

Reaching Consensus in the Presence of Contention-Related Crash Failures

Anaïs Durand, Michel Raynal, Gadi Taubenfeld

▶ **To cite this version:**

Anaïs Durand, Michel Raynal, Gadi Taubenfeld. Reaching Consensus in the Presence of Contention-Related Crash Failures. SSS 2022 - 24th International Symposium on Stabilization, Safety, and Security of Distributed Systems, Nov 2022, Clermont-Ferrand, France. pp.193-205, 10.1007/978-3-031-21017-4_13 . hal-03853639

HAL Id: hal-03853639
https://hal.science/hal-03853639

Submitted on 15 Nov 2022

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

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

Reaching Consensus in the Presence of Contention-Related Crash Failures

Anaïs Durand*, Michel Raynal†, and Gadi Taubenfeld§

*LIMOS, Université Clermont Auvergne CNRS UMR 6158, Aubière, France
†IRISA, CNRS, Inria, Univ Rennes, 35042 Rennes, France
§Reichman University, Herzliya 4610101, Israel

Abstract. While consensus 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 λ-*constrained crashes*, where λ 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 $k \geq 1$, consensus for n processes can be solved for any $n \geq k$ assuming up to k 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, it shows that if the system is enriched with objects whose consensus number is $k \geq 1$, then

- when $\lambda = n - k$, consensus can be solved despite up to k λ-constrained crashes, for any $n \geq k$, and
- when $\lambda = n - 2k + 1$, consensus can be solved despite up to $2k - 1$ λ-constrained crashes, assuming k divides n.

Finally, impossibility results are presented for the number of λ-constrained failures that can be tolerated.

Keywords: Consensus algorithm, Asynchronous system, Atomic register, Concurrency, Consensus number, Contention, λ-constrained failure, Participating process, Process crash failure, Read/write register.

1 Introduction

Consensus 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 [6], or atomic read/write registers [10].

In a very interesting way, Fischer, Lynch, and Paterson presented in Section 4 of [6] 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 [19]). 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 [18] the explicit notion of *weak failures*, then renamed *contention-related crash failures* in [5].

More precisely, given a predefined contention threshold λ, a λ-constrained crash failure is a crash that occurs while process contention is smaller or equal to λ. Considering read/write shared memory systems and $\lambda = n - 1$, a consensus algorithm is presented in [5, 18] that tolerates one λ-constrained crash (*i.e.*, at most one process may crash, which may occur only when the contention degree is $\leq (n - 1)$), and it is shown that this bound (on the number of failures) is tight.[1] In addition, upper and lower bounds for solving the k-set agreement problem [2] in the presence of multiple contention-related crash failures for $k \geq 2$ are presented in [5, 18].

Motivation: Why λ-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 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], the new type of λ-constrained failures enables the design of algorithms that can tolerate several traditional "any-time" failures plus several additional λ-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 λ-constrained failures, where $t_1 + t_2 > t$.

Adding the ability to tolerate λ-constrained failures to algorithms that are already designed to circumvent various impossibility results, such as the Paxos algorithm [12] and indulgent algorithms in general [7, 8], 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 λ-constrained failures may, in many cases, satisfy its liveness property even before the synchrony assumptions it relies on are satisfied.

[1] The consensus algorithm described in [5, 18] does not use adopt/commit objects as done in the present article. As we will see, this object is crucial for the present paper.

When facing a failure-related impossibility result, such as the impossibility of consensus in the presence of a single faulty process, discussed earlier [6], one is often tempted to use a solution that guarantees no resiliency at all. We point out that there is a middle ground: tolerating λ-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 [6], 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. This article investigates the interplay between asynchrony, process crashes, contention threshold, and the computability power of base objects as measured by their consensus number [9]. 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, the article is made up of three 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 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 $k \geq 1$.
  - The first algorithm tolerates up to k process crashes that may occur before the contention degree bypasses $n-k$.
  - The second algorithm, assumes k divides n, and tolerates up to $2k-1$ process crashes that may occur before the contention degree bypasses $n-2k+1$.
- Finally, Section 5 presents impossibility results that address the limits of the proposed approach.

