# Reaching agreement in the presence of contention-related crash failures

Anaïs Durand, Michel Raynal, Gadi Taubenfeld

# Reaching Agreement in the Presence of Contention-Related Crash Failures

Anaïs Durand[1], Michel Raynal[2], and Gadi Taubenfeld[3]

[1]*LIMOS, Université Clermont Auvergne CNRS UMR 6158, Aubière, France*
[2]*IRISA, CNRS, Inria, Univ Rennes, 35042 Rennes, France*
[3]*Reichman University, Herzliya 4610101, Israel*

**Abstract**

While consensus (and more generally agreement problems) is at the heart of many coordination problems in asynchronous distributed systems prone to process crashes, it has been shown to be impossible to solve in such systems where processes communicate by message-passing or by reading and writing a shared memory. Hence, these systems must be enriched with additional computational power for consensus to be solved on top of them. This article presents a new restriction of the classical basic computational model that combines process participation and a constraint on failure occurrences that can happen only while a predefined contention threshold has not yet been bypassed. This type of failure is called $\lambda$-*constrained crashes*, where $\lambda$ defines the considered contention threshold. It appears that when assuming such contention-related crash failures and enriching the system with objects whose consensus number is $x \geq 1$, consensus for $n$ processes can be solved for any $n \geq x$ assuming up to $x$ failures. The article proceeds incrementally. It first presents an algorithm that solves consensus on top of read/write registers if at most one crash occurs before the contention threshold $\lambda = n-1$ has been bypassed. Then, the article considers two extensions. The first one assumes that the system is enriched with objects whose consensus number is $x \geq 1$, and shows that when $\lambda = n-x$, consensus can be solved despite up to $x$ $\lambda$-constrained crashes, for any $n \geq x$, and when $\lambda = n-2x+1$, consensus can be solved despite up to $2x-1$ $\lambda$-constrained crashes, assuming $x$ divides $n$. The second extension prolongs the previous results to the $k$-set agreement problem (which is a natural generalization of consensus). Impossibility results are also presented for the number of $\lambda$-constrained failures that can be tolerated.

**Keywords.** Algorithm · Asynchronous system · Atomic register · Concurrency · Consensus · Consensus number · Contention · Impossibility · $k$-set agreement · $\lambda$-constrained failure · Participating process · Process crash failure · Read/write register

# 1 Introduction

**Consensus, $k$-set agreement, and contention-related crash failures** Consensus is one of the most important problems encountered in crash-prone asynchronous distributed systems. Its statement is pretty simple. Let us consider a system of $n$ asynchronous sequential processes denoted $p_1$, ..., $p_n$. Each process $p_i$ is assumed to propose a value and, if it does not crash, must decide a value (Termination property) such that no two processes decide different values (Agreement property) and the decided value is a proposed value (Validity property). Despite its very simple statement, consensus is impossible to solve in the presence of asynchrony and process crashes, even if a single process may crash, be the communication medium message-passing [7], or atomic read/write registers [12].

$k$-Set agreement has been introduced in [3]. It is a simple generalization of consensus. It has the same validity and termination properties and a weaker agreement property, namely, at most $k$ different values can be decided. So consensus is 1-set agreement. It has been shown in [2, 11, 19] that $k$-set agreement cannot be solved when the number of process crashes is equal or greater than $k$.

As far as consensus is concerned, in a very interesting way, Fischer, Lynch, and Paterson presented in Section 4 of [7] an algorithm for asynchronous message-passing systems that solves consensus if a majority of processes do not crash and the processes that crash do it initially (the number of crashes being unknown to the other processes [23]). This poses the following question: Can some a priori knowledge on the timing of failures impact the possibility/impossibility of consensus in the presence of process crash failures? As the notion of "timing" is irrelevant in an asynchronous system, Taubenfeld replaced the notion of time with the notion of contention degree and, to answer the previous question, he introduced in [22] the explicit notion of *weak failures*, then renamed *contention-related crash failures* in [5][1].

More precisely, given a predefined contention threshold $\lambda$, a $\lambda$-constrained crash failure is a crash that occurs while process contention is smaller or equal to $\lambda$. Considering read/write shared memory systems and $\lambda = n - 1$, a consensus algorithm is presented in [5, 22] that tolerates one $\lambda$-constrained crash (*i.e.*, at most one process may crash, which may occur only when the contention degree (namely, the number of participating processes) is $\leq n-1$), and it is shown that this bound (on the number of failures) is tight.[2] In addition, upper and lower bounds for solving the $k$-set agreement problem [3] in the presence of multiple contention-related crash failures for $k \geq 2$ are presented in [5, 22]. Unlike the results presented in this article, all the results presented in [5, 22] are only for read/write shared memory systems, that is, systems in which only registers that support read and write operations can be used.

**Motivation: Why $\lambda$-Constrained Failures?** The first and foremost motivation for this study is related to the basics of computing, namely, increasing our knowledge of what can (or

[1] Contention refers here to the number of processes that started participating to the algorithm, where a process starts participating when it accesses the shared memory for the first time.

[2] The consensus algorithm described in [5, 22] does not use adopt/commit objects as done in the present article. As we will see, this object is crucial for the present paper.

cannot) be done in the context of asynchronous failure-prone distributed systems. Providing necessary and sufficient conditions helps us determine and identify under which type of (weak) process failures the fundamental consensus problem is solvable.

As discussed and demonstrated in [5, 22], the new type of $\lambda$-constrained failures enables the design of algorithms that can tolerate several traditional "any-time" failures plus several additional $\lambda$-constrained failures. More precisely, assume that a problem can be solved in the presence of $t$ traditional failures but cannot be solved in the presence of $t+1$ such failures. Yet, the problem might be solvable in the presence of $t_1 \leq t$ "any-time" failures plus $t_2$ $\lambda$-constrained failures, where $t_1 + t_2 > t$.

Adding the ability to tolerate $\lambda$-constrained failures to algorithms that are already designed to circumvent various impossibility results, such as the Paxos algorithm [14] and indulgent algorithms in general [8, 9], would make such algorithms even more robust against possible failures. An indulgent algorithm never violates its safety property and eventually satisfies its liveness property when the synchrony assumptions it relies on are satisfied. An indulgent algorithm which in addition (to being indulgent) tolerates $\lambda$-constrained failures may, in many cases, satisfy its liveness property even before the synchrony assumptions it relies on are satisfied.

