Contention-Related Crash Failures: Definitions, Agreement Algorithms, and Impossibility Results

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

*LIMOS, Université Clermont Auvergne CNRS UMR 6158, Aubière, France
†IRISA, Université de Rennes, 35042 Rennes, France
‡Department of Computing, Polytechnic University, Hong Kong
§Reichman University, Herzliya 46150, Israel

Abstract

This article explores an interplay between process crash failures and concurrency. Namely, it aims at answering the question, "Is it possible to cope with more crash failures when some number of crashes occur before some predefined contention point happened?". These crashes are named λ-*constrained crashes*, where λ is the predefined contention point (known by the processes). Hence, this article considers two types of process crashes: λ-constrained crashes and classical crashes (which can occur at any time and are consequently called *any-time* crashes).

Considering a system made up of n asynchronous processes communicating through atomic read/write registers, the article focuses on the design of two agreement-related algorithms. Assuming $\lambda = n - 1$ and no any-time failure, the first algorithm solves the consensus problem in the presence of one λ-constrained crash failure, thereby circumventing the well-known FLP impossibility result. The second algorithm considers k-set agreement for $k \geq 2$. It is a k-set agreement algorithm such that $\lambda = n - \ell$ and $\ell \geq k = m + f$ that works in the presence of up to $(2m + \ell - k)$ λ-constrained crashes and $(f - 1)$ any-time crashes, i.e., up to $t = (2m + \ell - k) + (f - 1)$ process crashes. It follows that considering the timing of failures with respect to a predefined contention point enlarges the space of executions in which k-set agreement can be solved despite the combined effect of asynchrony, concurrency, and process crashes. The paper also presents agreement-related impossibility results for consensus and k-set agreement in the context of λ-constrained process crashes (with or without any-time crashes).

Keywords: Agreement algorithm, Asynchronous system, Concurrency, Contention, Process crash.

1 Introduction

1.1 Context and Related Works

Distributed computing is about cooperation between predefined computing entities (processes). Each entity has its own local input and must compute a local output that depends on the set of inputs [28]. It is well-known that in the presence of asynchrony and failures, some distributed computing problems become impossible to solve, be the communication medium a message-passing network [13] or a shared read/write memory [23].

Several approaches have been proposed to counteract such impossibilities. We consider here only a few of them. One enriches the system with oracles that provide the processes with information on failures [6] and, given a specific problem P, finds the weakest information on failures that allows P to be solved [5]. Another approach consists in weakening the progress condition associated with each cooperation operation, namely, the termination of such an operation is required only in specific circumstances (e.g., when there is no concurrency [19]). Another approach is based on the notions of adversary disagreement power [9]. It consists of defining subsets of processes such that the considered problem must be solved despite asynchrony and process crashes when the set of processes that do not crash in execution is precisely one of the predefined subsets.[1] This approach has been made constructive in [16]. The interested reader will find more developments on this topic in the following articles [8, 14–16, 29]. Yet another approach described in [14] consists in adapting the concurrency level to the level of synchrony. In [32], the traditional notion of fault tolerance is generalized by allowing a limited number of participating correct processes not to terminate in the presence of faults. Every process that does terminate is required to return a correct result. Thus, the new definition guarantees safety but may sacrifice liveness (termination), for a limited number of processes, in the presence of faults. Initial failures were investigated in [34].

The present work introduces a new approach that allows impossibility results related to consensus and k-set agreement to be circumvented. This approach, based on a predefined contention point, considers that some number of process crashes occur at the latest when a predefined contention point occurs.[2] Let us notice that, while it considers a predefined contention threshold, the proposed model differs from the k-concurrency model, which bounds the number of processes that can be concurrently active.

1.2 Processes, Communication, Participation, and Failure model

The system is composed of n asynchronous sequential processes, denoted p_1, ..., p_n, which communicate by reading and writing atomic registers. Without loss of generality, it is assumed that the identity of process p_i is its index i. The model parameter t denotes the maximal number of processes that may crash during a run. A process crash is a premature definitive halting. A process that crashes is called *faulty*; otherwise, it is *correct*. All correct processes are assumed to participate, i.e., execute their local algorithm. (Let us notice that this assumption is a classical –very often left implicit– assumption encountered in message-passing distributed algorithms [28].)

Let us call *contention* the current number of processes that started executing. A process starts executing when it writes an atomic register for the first time (i.e., from this write it starts cooperating with the other processes). The model parameter λ denotes a predefined contention threshold. So, execution can be divided into two parts: a prefix in which the contention is $\leq \lambda$ and a suffix in which contention is $> \lambda$. Hence, we consider a failure model in which there are two types of crashes: the ones that can occur only when contention is $\leq \lambda$ that we call "λ-constrained",[3] and the ones that can occur at "any time." If we suppress λ-constrained crashes, the computing model boils down to the classical crash-prone model. It is important to notice that any-time failures can occur before or after the predefined contention threshold has been attained. So, it is as if there are two cooperating adversaries:

- A constrained adversary that can crash x processes, $0 \leq x \leq \alpha$, while the contention threshold is $\leq \lambda$ (α is a predefined value depending on both the contention threshold and the agreement problem),
- and an unconstrained adversary that can crash $(t - x)$ processes, and it can crash them at any time.

1.3 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

[1]The disagreement power of a set of processes is the biggest integer ℓ for which the adversary can prevent processes from agreeing on ℓ values when using registers only.

[2]Preliminary versions of these results appeared in [10, 33].

