Better Sooner Rather Than Later

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

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

January 30, 2024

Abstract

This article unifies and generalizes fundamental results related to n-process asynchronous crash-prone distributed computing. More precisely, it proves that for every $0 \leq k \leq n$, assuming that process failures occur only before the number of participating processes bypasses a predefined threshold that equals $n - k$ (a participating process is a process that has executed at least one statement of its code), an asynchronous algorithm exists that solves consensus for n processes in the presence of f crash failures *if and only if* $f \leq k$. In a very simple and interesting way, the "extreme" case $k = 0$ boils down to the celebrated FLP impossibility result (1985, 1987). Moreover, the second extreme case, namely $k = n$, captures the celebrated mutual exclusion result by E.W. Dijkstra (1965) that states that mutual exclusion can be solved for n processes in an asynchronous read/write shared memory system where any number of processes may crash (but only) before starting to participate in the algorithm (that is, participation is not required, but once a process starts participating it may not fail). More generally, the possibility/impossibility stated above demonstrates that more failures can be tolerated when they occur earlier in the computation (hence the title).

Keywords: Adopt/commit, Asynchronous read/write system, Concurrency, Consensus, Contention, Mutual exclusion, Process participation, Process crash, Time-constrained crash failure, Simplicity.

1 Introduction

1.1 Two fundamental problems in distributed computing

On the nature of distributed computing. *Parallel computing* aims to track and exploit data independence in order to obtain efficient algorithms: the decomposition of a problem into data-independent sub-problems is under the control of the programmer. The nature of *distributed computing* is different; namely, distributed computing is the science of cooperation in the presence of adversaries (the most common being asynchrony and process failures): a set of predefined processes, each with its own input (this is not on the control of the programmer) must exchange information in order to attain a common goal. The two most famous distributed computing problems are *consensus* and *mutual exclusion*.

The consensus problem. The consensus problem was initially introduced in the context of synchronous message-passing systems in which some processes are prone to Byzantine failures [9, 11]. Consensus is a one-shot object providing the processes with a single operation denoted $\mathsf{propose}()$. This operation takes an input parameter and returns a value. When a process invokes $\mathsf{propose}(v)$, we say it proposes the value v. If $\mathsf{propose}()$ returns the value v', we say that it decides v'. The following set of properties defines consensus. A *faulty* process is a process that commits a failure (crash in our case, i.e., an unexpected premature and definitive stop). An *initial failure* is a process crash that occurs before the process starts participating [18]. A process that is not faulty is said to be *correct*.

- Validity. If a process decides value v, then v was proposed by some process.
- Agreement. No two processes decide different values.
- Termination. If a correct process invokes $\mathsf{propose}(v)$ then it decides on a value.

A fundamental result related to consensus in asynchronous crash-prone systems where processes communicate by reading and writing atomic registers only is its impossibility if even only one process may crash [10] (read/write counterpart of the famous FLP result stated for asynchronous message-passing systems [4]).

The mutual exclusion problem. Mutual exclusion is the oldest and one of the most important synchronization problems. Formalized by E.W. Dijkstra in the mid-sixties [1], it consists of building what is called a lock (or mutex) object, defined by two operations, denoted $\mathsf{acquire}()$ and $\mathsf{release}()$. The invocation of these operations by a process p_i follows the following pattern: "$\mathsf{acquire}()$; *critical section*; $\mathsf{release}()$", where "critical section" is any sequence of code. It is assumed that, once in the critical section, a process eventually invokes $\mathsf{release}()$. A mutex object must satisfy the following two properties.

- Mutual exclusion: No two processes are simultaneously in their critical section.
- Deadlock-freedom progress condition: If there is a process p_i that has a pending operation $\mathsf{acquire}()$ (i.e., it invoked $\mathsf{acquire}()$ and its invocation is not terminated) and there is no process in the critical section, there is a process p_j (maybe $p_j \neq p_i$) that eventually enters the critical section.

A fundamental result related to mutual exclusion in asynchronous fault-free systems where processes communicate by reading and writing atomic registers only is that mutual exclusion can be solved for any finite number of processes even when (process) participation is not required [1].

Observation 1 *In a shared memory system with no failures and where participation is not required, mutual exclusion is solvable if and only if consensus is solvable.*

The proof is straightforward. To solve consensus using mutual exclusion, we can simply let everybody decide on the proposed value of the first process to enter the critical section. To solve mutual exclusion using consensus, the processes can participate in a sequence of consensus objects to decide on the next process to enter the critical section.