When facing a failure-related impossibility result, such as the impossibility of consensus in the presence of a single faulty process, discussed earlier [7], one is often tempted to use a solution that guarantees no resiliency at all. We point out that there is a middle ground: tolerating $\lambda$-constrained failures enables to tolerate failures some of the time. Notice that traditional $t$-resilient algorithms also tolerate failures only some of the time (i.e., as long as the number of failures is at most $t$). After all, *something is better than nothing.* As a simple example, an algorithm is described in [7], which solves consensus despite asynchrony and up to $t < n/2$ processes crashes if these crashes occur initially (hence no participating process crashes).

**Content of the article** As already said, this article investigates the interplay between asynchrony, process crashes, contention threshold, and the computability power of base objects as measured by their consensus number [10]. Let us recall that the consensus number of an object $O$ (denote $CN(O)$) is the maximal number of processes for which consensus can be solved despite any number of process crashes (occurring at any time) with any number of objects $O$ and read/write registers. If there is no such integer, $CN(O) = +\infty$.

After a presentation of the computing model in Section 2, the article is made up of four main sections.

- Section 3 presents a consensus algorithm built on top of read/write registers (RW), which tolerates one process crash occurring before the contention degree $\lambda$ bypasses $n-1$.
- Section 4 generalizes the previous algorithm by presenting two (reduction) algorithms that solve consensus on top of objects whose consensus number is $x \geq 1$.
  - The first algorithm tolerates up to $x$ process crashes that may occur before the contention degree $\lambda$ bypasses $n-x$.

  - The second algorithm, which assumes $x$ divides $n$, tolerates up to $2x-1$ process crashes that may occur before the contention degree $\lambda$ bypasses $n-2x+1$.
- Section 5 generalizes the first algorithm by presenting a $k$-set agreement algorithm which tolerates up to $k$ process crashes occurring before the contention degree $\lambda$ bypasses $n-k$.
- Section 6 presents our most general algorithm, which solves $k$-set agreement on top of objects whose consensus number is $x \geq 1$. This algorithm tolerates up to $kx$ crashes that may occur before the contention degree $\lambda$ bypasses $n-kx$.
- Finally, Section 7 presents impossibility results that address the limits of the proposed approach.

# 2 Computing Model

**Process and communication model** The system is composed of $n$ asynchronous sequential processes denoted $p_1$, ..., $p_n$. The index of $p_i$ is the integer $i$. Asynchronous means that each process proceeds at its own speed, which can vary with time and remains unknown to the other processes [17, 20].

A process can crash (a crash is an unexpected premature silent halt). Given an execution, a process that crashes is said to be *faulty* in that execution, otherwise, it is *correct*. Notice that a faulty process $p$ either fail before taking any steps in which case we will say that $p$ failed *initially*, or $p$ may fail after it has started participating is the algorithm.

As already indicated, the word *contention* denotes the current number of processes that started executing. A $\lambda$-constrained crash is a crash that occurs before the contention degree bypasses $\lambda$.[3]

The processes communicate through a shared memory made of the following base objects:

- Read/write atomic registers (RW).
- Atomic objects with consensus number $x \geq 1$ (these objects, denoted $x$-CONS, will be used in Sections 4 and 6).
- Adopt/commit objects (see below).

**The adopt-commit object** This object can be built in asynchronous read/write systems prone to any number of process crashes. Hence, its consensus number is 1. It was introduced by Gafni in [13]. It provides the processes with a single operation (that a process can invoke only once) denoted ac_propose(). This operation takes a value as input parameter and returns a pair $\langle tag, v\rangle$, where $tag \in \{\texttt{adopt}, \texttt{commit}\}$ and $v$ is a proposed value (we say that the process decides a pair). The following properties define the object.

- *Termination.* A correct process that invokes ac_propose() returns from its invocation.
- *Validity.* If a process returns the pair $\langle -, v\rangle$, then $v$ was proposed by a process.

[3]There is no restriction on the the number of processes that may crash. If all the processes crash, the problem we want to solve becomes irrelevant. So the proofs consider runs in which at least one process does not crash.

- *Obligation.* If the processes that invoke ac_propose() propose the same input value $v$, only the pair $\langle \texttt{commit}, v \rangle$ can be returned.
- *Weak agreement.* If a process decides $\langle \texttt{commit}, v \rangle$, then any process that decides returns the pair $\langle \texttt{commit}, v \rangle$ or $\langle \texttt{adopt}, v \rangle$.

**Process participation** As in message-passing systems (see *e.g.*, [1, 18]), it is assumed that all the processes participate in the algorithm. (Equivalently, a process that does not participate is considered as having crashed initially.)

**Consensus and $k$-set agreement** The *consensus* object was defined in [16]. This *one-shot* object provides a single operation, denoted propose() to the processes. This operation allows the invoking process to propose a value and obtain a result (called *decided* value). Assuming each correct process proposes a value, each process must decide on a value such that the following properties are satisfied.

- *Validity.* A decided value is a proposed value.
- *Agreement.* No two processes decide different values.
- *Termination.* Every correct process decides a value.

A *$k$-set agreement* object is a one-shot object introduced by S. Chaudhuri [3] to study the relation linking the number of failures and the agreement degree attainable in a set of crash-prone asynchronous processes. Such an object is similar to a consensus object. It is defined by the same validity and termination properties, and the following weaker agreement property.

- *Agreement.* At most $k$ different values are decided.

Hence, when $k = 1$, $k$-set agreement boils down to consensus.

**Proposed values** Without loss of generality, it is assumed that the values proposed in a consensus instance are non-negative integers, and $\perp$ is greater than any proposed value.

**Line numbering** The lines of the presented algorithms are identified as follows. All the lines with the same identifier are the very same line. In Algorithm 1, the lines are numbered from 1 to 7. Let $\ell \in \{1, ..., 7\}$. In Algorithm 2 a line identified N$\ell$ is a new line, a line identified $\ell$-X is line $\ell$ of Algorithm 1 that has been modified to take into account $x$-CONS objects, a line identified $\ell$-K is line $\ell$ of Algorithm 3 that has been modified (to solve $k$-set agreement), and finally, a combination of these notations is used in Algorithm 4.

# 3 Base Algorithm $k = 1, \quad x = 1$: Consensus from Read/Write Registers

This section presents an algorithm that solves consensus on top of RW registers (the consensus number of which is 1) while tolerating one crash that occurs before the contention degree bypasses $\lambda = n - 1$.