[3]λ-constrained crashes were introduced in [33] under the name "weak failures".

systems. Providing necessary and sufficient conditions for agreement problems helps us determine and identify under which type of (weak) process failures the fundamental consensus and set-agreement problems are solvable.

As discussed in [10], the new type of λ-constrained failures enables us to design 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 [22] and indulgent algorithms in general [17, 18], 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.

When facing a failure-related impossibility result, such as the impossibility of consensus in the presence of a single faulty process [13], 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 [13], which solves consensus despite asynchrony and up to $t < n/2$ processes crashes if these crashes occur initially (hence no participating process crashes).

Let us observe that the λ-constrained failure model establishes a link between contention and failures, enabling us to better understand various known impossibility results, like the impossibility result for consensus [13] and its generalizations for k-set agreement [4, 20, 30].[4]

1.4 Content of the Paper

The paper is composed of three parts. Considering $\lambda = n - 1$, the first part (Section 2) presents an algorithm that solves the consensus problem in the presence of one λ-constrained crash failure and no any-time crash failure, thereby circumventing the well-known FLP impossibility result [13].

The second part (Section 3) presents a generic k-set agreement algorithm for $k \geq 2$, which copes with the presence of both λ-constrained and any-time crash failures. Its genericity dimension lies in the value of $\lambda = n - \ell$ where $\ell \geq k = m + f$, that tolerates up to $2m + \ell - k$ λ-constrained failures and $(f-1)$ any-time failures, i.e., up to $t = (2m + \ell - k) + (f - 1)$ process crashes. When instantiated with $\ell = k = m + f$, we have $\lambda = n - k$, the algorithm solves $(m + f)$-set agreement while tolerating $t = 2m + f - 1$ crash failures, up to $2m$ being $(n-k)$-constrained failures, and $(f-1)$ being any-time failures. Let us observe that, in this case, the "weight" associated with a λ-constrained failure appears as being twice the "weight" associated with an any-time failure. This seems to be the price to pay to circumvent k-set agreement impossibility result with the help of the added information provided by the timing of process crashes with respect to process contention.

Finally, the last part of the paper (Section 4) presents agreement-related impossibility results in the λ-constrained failures model. These impossibility results involve the number of processes n, the predefined contention threshold λ, and the maximal number of values k that can be decided.

[4]The λ-constrained failure model and the classical round-based model share a same feature, namely participation of each process is required in both. They differ in the way the processes synchronize. The λ-constrained failure model is based on a predefined contention degree λ which allows the processes to synchronize only once (namely, when the λ contention degree is attained). In the round-based model the processes must synchronize at every round.

2 Consensus

As previously announced, this section presents an algorithm that solves consensus in the presence of a single λ-constrained crash failure where $\lambda = n - 1$. This bound on the number of $(n-1)$-constrained crash failures is tight. In Section 4, it is shown that there does not exist a consensus algorithm for n processes, using read/write registers, that can tolerate *two* $(n-1)$-constrained crash failures, for any $n > 2$.

2.1 Consensus: Definition

The consensus object was defined in [24]. Such an object provides the processes with a single operation, denoted $\mathsf{propose}()$, that a process invokes once (one-shot object). 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.

It has been shown that consensus cannot be solved in an asynchronous crash-prone system, be the communication be through atomic read/write registers [23], or message-passing [13].

2.2 Consensus in the Presence of one (*n*–1)-Constrained Crash Failure: Algorithm

Without loss of generality, it is assumed that the proposed values are non-negative integers.

Shared base objects The processes communicate through atomic read/write registers. *Atomic* means that the invocations of the read and write operations appear as if they have been issued sequentially. Moreover, this total order respects real-time order, and a read returns the value written by the closest preceding write (or the initial value if there is no such write) [21].

- $STATE[1..n]$ is an array of atomic single-writer multi-reader registers. For any i, $STATE[1..n]$ contains the current state of process p_i, namely a value in the set $\{0, 1, 2, 3\}$ (initialized to 0). $STATE[i]$ denotes the progress of p_i.
- $INPUT[1..n]$ is an array of atomic single-writer multi-reader. $INPUT[1..n]$ 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 $\perp$.

Local objects Each process p_i manages three local variables: $counter_i$, max_i, and $round_i$. Their initial values are irrelevant.

Overview of the algorithm The algorithm is based on three asynchronous rounds, the same code being executed at each round, with the third round having additional post-fixed code.

The local counter $counter_i$ is used by p_i to count the number of processes that already entered the round p_i is entering, and max_i contains the greatest value proposed by the processes that started the algorithm.

