omega}_y \rangle$ is presented in [7] that solves k-set agree-

ment for $k \geq xy$ (let BT-2010 denote this algorithm). Moreover, it is shown in that paper that there is no $\langle \Sigma_x, \overline{\Omega}_y \rangle$-based k-set agreement algorithm when $(k < xy) \wedge (n \geq 2xy)$.
Actually, an appropriate modification of BT-2010 provides us with a $\Pi\Sigma_{x,y}$-based k-set algorithm that has the same properties (listed above) as BT-2010. The important point is here the following one: while BT-2010 and the proposed algorithm work for the same pairs (x, y), it follows from item (c) that the proposed algorithm is based on weaker information on failures than BT-2010.

- The second contribution of the paper (which has been obtained thanks to the previous failure detector $\Pi\Sigma_{x,y}$) is the following: Ω_k is not necessary to solve k-set agreement when $1 < k < n-1$ [1]. Combined with the fact that Σ_k is necessary [5], this result restricts the area we have to look for in order to discover the weakest failure detector for k-set agreement in message-passing systems for $1 < k < n - 1$.

g) Roadmap: The paper is made up of 8 sections. Section II presents the base computation model and the k-set agreement problem. Section III presents the eventual leaders and generalized quorums failure detectors. Section IV defines the new failure detector $\Pi\Sigma_{x,y}$ and shows that it is strictly weaker than $\langle \Sigma_x, \overline{\Omega}_y \rangle$ for $1 < y < x < n$. Section V presents a basic abstraction called Alpha_x which is used in Section VI (assuming $k \geq xy$) to build an algorithm that solves the k-set agreement problem in the asynchronous message-passing model enriched with $\Pi\Sigma_{x,y}$. Section VII shows that Ω_k is not necessary for solving k-set agreement (when $1 < k < n - 1$). Finally, Section VIII concludes the paper.

II. Base computation model and k-set agreement

A. Computation model

h) Process model: The system consists of a set of n sequential processes denoted p_1, ..., p_n. $\mathcal{P} = \{1, \ldots, n\}$ is the set of process identities. Each process executes a sequence of (internal or communication) atomic steps. A process executes its code until it possibly crashes (if it ever crashes). After it has crashed, a process executes no more steps. A process that crashes in a run is said *faulty* in that run, otherwise it is *correct*. Given a run, $\mathcal{C}$ and $\mathcal{F}$ denote the set processes that are correct and the set of processes that are faulty in that run, respectively. Up to $t = n - 1$ processes may crash in a run, hence, $1 \leq |\mathcal{C}| \leq n$.

i) Communication model: The processes communicate by executing atomic communication steps which are the sending or the reception of a message. Every pair of processes is connected by a bidirectional channel. The channels are failure-free (no creation, alteration, duplication or loss of messages) and asynchronous (albeit the time taken by a message to travel from its sender to its receiver is finite, there is no bound on transfer delays). The notation "broadcast MSG_TYPE(m)" is used as a (non-atomic) shortcut for "**for each** $j \in \mathcal{P}$ **do** send MSG_TYPE(m) to p_j **end for**" (let us observe that p_i sends then the message also to itself).

j) Underlying time model: The underlying *time model* is the set $\mathbb{N}$ of natural integers. As we are in an asynchronous system, this time notion is not accessible to the processes (hence, the model is sometimes called time-free model). It can only be used from an external observer point of view to state or prove properties. Time instants are denoted τ, τ', etc.

k) Notation: The previous asynchronous crash-prone message-passing system model is denoted $\mathcal{AMP}[\emptyset]$. $\mathcal{AMP}$ stands for "$\mathcal{A}$synchronous $\mathcal{M}$essage-$\mathcal{P}$assing"; $\emptyset$ means this is the "base" system (not enriched with a failure detector).

B. The k-Set agreement problem

As already indicated in the Introduction, the k-set agreement problem has been introduced by Soma Chaudhuri [10]. It generalizes the consensus problem (that corresponds to $k = 1$). It is defined as follows. Each process proposes a value and has to decide a value in such a way that the following properties are satisfied:

- Termination. Every correct process decides a value.
- Validity. A decided value is one of the proposed values.
- Agreement. At most k different values are decided.

III. Existing families of failure detectors

This section presents failure detectors that have been proposed in the quest of the weakest failure detector for k-set agreement. The system model $\mathcal{AMP}[\emptyset]$ enriched with a failure detector A is denoted $\mathcal{AMP}[A]$.

A failure detector provides each alive process with a read-only local variable. Let xxx_i be such a variable of process p_i. Let xxx_i^τ denotes the value of xxx_i at time τ.

A. The Ω_k and $\overline{\Omega}_k$ families

The *eventual leaders* failure detectors of the families Ω_k and $\overline{\Omega}_k$ provide each process p_i with a local variable denoted $leaders_i$. They originates from Ω [9] ($\Omega_1 \equiv \overline{\Omega}_1 \equiv \Omega$). $\overline{\Omega}_k$ is a straightforward generalization of $\overline{\Omega}_{n-1}$ (introduced in [26]). $\overline{\Omega}_k$ has been shown to be the weakest failure detector to solve k-set agreement in asynchronous shared memory systems with any number of process crashes in [14]. Let us consider the following properties on the sets $leaders_i$.

- Validity. $\forall i, \forall \tau$: $leaders_i^\tau$ is a set of k process identities.
- Strong eventual leadership. $\exists\ LD, \tau$: $(LD \cap \mathcal{C} \neq \emptyset) \wedge (\forall \tau' \geq \tau,\ \forall i \in \mathcal{C} :\ leaders_i^{\tau'} = LD)$.
- Weak eventual leadership. $\exists\ \ell \in \mathcal{C}, \tau$: $(\forall \tau' \geq \tau,\ \forall i \in \mathcal{C} :\ \ell \in leaders_i^{\tau'})$.

