Set Agreement and Renaming in the Presence of Contention-Related Crash Failures*

Anaïs Durand[†], Michel Raynal[†,⋆] Gadi Taubenfeld[‡]

[†]IRISA, Université de Rennes, 35042 Rennes, France
[⋆]Department of Computing, Polytechnic University, Hong Kong
[‡]The Interdisciplinary Center, Herzliya 46150, Israel

Abstract

A new notion of process failure explicitly related to contention has recently been introduced by one of the authors (NETYS 2018). More precisely, given a predefined contention threshold λ, this notion considers the executions in which process crashes are restricted to occur only when process contention is smaller than or equal to λ. If crashes occur after contention bypassed λ, there are no correctness guarantees (e.g., termination is not guaranteed). It was shown that, when $\lambda = n - 1$, consensus can be solved in an n-process asynchronous read/write system despite the crash of one process, thereby circumventing the well-known FLP impossibility result. Furthermore, it was shown that when $\lambda = n-k$ and $k \geq 2$, k-set agreement can be solved despite the crash of $2(k-1)$ processes.

This article considers two types of process crash failures: "λ-constrained" crash failures (as previously defined), and classical crash failures (that we call "any time" failures). It presents two algorithms suited to these types of failures. The first one is a generic k-set agreement algorithm, whose genericity dimension is related to the value of λ. For $\lambda = n - k$, it solves k-set agreement, where $k = m + f$, in the presence of $t = 2m + f - 1$ crash failures, $2m$ being λ-constrained failures, and $(f - 1)$ being any time failures. The second algorithm solves $(n + f)$-renaming in the presence of $t = m + f$ crash failures, m being $(n - t - 1)$-constrained failures, and f being any time failures. It follows that the differentiation between λ-constrained crash failures and any time crash failures enlarges the space of executions in which the impossibility of k-set agreement and renaming in the presence of asynchrony and process crashes can be circumvented. In addition to its behavioral properties, both algorithms have a noteworthy first class property, namely, their simplicity.

Keywords: Agreement algorithm, Asynchronous system, Atomic register, Concurrency, Contention, ℓ-Mutual exclusion, Participating process, Process crash failure, Read/write register, Renaming, k-Set agreement.

1 Definitions and Motivation

1.1 Processes, Failures, Communication

The system is composed of n asynchronous sequential processes, denoted p_1, ..., p_n, which communicate by reading and writing atomic registers. 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*. The model parameter t denotes the maximal number of processes that can be faulty in an execution. It is assumed that all correct processes participate, i.e.,

*A preliminary version of this paper was presented at the 20th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS'18), Springer LNCS 11201 (2018).

execute their local algorithm. (Let us notice that this assumption is a classical –very often left implicit– assumption encountered in message-passing distributed algorithms [17].)

Let us call *contention* the current number of processes that started executing. The model parameter λ denotes a predefined contention threshold. So, an 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", and the ones that can appear at "any time"; λ-constrained crashes were introduced in [20] under the name "weak failures".

1.2 Motivation for Considering λ-Constrained Failures

As discussed in [20], 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 (i.e., weak 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 [14] and indulgent algorithms in general [11, 12], 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 [10], one is often tempted to use a solution which guarantees no resiliency at all. We point out that there is a middle ground: tolerating λ-constrained (weak) failures enables to tolerate failures some of the time. Also, traditional t-resilient algorithms 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.

The type of λ-constrained failures which are assumed to occur only *before* a specific predefined threshold on the level of contention is reached, is in particular useful in systems in which contention is usually low. Another possible type of weak failures, also defined in [20], in which failures are assumed to occur only *after* a specific predefined threshold on the level of contention is reached, may correspond to a situation where, when there is high contention, processes are slowed down and, as a result, give up and abort.

Finally, the new failure model establishes a link between contention and failures, which enables us to better understand various known impossibility results, like the impossibility result for consensus [10] and its generalizations [6, 13, 18].