A short look at consensus solvability The article [19] was one of the very first articles (if not the first one) that considered the case of initial failures for distributed tasks solvability. The reader will find in [4, 13] an approach to task solvability based on the theory of knowledge. When considering the close case of synchronous network-based systems the reader will find in [20] an overview of results for the case of consensus algorithms where links can appear and disappear at every communication step.

The usual notion of fault tolerance states that algorithm is crash-resilient if, in the presence of crash faults, all the non-faulty processes complete their operations and terminate. The article [17] considers a weaker liveness property namely a limited number of participating correct processes are allowed not to terminate in the presence of faults. As stated in [17] "sacrificing liveness for few of the processes allows us to increase the resiliency of the whole system".

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 [14, 16].

A process can crash (a crash is an unexpected premature halt). Given an execution, a process that crashes is said to be *faulty* in that execution, otherwise, it is *correct*. Let us call *contention* the current number of processes that started executing. A λ-constrained crash is a crash that occurs before the contention degree bypasses λ.

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

- Read/write atomic registers (RW).
- Atomic objects with consensus number $k \geq 1$ (these objects, denoted kCONS, will be used in Section 4).
- 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 [11]. 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 $tag \in \{\texttt{commit}, \texttt{adopt}\}$ 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.
- *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., [1,3,15]), it is assumed that all the processes participate in the algorithm. (Equivalently, a process that does not participate is considered as having crashed initially).

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.

3 Base Algorithm ($k = 1$): 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 shared 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)

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 $tag_i = \texttt{adopt}$, 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_ℓ be the last process that writes the value it proposes in $INPUT[1..n]$. It follows from Line 3 that p_ℓ computes the smallest value in the array, and from Line 2 and Line 4 that, no index different from ℓ 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_k 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_ℓ (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_ℓ 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_k crashes, it can do it only when the contention is lower or equal to $(n-1)$. We consider two cases.

- If p_k crashes initially (i.e., before writing the value it proposes in $INPUT[k]$, 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_k crashes after it writes the value it proposes in $INPUT[k]$. 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 x such that, when exiting Line 2, there is some process p_i where $input1_i[x] = \perp$ and all other entries are different than $\perp$ (let call A this set of processes), while other process p_j is such that all entries of $input1_j$ are different than $\perp$ (set B). p_x is the last process to write into $INPUT$ and belongs to B.
    Notice that p_k is not the last process to write into $INPUT$ since it crashes when the contention threshold is lower or equal to $(n-1)$. Thus $x \neq k$ and p_x is correct.
    Processes of set A write x 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. *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_k, this process assigns a value to DEC. If the tag is `adopt`, we claim that the process p_k such that $k = LAST$ 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 $LAST = k$ at Line 6, there is a process p_i that wrote k in $LAST$ at Line 4. This means that p_i found $input1_i[k] = \perp$ 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 k in $LAST$. But, by assumption, no process crashes after the contention threshold $\lambda = n - 1$ has been attained. So, p_k is a correct process. $\square_{Lemma\ 4}$

Theorem 1. *Let $\lambda = n - 1$. Considering an asynchronous RW system, Algorithm* 1 *solves consensus in the presence of at most one λ-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, [5, 18]).

4 General Algorithms ($k \geq 1$): Consensus from Objects whose Consensus Number is k

As we are about to see, these algorithms are reductions to Algorithm 1. At Line 5, it exploits the additional power provided by objects whose consensus number is k. We present below two consensus algorithms:

- Algorithm 2, which tolerates up to k $(n-k)$-constrained failures, and
- Algorithm 3, which tolerates up to $2k-1$ $(n-2k+1)$-constrained failures, assuming k 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 $INPUT[1..\lceil n/k \rceil]$ where each entry $INPUT[x]$ (instead of being a simple read/write register) is a kCONS object, and
- A Boolean array denoted $PARTICIPANT[1..n]$, initialized to $[\texttt{false}, ..., \texttt{false}]$.