Validity combined with strong eventual leadership states that, after some unknown but finite time, all correct processes have the same set of k leaders and at least one of them is a correct process. Validity combined with weak eventual leadership requires only that the correct processes eventually share a common correct leader.

[1] In [11], a slightly different result is shown, namely $\langle \Sigma, \Omega_k \rangle$ is not necessary to solve k-set agreement in message-passing. However, this is not sufficient to prove our theorem since $\langle \Sigma, \Omega_k \rangle$ has a strictly stronger computability power than Ω_k in message-passing.

l) The Ω_k family: This family (introduced in [22]) includes the failure detectors that satisfy the validity and strong eventual leadership properties.

m) The $\overline{\Omega}_k$ family: This family (introduced in [23]) includes the failure detectors that satisfy the validity and weak eventual leadership properties.

B. The Σ_k and Π_k families

n) The Σ_k family: As noticed in the Introduction, the *generalized quorum* failure detector Σ_k (introduced in [5]) is a generalization of the quorum failure detector Σ introduced in [12] where it is shown to be the weakest failure detector to implement a register in $\mathcal{AMP}[\emptyset]$.

Σ_k provides each process p_i with a set qr_i (called quorum) that satisfies the following properties (after a process p_i has crashed, we have $qr_i = \mathcal{P}$ by definition). The self-inclusion property (which does not appear in [5]) is considered here because it allows for a simpler formulation of algorithms.

- Self-inclusion. $\forall\ i \in \mathcal{P}, \forall\ \tau$: $i \in qr_i^{\tau}$.
- Quorum liveness. $\exists\ \tau$: $\forall\ i \in \mathcal{C}, \forall\ \tau' \geq \tau$: $qr_i^{\tau'} \subseteq \mathcal{C}$.
- Quorum intersection. $\forall\ \ id_1, \ldots, id_{k+1} \in \mathcal{P}$, $\forall\ \tau_1, \ldots, \tau_{k+1}$: $\exists\ i, j : (i \neq j) \wedge (qr_{id_i}^{\tau_i} \cap qr_{id_j}^{\tau_j} \neq \emptyset)$.

It is shown in [5] that Σ_k is necessary when one wants to solve k-set agreement in $\mathcal{AMP}[\emptyset]$.

o) The Π_k family: The failure detector Π_k (introduced in [5]) is an extension of Σ_k to which it adds the following property.

- Eventual leadership. $\exists\ LD, \tau$: $(|LD| = k)\ \wedge\ (\forall \tau' \geq \tau,\ \forall i \in \mathcal{C} :\ qr_i^{\tau'} \cap LD \neq \emptyset)$.

It is shown in [5] that Π_k and the pair $\langle \Sigma_k, \Omega_k \rangle$ are equivalent, i.e., Π_k can be built in $\mathcal{AMP}[\Sigma_k, \Omega_k]$ and $\langle \Sigma_k, \Omega_k \rangle$ can be built in $\mathcal{AMP}[\Pi_k]$. It is also shown in [5] that Π_{n-1} and $\mathcal{L}$ are equivalent. If follows from these observations that Π_1 and Π_{n-1} are the weakest failure detectors for $k = 1$ and $k = n - 1$. Unfortunately, as shown in [3], [7], Π_k does not allow to solve k-set agreement for $1 < k < n - 1$.

IV. The family of failure detectors $\Pi\Sigma_{x,y}$

A. Definition

The definition of $\Pi\Sigma_{x,y}$ is incremental, first is defined $\Pi\Sigma_x$ and then $\Pi\Sigma_{x,y}$.

p) The failure detector $\Pi\Sigma_x$: A failure detector $\Pi\Sigma_x$ provides each process p_i with a set qr_i and a variable $leader_i$ which define the current quorum and the current leader of p_i. It is defined by the following properties.

- Self-inclusion. $\forall\ i \in \mathcal{P}, \forall\ \tau$: $i \in qr_i^{\tau}$.
- Quorum liveness. $\exists\ \tau$: $\forall\ i \in \mathcal{C}, \forall\ \tau' \geq \tau$: $qr_i^{\tau'} \subseteq \mathcal{C}$.
- Quorum intersection. $\forall\ \ id_1, \ldots, id_{x+1} \in \mathcal{P}$, $\forall\ \tau_1, \ldots, \tau_{x+1}$: $\exists\ i, j : (i \neq j) \wedge (qr_{id_i}^{\tau_i} \cap qr_{id_j}^{\tau_j} \neq \emptyset)$.
- Eventual partial leadership. $\exists \ell \in \mathcal{C} :\ \forall i \in\ \mathcal{C}$: $\big(\forall \tau : \exists \tau_i, \tau_\ell \geq \tau : qr_i^{\tau_i} \cap qr_\ell^{\tau_\ell} \neq \emptyset\big)\ \Rightarrow\ \big(\exists \tau : \forall \tau' \geq \tau : leader_i^{\tau'} = \ell\big)$.

The self-inclusion, liveness and intersection properties are the properties that define Σ_x: after some time the quorum of any correct process contains only correct processes (liveness) and any set of $x + 1$ quorums contains two intersecting quorums (intersection). Hence, $\Pi\Sigma_x \succeq \Sigma_x$.

Eventual partial leadership states that there is a correct process p_ℓ such that, for any correct process p_i whose quorum qr_i intersects infinitely often its quorum qr_ℓ (left part of the implication), then eventually p_ℓ is forever the leader of p_i (right part of the implication).

q) The failure detector $\Pi\Sigma_{x,y}$: $\Pi\Sigma_x$ is $\Pi\Sigma_{x,1}$. More generally, $\Pi\Sigma_{x,y}$ provides each process p_i with an array $FD_i[1..y]$ such that for each each $j \in \{1, \ldots, y\}$, $FD_i[j]$ is a pair containing a quorum $FD_i[j].qr$ and a process index $FD_i[j].leader$. Let $FD[j]$ denote the corresponding distributed object (ie, $FD[j]$ on p_i is represented by $FD_i[j]$). The failure detector $\Pi\Sigma_{x,y}$ consists of an array $FD[1..y]$ that satisfies the following properties:

- Vector safety. $\forall j \in [1..y]$: $FD[j].qr$ satisfies the liveness and intersection properties of $\Pi\Sigma_x$.
- Vector liveness. $\exists j \in [1..y]$: $FD[j].leader$ satisfies the eventual partial leadership property of $\Pi\Sigma_x$ related to $FD[j].qr$ for the same j.