Let us observe that, as by assumption, the predefined contention threshold is $\lambda = n - 1$, and at most one process may crash, it follows that when a process p_i sees that the $(n-1)$ other processes have attained the round $round_i$, it knows that no process will crash.

```
operation propose($in_i$) is
(1)  $INPUT[i] \leftarrow in_i$;
(2)  for $round_i = 1, 2, 3$ do
(3)    $counter_i \leftarrow 0$; $max_i \leftarrow 0$;
(4)    for $j$ from $1$ to $n$ do              % up to Line 8: $STATE[i] = round_i - 1$
(5)      if $STATE[j] \geq round_i$ then $counter_i \leftarrow counter_i + 1$;
                                          $max_i \leftarrow$ max($max_i, INPUT[j]$) end if
(6)    end for;
(7)    if ($counter_i = n - 1$) then $DEC \leftarrow max_i$; return($max_i$) end if;
(8)    $STATE[i] \leftarrow round_i$;
(9)    repeat $counter_i \leftarrow 0$;
(10)          for$j$ from $1$ to $n$ do
(11)            if $STATE[j] \geq round_i$ then $counter_i \leftarrow counter_i + 1$ end if
(12)          end for;
(13)   until $counter_i \geq n - 1 \vee DEC \neq \perp$ end repeat;
(14)   if ($DEC \neq \perp$) then return($DEC$) end if
(15) end for;    % $p_i$ has terminated its three rounds

(16) if ($counter_i = n - 1$) then
(17)   $counter_i \leftarrow 0$; $max_i \leftarrow 0$;
(18)   for $j$ from $1$ to $n$ do
(19)    if ($STATE[j] \geq 2$) then
           $counter_i \leftarrow counter_i + 1$; $max_i \leftarrow$ max($max_i, INPUT[j]$) end if
(20)   end for;
(21)   if ($counter_i = n - 1$) then $DEC \leftarrow max_i$; return($max_i$) end if
(22) end if;

(23) for $j$ from $1$ to $n$ do
(24)   wait($STATE[j] = 3 \vee DEC \neq \perp$);
(25)   if ($DEC \neq \perp$) then return($DEC$) end if;
(26)   $max_i \leftarrow$ max($max_i, INPUT[j]$)
(27) end for;
(28) $DEC \leftarrow max_i$; return($max_i$).
```

Algorithm 1: Consensus with $\lambda = n - 1$, one λ-constrained failure and no any-time failure

Behavior of a process p_i A process decides when it executes the return() statement, which occurs at Line 7, 14, 21, 25, or 28.

Intuitively, the aim is to direct the processes to decide the highest proposed value they see. To this end, each process first executes a sequence of rounds from 1 to 3. Initially process p_i is in state $STATE[i] = 0$. It will change its state according to its progress in the round numbers (Line 8). We say "process p_i is in Round r" if $round_i = r$.

When it invokes propose(in_i), process p_i first deposits in_i in $INPUT[i]$ to make it known by all the processes, and starts executing three rounds (Lines 2-15). The current round number of process p_i is saved in the atomic register $STATE[i]$.

At each round, p_i first asynchronously reads $STATE[1]$, ..., $STATE[n]$, and counts how many other processes are at the same or a higher round than its round $round_i$, and computes the highest value seen by these processes (Line 5). If it counts $(n - 1)$ such processes, p_i is the last one in $round_i$ and max_i contains the highest value proposed by the other $(n - 1)$ processes. So p_i writes this value in DEC and decides it (Line 7).

If p_i does not decide, it updates $STATE[i]$ (Line 8), and loops (Lines 9-13) until it sees that a value has been decided (predicate $DEC \neq \perp$ in Line 13), or it sees $(n-1)$ processes at a round equal or greater than its current round (predicate $counter_i \geq n-1$ in Line 13). In the former case, p_i decides the value DEC (Line 14). In the latter case, it proceeds to the next round if $round_i \leq 2$.

If $round_i = 3$, p_i continues executing the Lines 16-28. So, p_i checks if there exists a process, say p_j, that had not written the value 2 into $STATE[j]$ (Lines 16-20). In case of a positive answer, it concludes that process p_j will never be able to reach round three, and thus, p_j will never set $STATE[j]$ to 3. This is so because p_j will notice that $(n-1)$ other processes have already set their $STATE[]$ registers to 3, and will decide (Line 7) before increasing their $STATE[]$ register (Line 8). Thus, process p_i sets the atomic register DEC to the maximum input value among all the processes, excluding process p_j, decides on that maximum value and terminates (Line 21).

Otherwise, if all the n processes have written the round number 2 into their $STATE[]$ registers, process p_i concludes that all the n processes are still active and –due to the λ-constrained failure assumption– are guaranteed not to fail. So, p_i waits until either a decision is made, or until all the processes complete Round 3, whatever comes first (Lines 23-27). In the former case, p_i adopts the value of DEC (Line 24). In the latter case it decides on the maximum input value among the input values of all the n processes (Line 28).

2.3 Proof of Algorithm 1 (Consensus)

Reminder: a process p_i is in Round r if $round_i = r$.

Lemma 1 *For every $i \in \{1, ..., n\}$, when process p_i sets $STATE[i]$ to 2 (Line 8), either the contention point is already n or there is a $j \neq i$ such that $STATE[j]$ will always be 0.*

Proof If the contention point is not n when p_i sets $STATE[i]$ to 2 (Line 8), it follows that (1) by definition, some process, say p_j, has not taken any steps yet, and (2) except for process p_j, all the other $(n-1)$ processes p_k have already incremented their register $STATE[k]$. If p_j is a correct process, it will eventually reach Line 7 at which point $counter_j$ will become equal to $(n-1)$. Thus, p_j will decide and terminate without ever increasing the atomic register $STATE[i]$. $\Box_{Lemma\ 1}$

Lemma 2 *If at some point in time, for every $i \in \{1, ..., n\}$ $STATE[i] \geq 2$, then the n processes are active, no process has failed before that point, and no process will fail after that point.*

Proof Assume that for every $i \in \{1, ..., n\}$, $STATE[i] \geq 2$. It follows from this assumption and Lemma 1 that, for every $i \in \{1, ..., n\}$, when p_i has set $STATE[i]$ to 2 (Line 8), the contention point was already n. Thus, process p_i will never fail since it is assumed that no process fails once the contention point is n. $\Box_{Lemma\ 2}$

Lemma 3 *For every $i \in \{1, ..., n\}$:*

1. *If a process p_i writes into DEC in Line 7, no other process writes into DEC in Line 7.*
2. *If a process p_i writes into DEC in Line 7, no other process writes into DEC in Line 28.*
3. *If a process p_i writes into DEC in Line 21, no other process writes into DEC in Line 28.*

Proof Suppose process p_i writes into DEC in Line 7 in Round $r \in \{1, 2, 3\}$. This means that each other process p_k has already written r into its state register $STATE[k]$, and hence has not written into DEC in Line 7 in Round r or in a previous round. After p_i writes into DEC, it immediately terminates.