## 3.1 Presentation of the Algorithm

**Shared base objects** The processes cooperate through the following objects.

- $INPUT[1..n]$ is an array of atomic single-writer multi-reader registers. Each of its entries is initialized to $\perp$, a value that cannot be proposed by the processes and is greater than any of these values. $INPUT[i]$ will contain the value proposed by $p_i$.
- $DEC$ is a multi-writer multi-reader atomic register, the aim of which is to contain the decided value. It is initialized to $\perp$.
- $LAST$ will contain the index of a process.
- $AC$ is an adopt/commit object.

**Local objects** Each process $p_i$ manages:

- three local variables denoted $val_i$, $res_i$ and $tag_i$, and
- two arrays denoted $input1[1..n]$ and $input2[1..n]$.

The initial values of the previous local variables are irrelevant. The value proposed by $p_i$ is denoted $in_i$.

```
operation propose(in_i) is                                              code for p_i
(1)  INPUT[i] ← in_i;
(2)  repeat input1_i ← asynchronous non-atomic reading of INPUT[1..n];
            input2_i ← asynchronous non-atomic reading of INPUT[1..n]
     until (input1_i = input2_i ∧ input1_i contains at most one ⊥ ) end repeat;
(3)  val_i ← min(values deposited in input1_i[1..n]);
(4)  if (∃ j such that input1_i[j] = ⊥) then LAST ← j end if;
(5)  ⟨tag_i, res_i⟩ ← AC.ac_propose(val_i);
(6)  if (tag_i = commit ∨ LAST = i) then DEC ← res_i else wait(DEC ≠ ⊥) end if;
(7)  return(DEC).
```

Algorithm 1: Consensus tolerating one $(n-1)$-constrained failure (on top of atomic RW registers)

**The behavior of a process $p_i$: Algorithm *1*** When a process $p_i$ invokes propose($in_i$), it first deposits the value $in_i$ in $INPUT[i]$ (Line 1) and waits until the array $INPUT[1..n]$ contains at least $(n-1)$ entries different from their initial value $\perp$ (Line 2). Because at most one process may crash, and the process participation assumption, the wait statement eventually terminates.

After this occurs, $p_i$ computes the smallest value deposited in the array $INPUT[1..n]$ (Line 3, remind that $\perp$ is greater than any proposed value). If $INPUT[1..n]$ contains an entry equal to $\perp$, say $INPUT[j]$, $p_i$ observes that $p_j$ is a belated process (or $p_j$ the only process that may crash and it crashed before depositing its value in $INPUT[j]$) and posts this information in the shared register $LAST$ (Line 4).

Then, $p_i$ champions its value $val_i$ for it to be decided. To this end, it uses the underlying adopt/commit object, namely, it invokes $AC$.ac_propose($val_i$) from which it obtains a pair

$\langle tag_i, res_i \rangle$ (Line 5). There are three possible cases for a process $p_i$; at the end of which it decides at Line 7.

- If $tag_i = \texttt{commit}$, due to the Weak Agreement property of the object $AC$, no value different from $res_i$ can be decided. Consequently, $p_i$ writes $res_i$ in the shared register $DEC$ (Line 6) and returns it as the consensus value (Line 7).
- The same occurs if, while $tag_i = \texttt{adopt}$, $p_i$ is such that $LAST = i$. In this case, $p_i$ has seen all the entries of the array $INPUT[1..n]$ filled with non-$\perp$ values and imposes $res_i$ as the consensus value.
- If $p_i$ is such that $tag_i = \texttt{adopt} \wedge LAST \neq i$, it waits until it sees $DEC \neq \perp$, and decides.

## 3.2 Proof of Algorithm *1*

**Lemma 1.** *Algorithm* 1 *satisfies the* Validity *property of consensus.*

**Proof.** It is easy to see that a value written in $DEC$ is obtained from the adopt/commit object at Line 5. Moreover, due to Line 1 and Line 3 (where $\perp$ is greater than any proposed value), only values proposed to consensus can be proposed to the adopt/commit object. $\square_{Lemma\ 1}$

**Lemma 2.** *Let us consider an execution in which no process crashes. Algorithm* 1 *satisfies the* Agreement *property of consensus.*

**Proof.** Let $p_\ell$ be the last process that writes the value it proposes in $INPUT[1..n]$. It follows from Line 3 that $p_\ell$ computes the smallest value in the array, and from Line 2 and Line 4 that, no index different from $\ell$ can be assigned to $LAST$. There are then two cases according to the value of the pair $\langle tag, res \rangle$ returned at Line 5.

- If a process $p_i$ obtains $\langle \texttt{commit}, res \rangle$, it follows from the Weak Agreement property of the adopt/commit object that any other process can obtain $\langle \texttt{commit}, res \rangle$ or $\langle \texttt{adopt}, res \rangle$ only. We then have $DEC = res$ after the execution of Line 6. This is because the assignment at Line 6 can be executed only by a process that obtained $\langle \texttt{commit}, res \rangle$ or by $p_\ell$ (which is $p_{LAST}$) which obtained $\langle \texttt{commit}, res \rangle$ or $\langle \texttt{adopt}, res \rangle$ from its invocation of the $AC$ object.
- If at Line 5 no process obtains $\langle \texttt{commit}, - \rangle$, it follows from Line 6 that only $p_\ell$ assigns a value to $DEC$, and consequently, no other value can be decided.

$\square_{Lemma\ 2}$

**Lemma 3.** *Let us consider executions in which one process crashes. Algorithm* 1 *satisfies the* Agreement *property of consensus.*

**Proof.** Let us recall that by assumption (namely, contention-related crash failures) if a process $p_j$ crashes, it can do it only when the contention is lower or equal to $(n-1)$. We consider two cases.