Behavior of a process p_i. Algorithm 2 is very close to Algorithm 1.

- The lines N1 and N2 are new. They aim to ensure that no process will block forever despite up to k crashes.
- The lines with the same number have the same meaning in both algorithms.
- Each set of at most k processes p_i, p_j, etc. such that $\lceil i/k \rceil = \lceil j/k \rceil$, defines a cluster of processes that share the same kCONS 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 $INPUT[\lceil i/k \rceil]$ by processes belonging to the same cluster.

4.2 Further explanations

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

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

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

```
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 k entries with false end repeat;
(1-M)  INPUT[⌈i/k⌉] ← kCONS[⌈i/k⌉].propose(in_i);
(2-M)  repeat input1_i ← asynchronous non-atomic reading of INPUT[1..⌈n/k⌉];
              input2_i ← asynchronous non-atomic reading of INPUT[1..⌈n/k⌉]
       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/k⌉)
                                  then DEC ← res_i else wait(DEC ≠ ⊥) end if ;
(7)    return(DEC);
```

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

Question 2. Are the lines N1-N2 needed?

Let us consider Algorithm 2 without the lines N1-N2 and with $k = 2$, 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-M and writes in $INPUT[1]$.
- Process p_3 executes Line 1-M 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-M 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 $n \geq k$ and $\lambda = n - k$. Considering an asynchronous* RW *system enriched with k-*CONS *objects, Algorithm* 2 *solves consensus in the presence of at most k λ-constrained crash failures.*

Proof Let us first observe that, as at most k 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-M to 7. To this end, let us consider the n processes are partitioned in clusters of at most k processes so that p_i belongs to the cluster identified $\lceil i/k \rceil$. 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 blocking 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.[2] 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 3.

$\square_{Theorem\ 2}$

4.4 When k divides n: Tolerating $k-1$ 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 $n \geq k+1$ processes that can tolerate k any-time failures, using registers and wait-free consensus objects for k processes [9]. In such a model, Algorithm *2* has the property captured by the following theorem.

Theorem 3. *If k divides n, Algorithm 2 tolerates $k-1$ any-time failures.*

Proof Using the cluster terminology defined in the previous proof, k divides n, each cluster contains k processes exactly. As at most $(k-1)$ 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. $\square_{Theorem\ 3}$

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

Algorithm 3. Let Algorithm 3 be the same as Algorithm 2, except that line 3,
"**until** $participant_i[1..n]$ contains at most k entries with `false` **end repeat**;"
is replaced with,
"**until** $participant_i[1..n]$ contains at most $2k-1$ entries with `false` **end repeat**;"
Then, the following theorem holds,

Theorem 4. *Assume that k divides n, $n \geq 2k-1$, and $\lambda = n-2k+1$. Considering an asynchronous* RW *system enriched with k-*CONS *objects, Algorithm* 3 *solves consensus in the presence of up to $(2k-1)$ λ-constrained crash failures.*

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

Proof Using the cluster terminology defined in the proof of Algorithm 2, k divides n, implies that each cluster contains k processes exactly. As at most $2k-1$ 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. $\square_{Theorem\ 4}$

5 Impossibility Results

This section presents impossibility results for an asynchronous model which supports atomic read/write registers and kCONS objects, in which λ-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, λ-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 λ-constrained failures.

Theorem 5. *For every $\ell \geq 0$, $k \geq 1$, $n > \ell + k$, and $\lambda = n - \ell$, there is no consensus algorithm for n processes, using atomic* RW *registers and k*CONS *objects, that tolerates $(\ell + k)$ λ-constrained crash failures (even when assuming that there are no any-time crash failures).*

Proof Assume to the contrary that for some $\ell \geq 0$, $k \geq 1$, $n > \ell + k$, and $\lambda = n - \ell$, there is a consensus algorithm, say A, that (1) uses atomic registers and kCONS objects, and (2) tolerates $\ell + k$ λ-constrained crash failures.