Thus, p_i will never write a value $r' \geq r$ into its state register $STATE[i]$. Thus, for every other process p_k, after executing the for loop in Lines 4-6, the value of $counter_k$ will be at most $(n-2)$, and the test in Line 7 will fail. Also, since $STATE[i]$ will never equal 3, no process will ever reach Line 28.

Suppose now process p_i writes into DEC in Line 21. This means that there exists a process, say process p_j, that has not written the value 2 into its state register $STATE[j]$, at the time when p_i checked $STATE[j]$ in Line 19. Although process p_j may still set $STATE[j]$ to 2 at a later time, it will never be able to set $STATE[j]$ to 3 at a later time because, in Round 3, the counter of p_j will reach $(n-1)$ when p_j executes the for loop in Lines 4-6, and if continues it will terminate at Line 7. For that reason, when some other process executes Line 24 (predicate wait($STATE[j] = 3 \vee DEC \neq \bot$)), the waiting may terminate only because $DEC \neq \bot$. Thus, no process will ever reach and execute Line 28. $\Box_{Lemma\ 3}$

Lemma 4 *For every two processes p_i and p_j:*

1. *if p_i writes v into DEC in Line 7, and p_j writes v' into DEC in Line 21, then $v = v'$.*
2. *if p_i writes v into into DEC in Line 21, and p_j writes v' into DEC also in Line 21, then $v = v'$.*
3. *if p_i writes v into DEC in Line 28, and p_j writes v' into DEC also in Line 28, then $v = v'$.*

Proof

1. Assume that p_i writes v into in Line 7, and p_j writes v' into DEC in Line 21. When p_i terminates, the value of its state register $STATE[i]$ is either 0,1 or 2. In the first two cases (0 and 1), the value of max_j that p_j computes in Line 19 does not depend on the input value of p_i, and hence $v = v'$.

   Let us consider the case that when p_i decides at Line 7, the value of its state register $STATE[i]$ is 2. Thus, when p_i terminates, the values of each other state register $STATE[k]$ must be 3. When p_j starts executing the for-loop in Line 18, the value of the state registers of $(n-1)$ processes must be 3. Thus, p_i and p_j set their max_i and max_j local variables max_i and max_j (in Lines 6 and 19, respectively) to the same value since they both choose the maximum input value from the set of $(n-1)$ input values which does not include the input value of process p_i. Thus, $v = v'$.
2. Assume that p_i writes v into DEC in Line 21, and p_j writes v' into DEC also in Line 21. When p_i started executing the for-loop in Line 18, the value of the state register $STATE[]$ of *exactly* one process, say process p_k, was less than 2. Similarly, when p_j started executing the for-loop in Line 18, the value of the state register $STATE[]$ of *exactly* one process, say process k', was less than 2. Since the value of a state register never decreases, it follows that $k = k'$. Thus, p_i and p_j set their local variables max_i and max_j (in Line 19) to the same value, since they choose the maximum input value from the same set of $(n-1)$ input values. Thus, $v = v'$.
3. Assume that p_i writes v into DEC in Line 28, and p_j writes v' into the DEC in Line 28. Both p_i and p_j set their local variables max_i and max_j in Line 26 to the same value, since they choose the maximum input value from the set of the n input values. Thus, $v = v'$.

$\Box_{Lemma\ 4}$

Theorem 1 (agreement & validity) *All the participating processes decide on the same value, and this decision value is the input of a participating process.*

Proof It follows from Lemma 3 and Lemma 4, that, whenever two processes write into the decision register DEC, they write the same value. Also, whenever a process writes into DEC, the written value is the input of a participating process. Each correct process decides only on a value written into DEC. $\Box_{Theorem\ 1}$

Theorem 2 (termination) *In the presence of at most a single $(n-1)$-constrained crash failure, every correct process eventually decides.*

Proof There are exactly two places in the algorithm where a process may need to wait for some other process to take a step: (1) in the repeat-until loop in Lines 9-13, and (2) in the wait statement in Line 24. In both places, whenever a process needs to wait, it continuously examines the value of the decision register DEC, and if it finds out that $DEC \neq \perp$, it decides on the value written in DEC and terminates. Thus, we can conclude that: if some process decides, then every correct process eventually decides.

So, let us assume, by contradiction, that no correct process ever decides. There are at least $n - 1$ correct processes. At least $(n-1)$ correct process will execute the for-loop in Lines 2-15 with $round =$ 1. They all will eventually execute the assignment in Line 8; setting their state registers $STATE[]$ to 1. Thus, each correct process with $round = 1$, will eventually exit the repeat-loop in Lines 9-13, and will move to Round 2. By a similar argument, each correct process will eventually complete Rounds 2 and 3 (i.e., will complete the for-loop in Lines 2-15).

A process reaches the for loop in Lines 23-27, only if its local variable $counter_i$ equals n, which implies that for every $i \in \{1, ..., n\}$ $STATE[i] \geq 2$. Thus, by Lemma 2, if some process executes the for-loop in Lines 23-27 all the n processes are active and will never fail. Since, by contradiction, no process terminates, all the n processes must eventually get stuck in the wait statement on Line 24. However, this is not possible since the value of the state register $STATE[k]$ of each process p_k which reaches Line 24 must be 3. Thus, all the waiting processes in Line 24 will be able to proceed beyond the wait statement and decide. A contradiction. $\Box_{Theorem\ 2}$

3 k-Set Agreement ($k \geq 2$)

This section presents a k-set agreement algorithm that circumvents the known impossibility result for solving k-set agreement in t-resilient crash-prone asynchronous read/write systems where $t \geq k$ [4, 20, 30].