1.3 Content of the Paper

This paper presents algorithms suited to the previously defined types of crash failures (namely, λ-constrained and any time crash failures). As announced in the abstract, the first algorithm is a generic k-set agreement algorithm for $k \geq 2$ (Section 2). Its genericity dimension lies in the value of $\lambda = n - \ell$, $\ell \geq k$. When instantiated with $\lambda = n - k$, it solves $(m + f)$-set agreement (hence $k = m + f$), while tolerating $t = 2m + f - 1$ crash failures, up to $2m$ being $(n - k)$-constrained failures, and $(f - 1)$ being any time failures. The second algorithm (Section 3) solves the $(n + f)$-renaming problem in the presence of $t = m + f$ crash failures, m of them being $(n - t - 1)$-constrained failures, and f being any time failures.

1.4 High Level Objects

To make the presentation of the proposed algorithms 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, which provides the processes with the operations acquire() and release(), allows up to ℓ of them 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 acquire(), at least one of them will terminate its invocation.

It is shown in [2, 9, 19] 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 acquire() and release() at most once.

Snapshot A snapshot object provides two operations denoted write() and snapshot() [1, 3]. 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 write(v), it writes v into $SN[i]$; and
(b). when p_i invokes snapshot(), it obtains the value of the array $SN[1..n]$ as if it read simultaneously and instantaneously all its entries.

Said another way, the operations write() and snapshot() are atomic. Snapshot objects can be implemented on top of asynchronous crash-prone read/write systems [1, 3, 16].

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

This section presents a k-set agreement algorithm that allows to circumvent the known impossibility result for solving k-set agreement in crash-prone asynchronous read/write systems where $t \geq k$ [6, 13, 18].

2.1 k-Set Agreement: Definition

A k-set agreement (k-SA) object is a one-shot object introduced by S. Chaudhuri [8] 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 provides a single operation denoted propose(), which 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.* At most k different values are decided.
- *Termination.* Every correct process decides a value.

When $k = 1$, k-set agreement boils down to consensus, whose impossibility in the presence of asynchrony and a single process crashed was proved in [10] for message-passing systems, and in [15] for read/write systems. It was later shown in [6, 13, 18] that it is impossible to solve k-set agreement in crash-prone asynchronous read/write systems where $t \geq k$. Hence, as the k-set agreement read/write-based algorithm presented in [20] works despite up to $t = 2(k-1)$ λ-constrained failures (where $\lambda = n - k$), the introduction of contention-related failures in [20] is a noteworthy advance in fault-tolerance, which enlarges the space of executions in which k-set agreement can be solved.

2.2 k-Set Agreement with $(n-k)$-Constrained Crash failures

For simplicity in the presentation, the generic k-set agreement algorithm is presented incrementally. Its base version, that considers $\lambda = n - k$ (i.e., $\ell = k$ when considering the general version, Section 2.3), is presented in this section.

Properties of the base algorithm In addition to the model and problem parameters n, t, k, and $\lambda = n-k$, the algorithm considers two integers $m \geq 0$ and $f \geq 1$, such that $m+f = k$ and $t = 2m+f-1$ (or, equivalently, $t = 2k - f - 1$). Its fault-tolerance properties are summarized in Table 1.

The k-set agreement algorithm: Fault-tolerance properties	
total # of failures tolerated	$t = 2m + f - 1$
"λ-constrained" crash failures	$2m$
"any time" crash failures	$f - 1$

Table 1: k-Set agreement: tolerated crash failures with $\lambda = n - k$ and $k = m + f$

More generally, the parameters m and f, where $k = m + f$, can be seen as parameters allowing 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, and allows up to $k-1$ any time crash failures. At the other extreme, the pair $\langle m, f\rangle = \langle k-1, 1\rangle$ maximizes the number of λ-constrained failures: it allows up to $2(k-1)$ λ-constrained failures and no any time failure.

Since $t = 2m + f - 1$ we can say that, intuitively, *one* any time failure "equals" *two* $(n-k)$-constrained failures. That is, it is possible to trade *one strong* (any time) failure for *two weak* (λ-constrained) failures and vice versa, as demonstrated in Table 2.