Given any execution of A, let us remove any set of ℓ 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 $n' > k$, using atomic registers and k-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 $n' > k$, that (1) uses only atomic registers and k-cons objects, and (2) tolerates k any-time crash failures. But, this is known to be impossible as shown in [9]. A contradiction. $\square_{Theorem\ 5}$

Consensus using atomic registers only. For the special case of consensus using atomic registers only, the equation $n > \ell + k$ becomes $n > \ell + 1$. The following corollary is then an immediate consequence of Theorem 5.

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 $(\ell+1)$ λ-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 λ-constrained and any-time failures.

Theorem 6. *For every $\ell \geq 0$, $k \geq 1$, $n > \ell + k$, $g \geq 0$, and $\lambda = n - \ell$, there is no consensus algorithm for n processes, using atomic* RW *registers and k*CONS *objects, that,tolerates $(\ell + k - g)$ λ-constrained crash failures and g any-time crash failures.*

Proof Follows immediately from Theorem 5 by observing that any-time crash failures belong to a more severe type of a failure than λ-constrained crash failures when $\lambda < n$, and is the same as a λ-constrained crash failure when $\lambda = n$. $\square_{Theorem\ 6}$

6 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 in an n-process asynchronous shared memory system enriched with objects the consensus number of which is equal to k. It has been shown that for $n \geq k$, consensus can be solved in such a context in the presence of up to k process crashes if these crashes occur before process contention has attained the value $\lambda = n - k$. Furthermore, for the case where k divides n, it has been shown that consensus can be solved in such a context in the presence of up to $2k - 1$ process crashes if these crashes occur before process contention bypasses the threshold $\lambda = 2n - k + 1$.

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 k. 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

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. Chaudhuri S., More choices allow more faults: set consensus problems in totally asynchronous systems. *Information and Computation*, 105(1):132-158 (1993)
3. Cachin Ch., Guerraoui R., and Rodrigues L., *Reliable and secure distributed programming*, Springer, 367 pages, ISBN 978-3-642-15259-7 (2011)
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. Fischer M.J., Lynch N.A., and Paterson M.S., Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374-382 (1985)
7. Guerraoui R., Indulgent algorithms. *Proc. 19th Annual ACM Symposium on Principles of Distributed Computing (PODC'00)*, ACM Press, pp. 289-297 (2000)
8. Guerraoui R. and Raynal M., The information structure of indulgent consensus. *IEEE Transactions on Computers*, 53(4):453-466 (2004)
9. Herlihy M. P., Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124-149 (1991)
10. 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)
11. 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)
12. Lamport L., The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169 (1998)
13. Moses Y. and Rajsbaum S., A layered analysis of consensus. *SIAM Journal of Computing*, 31(4):989-1021 (2002)
14. Raynal M., *Concurrent programming: algorithms, principles and foundations*. Springer, 515 pages, ISBN 978-3-642-32026-2 (2013)
15. Raynal M., *Fault-tolerant message-passing distributed systems: an algorithmic approach*. Springer, 492 pages, ISBN 978-3-319-94140-0 (2018)
16. Taubenfeld G., *Synchronization algorithms and concurrent programming*. Pearson Education/Prentice Hall, 423 pages, ISBN 0-131-97259-6 (2006)
17. 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)
18. 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)
19. Taubenfeld G., Katz S., and Moran S., Initial failures in distributed computations. *International Journal of Parallel Programming*, 18(4):255–276 (1989)
20. Winkler K. and Schmid S., An Overview of recent results for consensus in directed dynamic networks. *Bulletin of the European Association for Theoretical Computer Science*, Vol. 128, 30 pages (2019)