3.1 k-Set Agreement: Definition

A k-set agreement (k-SA) object is a one-shot object introduced by S. Chaudhuri [7] 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. It is shown in [4, 20, 30] that it is impossible to solve k-set agreement in asynchronous read/write systems where up to $t \geq k$ processes may crash at any time.

3.2 Basic Model and High-Level Objects

Basic model As indicated in the introduction, the basic model is the classical crash-prone model of n processes communicating through atomic multi-writer multi-reader registers.

To make the presentation of the proposed algorithm easier, the basic read/write system is enriched with two types of objects, namely ℓ-mutual exclusion and snapshot. Both can be built on top of a crash-prone asynchronous read/write system.

Deadlock-free ℓ-mutual exclusion Such an object was first defined and solved in [11, 12]. Several papers have proposed ℓ-exclusion algorithms for solving the problem using atomic read/write registers satisfying various progress properties (see, for example, [2, 25, 26]). This object provides the processes with the operations denoted $\mathsf{acquire}()$ and $\mathsf{release}()$. It allows up to ℓ processes to simultaneously execute their critical section. It is defined by the following properties.

- *Mutual exclusion.* No more than ℓ processes can simultaneously be in their critical section.
- *Deadlock-freedom.* If less than ℓ processes crash and processes are invoking the operation $\mathsf{acquire}()$, at least one of them will terminate its invocation.

It is shown in [2, 11, 31] that ℓ-mutual exclusion can be built on top of an asynchronous crash-prone read/write system. In the *one-shot* version, a process invokes $\mathsf{acquire}()$ and $\mathsf{release}()$ at most once.

Atomic snapshot This object was introduced in [1, 3]. It provides the processes with two atomic operations denoted $\mathsf{write}()$ and $\mathsf{snapshot}()$. Such an object can be seen as an array of single-writer multi-reader atomic register $SN[1..n]$ such that:
(a) when p_i invokes $\mathsf{write}(v)$, it writes v into $SN[i]$; and
(b) when p_i invokes $\mathsf{snapshot}()$, it obtains the value of the array $SN[1..n]$ as if it simultaneously reads and instantaneously all its entries.
Put another way, the operations $\mathsf{write}()$ and $\mathsf{snapshot}()$ appear to the processes as if they were totally ordered. Snapshot objects can be implemented on top of asynchronous crash-prone read/write systems [1, 3, 27].

3.3 k-Set Agreement with λ-Constrained and Any-time Crash failures: Algorithm

Shared objects The processes communicate through the following objects.

- $PART[1..n]$: snapshot object, initialized to $[\mathtt{down}, \cdots, \mathtt{down}]$, used to indicate participation.
- DEC: atomic register initialized to $\perp$ (a value which cannot be proposed). It will contain values (one at a time) that can be decided.
- $MUTEX[1]$: one-shot deadlock-free f-mutex object.
- $MUTEX[2]$: one-shot deadlock-free m-mutex object. (As we will see, no process will ever try to access a 0-mutex $MUTEX[2]$ object.)

Local variables Each process p_i manages the following local variables: $part_i$ is used to locally store a copy of the snapshot object $PART$; $count_i$ is a local counter; and $group_i$ a binary variable whose value belongs to $\{1, 2\}$.

The behavior of a process p_i Algorithm 2 describes the behavior of a process p_i. When it invokes $\mathsf{propose}(in_i)$ (where in_i is the value it proposes), p_i first indicates it is participating (Line 1). Then it invokes the snapshot object until at least $n-t$ (i.e., $n-(2m+\ell-k)-(f-1)$) processes are participating (Lines 2-4). When this occurs, p_i enters Group 1 or Group 2 according to the value of its counter $count_i$ (Line 5), and launches in parallel two threads $T1$ and $T2$ (Line 6).

In the thread $T1$, p_i loop forever until DEC contains a proposed value. When this happens, p_i decides it (Line 7). The execution of $\mathsf{return}()$ at Line 7 or 12 terminates the invocation of $\mathsf{propose}()$.

The thread $T2$ is the core of the algorithm. Process p_i tries to enter the critical section controlled by either the f-mutex or the m-mutex object $MUTEX[group_i]$ (Line 9). If it succeeds and DEC still has its initial default value, p_i assigns it the value in_i it proposed (Line 10). Finally, p_i releases the critical section (Line 11), and decides (Line 12). Let us remind that, as far as $MUTEX[1]$ (respectively, $MUTEX[2]$) is concerned, up to f (respectively, m) processes can simultaneously execute Line 10. This intuitively explains why at most $k = m + f$ different values can be decided.

```
**operation** propose($in_i$) **is**
(1)  $PART$.write(up);
(2)  **repeat** $part_i \leftarrow PART$.snapshot();
(3)          $count_i \leftarrow |\{x$ such that $part_i[x] = \text{up}\}|$;
(4)  **until** $count_i \geq n - t$ **end repeat**;
(5)  **if** $count_i \leq \lambda$ **then** $group_i \leftarrow 2$ **else** $group_i \leftarrow 1$ **end if**;
(6)  launch in parallel the threads $T1$ and $T2$.
% Both threads and the operation terminate when $p_i$ invokes return() (Line 7 or 12).

**thread** $T1$ **is**
(7)  **loop forever if** $DEC \neq \perp$ **then** return($DEC$) **end if end loop**.

**thread** $T2$ **is**
(8)  **if** $group_i = 1 \vee m > 0$ **then**
(9)          $MUTEX[group_i]$.acquire();
(10)             **if** $DEC = \perp$ **then** $DEC \leftarrow in_i$ **end if**;
(11)         $MUTEX[group_i]$.release();
(12)         return($DEC$)
(13) **end if**.
```