- If $p_j$ crashes initially (i.e., before writing the value it proposes in $INPUT[j]$, this array will eventually contain $(n-1)$ non-$\perp$ entries, and all the correct processes will consequently compute the same minimal value $val$ that they will propose to the underlying adopt/commit object (Line 5). It then follows from the Obligation property of this object that all the correct processes will obtain the same pair $\langle \texttt{commit}, res \rangle$, from which we conclude that a single value can be decided.
- The process $p_j$ crashes after it writes the value it proposes in $INPUT[j]$. There are two cases.
  - When exiting the repeat loop (Line 2), the local array $input1_i$ of all processes does not contain $\perp$. In this case, we are as in the previous item (replacing $INPUT[1..n]$ with one $\perp$ value by $INPUT[1..n]$ with no $\perp$ value).
  - There is an entry $\ell$ such that, when exiting Line 2, there is some process $p_a$ where $input1_a[\ell] = \perp$ and all other entries are different than $\perp$ (let call $A$ this set of processes), while other process $p_b$ is such that all entries of $input1_b$ are different from $\perp$ (set $B$). $p_\ell$ is the last process to write into $INPUT$ and belongs to $B$. Notice that $p_j$ is not the last process to write into $INPUT$ since it crashes when the contention threshold is lower or equal to $(n-1)$. Thus $\ell \neq j$ and $p_\ell$ is correct. Processes of set $A$ write $\ell$ in $LAST$ at Line 4. Thus $LAST$ contains the identity of a correct process. The rest of the proof is the same as the proof of Lemma 2.

$\square_{Lemma\ 3}$

**Lemma 4.** *If $j = LAST$ then process $p_j$ is either correct (that is, will never fail) or it has failed initially (that is, before taking a step).*

**Proof.** If $LAST = j$ at Line 6, there is a process $p_i$ that wrote $j$ in $LAST$ at Line 4. This means that $p_i$ found $input1_i[j] = \perp$ at Line 4 and every other entry of $input1_i$ was different than $\perp$. Thus, we conclude that the contention threshold $\lambda = n - 1$ was attained when $p_i$ wrote $j$ in $LAST$. But, by assumption, no process crashes after the contention threshold $\lambda = n - 1$ has been attained. So, either $p_j$ failed initially or it is correct process. $\square_{Lemma\ 4}$

**Lemma 5.** *Algorithm* 1 *satisfies the* Termination *property of consensus.*

**Proof.** Due to the assumption that all the processes participate and at most one process can crash, no process can block forever at Line 2.

Hence, all the correct processes invoke $AC.\mathsf{ac\_propose}(val_i)$ and, due the Termination of the adopt/commit object, return from their invocation. If the tag `commit` is returned at some correct process $p_j$, this process assigns a value to $DEC$. If the tag is `adopt`, then it must be the case that $p_j$ has invoked $AC.\mathsf{ac\_propose}(val_j)$, and hence $p_j$ did not failed initially.

Due to Lemma 4, the process $p_j$ such that $j = LAST$ must be a correct process. Hence, it then assigns a non-$\perp$ value to $DEC$. So, in all cases, we have eventually $DEC \neq \perp$, which concludes the proof. $\square_{Lemma\ 5}$

**Theorem 1.** *Let $\lambda = n - 1$. Considering an asynchronous* RW *system, Algorithm* 1 *solves consensus for $n$ processes in the presence of at most one $\lambda$-constrained failure.*

**Proof.** The proof follows from the previous lemmas. $\square_{Theorem\ 1}$

We notice that this bound is tight. When using only atomic registers, there is no consensus algorithm for $n$ processes that can tolerate two $(n-1)$-constrained crash failures (Corollary 1 in [5, 22]).

# 4 Generalization for $k = 1,\ x \geq 1$: Consensus from Objects whose Consensus Number is $x$

As we are about to see, Algorithm 2 is a reduction to Algorithm 1, which exploits the additional power provided by objects (denoted $x$-CONS) whose consensus number is $x$. We present below two consensus algorithms:

- Algorithm 2, which tolerates up to $x$ $(n-x)$-constrained failures, and
- Algorithm 4.5, which tolerates up to $2x-1$ $(n-2x+1)$-constrained failures, assuming $x$ divides $n$.

## 4.1 Presentation of Algorithm *2*

**Shared objects** Algorithm 2 uses the same shared registers $DEC$, $LAST$, and $AC$ as Algorithm 1. It also uses:

- An array $INPUT[1..\lceil n/x \rceil]$ where each entry $INPUT[j]$ (instead of being a simple read/write register) is a $x$-CONS object, and
- A Boolean array denoted $PARTICIPANT[1..n]$, whose entries are initialized to `false`.
- An array $XCONS[1..\lceil n/x \rceil]$ of $x$-CONS objects.

**The behavior of a process $p_i$** As already said, Algorithm 2 is a simple adaptation of Algorithm 1 to the use of $x$-CONS objects.

- The lines N1 and N2 are new. They aim to ensure that no process will block forever despite up to $x$ crashes.
- Each set of at most $x$ processes $p_i$, $p_j$, etc. such that $\lceil i/x \rceil = \lceil j/x \rceil$, defines a cluster of processes that share the same $x$-CONS object. Consequently, all the processes of a cluster act as if they were a single process, namely, no two different values can be written in $INPUT[\lceil i/x \rceil]$ by processes belonging to the same cluster.

## 4.2 Further Explanations

Before proving Algorithm 2, let us analyze it with two questions/answers.

```
operation propose(in_i) is                                              code for p_i
(N1)   PARTICIPANT[i] ← true;
(N2)   repeat participant_i ← asynchronous reading of PARTICIPANT[1..n]
       until participant_i[1..n] contains at most x entries with false end repeat;
(1-X)  INPUT[⌈i/x⌉] ← XCONS[⌈i/x⌉].propose(in_i);
(2-X)  repeat input1_i ← asynchronous non-atomic reading of INPUT[1..⌈n/x⌉];
              input2_i ← asynchronous non-atomic reading of INPUT[1..⌈n/x⌉]
       until input1_i = input2_i ∧ input1_i contains at most one ⊥ end repeat;
(3)    val_i ← min(values deposited in input1_i);
(4)    if (∃ j such that input1_i[j] = ⊥) then LAST ← j end if;
(5)    ⟨tag_i, res_i⟩ ← AC.ac_propose(val_i);
(6)    if (tag_i = commit ∨ LAST = ⌈i/x⌉)
                                  then DEC ← res_i else wait(DEC ≠ ⊥) end if ;
(7)    return(DEC);
```

Algorithm 2: Consensus tolerating up to $x$ $(n-x)$-constrained failures (on top of $x$-CONS objects)

**Question *1*** Can Algorithm 2 where $x \geq 1$, tolerates $(x+1)$ $(n-(x+1))$-constrained process crashes?

The answer is "no." This is because if $(x+1)$ processes crash, for example, initially (as allowed by the $(n-(x+1))$-constrained assumption), the other processes will remain blocked forever in the loop of Line N2. This entails the second question.