B. $\Pi\Sigma_{x,y}$ *vs* $\langle \Sigma_x, \overline{\Omega}_y \rangle$

A failure detector $\langle \Sigma_x, \overline{\Omega}_y \rangle$ provides each process p_i with two *independent* read-only local variables: qr_i that satisfies the properties defined by Σ_x, and $leaders_i$ that satisfies the properties defined by $\overline{\Omega}_y$.

Lemma 1: Let $1 \leq x, y \leq n - 1$. $\Pi\Sigma_{x,y} \preceq_{\mathcal{AMP}} \langle \Sigma_x, \overline{\Omega}_y \rangle$.

Algorithm 1 From $\langle \Sigma_x, \overline{\Omega}_y \rangle$ to $\Pi\Sigma_{x,y}$ (code for p_i)

(01) **repeat forever** rel_broadcast LEADER($leaders_i$) **end repeat**.

(02) **when** LEADER(ld) **is delivered**:
(03) **for each** $j \notin ld$ **do** $susp\_nb_i[j] \leftarrow susp\_nb_i[j] + 1$ **end for**;
(04) **let** $j_1, j_2, \ldots, j_n$ be a permutation of $\{1, \ldots, n\}$ such that $(susp\_nb_i[j_1], j_1) < (susp\_nb_i[j_2], j_2) < \cdots < (susp\_nb_i[j_n], j_n)$;
(05) **for each** $x \in \{1, \ldots, y\}$ **do** $FD_i[x].leader \leftarrow j_x$ **end for**.

Theorem 1: Let $1 \leq y \leq n - 1$. $\Pi\Sigma_{1,y} \simeq_{\mathcal{AMP}} \langle \Sigma_1, \overline{\Omega}_y \rangle$.

Proof Taking $x = 1$ in Lemma 1 we have $\Pi\Sigma_{1,y} \preceq_{\mathcal{AMP}} \langle \Sigma_1, \overline{\Omega}_y \rangle$. Hence, we have only to show that $\langle \Sigma_1, \overline{\Omega}_y \rangle \preceq_{\mathcal{AMP}} \Pi\Sigma_{1,y}$.

Let qr_i of Σ_1 be the output of $FD_i[1].qr$. Hence, the quorums qr_i inherit the liveness and intersection properties of $FD_i[1].qr$ that trivially satisfy the properties defining Σ_1.

Let $leaders_i$ be any subset of size y that contains $\cup_{1 \leq j \leq y} \{FD_i[j].leader\}$. Due to the vector liveness property of $\Pi\Sigma_{1,y}$, there is an entry j such that $FD[j]$ satisfies the eventual partial leadership (Observation $O1$). Moreover, as $x = 1$, any pair of quorums output by $FD[j]$ do intersect (Observation $O2$). It follows from $O1$ and $O2$ that there is a correct process p_ℓ such that, for each correct process p_i, there is a time after which the predicate $FD_i[j].leader = \ell$ remains forever true. Consequently, there is a finite time after which the predicate $\ell \in leaders_i$ remains forever true at any correct process p_i, from which follows the weak eventual leadership property of $\overline{\Omega}_y$. $\square_{Theorem\ 1}$

Theorem 2: Let $1 \leq x \leq n-1$. $\Pi\Sigma_{x,n-1} \simeq_{\mathcal{AMP}} \Sigma_x$.

Proof Taking $y = n-1$ in Lemma 1 we have $\Pi\Sigma_{x,n-1} \preceq_{\mathcal{AMP}} \langle \Sigma_x, \overline{\Omega}_{n-1} \rangle$. Hence, we have only to show that $\langle \Sigma_x, \overline{\Omega}_{n-1} \rangle \preceq_{\mathcal{AMP}} \Pi\Sigma_{x,n-1}$.

It is shown in [5] (Corollary 2 in [5]) that $\Sigma_x \succeq_{\mathcal{AMP}} \Sigma_{x+1} \succeq_{\mathcal{AMP}} \cdots \succeq_{\mathcal{AMP}} \Sigma_{n-1} \succeq_{\mathcal{AMP}} \Omega_{n-1}$. Moreover, it follows directly from their definitions that $\Omega_{n-1} \succeq_{\mathcal{AMP}} \overline{\Omega}_{n-1}$. It follows that $\Sigma_x \succeq_{\mathcal{AMP}} \langle \Sigma_x, \overline{\Omega}_{n-1} \rangle$ which completes the proof of the theorem. $\square_{Theorem\ 2}$

Lemma 2: Let $1 \leq y < n$. $\Pi\Sigma_{y+1,1} \not\succeq_{\mathcal{AMP}} \overline{\Omega}_y$.

Theorem 3: Let $1 < y < x < n$. $\Pi\Sigma_{x,y} \prec_{\mathcal{AMP}} \langle \Sigma_x, \overline{\Omega}_y \rangle$.