Algorithm 2: k-Set agreement (where $k = m + f$, $\lambda = n - \ell$ and $\ell \geq k$) tolerating up to $2m + \ell - k$ λ-constrained failures and up to $f - 1$ any-time failures

total # of failures tolerated	$t = 2m + \ell - k + f - 1 = m + \ell - 1$
"λ-constrained" crash failures	$2m + \ell - k = m + \ell - f$
"any-time" crash failures	$f - 1$

Table 1: Tolerated crash failures of Algorithm 2 ($k = m + f$)

Properties summary The fault-tolerance properties of Algorithm 2 are summarized in Table 1. The parameters m and f allow the user to tune the type of crash failures that are dominant in the considered application context. At one extreme, the pair of values $\langle m, f \rangle = \langle 0, k \rangle$ maximizes the number of any time failures ($k - 1$). At the other extreme, the pair $\langle m, f \rangle = \langle k - 1, 1 \rangle$ maximizes the number of λ-constrained failures (up to $k + \ell - 2$ λ-constrained failures and no any-time failure). This is summarized in Table 2.

total # of failures $t = 2m + \ell - k + f - 1$	$m = 0$ $f = k$	$m = \lceil k/2 \rceil$ $f = \lfloor k/2 \rfloor$	$m = k - 1$ $f = 1$
$2m + \ell - k$ "λ-constrained" crash failures	$\ell - k$	$2\lceil k/2 \rceil + \ell - k$	$\ell + k - 2$
$f - 1$ "any-time" crash failures	$k - 1$	$\lfloor k/2 \rfloor - 1$	0

Table 2: k-Set agreement: trade-offs "λ-constrained/any-time" crash failures when $\lambda = n - \ell$

Two instances of Algorithm 2 Among all its possible instances of the proposed generic algorithm, let us consider two specific cases for which algorithms have already been proposed.

- Case $m = 0$ and $\ell = k$. In this case, there are no λ-constrained failures. We then have $k = m + f = f$, and $(f - 1)$ any-time failures. This is the best that can be done in the classical (any-time) failure model [4,20,30].
- Case $f = 1$. In this case there is no any-time failures. We have then $k = m + f = m + 1$ and, as $m = k - 1$, at most $2m + \ell - k = 2(k - 1) + \ell - k = \ell + k - 2$ λ-constrained failures are tolerated. This is what is solved by the k-set agreement algorithm proposed in [33].

3.4 Proof of Algorithm 2 (k-Set, $k \geq 2$)

In the following, it is assumed that $\lambda = n - \ell$ and $\ell \geq k$.

Lemma 5 *At most $n - \ell$ processes have a counter less or equal to $n - \ell$ when leaving the repeat loop (Lines 2-4).*

Proof Assume by contradiction that more than $n - \ell$ processes have their counter less or equal to $n - \ell$ when leaving the repeat loop (2-4). P being this set of processes, we have $|P| \geq n - \ell + 1$. Moreover, let p_i be the last process of P that invokes $PART.\mathsf{snapshot}()$ (on Line 1[5]). It follows from the atomicity of the $\mathsf{write}()$ and $\mathsf{snapshot}()$ operations on the object $PART$ that $count_i \geq |P| \geq n - \ell + 1$, a contradiction. $\square_{Lemma\ 5}$

Lemma 6 *In the presence of at most $t = 2m + \ell - k + f - 1$ crash failures, $2m + \ell - k$ of them being $(n - \ell)$-constrained, if processes participate in $MUTEX[1]$, at most $f - 1$ of them can fail.*

Proof If a process p_i participates in $MUTEX[1]$, it follows from Line 5 that $count_i > n - \ell$ when it exited the repeat loop (Lines 2-4). Thus, the contention was at least $n - \ell + 1$ when p_i exited the loop and, due to the definition of "$(n - \ell)$-constrained crash failures", there is no more such failures. As $t = 2m + \ell - k + f - 1$, it follows that, if processes participate in $MUTEX[1]$, at most $f - 1$ of them can fail. $\square_{Lemma\ 6}$

Theorem 3 (Termination) *In the presence of at most $t = 2m + \ell - k + f - 1$ crash failures, $2m + \ell - k$ of them being $(n - \ell)$-constrained, every correct process eventually terminates.*

Proof Since there are at most t processes that may fail and participation is required, at least $n - t$ processes eventually set their participating flag to up in the snapshot object $PART$ (Line 1). Thus, no correct process remains stuck forever in the repeat loop (Lines 2-4).

First, assume $m = 0$. By Lemma 5, at most $n - \ell$ processes have a counter less or equal to $n - \ell$ when they exit the repeat loop (Lines 2-4). Thus, at most $n - \ell$ processes belong to Group 2. As $m = 0$, there are at least $n - t = n - (\ell - k + f - 1)$ correct processes. Since $k = m + f = f$, $n - (\ell - k + f - 1) = n - (\ell - 1) = n - \ell + 1 > n - \ell$. So, among the processes participating in $MUTEX[1]$, at least one is correct and at most $f - 1$ crash before returning from $MUTEX[1].\mathsf{release}()$ (Line 11). Due to the deadlock-freedom property of the one-shot f-mutex object $MUTEX[1]$, at least one correct process eventually enters its critical section and, if DEC has not already been written, writes its input into DEC. It then follows from task $T1$ that, if it does not terminate at Line 12, every other correct process will decide and terminate.