**Question *2*** Are the lines N1-N2 needed?

Let us consider Algorithm 2 without the lines N1-N2 and with $x = 2$, and let us examine the following possible scenario which involves five processes $p_1$, ..., $p_5$. So, $p_1$ and $p_2$ belong the cluster 1, $p_3$ and $p_4$ belong the cluster 2, and $p_5$ belongs to cluster 3. Let us assume that the value $in_5$ proposed by $p_5$ is smaller than the other proposed values.

- Process $p_1$ executes Line 1-X and writes in $INPUT[1]$.
- Process $p_3$ executes Line 1-X and writes in $INPUT[2]$.
- Both processes $p_1$ and $p_3$ execute Line 4 and write the cluster number 3 in $LAST$.
- Then, process $p_5$ executes from Line 1-M until Line 4.
- Then, the processes $p_1$, $p_3$, and $p_5$ execute Line 5, and obtain the tag `adopt`.
- Then $p_5$ crashes. It follows that $p_5$ will never write in $DEC$ which forever remains equal to $\perp$.
- Then $p_2$ and $p_4$ execute Line 1-X to Line 4, and obtain `adopt` from the adopt/commit object.
- It follows that, when the processes $p_1$, $p_2$, $p_3$, and $p_4$ execute Line 6 they remain forever blocked in the wait statement.

Hence, Lines N1 and N2 cannot be suppressed from Algorithm 2.

## 4.3 Proof of Algorithm *2*

**Theorem 2.** *Let $n \geq x$ and $\lambda = n - x$. Considering an asynchronous* RW *system enriched with $x$-*CONS *objects, Algorithm* 2 *solves consensus for $n$ processes in the presence of at most $x$ $\lambda$-constrained crash failures.*

**Proof.** Let us first observe that, as at most $x$ processes may crash, no process can block forever at Line N2.

Now, let us show that the lines N1-N2 cannot entail a process to block forever at any line from 1-X to 7. To this end, let us consider the $n$ processes are partitioned in clusters of at most $x$ processes so that $p_i$ belongs to the cluster identified $\lceil i/x \rceil$. A cluster crashes if all its processes crash. A cluster is alive if at least one of its processes does not crash. There are two cases.

- Each cluster contains at least one process that does not crash, so all the clusters are alive. It follows that, when a process executes Line 4 and assigns a cluster identity to $LAST$, it is the identity of an alive cluster, from which follows that (if needed due to the predicate of Line 4) a correct process will be able to write a value in $DEC$, thereby preventing processes from being blocked forever in the wait statement at Line 6.
- All the processes in a cluster crash. Let us notice that at most one cluster can crash.[4] In this case, considering the clusters (instead of the processes) and replacing $n$ by $\lceil n/x \rceil$, we are in the same case as in the proof of Lemma 3.

$\square_{Lemma\ 2}$

Since an $x$-CONS object is also a $k$-CONS object for every $k \leq x$, the following theorem is a direct consequence of the previous one.

**Theorem 3.** *Let $n \geq x$ and consider an asychronous* RW *system enriched with enriched with $x$-*CONS *objects. For every $k \leq x$ and $\lambda = n - k$, an algorithm exists that solves consensus for $n$ processes in the presence of at most $k$ $\lambda$-constrained crash failures.*

## 4.4 When $x$ Divides $n$: Tolerating $x - 1$ Classical Any-time Failures

Let us consider the case where crash failures are not constrained. Those are the classical crashes that can occur at any time (they are called *any-time* failures in [5]). It is known that there is no consensus algorithm for $n \geq x + 1$ processes that can tolerate $x$ any-time failures, using registers and wait-free consensus objects for $x$ processes [10]. In such a model, Algorithm *2* has the property captured by the following theorem.

**Theorem 4.** *If $x$ divides $n$, Algorithm* 2 *tolerates $x - 1$ any-time failures.*

**Proof.** Using the cluster terminology defined in the previous proof, $x$ divides $n$, each cluster contains $x$ processes exactly. As at most $(x - 1)$ processes may crash, it follows that all the clusters must be alive. The rest of the proof is the same as the proof of Theorem 2.

$\square_{Theorem\ 4}$

[4] If $x$ does not divides $n$, and the cluster that crashes contains less than $x$ processes, no other cluster can crash.

### 4.5 When $x$ divides $n$: Tolerating $2x-1$ contention-related crash failures

Let Algorithm be Algorithm 2 where, at line 2 the predicate "$participant_i[1..n]$ contains at most $x$ entries with `false`" is replaced with "$participant_i[1..n]$ contains at most $2x-1$ entries with `false`". Then, the following theorem holds.

**Theorem 5.** *Assume that $x$ divides $n$, $n \geq 2x-1$, and $\lambda = n-2x+1$. Considering an asynchronous* RW *system enriched with $x$-*CONS *objects, Algorithm 2 solves consensus for $n$ processes in the presence of up to $(2x-1)$ $\lambda$-constrained crash failures.*

**Proof.** Using the cluster terminology defined in the proof of Algorithm 2, $x$ divides $n$, implies that each cluster contains $x$ processes exactly. As at most $2x-1$ processes may crash, it follows that all the clusters, except maybe one, must be alive. The rest of the proof is similar to the proof of Theorem 2. $\square_{Theorem\ 5}$

### 4.6 A few refinements

As suggested by a reviewer, the previous results can be improved when the distribution of the $n \geq x$ processes among the $b = \lceil \frac{n}{x} \rceil$ clusters is balanced. This can be achieved by assigning process $p_i$ to cluster $(i \bmod b)+1$ (instead of assigning it to $\lceil \frac{i}{x} \rceil$). The consequence of this is that all clusters have $(x-1)$ or $x$ processes. Then, there are three cases for which the previous theorems remain valid.

- If $n$ is a multiple of $x$, all clusters have $x$ processes.
- If $(n+1)$ is a multiple of $x$, all clusters have $x$ processes except one that has $(x-1)$ processes. Then, Theorem 5 can be adapted to prove that consensus can be solved if $n \geq 2x-2$, $\lambda = n-2x+2$, and the number of failures is up to $(2x-2)$.
- Otherwise, all clusters have at least $(x-1)$ processes. Then, Theorem 5 can be adapted to prove that consensus can be solved if $n \geq 2x-3$, $\lambda = n-2x+3$, and the number of failures is up to $(2x-3)$.

## 5 Generalization for $k \geq 1$, $x = 1$: $k$-set Agreement from Read/Write Registers

Algorithm 1 can be generalized to $k$-set agreement with $k \geq 1$. The obtained algorithm tolerates up to $k$ $(n-k)$-constrained failures.