The k-set agreement algorithm: tradeoffs			
total # of failures $t = 2m + f - 1$	$m = 0$ $f = k$	$m = \lceil k/2 \rceil$ $f = \lfloor k/2 \rfloor$	$m = k - 1$ $f = 1$
$2m$ "λ-constrained" crash failures	0	$2\lceil k/2 \rceil$	$2(k-1)$
$f - 1$ "any time" crash failures	$k - 1$	$\lfloor k/2 \rfloor - 1$	0

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

Interestingly, the particular instance $\langle m, f\rangle = \langle k-1, 1\rangle$ boils down to a specific case of the algorithm described in [20]. Additionally, as it will become clear in its description, Algorithm 1 presented below sheds new light on a relation linking k-set agreement and ℓ-mutual exclusion.

It is then shown that this algorithm works in a more general model where the contention threshold $\lambda = n - k$ is replaced by any contention threshold value $\lambda = n - \ell$ where $\ell \geq k$.

Base algorithm The algorithm, which, as announced in the Introduction, assumes assumes all correct processes participate, is characterized by the following theorem.

Theorem 1 *For any $n \geq 1$, $n > t \geq 0$, $m \geq 0$, and $f \geq 1$ such that $t = 2m + f - 1$ and $k = m + f$, it is possible to solve k-set agreement for n processes in the presence of at most t crash failures, $2m$ of them being λ-constrained failures (where $\lambda = n - k$), and $f - 1$ of them being any time failures.*

```
operation propose(in_i) is
(1)  PART.write(up);
(2)  repeat part_i ← PART.snapshot();
(3)       count_i ← |{x such that part_i[x] = up}|;
(4)  until count_i ≥ n − t end repeat;
(5)  if count_i ≤ λ then group_i ← 2 else group_i ← 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 ≠ ⊥ then return(DEC) end if end loop.

thread T2 is
(8)  if group_i = 1 ∨ m > 0 then
(9)       MUTEX[group_i].acquire();
(10)         if DEC = ⊥ then DEC ← in_i end if;
(11)      MUTEX[group_i].release();
(12)      return(DEC)
(13) end if.
```

Algorithm 1: k-SA despite up to $2m$ "λ-constrained" and $f-1$ "any time" failures ($\lambda = n-k$)

Remark 1 The proof of Theorem 1 is given in Section 2.4, when taking $\ell = k$.

In the algorithm described below, it is assumed that the identity of a process p_i is its index i.

Shared objects The processes cooperate through the following objects.

- $PART[1..n]$: snapshot object, initialized to $[\texttt{down}, \cdots, \texttt{down}]$, 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.

For the special case where $m = 0$ and $f = k$, in the proposed algorithm no process will ever try to access the $MUTEX[2]$ object. Thus, there is no need to define the notion of a 0-mutex 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 $\{1, 2\}$.

Behavior of a process p_i Algorithm 1 describes the behavior of a process p_i. When it invokes 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 $n-t$ 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 return() at line 7 or 12 terminates the invocation of 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 has still 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. Intuitively this explains why at most $k = m + f$ different values can be decided.

Remark 2 The reader can check that the line 8 (together with line 13) and line 11 can be suppressed without compromising the correctness of the algorithm. This is a side-effect of task $T1$. For clarity, we nevertheless keep these lines.

2.3 k-Set Agreement with $(n - \ell)$-Constrained Crash failures

The generic algorithm considers an additional contention-related parameter ℓ such that $\ell \geq k$. Instead of $\lambda = n - k$, its contention threshold is now $\lambda = n - \ell$. Hence, Section 2.2 corresponds to the particular case where $k = \ell$. In a very interesting way, this generic algorithm is nothing else than ... Algorithm 1. The only change is not in the text of the algorithm, but in the value of the contention threshold λ. The fault-tolerance properties of Algorithm 1 where $\lambda = n - \ell$ are summarized in Table 3, which extends Table 1.

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

Table 3: Tolerated crash failures when $\lambda = n - \ell$, $k = m + f$, and $\ell \geq k$.