Now, assume $m > 0$. There are two cases.

[5]As the snapshot object is atomic, the notion of "last process of P that ..." is well-defined.

- If at least $y \geq f$ processes participate in $MUTEX[1]$, it follows from Lemma 6 that at most $f-1$ of them crash before returning from $MUTEX[1].\mathsf{release}()$ (Line 11), and consequently, all other processes participating in $MUTEX[1]$ are correct. As $y > f-1$ and $f > 0$, there is at least one such correct process, say p_x. Due to the deadlock-freedom property of the one-shot f-mutex object $MUTEX[1]$, p_x eventually enters its critical section and, if DEC has not already been written, writes its input into DEC.
- Otherwise, less than f processes participate in $MUTEX[1]$. There are two sub-cases.
  - If a correct process p_i participates in $MUTEX[1]$, it follows from this sub-case assumption and the deadlock-freedom property of the one-shot f-mutex object $MUTEX[1]$, that p_i eventually enters its critical section and, if $DEC = \perp$, writes its input in_i into this atomic register.
  - Otherwise, no correct process participates in $MUTEX[1]$. By Lemma 5, at most $n-\ell$ processes have a counter less or equal to $n-\ell$ when they exit the repeat loop (Lines 2-4). So at most $n-\ell$ processes participate in $MUTEX[2]$. Since no correct process participates in $MUTEX[1]$, all correct processes (they are at least $n-t$) participate in $MUTEX[2]$. Thus, at most $(n-\ell)-(n-t) = t-\ell = 2m+\ell-k+f-1-\ell = 2m-k+f-1 = 2m-(m+f)+f-1 = m-1$ processes that participate in $MUTEX[2]$ fail. Hence, due to the deadlock-freedom property of the one-shot m-mutex object $MUTEX[2]$, at least one correct process enters its critical section and, if $DEC = \perp$, writes its input into DEC.

In both cases, every other correct process will decide and terminate.

$\Box_{Theorem\ 3}$

Theorem 4 (Agreement and validity) *At most k different values are decided, and each of them is the input of some process.*

Proof If a process decides (Line 7 or Line 12), it decides on the current value of DEC, which –due to the predicates of Line 7 or Line 10– has previously been set –at Line 10– to the value proposed by a process. Due to the predicate and the assignment of DEC at Line 10, and the fact that $MUTEX[1]$ is a f-mutex object, it follows that at most f processes assign a value to DEC in the critical section controlled by $MUTEX[1]$. Due to a similar argument, at most m processes assign a value to DEC in the critical section controlled by $MUTEX[2]$. Thus, at most $m+f=k$ different values can be written into DEC, and each of them is a proposed value. $\Box_{Theorem\ 4}$

4 Impossibility Results

This section presents impossibility results in the asynchronous read/write model with λ-constrained and any-time crash failures. 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).

Theorem 5 (k-set agreement with λ-constrained failures) *For every $\ell \geq 0$, $k \geq 1$, $n > \ell + k$, and $\lambda = n - \ell$, there is no k-set agreement algorithm for n processes, using registers, 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 k-set agreement algorithm, say A, that 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 ℓ-constrained > initial). It then follows (from the contradiction assumption) that the assumed algorithm A solves k-set agreement in a system of $n' = n - \ell$ processes, where $n' > k$, using read/write registers. But in a system of n' processes, process contention is always lower or equal to n', from which follows that, in this execution, n'-constrained crash failures are the same as any-time failures. Thus, algorithm A generates a read/write-based k-set agreement algorithm A' for $n' = n - \ell$ processes, where $n' > k$, that tolerates k any-time crash failures. But, this is known to be impossible as shown in [4, 20, 30]. $\Box_{Theorem\ 5}$

For the special case of consensus, the equation $n > \ell+k$ becomes $n > \ell+1$. The following theorem is then an immediate consequence of Theorem 5.

Theorem 6 (Consensus with λ-constrained failures) *For every $0 \leq \ell < n - 1$ and $\lambda = n - \ell$, there is no consensus algorithm for n processes, using 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.*

We have shown in Section 2 that there is a consensus algorithm for n processes, using registers, that tolerates a *single* $(n - 1)$-constrained crash failure. It follows from Theorem 6 that this bound is tight. Consequently the consensus algorithm presented in Section 2 is resilience-optimal for λ-constrained crash failures.

Theorem 7 (k-set agreement with λ-constrained and any-time failures) *For every $\ell \geq 0$, $k \geq 1$, $n > \ell + k$, $g \geq 0$, and $\lambda = n - \ell$, there is no k-set agreement algorithm for n processes, using registers, 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 $\lambda = n$. $\Box_{Theorem\ 7}$

5 Conclusion

The paper explored a "hybrid" computational power of a model where the adversary's power to induce process failures depends on contention, i.e., the number of concurrently active processes. In particular, assuming that up to t processes may fail, the adversary is restricted to fail x of the processes as long as contention remains below a predefined threshold λ, and may fail the rest $t - x$ processes "any-time." The paper determines conditions under which variations of the consensus and set-agreement problems can be solved in such a model.