Proof $\Pi\Sigma_{x,y} \preceq_{\mathcal{AMP}} \langle \Sigma_x, \overline{\Omega}_y \rangle$ follows from Lemma 1. Hence, we have to show that $\Pi\Sigma_{x,y} \not\succeq_{\mathcal{AMP}} \langle \Sigma_x, \overline{\Omega}_y \rangle$. Let us first observe that $\Pi\Sigma_{y+1,1} \succeq_{\mathcal{AMP}} \Pi\Sigma_{y+1,y}$. This is easily obtained by providing each $FD[j]$ of the array $FD[1..y]$ of $\Pi\Sigma_{y+1,y}$ with the outputs supplied by $\Pi\Sigma_{y+1,1}$. On an other side, (as shown in [5]) Σ_z is strictly stronger than Σ_{z+1} for $1 \leq z < n-1$ and, consequently, $\Pi\Sigma_{y+1,y} \succeq_{\mathcal{AMP}} \Pi\Sigma_{x,y}$ for $y < x$. It follows from Lemma 2 (i.e., $\Pi\Sigma_{y+1,1} \not\succeq_{\mathcal{AMP}} \overline{\Omega}_y$) that, as $\Pi\Sigma_{y+1,1}$ is stronger than $\Pi\Sigma_{x,y}$, we have $\Pi\Sigma_{x,y} \not\succeq_{\mathcal{AMP}} \overline{\Omega}_y$ which proves the theorem. $\square_{Theorem\ 3}$

r) Remark: A main difference between $\Pi\Sigma_{x,y}$ and $\langle \Sigma_x, \overline{\Omega}_y \rangle$ lies in the fact that the eventual correct leader elected by $\overline{\Omega}_y$ has to be the same for all correct processes, while $\Pi\Sigma_{x,y}$ requires only that the correct processes of a subset (dynamically defined by one of the y Σ_x failure detectors) agree on a common leader. Hence, the scope of the leadership provided by $\Pi\Sigma_{x,y}$ is not required to be the whole system but only a subset of it.

V. A Basic Building Block: the Alpha_x abstraction

A. Definition of Alpha_x

This paper presents an algorithm that solves the k-set agreement problem in $\mathcal{AMP}[\Pi\Sigma_{x,y}]$. This algorithm uses an abstraction called Alpha_x and has been introduced in [15] to capture the safety property of consensus and generalized in [24] to capture the safety property of k-set agreement in crash-prone systems. [15], [24] give corresponding implementations in read/write shared memory systems and send/receive message-passing systems.

Let $\perp$ be a default value that cannot be proposed by processes. Alpha_x is an object initialized to $\perp$ that may store up to x different values proposed by processes. It is an abstraction (object) that provides processes with a single operation denoted $\mathsf{propose}(r, v)$ (where r is a round number and v a proposed value) that returns a value to the invoking process. The round number plays the role of a logical time that allows identifying the $\mathsf{propose}()$ invocations. It is assumed that distinct processes use different round numbers and successive invocations by the same process use increasing sequence numbers. Alpha_x is a kind of *abortable* object in the sense that $\mathsf{propose}()$ invocations are allowed to return the default value $\perp$ (i.e., abort) in specific concurrency-related circumstances (as defined from the obligation property, see below). More precisely, the Alpha_x objects used in this paper are defined by the following specification in which the obligation property takes explicitly into account the fact that we are interested into an Alpha_x object that will be implemented on top of $\mathcal{AMP}[\Sigma_x]$ (which is a strictly stronger underlying model than $\mathcal{AMP}[\emptyset]$).

- Termination. Any invocation of $\mathsf{propose}()$ by a correct process terminates.
- Validity. If $\mathsf{propose}(r, v)$ returns $v' \neq \perp$, then $\mathsf{propose}(r', v')$ has been invoked with $r' \leq r$.
- Quasi-agreement. At most k different non-$\perp$ values can be returned by $\mathsf{propose}()$ invocations.
- Obligation. p_ℓ being a correct process let $Q(\ell, \tau) = \{i \in \mathcal{C} \mid \forall\ \tau_i, \tau_\ell \geq \tau :\ qr_i^{\tau_i} \cap qr_\ell^{\tau_\ell} = \emptyset\}$. If, after time τ, (a) only p_ℓ and processes of $Q(\ell, \tau)$ invoke $\mathsf{propose}()$ and (b) p_ℓ invokes $\mathsf{propose}()$ infinitely often, then at least one invocation issued by p_ℓ returns a non-$\perp$ value.

Differently from the obligation property stated in [7], [15], [24] the previous obligation property is Σ_x-aware which allows for a weaker property (the Alpha_x object used in [7] is implemented on top of $\mathcal{AMP}[\Sigma_x]$ but its specification is not Σ_x-aware). More precisely, our obligation property allows concurrent invocations of $\mathsf{propose}()$ to return non-$\perp$ values as soon as the quorums of the invoking processes do not intersect during these invocations.

B. The Alpha_x object used by Bouzid and Travers [7]

This section presents an implementation of an Alpha_x object on top of $\mathcal{AMP}[\Sigma_x]$. This implementation is obtained from a modification of the algorithm proposed in [7] which is first described.

s) The obligation property used in [7]: The Alpha_x object used in [7] has the same specification as the one defined in this paper (which is close to the one defined in [15], [24]) but for the obligation property which (similarly to [15]) is defined as follows.

- Obligation. Let $I = \mathsf{propose}(r, -)$ be a terminating invocation. If every invocation $I' = \mathsf{propose}(r', -)$ that starts before I returns is such that $r' < r$, then I returns a non-$\perp$ value.

It is easy to see that this specification is not Σ_x-aware. In presence of concurrent invocations, it directs at most one process to decide a non-$\perp$ value, namely, the one with the highest round number. The current outputs of Σ_x are irrelevant in this statement.

t) Principles: establish a priority on values: Each process p_i manages a local variable est_i (initialized to $\perp$) that represents its current estimate v of the value it will decide and a pair (lre_i, pos_i) that defines the priority associated with v from p_i's point of view (the aim is to decide values with the highest priority); $lre_i = r$ means that r is the highest round seen by p_i and $pos_i = \rho \in [1..2^r]$ is the position of v in round r. The pairs $\langle r, \rho \rangle$ are used to establish a priority on proposed values. The function $g(\rho, \delta) = 2^\delta(\rho - 1) + 1$

(where δ is a difference between two round numbers) is used to compute the priority of a value in the following rounds. More precisely, let (r, ρ) and (r', ρ') (such that $r \leq r'$) be the pairs associated with the values v and v', respectively. Value v has lower priority than value v' at round r' iff $g(\rho, r'-r) < \rho'$ or $(g(\rho, r'-r) = \rho') \wedge (v < v')$.