As before, the parameters m and f, where $k = m+f$, 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 4 which generalizes 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 4: k-Set agreement: tradeoffs "λ-constrained/any time" crash failures when $\lambda = n - \ell$

2.4 Proof of Algorithm 1 for $\lambda = n - \ell$ and $\ell \geq k$

Lemma 1 *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 $PART.\mathsf{snapshot}()$ (line 1). It follows from the atomicity of the

write() and snapshot() operations on the object $PART$ that $count_i \geq |P| \geq n - \ell + 1$, a contradiction. $\square_{Lemma\ 1}$

Lemma 2 *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. $\square_{Lemma\ 2}$

Theorem 2 (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 is at most t processes that may fail and participation is required, at least $n-t$ processes 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 1, 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. If $m = 0$, there is $n-t = n-f+1$ correct processes and, since $k = f$ and $\ell \geq k$, $n-f+1 > n-k \geq n-\ell$. So, among the processes participating in $MUTEX[1]$, at least one of them is correct and at most $f-1$ of them crash before returning from $MUTEX[1]$.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 11, every other correct process will decide and terminate.

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

- If at least $y \geq f$ processes participate in $MUTEX[1]$, it follows from Lemma 2 that at most $f - 1$ of them crash before returning from $MUTEX[1]$.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_x into this atomic register.
  - Otherwise, no correct process participates in $MUTEX[1]$. By Lemma 1, 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.

$\square_{Theorem\ 2}$

Theorem 3 (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. $\square_{Theorem\ 3}$

3 M-Renaming where $M = n + f$

3.1 M-Renaming: Definition

The renaming object was introduced in the context of message-passing system [4]. An introductory survey to renaming in crash-prone asynchronous read/write systems is presented in [7].

An M-renaming object allows n processes with initially distinct names from a large name space to acquire distinct new names from a smaller name space $\{1, ..., M\}$, where M is a predefined value known by the processes. A one-shot renaming object allows each process to acquire a distinct new name just once. A long-lived renaming object allows processes to repeatedly acquire distinct names and release them. In this paper, we consider one-shot renaming objects.

A process p_i accesses an M-renaming object R using the operation $R.\mathsf{rename}(id_i)$, where id_i is its original name, which returns a new name. A process p_i knows neither its index i, nor the original names of the other processes. The properties defining such an object are the following.

- *Validity.* A new name belongs to the set $\{1, ..., M\}$.
- *Agreement.* No two processes obtain the same new name.
- *Termination.* If a process invokes $R.\mathsf{rename}(id)$ and does not crash, it returns from its invocation.

In the classical n-process model (i.e., a model where only any time crash failures are considered), it is known that with t any time failures, there is a tight $(n + t)$ bound on the size of new name space for renaming for infinitely many values of n. We will show how this result can be circumvented. The interested reader will find renaming algorithms in textbooks such as [5, 16, 19].

3.2 Properties of the Algorithm

Considering a new name space of size $M = n+f$, Section 3.3 presents a general M-renaming algorithm that, in addition to the model and problem parameters n, t, and $\lambda = n - t - 1$, as previously, considers two integers $m \geq 0$ and $f \geq 0$, such that $t = m + f$. The fault-tolerance properties of this algorithm are summarized in Table 5.

Similarly to the case of k-set agreement, the parameters m and f, where $t = m + f$, allows the user to tune the type of crash failures and (here) the size of the name space that are dominant in the considered application context. At one extreme, the pair of values $\langle m, f\rangle = \langle 0, t\rangle$ maximizes the number of any time failures (which is good) but also maximizes the size of the name space (which is bad). At the other extreme, the pair $\langle m, f\rangle = \langle t, 0\rangle$ maximizes the number of λ-constrained failures and minimizes the size of the name space (which is good). This is demonstrated in Table 6.

The $(n+f)$-renaming algorithm: Fault-tolerance properties	
total # of failures tolerated	$t = m + f$
"λ-constrained" crash failures	m
"any time" crash failures	f

Table 5: M-Renaming: tolerated crash failures, with $\lambda = n - t - 1$

The $(n+f)$-renaming algorithm: Tradeoffs			
total # of failures $t = m + f$	$m = 0$ $f = t$	$m = \lceil k/2 \rceil$ $f = \lfloor k/2 \rfloor$	$m = t$ $f = 0$
m "λ-constrained" crash failures	0	$\lceil k/2 \rceil$	t
f "any time" crash failures	t	$\lfloor k/2 \rfloor$	0
The size of name space	$n + t$	$n + \lfloor k/2 \rfloor$	n

Table 6: M-Renaming: tradeoffs "λ-constrained/any time" crash failures, with $\lambda = n - t - 1$

3.3 Algorithm

The proposed renaming algorithm, which allows us to circumvent the $(n + t)$ tight bound on the size of name space for renaming for infinitely many values of n, is amazingly simple. As the previous k-set algorithm, it assumes that all correct processes participate. It is characterized by the following theorem (which follows from Theorems 5 and 6).