## 5.1 Presentation of Algorithm *3*

**Shared objects** Algorithm 3 uses the same shared objects $INPUT$, $DEC$, and $AC$ as Algorithm 1. However, the shared register $LAST$ is replaced by a Boolean array of atomic single-writer multi-reader registers. Each of its entries is initialized to `false`.

**The behavior of a process $p_i$** Algorithm 3 is nearly the same as Algorithm 1. The two differences are as follows.

- Up to $k$ processes may crash, instead of only one for Algorithm 1, thus processes can leave the waiting loop of Line 2-K when they have seen at least $n - k$ participants.
- The management of the late processes is different. Up to $k$ processes are not waited for in the loop of Line 2-K. Thus, a process $p_i$ may see up to $k$ $\perp$-values inside $input1_i$ after exiting Line 2-K. If $input1_i[j]$ equals $\perp$, $p_j$ is a belated process or has crashed before depositing its input values in $INPUT[j]$. $p_i$ posts this information by switching $LAST[j]$ to `true`.
  After the adopt/commit on Line 5, there are two cases:
  - If a process $p_i$ gets the pair $\langle \texttt{commit}, res_i \rangle$, due to the Weak Agreement property of the adopt/commit object, no value different from $res_i$ can be decided. So $p_i$ writes $res_i$ in the shared register $DEC$ (Line 6-K).
  - If no process $p_i$ gets the `adopt` tag, the only values that can be decided are the ones obtained by processes $p_j$ such that $LAST[j]$ equals `true`. But there can be at most $k$ such processes so they can put their $res_j$ in $DEC$ (Line 6-K) and at most $k$ different values will be decided.

```
operation propose(in_i) is                                              code for p_i
(1)    INPUT[i] ← in_i;
(2-K)  repeat input1_i ← asynchronous non-atomic reading of INPUT[1..n];
               input2_i ← asynchronous non-atomic reading of INPUT[1..n]
       until (input1_i = input2_i ∧ input1_i contains at most k ⊥) end repeat;
(3)    val_i ← min(values deposited in input1_i[1..n]);
(4-K)  for each j ∈ {1,..,n} such that input1_i[j] = ⊥ do LAST[j] ← true end for;
(5)    ⟨tag_i, res_i⟩ ← AC.ac_propose(val_i);
(6-K)  if (tag_i = commit ∨ LAST[i]) then DEC ← res_i else wait(DEC ≠ ⊥) end if;
(7)    return(DEC).
```

Algorithm 3: $k$-set agreement tolerating $k$ $(n-k)$-constrained failure (on top of atomic RW registers)

## 5.2 Proof of Algorithm *3*

**Lemma 6.** *Algorithm* 3 *satisfies the* Validity *property of $k$-set agreement.*

**Proof.** Same proof Lemma 1. $\square_{Lemma\ 6}$

**Lemma 7.** *$LAST[1..n]$ contains at most $k$ entries equal to* `true`.

**Proof.** By contradiction, assume there are at least $k+1$ `true`-values in $LAST[1..n]$. Initially $LAST$ contains only `false` values. By Lines 2-K and 4-K, it means that there are two processes $p_{i_1}$ and $p_{i_2}$ such that, when they execute Line 4-K, $\exists j_1, j_2$, $j_1 \neq j_2$, such that $input1_{i_1}[j_1] = \perp \wedge input1_{i_1}[j_2] \neq \perp$ and $input1_{i_2}[j_1] \neq \perp \wedge input1_{i_2}[j_2] = \perp$.

Since processes only exit the loop of Line 2-K after two similar consecutive reads of $INPUT$ and never write $\perp$ into $INPUT$, it is impossible. $\square_{Lemma\ 7}$

**Lemma 8.** *Algorithm* 3 *satisfies the* Agreement *property of $k$-set agreement.*

**Proof.** There are two cases according to the value of the pair $\langle tag, res\rangle$ returned at Line 5.

- If a process $p_i$ obtains $\langle$`commit`$, res\rangle$, it follows from the Weak Agreement property of the adopt/commit object that any other process can obtain $\langle$`commit`$, res\rangle$ or $\langle$`adopt`$, res\rangle$ only. We then have $DEC = res$ after the execution of Line 6-K. Indeed, the assignment at Line 6-K can only be executed by a process that obtained $\langle$`commit`$, res\rangle$ or a process $p_j$ such that $LAST[j] =$ `true`. In the latter case, $p_j$ obtained $\langle$`commit`$, res\rangle$ or $\langle$`adopt`$, res\rangle$ from its invocation of the $AC$ object.
- If no process obtains $\langle$`commit`$, res\rangle$, it follows from Line 6-K that only processes $p_i$ such that $LAST[i] =$ `true` assigns a value to $DEC$. By Lemma 7, at most $k$ different values can be assigned to $DEC$ and consequently, no other value can be decided.

$\square_{Lemma\ 8}$

**Lemma 9.** *Algorithm* 3 *satisfies the* Termination *property of $k$-set agreement.*

**Proof.** Due to the assumption that all the processes participate and at most $k$ processes can crash, no process can block forever at line 2-K.

Hence, all the correct processes invoke $AC.\mathsf{ac\_propose}(val_i)$ and, due the Termination of the adopt/commit object, return from their invocation. If the tag `commit` is returned at some correct process $p_j$, this process assigns a value to $DEC$. If the tag is `adopt`, we claim that at least one process $p_j$ such that $LAST[j] =$ `true` is a correct process. Hence, it then assigns a non-$\perp$ value to $DEC$. So, in all cases, we have eventually $DEC \neq \perp$, which concludes the proof.

Proof of the claim. If $LAST[j] =$ `true` at Line 6-K, there is a process $p_i$ that wrote `true` in $LAST[j]$ at Line 4-K. This means that $p_i$ found $input1_i[j] = \perp$ at Line 4-K and there was at most $(k-1)$ other $\perp$-entries in $input1_i$. Thus, we conclude that the contention threshold $\lambda = n - k$ was attained when $p_i$ wrote `true` in $LAST[j]$. But, by assumption, no process crashes after the contention threshold $\lambda = n - k$ has been attained. So, $p_j$ is a correct process. $\square_{Lemma\ 9}$

**Theorem 6.** *Let $\lambda = n - k$. Considering an asynchronous* RW *system, Algorithm* 3 *solves $k$-set agreement for $n$ processes in the presence of at most $k$ $\lambda$-constrained failures, where $1 \leq k < n$.*

**Proof.** The proof follows from the previous lemmas. $\square_{Theorem\ 6}$

# 6 Generalization for $k \geq 1, x \geq 1$: $k$-Set agreement from Objects whose Consensus Number is $x$

We combine the ideas of Algorithms 2 and 3 into Algorithm 4. This $k$-set agreement algorithm tolerates up to $kx$ $(n - kx)$-constrained failures by exploiting the additional power provided by objects whose consensus number is $x$.