Our description of Bouzid-Travers's algorithm (algorithm 2) is schematic. The reader will refer to [7] for more detailed presentation and a proof. The implementation of the operation propose() is made up of two sequential phases: a read phase followed by write phase.

Algorithm 2 Alpha_k in $\mathcal{AMP}[\Sigma_k]$: Bouzid-Travers's implementation [7]

init $lre_i \leftarrow 0$; $est_i \leftarrow \perp$; $pos_i \leftarrow 0$.

operation propose(r, v_i):
(01) broadcast REQ_R(r);
(02) **repeat** $Q_i \leftarrow qr_i$
(03) **until** $(\forall j \in Q_i$: RSP_R$(r, \langle lre_j, pos_j, est_j\rangle)$ received from $p_j)$
(04) **end repeat**;
(05) **let** $rcv_i = \{\langle lre_j, pos_j, est_j\rangle$: RSP_R$(r, \langle lre_j, pos_j, est_j\rangle)$ received$\}$;
(06) **if** $(\exists lre : \langle lre, -, -\rangle \in rcv_i : lre > lre_i)$ **then** return$(\perp)$ **end if**;
(07) $pos_i \leftarrow \max\{pos \mid \langle r, pos, v\rangle \in rcv_i\}$;
(08) $est_i \leftarrow \max\{v \mid \langle r, pos_i, v\rangle \in rcv_i\}$;
(09) **if** $(est_i = \perp)$ **then** $est_i \leftarrow v_i$ **end if**;
(10) **while**$(pos_i < 2^r)$ **do**
(11) $pos_i \leftarrow pos_i + 1$; $pst_i \leftarrow pos_i$; // this line is executed atomically
(12) broadcast REQ_W(r, pst_i, est_i);
(13) **repeat** $Q_i \leftarrow qr_i$
(14) **until** $(\forall j \in Q_i$:
(15) RSP_W$(r, pst_i, \langle lre_j, pos_j, est_j\rangle)$ received from $p_j)$
(16) **end repeat**;
(17) **let** $rcv_i = \{\langle lre_j, pos_j, est_j\rangle$:
(18) RSP_W$(r, pst_i, \langle lre_j, pos_j, est_j\rangle)$ received$\}$;
(19) **if** $(\exists lre : \langle lre, -, -\rangle \in rcv_i : lre > r)$ **then** return$(\perp)$ **end if**;
(20) $pos_i \leftarrow \max\{pos \mid \langle r, pos, v\rangle \in rcv_i\}$;
(21) $est_i \leftarrow \max\{v \mid \langle r, pos_i, v\rangle \in rcv_i\}$
(22) **end while**;
(23) return(est_i).

when REQ_R(rd) **received from** p_j:
(24) **if** $rd > lre_i$ **then** $pos_i \leftarrow$ g$(pos_i, rd - lre_i)$; $lre_i \leftarrow rd$ **end if**;
(25) send RSP_R$(rd, \langle lre_i, pos_i, est_i\rangle)$ to p_j.

when REQ_W(rd, pos, est) **received from** p_j:
(26) **if** $rd \geq lre_i$ **then** $pos_i \leftarrow$ g$(pos_i, rd - lre_i)$; $lre_i \leftarrow rd$
(27) **case** $pos_j > pos_i$ **then** $est_i \leftarrow est$; $pos_i \leftarrow pos$
(28) $pos_j = pos_i$ **then** $est_i \leftarrow \max\{v_i, est\}$
(29) $pos_j < pos_i$ **then** nop
(30) **end case**
(31) **end if**;
(32) send RSP_W$(rd, pos, \langle lre_i, pos_i, est_i\rangle)$ to p_j.

u) Succinct description of the algorithm: the read phase: When it invokes propose(r, v), a process p_i first broadcasts a read-request message (line 01) to (a) obtain information on values proposed in previous rounds (if any) and (b) learn if other processes have started higher rounds.

When a process p_j receives such a message REQ_R(rd) (where rd is a round number) it redefines its pair (lre_j, pos_j) if $rd > lre_j$ (line 24) (the new position pos_j of est_j is recomputed according to the values of rd and lre_j). In all cases, p_j sends back an answer to p_i carrying its current value est_j and the associated pair $\langle lre_j, pos_j\rangle$ (line 25).

Then, when it has received a response from each process in its current quorum qr_i as supplied by Σ_x (lines 02-04), p_i returns $\perp$ if it has received an answer indicating that another process has started a round higher than lre_i (lines 05-06). Otherwise, $lre_i = r$ is the greatest round number known by p_i. In that case, p_i update pos_i to the greatest position associated with round r it has seen and adopts the corresponding value v as its current estimate est_i (lines 07 and 08). Moreover, if $v = \perp$, p_i adopts v_i into est_i, namely, the value it proposes to the Alpha_x object (line 09).

v) Succinct description of the algorithm: the write phase: Process p_i enters then a loop that it will exit either by returning $\perp$ (line 19) or its current estimate value (line 23). The maximum number of times that this loop can be executed depends on the round number r and the current position value pos_i (line 10). The part of the loop body defined by lines 12-21 is the same as lines 01-08. The difference is that, instead of obtaining information on the current state, p_i cooperate with the processes of its current quorum qr_i in order to try to increase the priority of its current est_i. Hence instead of a read-request, p_i broadcasts write-request messages.

Each time a process p_j receives such a message that carries a triplet $\langle rd, pos, val\rangle$ it updates its current state in order this local state contains the value with the highest priority (and the associated control data). Operationally, if $rd \geq lre_j$, p_j first updates pos_j and lre_j (line 26) exactly as it did at line 24 when it received a read-request message. Then, according to the value of pos_j and pos (lines 27-30), p_j updates est_j and pos_j if $pos > pos_j$ or updates only est_j if $pos_j = pos$. In all cases, p_j sends back a response carrying its local state to the process that sent the write-request.