Theorem 4 *For any $n \geq 1$, $n > t \geq 0$, $m \geq 0$, and $f \geq 0$ such that $t = m + f$, it is possible to solve $(n + f)$-renaming for n processes in the presence of at most t crash failures, m of them being λ-constrained failures (where $\lambda = n - t - 1$), and f of them being any time failures.*

Shared objects The processes cooperate through the following objects.

- $PART[1..n]$: snapshot object, initialized to $[\texttt{down}, \cdots, \texttt{down}]$, used to indicate participation.
- $RENAMING_f$: $(n + f)$-renaming object which can tolerate up to f any time crash failures for a model where participation is not required. The fact that participation is not required means that a process that does not participate is *not* consider faulty. The object is not assumed to tolerate any additional λ-constrained failures. An example of such an algorithm is described in [5] (pages 359-360).

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; id_i and new_name_i are used to store the original and new names, respectively.

Behavior of a process p_i Algorithm 2 describes the behavior of a process p_i. Every process p_i keeps on taking snapshots until it notices that $n - t$ processes (including itself) are participating. Then, the process invokes the rename operation of the underlying object $RENAMING_f$, stores the value of its new name in new_name_i, and returns this value.

```
operation rename(id_i) is
(1)  PART.write(up);
(2)  repeat part_i ← PART.snapshot();
(3)       count_i ← |{x such that part_i[x] = up}|
(4)  until count_i ≥ n − t end repeat;
(5)  new_name_i ← RENAMING_f.rename(id_i);
(6)  return(new_name_i).
```

Algorithm 2: $(n + f)$-renaming despite up to m "$(n - t - 1)$-constrained" and f "any time" failures, where $t = m + f$

3.4 Proof

Lemma 3 *In the presence of at most $t = m+f$ crash failures, m of them being $(n-t-1)$-constrained, if processes participate in $RENAMING_f$, at most f of them can fail.*

Proof If a process p_i participates in $RENAMING_f$ it follows from line 4 that the predicate $count_i \geq n - t$ is satisfied when it exited the repeat loop (lines 2-4). Thus, the contention was at least $n - t$ when p_i exited the loop and, due to the definition of "$(n - t - 1)$-constrained crash failures", there is no more such failures. As $t = m + f$, it follows that, if processes participate in $RENAMING_f$, at most f of them can fail. $\square_{Lemma\ 3}$

Theorem 5 (Termination) *In the presence of at most $t = m + f$ crash failures, m of them being $(n - t - 1)$-constrained, every correct process eventually terminates.*

Proof Since there is at most t processes that may fail and participation is required, at least $n-t$ processes 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).

By Lemma 3, if processes participate in $RENAMING_f$, at most f of them can fail. Since, by definition, (1) $RENAMING_f$ can tolerate f any time failures, and (2) in $RENAMING_f$ participation is not required, it follows that every operation invoked by a correct processes on $RENAMING_f$ must return a value. Thus, every correct process eventually terminates. $\square_{Theorem\ 5}$

Theorem 6 (Agreement and validity) *In the presence of at most $t = m + f$ crash failures, m of them being $(n - t - 1)$-constrained,* (1) *no two processes decide on the same new name, and* (2) *the new names are in the range $[1..n + f]$.*

Proof By Lemma 3, at most f processes can fail while executing $RENAMING_f$. Since, $RENAMING_f$ is an $(n + f)$-renaming object which can tolerate up to f crash failures for a model where participation is not required, any correct process that participates in $RENAMING_f$ must acquire a unique new name in the range $[1..n + f]$. $\square_{Theorem\ 6}$

3.5 From M-Renaming to One-shot Concurrent Objects

Let us consider any one-shot concurrent object OB, which provides a single operation op(), and tolerates up to x any time crash failures in a model where participation is not required.