## 6.1 Presentation of Algorithm 4

**Shared objects** Algorithm 4 uses the shared objects $DEC$, $LAST$, and $AC$, $PARTICIPANT[1..n]$, $INPUT[1..\lceil n/x \rceil]$ and $XCONS[1..\lceil n/x \rceil]$ used in the previous algorithms.

**The behavior of process $p_i$** Similarly to Algorithm 2, processes wait for $n - kx$ participants (Lines N1 and N2-K). Then, processes are separated into clusters of at most $x$ processes such that two processes $p_i$ and $p_j$ belong to the same cluster if $\lceil i/x \rceil = \lceil j/x \rceil$. Each cluster of processes then emulates the behavior of one process in Algorithm 3, using the $x$-CONS object to determine the value proposed by all processes of the cluster.

```
operation propose(in_i) is                                              code for p_i
(N1)    PARTICIPANT[i] ← true;
(N2-K)  repeat participant_i ← asynchronous reading of PARTICIPANT[1..n]
        until participant_i[1..n] contains at most kx entries with false end repeat;
(1-X)   INPUT[⌈i/x⌉] ← XCONS[⌈i/x⌉].propose(in_i);
(2-KX)  repeat input1_i ← asynchronous non-atomic reading of INPUT[1..⌈n/x⌉];
                  input2_i ← asynchronous non-atomic reading of INPUT[1..⌈n/x⌉]
        until input1_i = input2_i ∧ input1_i contains at most k ⊥ end repeat;
(3)     val_i ← min(values deposited in input1_i);
(4-KX)  for each  j ∈ {1, .., ⌈n/x⌉} such that input1_i[j] = ⊥)
                                                  do LAST[j] ← true end for;
(5)     ⟨tag_i, res_i⟩ ← AC.ac_propose(val_i);
(6-KX)  if (tag_i = commit ∨ LAST[⌈i/x⌉])
                                  then DEC ← res_i else wait(DEC ≠ ⊥) end if;
(7)     return(DEC).
```

Algorithm 4: $k$-set agreement tolerating up to $kx$ $(n - kx)$-constrained failures (on top of $x$-CONS objects)

## 6.2 Proof of Algorithm 4

**Theorem 7.** *Let $n \geq kx$ and $\lambda = n - kx$. Considering an asynchronous* RW *system enriched with $x$-*CONS *objects, Algorithm 4 solves $k$-set agreement for $n$ processes in the presence of at most $kx$ $\lambda$-constrained crash failures.*

**Proof.** Let us first observe that, as at most $k$ processes may crash, no process can block forever at Line N2-K.

Now, let us show that lines N1 and N2-K cannot entail a process to block forever at any line from 1-M to 7. To this end, let us consider the $n$ processes are partitioned in clusters of at most $x$ processes so that $p_i$ belongs to the cluster identified $\lceil i/x \rceil$. A cluster crashes if all its processes crash. A cluster is alive if at least one of its processes does not crash. There are two cases.

- Each cluster contains at least one process that does not crash, so all the clusters are alive. It follows that, when a process executes Line 4-KM and assigns `true` to $LAST[j]$, $j$ is the identity of an alive cluster, from which follows that (if needed due to the predicate of Line 4-KX) a correct process will be able to write a value in $DEC$, thereby preventing processes from being blocked forever in the wait statement at Line 6-KX.
- All the processes in a cluster crash. Let us notice that at most $k$ clusters can crash.[5] In this case, considering the clusters (instead of the processes) and replacing $n$ by $\lceil n/k \rceil$, we are in the same case as in the proof of Lemma 9.

$\square_{Theorem\ 7}$

# 7 Impossibility Results

This section presents impossibility results for an asynchronous model which supports atomic read/write registers and $x$-CONS objects, in which $\lambda$-constrained and any-time crash failures are possible. Let an *initial* crash failure be the crash of a process that occurs before it executes its first access to an atomic read/write register.

Hence, there are three types of crash failures: initial, $\lambda$-constrained, and any-time. Let us say that a failure type T1 is *more severe* than a failure type T2 (denoted T1 $>$ T2) if any crash failure of type T2 is also a crash failure of type T1 but not vice-versa. Considering an $n$-process system, the following severity hierarchy follows from the definition of the failure types: any-time $>$ $(n-1)$-constrained $>$ $(n-2)$-constrained $\cdots$ $>$ 1-constrained $>$ initial (let us observe that any-time is the same as $n$-constrained and initial is the same as 0-constrained).

### Consensus with $\lambda$-constrained failures

**Theorem 8.** *For every $\ell \geq 0$, $x \geq 1$, $n > \ell + x$, and $\lambda = n - \ell$, there is no consensus algorithm for $n$ processes, using atomic* RW *registers and $x$-*CONS *objects, that tolerates $(\ell + x)$ $\lambda$-constrained crash failures (even when assuming that there are no any-time crash failures).*

**Proof.** Assume to the contrary that for some $\ell \geq 0$, $x \geq 1$, $n > \ell + x$, and $\lambda = n - \ell$, there is a consensus algorithm, say $A$, that

[5] If $x$ does not divides $n$, and one cluster that crashes contains less than $x$ processes, at most $k-1$ other clusters can crash.

(1) uses atomic registers and $x$-CONS objects, and
(2) tolerates $\ell + x$ $\lambda$-constrained crash failures.

Given any execution of $A$, let us remove any set of $\ell$ processes by assuming they fail initially (this is possible because $(n-\ell)$-constrained $>$ initial). It then follows (from the contradiction assumption) that the assumed algorithm $A$ solves consensus in a system of $n' = n - \ell$ processes, where $n' > x$, using atomic registers and $x$-CONS objects.