As already indicated, a proof that this algorithm implements an Alpha_x object that satisfies the round-based obligation property stated at the beginning of this section is given in [7].

C. An implementation of Alpha_x as defined in this paper

Algorithm 3 describes an implementation of the Σ_x-aware specification of Alpha_x defined in Section V-A. This algorithm is an appropriate improvement of Algorithm 2.

To make the presentation easier, the line numbers are the same in both algorithms. The lines that are new or modified are prefixed by the letter N. These modifications concern message filtering. This filtering is used to prevent a process p_i from sending read or write-requests to the processes p_j such that p_i does not need information from p_j to complete its current invocation of propose(). Hence, such a process p_j cannot direct p_i to return $\perp$ while it could return a non-$\perp$ value (and additionally the values not sent by p_i cannot force other processes to return $\perp$).

w) Modification of the read phase: A process p_i records in $r\_req\_set_i$ the set of processes to which it has already sent a REQ_R(r) message (lines N01 and N02-2) and sends read-request messages REQ_R(r) only to the processes p_j that belong to its quorum qr_i (line N02-1).

Then, when it stops waiting for response messages, it considers only the responses sent by the processes of its last quorum plus its own response (lines N05-1 and N05-2).

x) Modification of the write phase: The message exchange pattern of that phase is modified similarly to what has been done for the read phase (lines N12, N14-1, N14-2, N17-1 and N17-2).

Algorithm 3 Alpha_k in $\mathcal{AMP}[\Sigma_k]$ as defined in Section V

init $lre_i \leftarrow 0$; $est_i \leftarrow \perp$; $pos_i \leftarrow 0$.

operation $\mathsf{propose}(r, v_i)$:
(N01) $r\_req\_set_i \leftarrow \emptyset$;
(02) **repeat** $Q_i \leftarrow qr_i$
(N02-1) send REQ_R(r) to each $p_j, j \in Q_i \setminus r\_req\_set_i$;
(N02-2) $r\_req\_set_i \leftarrow r\_req\_set_i \cup Q_i$
(03) **until** $(\forall j \in Q_i :$ RSP_R$(r, \langle lre_j, pos_j, est_j\rangle)$ received from $p_j)$
(04) **end repeat**;
(N05-1) **let** $rcv_i = \{ \langle lre_j, pos_j, est_j\rangle :$
(N05-2) RSP_R$(r, \langle lre_j, pos_j, est_j\rangle)$ received from Q_i $\}$;
(06) **if** $(\exists lre : \langle lre, -, -\rangle \in rcv_i : lre > lre_i)$ **then** return$(\perp)$ **end if**;
(07) $pos_i \leftarrow \max\{pos \mid \langle r, pos, v\rangle \in rcv_i\}$;
(08) $est_i \leftarrow \max\{v \mid \langle r, pos_i, v\rangle \in rcv_i\}$;
(09) **if** $(est_i = \perp)$ **then** $est_i \leftarrow v_i$ **end if**;
(10) **while**$(pos_i < 2^r)$ **do**
(11) $pos_i \leftarrow pos_i + 1$; $pst_i \leftarrow pos_i$; // this line is executed atomically
(N12) $w\_req\_set_i \leftarrow \emptyset$;
(13) **repeat** $Q_i \leftarrow qr_i$
(N14-1) send REQ_W(r, pst_i, est_i) to each $p_j, j \in Q_i \setminus w\_req\_set_i$;
(N14-2) $w\_req\_set_i \leftarrow w\_req\_set_i \cup Q_i$
(14) **until** $(\forall j \in Q_i :$
(15) RSP_W(r, pst_i, $\langle lre_j, pos_j, est_j\rangle$) received from $p_j)$
(16) **end repeat**;
(N17-1) $rcv_i \leftarrow \{\langle lre_j, pos_j, est_j\rangle :$
(N17-2) RSP_W(r, pst_i, $\langle lre_j, pos_j, est_j\rangle$) received from Q_i $\}$;
(19) **if** $(\exists lre : \langle lre, -, -\rangle \in rcv_i : lre > r)$ **then** return$(\perp)$ **end if**;
(20) $pos_i \leftarrow \max\{pos \mid \langle r, pos, v\rangle \in rcv_i\}$;
(21) $est_i \leftarrow \max\{v \mid \langle r, pos_i, v\rangle \in rcv_i\}$
(22) **end while**;
(23) return(est_i).

when REQ_R(rd) **received from** p_j:
(24) **if** $rd > lre_i$ **then** $pos_i \leftarrow \mathsf{g}(pos_i, rd - lre_i)$; $lre_i \leftarrow rd$ **end if**;
(25) send RSP_R(rd, $\langle lre_i, pos_i, est_i\rangle$) to p_j.

when REQ_W(rd, pos, est) **received from** p_j:
(26) **if** $rd \geq lre_i$ **then** $pos_i \leftarrow \mathsf{g}(pos_i, rd - lre_i)$; $lre_i \leftarrow rd$
(27) **case** $pos_j > pos_i$ **then** $est_i \leftarrow est$; $pos_i \leftarrow pos$
(28) $pos_j = pos_i$ **then** $est_i \leftarrow \max\{v_i, est\}$
(29) $pos_j < pos_i$ **then** nop
(30) **end case**
(31) **end if**;
(32) send RSP_W(rd, pos, $\langle lre_i, pos_i, est_i\rangle$) to p_j.

Theorem 4: Algorithm 3 implements an Alpha_x object as defined in Section *V-A*.