This section presents an algorithm that transforms OB in an object OB' where, assuming all processes participate (i.e., invoke op()), allows to withstand additional λ-constrained crash failures. As in the previous section, the transformation considers the parameters n, t, $\lambda = n - t - 1$, $m \geq 0$, and $0 \leq f \leq x - 1$. The fault-tolerance properties of the resulting object OB' are summarized in Table 7 (where, let us remind, x is the number of any time crash failures tolerated by the underlying object OB').

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

Table 7: Crash failures tolerated by OB', where $\lambda = n - t - 1$

As before, the parameters m and f are parameters that allow the user to tune the type of crash failures that are dominant in the considered application context.
Algorithm 3 transforms of OB into OB'. It is the same as Algorithm 2, which implements an M-renaming object coping with both λ-constrained failures and any time failures. The meaning of the underlying shared objects and local variables are the same as in Algorithm 2. In addition, res_i contains the result of the underlying invocation $OB.\mathsf{op}(in)$ (line 5), where in is the input parameter of $\mathsf{op}()$. The proof, which is the same as the one given in Section 3.4, is left to the reader.

```
operation op(in) is % applied to OB'
(1)  PART.write(up);
(2)  repeat part_i ← PART.snapshot();
(3)          count_i ← |{x such that part_i[x] = up}|
(4)  until count_i ≥ n − t end repeat;
(5)  res_i ← OB.op(in);
(6)  return(res_i).
```

Algorithm 3: Transformation of the operation *op* of a one-shot object tolerating up to m "$(n - t - 1)$-constrained" failures and f "any time" failures, where $t = m + f$

4 Conclusion

This paper addressed a process crash failure model in which some number of processes may crash only when process contention has not bypassed a predefined threshold λ, while another number of processes may crash at any time. It has been shown that this failure model allows impossibility results to be circumvented. To this end, the paper has presented algorithms building k-set agreement and renaming objects in such a model. So, it extends the set of possible executions in which k-set agreement and renaming can be solved despite asynchrony and process crashes. The proposed algorithms allow their users to tune them 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 $2m = 2(k - f)$, and the number of failures which can occur at any time, namely, $f - 1$. That is, it is possible to trade *one strong* "any time" failure for *two weak* "$(n - k)$-constrained" failures, and vice versa.

Finally, some issues remain challenging on the open problem side. More specifically, on the complexity/computability side of k-set agreement, it would be interesting to find out whether the upper bound we have proved on the number of failures $t = 2m + f - 1$ (where $2m$ failures are $(n - k)$-constrained and $f - 1$ failures are any time failures) is tight for $k \geq 2$. On the algorithm design side, as there is an algorithm (and a tight bound) for 1-agreement (see [20]), it would be interesting to find a more general algorithm, i.e., an algorithm which works for $k \geq 1$ (and not only for $k \geq 2$).

Acknowledgments

We thank Armando Castañeda for helpful discussions on the renaming problem. This work has been partially supported by the Franco-German DFG-ANR Project 40300781 DISCMAT (devoted to connections between mathematics and distributed computing), and the French ANR project ANR-16-CE40-0023-03 DESCARTES (devoted to layered and modular structures in distributed computing).

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] Attiya H., Bar-Noy A., Dolev D., Peleg D., and Reischuk R., Renaming in an asynchronous environment. *Journal of the ACM*, 37(3):524-548 (1990)

[5] 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)

[6] 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)

[7] Castañeda A., Rajsbaum S., and Raynal M., The renaming problem in shared memory systems: an introduction. *Elsevier Computer Science Review*, 5:229-251 (2011)

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

[9] 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)

[10] 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)

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

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

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

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

[15] 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)

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

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

[18] 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)

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

[20] Taubenfeld G., Weak failures: definition, algorithms, and impossibility results. *Proc. 6th Int'l Conference on Networked Systems (NETYS'18)*, Springer LNCS 11028, 15 pages (2018)