However, in a system of $n' = n - \ell$ processes, process contention is always lower or equal to $n'$, from which follows that, in such an execution, $n'$-constrained crash failures are the same as any-time failures. Thus, algorithm $A$ can be used to generate a consensus algorithm $A'$ for $n' = n - \ell$ processes, where $n' > x$, that
(1) uses only atomic registers and $x$-CONS objects, and
(2) tolerates $x$ any-time crash failures.
But, this is known to be impossible as shown in [10]. A contradiction. $\square_{Theorem\ 8}$

**Consensus using atomic registers only** For the special case of consensus using atomic registers only, the equation $n > \ell + x$ becomes $n > \ell + 1$. The following corollary is then an immediate consequence of Theorem 8.

**Corollary 1.** *For every $0 \leq \ell < n-1$ and $\lambda = n - \ell$, there is no consensus algorithm for $n$ processes, using atomic* RW *registers, that can tolerate $(\ell+1)$ $\lambda$-constrained crash failures (even when assuming that there are no any-time crash failures). In particular, when $\ell = 1$, there is no consensus algorithm for $n$ processes that can tolerate two $(n-1)$-constrained crash failures.*

**Consensus with $\lambda$-constrained and any-time failures**

**Theorem 9.** *For every $\ell \geq 0$, $x \geq 1$, $n > \ell + x$, $g \geq 0$, and $\lambda = n - \ell$, there is no consensus algorithm for $n$ processes, using atomic* RW *registers and $x$-CONS objects, that,tolerates $(\ell + x - g)$ $\lambda$-constrained crash failures and $g$ any-time crash failures.*

**Proof.** Follows immediately from Theorem 8 by observing that any-time crash failures belong to a more severe type of a failure than $\lambda$-constrained crash failures when $\lambda < n$, and is the same as a $\lambda$-constrained crash failure when $\lambda = n$. $\square_{Theorem\ 9}$

# 8 Conclusion

This article has investigated the computability power of the pair made up of process participation plus contention-related crashes, when one has to solve consensus and the more general $k$-set agreement problem in an $n$-process asynchronous shared memory system enriched with objects the consensus number of which is equal to $x$. It has been shown that for $n \geq x$, consensus can be solved in such a context in the presence of up to $x$ process crashes if these crashes occur before process contention has attained the value $\lambda = n - x$. Furthermore,

for the case where $x$ divides $n$, it has been shown that consensus can be solved in such a context in the presence of up to $2x-1$ process crashes if these crashes occur before process contention bypasses the threshold $\lambda = 2n - x + 1$. Then, enriching the computing model with $x$-CONS objects (objects that allow solving consensus among $x$ processes), the previous results have been extended for consensus and $k$-set agreement.

The corresponding consensus algorithms have been built in an incremental way. Namely, a read/write algorithm based on adopt/commit object has first been given, and then generalized by replacing atomic read/write registers by objects whose consensus number is $x$. Developments of the power/limit of this approach have also been presented, increasing our knowledge on an important topic of fault-tolerant process synchronization in asynchronous distributed systems.

# Acknowledgments

The authors want to thank the referees for suggestions that helped improve both the content and the presentation of the article.

M. Raynal has been partially supported by the French projects ByBloS (ANR-20-CE25-0002-01) and PriCLeSS (ANR-10-LABX-07-81) devoted to the design of modular building blocks for distributed applications.

# References

[1] Attiya H. and Welch J.L., *Distributed computing: fundamentals, simulations and advanced topics, (2nd Edition)*, Wiley-Interscience, 414 pages, ISBN 0-471-45324-2 (2004)

[2] Borowsky E. and Gafni E., Generalized FLP impossibility results for $t$-resilient asynchronous computations. *Proc. 25th ACM Symposium on Theory of Computing (STOC'93)*, ACM Press, pp. 91-100 (1993)

[3] Chaudhuri S., More choices allow more faults: set consensus problems in totally asynchronous systems. *Information and Computation*, 105(1):132-158 (1993)

[4] Castañeda A., Gonczarowski Y.A., and Moses Y., Unbeatable consensus. *Distributed Computing*, 35(2): 123-143 (2022)

[5] Durand A., Raynal M., and Taubenfeld G., Contention-related crash-failures: definitions, agreement algorithms and impossibility results. *Theoretical Computer Science*, 909:76-86 (2022)

[6] Reaching consensus in the presence of contention-related crash failures. *Proc. 24th Int'l Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS'22)* Springer LNCS 13751, pp. 193-205 (2022)

[7] 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)

[8] Guerraoui R., Indulgent algorithms. *Proc. 19th Annual ACM Symposium on Principles of Distributed Computing (PODC'00)*, ACM Press, pp. 289-297 (2000)

[9] Guerraoui R. and Raynal M., The information structure of indulgent consensus. *IEEE Transactions on Computers*, 53(4):453-466 (2004)

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

[11] Herlihy M.P. and Shavit N., The topological structure of asynchronous computability. *Journal of the ACM*, 46(6):858-923 (1999)

[12] Loui M. and Abu-Amara H., Memory requirements for agreement among unreliable asynchronous processes. *Advances in Computing Research*, 4:163-183, JAI Press Inc. (1987)

[13] Gafni E., Round-by-round fault detectors: unifying synchrony and asynchrony. *Proc. 17th ACM Symposium on Principles of Distributed Computing (PODC)*, ACM Press, pp. 143-152 (1998)

[14] Lamport L., The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169 (1998)

[15] Moses Y. and Rajsbaum S., A layered analysis of consensus. *SIAM Journal of Computing*, 31(4):989-1021 (2002)

[16] Pease M., Shostak R., and Lamport L. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234 (1980)

[17] Raynal M., *Concurrent programming: algorithms, principles and foundations*. Springer, 515 pages, ISBN 978-3-642-32026-2 (2013)

[18] Raynal M., *Fault-tolerant message-passing distributed systems: an algorithmic approach*. Springer, 492 pages, ISBN 978-3-319-94140-0 (2018)

[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] Taubenfeld G., *Synchronization algorithms and concurrent programming*. Pearson Education/Prentice Hall, 423 pages, ISBN 0-131-97259-6 (2006)

[21] Taubenfeld G., A closer look at fault tolerance. *Theory of Computing Systems*, 62(5) :1085–1108 (2018). (First version in *Proceedings of PODC* 2012, 261–270)

[22] Taubenfeld G., Weak failures: definition, algorithms, and impossibility results. *Proc. 6th Int'l Conference on Networked Systems (NETYS'18)*, Springer LNCS 11028, pp. 269-283 (2018)

[23] Taubenfeld G., Katz S., and Moran S., Initial failures in distributed computations. *International Journal of Parallel Programming*, 18(4):255–276 (1989)