VI. Solving k-Set agreement in $\mathcal{AMP}[\Pi\Sigma_{x,y}]$

This section presents a simple algorithm that implements k-set agreement in $\mathcal{AMP}[\Pi\Sigma_{x,y}]$ for $k \geq xy$. It is similar to the one presented in [7]: it uses a base algorithm (similar to the one introduced in [15]) that solves x-set agreement in $\mathcal{AMP}[\Pi\Sigma_{x,1}]$ and then assuming $k \geq xy$ (as in [1], [7]) it uses y instances of this base to solve k-set agreement in $\mathcal{AMP}[\Pi\Sigma_{x,y}]$.

y) x-Set agreement in $\mathcal{AMP}[\Pi\Sigma_{x,1}]$: Algorithm 4 solves x-set agreement in $\mathcal{AMP}[\Pi\Sigma_{x,1}]$. A process p_i invokes $\mathsf{ks\_propose}_{x,1}$ (v_i) where v_i is the value it proposes. It decides a value d when it executes the statement return (d) which terminates its invocation. The local variable r_i is the local round number (as it is easy to see, each process uses increasing round numbers and no two distinct processes use the same round numbers).

A process loops until it decides. If during a loop iteration p_i is such that $leader_i = i$ ($leader_i$ is one of the two local outputs provided by $\Pi\Sigma_{x,1}$), p_i invokes the Alpha_x object to try to deposit its value v_i into it (the success depends on the concurrency and quorums pattern). If a non-$\perp$ value is returned by this invocation, p_i broadcasts it (with the reliable broadcast operation). A process decides as soon as it is delivered a DECISION() message.

Algorithm 4 x-set agreement in $\mathcal{AMP}[\Pi\Sigma_{x,1}]$ (p_i's code)

operation $\mathsf{ks\_propose}_{x,1}$ (v_i):
(01) $dec_i \leftarrow \perp$; $r_i \leftarrow i$;
(02) **while** $(dec_i = \perp)$ **do**
(03) **if** $(leader_i = i)$ **then** $\mathsf{Alpha}_x.\mathsf{propose}(r_i, v_i)$;
$r_i \leftarrow r_i + n$ **end if**
(04) **end while**;
(05) rel_broadcast DECISION(dec_i).

when DECISION(d) **is delivered**: return (d).

Theorem 5: Algorithm 4 solves the x-set agreement in $\mathcal{AMP}[\Pi\Sigma_{x,1}]$.

Proof Validity and agreement properties. Let us first observe that, due to the test of line 02, the default value $\perp$ cannot be decided. The fact that a decided value is a proposed value follows then from the validity of the underlying Alpha_x object. Similarly, the fact that at most k non-$\perp$ values are decided follows directly from the quasi-agreement property of the underlying Alpha_x object.

Termination property. It follows from the reliable broadcast operation that, at soon as a process decides (invokes return()) each correct process eventually delivers the same DECISION(d) message and decides (if not yet done). The proof is by contradiction: assuming that no process decides, we show that at least one correct process executes rel_broadcast() (and consequently, all correct processes decide).

Let p_ℓ be a correct process that appears in the definition of the eventual partial leadership property of $\Pi\Sigma_x$. It follows from the definition of p_ℓ that we eventually have forever $leader_\ell = \ell$.

Let R_ℓ be the set of the identities of the processes p_j (with $j \neq \ell$) such that we have $leader_j = j$ infinitely often. It follows from the contrapositive of the eventual partial leadership property of $\Pi\Sigma_x$ that there is a time τ_{R_ℓ} such that $\forall\ j \in R_\ell$, $\forall\ \tau 1, \tau 2 \geq \tau_{R_\ell}$: $qr_j^{\tau 1} \cap qr_\ell^{\tau 2} = \emptyset$, from which we conclude that $R_\ell \subseteq Q(\ell, \tau_{R_\ell})$ (this is the set defined in the obligation property of Alpha_x).

Let us notice that, due to test of line 03, there is a

finite time τ_a after which the only processes that invoke $\mathsf{Alpha}_x.\mathsf{propose}()$ are the processes in $R_\ell \cup \{\ell\}$. Moreover (as by the contradiction assumption no process decides) it follows that, after τ_a, p_ℓ invokes $\mathsf{Alpha}_x.\mathsf{propose}()$ infinitely often. Let τ_b be a time greater than $\max(\tau_{R_\ell}, \tau_a)$ from which we have $R_\ell \subseteq Q(\ell, \tau_{R_\ell}) \subseteq Q(\ell, \tau_b)$.

As after τ_b (a) only processes in $R_\ell \cup \{\ell\}$ invoke $\mathsf{Alpha}_x.\mathsf{propose}()$, (b) p_ℓ invokes $\mathsf{Alpha}_x.\mathsf{propose}()$ infinitely often and (c) $R_\ell \subseteq Q(\ell, \tau_b)$, we conclude from the obligation property of Alpha_x that at least one invocation of p_ℓ returns a value $d \neq \perp$ and consequently executes rel_broadcast DECISION(d). This contradicts the fact that no process decides and concludes the proof of the theorem. $\square_{Theorem\ 5}$

z) k-Set agreement in $\mathcal{AMP}[\Pi\Sigma_{x,y}]$: As in [1], [7], a simple k-set algorithm can be obtained by launching concurrently y instances of Algorithm 4, the jth one relying on the component $FD[j]$ of the failure detector $\mathcal{AMP}[\Pi\Sigma_{x,y}]$. A process decides the value returned by the first of the y instances that locally terminates. As there are y instances of Algorithm 4 and at most x values can be decided in each of them, it follows that at most xy different values can be decided. Moreover, as at least one $FD[j]$ is a $\Pi\Sigma_{x,y}$ failure detector, it follows that the correct processes decide (if not done before) in at least one of the y instances of Algorithm 4. Let us observe that, in such a "worst" case where the processes decide in the same instance, at most x values are decided).

VII. Ω_k IS NOT NECESSARY FOR k-SET AGREEMENT WHEN $1 < k < n-1$

Lemma 3: Let $1 < k < n-1$. Ω_k cannot be built in $\mathcal{AMP}[\Pi\Sigma_{k,1}]$.