It has been shown that this failure model allows impossibility results to be circumvented [33]. To this end, the paper has first presented an algorithm that, considering the process contention threshold $\lambda = n - 1$, solves consensus in the presence of one λ-constrained crash failure.

The paper has then presented a generic algorithm solving the k-set agreement problem, for $k \geq 2$. So, it extends the set of possible executions in which k-set agreement can be solved despite asynchrony and process crashes. The proposed algorithm allows its users to tune it to specific failure-prone environments. This can be done by appropriately defining the pair of integers $\langle m, f \rangle$. As an example, considering k-set agreement and a contention threshold $\lambda = n - k$, these parameters control the number of crashes allowed to occur before the contention threshold λ is bypassed, namely $2m + \ell - k$, and the number of failures which can occur at any-time, namely, $f - 1$. That is, it is possible to trade one "any-time" failure for several "λ-constrained" failures, and vice versa. The paper has also presented impossibility results in the presence of λ-constrained crash failures.

The two algorithms that have been presented are based on totally different design principles, despite the fact that consensus is a special instance of k-set agreement where $k = 1$. Hence the question: Does a *non-trivial generic* algorithm exist[6] that could be instantiated for any value of $k \geq 1$ or is $k = 1$ a "special" value?

Acknowledgments

The authors want to thank the reviewers whose constructive comments helped improve the article's content and presentation.

References

[1] Afek Y., Attiya H., Dolev D., Gafni E., Merritt M., and Shavit N., Atomic snapshots of shared memory. *Journal of the ACM*, 40(4):873-890 (1993)

[2] Afek Y., Dolev D., Gafni E., Merritt M. and Shavit S., A bounded first-in, first-enabled solution to the ℓ-exclusion problem. *ACM Transactions On Programming Languages and Systems*, 16(3):939-953 (1994)

[3] Anderson J., Multi-writer composite registers. *Distributed Computing*, 7(4):175-195 (1994)

[4] 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)

[5] Chandra T., Hadzilacos V., and Toueg S., The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685-722 (1996)

[6] Chandra T. and Toueg S., Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225-267 (1996)

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

[8] Delporte-Gallet C., Fauconnier H., Gafni E., and Kuznetsov P., Set-consensus collections are decidable. *Proc. 20th Int'l Conference on Principles of Distributed Systems (OPODIS'10)*, LIPICs Vol; 70, pp. 7:1–7:17 (2016)

[9] Delporte-Gallet C., Fauconnier H., Guerraoui G. and Tielmanns A., The disagreement power of an adversary. *Distributed Computing*, 24(3):137–147 (2011)

[10] Durand A., Raynal M., Taubenfeld G., Set agreement and renaming in the presence of contention-related crash failures. *20th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2018)*, Springer LNCS 11201, pp. 269-283 (2018)

[11] Fischer M.J., Lynch N.A., Burns J.E., Borodin A., Resource allocation with immunity to limited process failure (Preliminary Report). *Proc. 20th IEEE Symposium on Foundations Of Computer Science (FOCS'79)*, IEEE Press, pp. 234-254 (1979)

[12] Fischer M.J., Lynch N.A., Burns J.E., Borodin A., Distributed FIFO allocation of identical resources using small shared space. *ACM Transactions on Programming Languages and Systems*, 11(1):90–114 (1989)

[13] 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)

[14] Fraigniaud P., Gafni E., Rajsbaum S., and Roy M., Automatically adjusting concurrency to the level of synchrony. *Proc. 28th Int'l Symposium on Distributed Computing (DISC'14)*, Springer LNCS 8784, pp. 1-15 (2014)

[15] Gafni E. and Guerraoui R., Generalizing universality. *Proc. 22nd Int'l Conference on Concurrency Theory (CONCUR'11)*, Springer LNCS 6901, pp. 17-27 (2011)

[6] *Non-trivial generic* means here that the algorithm has not to be a case statement that directs to sub-algorithms for the two cases $k = 1$ and $k > 1$. The code must be the same for all the values of $k \geq 1$ (as done in Algorithm 2 for $k \geq 2$).

[16] Gafni E. and Kuznetsov P., Turning adversaries into friends: simplified, made constructive and extended. *Proc. 14th Int'l Conference on Principles of Distributed Systems (OPODIS'10)*, Springer LNCS 6490, pp. 380-394 (2010)

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

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

[19] Herlihy M.P., Luchangco V., and Moir M., Obstruction-free synchronization: double-ended queues as an example. *Proc. 23th Int'l IEEE Conference on Distributed Computing Systems (ICDCS'03)*, IEEE Press, pp. 522-529 (2003)

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

[21] Lamport L., On inter-process communications, part I: basic formalism. *Distributed Computing*, 1(2): 77-85 (1986)

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

[23] 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)

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

[25] Peterson G.L., Observations on ℓ-exclusion. *28th annual allerton conference on communication, control and computing*, pp. 568–577 (1990) 19

[26] Raynal M., A distributed solution to the k out of m resources allocation problem. *Proc. Int'l Conference on Computing and Information (ICCI'91)*, Springer LNCS 497, pp. 509-522 (1991)

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

[28] Raynal M., *Fault-tolerant message-passing distributed systems: an algorithmic approach. Springer*, 459 pages, ISBN: 978-3-319-94140-0 (2018)

[29] Raynal M., Stainer J., and Taubenfeld G., Distributed universality. *Algorithmica*, 76(2):502-535 (2016)

[30] 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)

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

[32] Taubenfeld G., A closer look at fault tolerance. *Proc. 31st ACM Symposium on Principles of Distributed Computing (PODC(18)*, ACM Press, pp. 261–270 (2012)

[33] 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)

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