Theorem 6: Let $1 < k < n-1$. Ω_k is not necessary for solving k-set agreement in $\mathcal{AMP}[\emptyset]$.

Let $X(k)$ denote the (still unknown) weakest failure detector such that k-set agreement can be solved in $\mathcal{AMP}[X(k)]$. The following corollary shows that, when $1 < k < n-1$, Ω_k is neither necessary nor sufficient for solving the k-set agreement problem.

Corollary 1: Let $1 < k < n-1$. $\Omega_k \not\simeq_{\mathcal{AMP}} X(k)$.

VIII. CONCLUSION

As indicated in the abstract, this paper is a new step in the quest for discovering the weakest failure detector for k-set agreement in asynchronous message-passing systems prone to any number of process failures. It has presented a new failure detector denoted $\Pi\Sigma_{x,y}$ which enjoys many noteworthy features and an associated algorithm that solves k-set agreement for $k \geq xy$. It has also shown that the weakest failure detector (that still remains to be discovered) and Ω_k (a well-studied failure detector) cannot be compared.

More generally, an important issue that remains to be solved lies in capturing the "weakest" type of shared memory that has to be emulated for solving k-set agreement in asynchronous message-passing systems.

REFERENCES

[1] Afek Y., Gafni E., Rajsbaum S., Raynal M. and Travers C., The k-Simultaneous Consensus Problem. *Dist. Computing*, 22:185-195, 2010.

[2] Attiya H. and Welch J., *Distributed Computing: Fundamentals, Simulations and Advanced Topics,* Wiley-Interscience, 414 pages, 2004.

[3] Biely M., Robinson P. and Schmid U., Easy Impossibility Proofs for k-Set Agreement in Message-passing Systems. *Brief Announcement, Proc. 30th ACM Symp. on Principles of Dist. Computing (PODC'11)*, 2010.

[4] Bonnet F. and Raynal M., A Simple Proof of the Necessity of the Failure Detector Σ to Implement an Atomic Register in Asynchronous Message-passing Systems. *Information Processing Letters*, 110(4):153-157, 2010.

[5] Bonnet F. and Raynal M., On the Road to the Weakest Failure Detector for k-Set Agreement in Message-passing Systems. *Theoretical Computer Science*, 412(33):4273-4284, 2011.

[6] 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.

[7] Bouzid Z. and Travers C., (Anti-$\Omega_k \times \Sigma_k$)-Based k-Set Agreement Algorithms. *Proc. 12th Int'l Conf. on Principles of Distributed Systems (OPODIS'10)*, Springer Verlag LNCS #6490, pp. 190-205, 2010.

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

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

[10] Chaudhuri S., More *Choices* Allow More *Faults:* Set Consensus Problems in Totally Asynchronous Systems. *Information and Computation,* 105:132-158, 1993.

[11] Chen W., Zhang J., Chen Y. and Liu X., Weakening failure detectors for k-set agreement via the partition approach. *21th Int'l Symp. on Dist. Comp. (DISC'07)*, Springer-Verlag LNCS #4731, pp. 123-138, 2007.

[12] Delporte-Gallet C., Fauconnier H. and Guerraoui R., Tight Failure Detection Bounds on Atomic Object Implementations. *Journal of the ACM*, 57(4):Article 22, 2010.

[13] Delporte-Gallet C., Fauconnier H., Guerraoui R. and Tielmann A., The Weakest Failure Detector for Message Passing Set-Agreement. *Proc. 22th Int'l Symposium on Distributed Computing (DISC'08)*, Springer-Verlag LNCS #5218, pp. 109-120, 2008.

[14] Gafni E. and Kuznetzov P., The Weakest Failure Detector for Solving k-Set Agreement. *Proc. 28th ACM Symposium on Principles of Distributed Computing (PODC'09)*, ACM Press, pp. 83-91, 2009.

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

[16] Herlihy M.P. and Shavit N., The Topological Structure of Asynchronous Computability. *Journal of the ACM*, 46(6):858-923, 1999.

[17] Lo W.-K. and Hadzilacos V., Using Failure Detectors to Solve Consensus in Asynchronous Shared-memory Systems. *Proc. 8th Int'l Workshop on Distributed Algorithms (WDAG'94, now DISC)*, LNCS #857, 1994.

[18] Lynch N.A., Distributed Algorithms. *Morgan Kaufmann Pub.*, San Francisco (CA), 872 pages, 1996.

[19] 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.

[20] Mostéfaoui A. and Raynal M., Randomized Set Agreement. *Proc. 13th ACM Symposium on Parallel Algorithms and Architectures (SPAA'01)*, ACM Press, pp. 291-297, 2001.

[21] Mostéfaoui A., Raynal M. and Stainer J., Relations Linking Failure Detectors Associated with k-Set Agreement in Message-passing Systems. *Proc. 13th Int'l Symp. on Stabilization, Safety and Security of Dist. Systems (SSS'11)*, Spinger Verlag LNCS #6976, pp. 341-355, 2011.

[22] Neiger G., Failure Detectors and the Wait-free Hierarchy. *Proc. 14th ACM Symposium on Principles of Distributed Computing (PODC'95)*, ACM Press, pp. 100-109, Las Vegas -NV), 1995.

[23] Raynal M., K-anti-Omega. *Rump Session at 26th ACM Symposium on Principles of Distributed Computing (PODC'07)*, 2007.

[24] Raynal M. and Travers C., In Search of the Holy Grail: Looking for the Weakest Failure Detector for Wait-free Set Agreement. *Proc. 10th Int'l Conf. on Principles of Dist. Systems (OPODIS'06)*, LNCS #4305, 2006.

[25] 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.

[26] Zielinski P., Anti-Omega: the Weakest Failure Detector for Set Agreement. *Proc. 27th ACM Symposium on Principles of Distributed Computing (PODC'08)*, ACM Press, pp. 55-64, Toronto (Canada), 2008.