1.2 Contention-related crash failures

The notion of λ-constrained crash failures. Consensus can be solved in crash-prone (read/write or message-passing) synchronous systems. So, an approach to solve consensus in crash-prone asynchronous systems consists in capturing a "logical time notion" that can be exploited to circumvent the consensus impossibility. In this article, the notion of time is captured by the increasing number of processes that started participating in the consensus algorithm (a process becomes *participating* when it accesses the shared memory for the first time).[1] Crash failures in such a context have given rise to the notion of λ-*constrained crash failures* (introduced in [17]) where they are named *weak* failures). Then, they have been investigated in [2,3]. The idea consists in allowing some number k of processes to crash only while the current number of participating processes has not bypassed some predefined threshold denoted λ. An example of a run with λ-constrained crash failures is presented in Fig. 1 for $n = 9$, $k = 3$, and $\lambda = n - k = 6$.

[1]Let us remind that such a process participation assumption is implicit in all asynchronous message-passing systems.

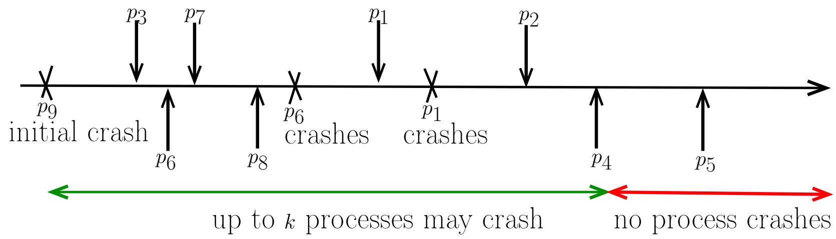

Figure 1: Asynchronous λ-constrained crash failures ($n = 9$, $k = 3$, $\lambda = 6$)

Observation 2 : A computability equivalence. The following observation follows immediately from the definitions concerning process participation, initial crashes, and λ-constrained crash failures. From a computability point of view, the three following statements are equivalent (each implies the two others).

It is possible to solve consensus and mutual exclusion in an asynchronous system

- in a fault-free system where participation is not required, or
- in the presence of any number of initial failures, or
- in the presence of any number of 0-constrained crash failures.

Motivation: Why study λ-constrained failures? As discussed and demonstrated in [2, 17], 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 (i.e., any-time) 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 [8] and indulgent algorithms in general [6, 7], 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 (discussed earlier [4, 10]) 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, a message-passing algorithm is described in [4], which solves consensus despite asynchrony and up to $t < n/2$ processes crashes if these crashes occur initially (hence no participating process crashes).

1.3 Computational model

Our model of computation consists of a collection of n asynchronous deterministic processes that communicate by atomically reading and writing shared registers. A process can read or write at each atomic step, but not both. A register that can be written and read by any process is a multi-writer multi-reader (MWMR) register. If a register can be written by a single (predefined) process and read by all, it is a single-writer multi-reader (SWMR) register. Asynchrony means that there is no assumption on the relative speeds of the processes. Each process has a unique identifier. The only type of failure considered in this paper is a process *crash* failure. As already said, a crash is a premature halt. Thus, until a process possibly crashes, it behaves correctly by executing its code. The following known observation implies that an impossibility results proved for the shared memory model also holds for such a message-passing system.

Observation 3 *A shared memory system that supports atomic registers can simulate a message-passing system that supports send, receive, and even broadcast operations.*

The proof is straightforward. The simulation is as follows. With each process p, we associate an unbounded array of shared registers which all processes can read from, but only p can write into. To simulate a broadcast (or sending) of a message, p writes to the next unused register in its associated array. When p has to receive a message, it reads the new messages from each process.

1.4 Contributions and related work

The article unifies and generalizes fundamental results about the mutual exclusion and consensus problems. To this end, it states and proves the following theorem.

Theorem 1 (Main result) *For every $0 \leq k \leq n$, an algorithm exists that solves consensus for n processes in the presence of f $(n-k)$-constrained crash failures if and only if $f \leq k$.*

There are two special cases that are of special interest.

- The first special case, when $k = 0$, indicates that in the presence of any number of n-constrained crash failures, not even a single failure can be tolerated. This implies the celebrated impossibility results (from 1985 and 1987) which states that consensus cannot be solved by n processes in an asynchronous message-passing or read/write shared memory system in which even a single process may crash at any time [4, 10]. Here, we use Observation 3 that a shared memory system can simulate a message passing system.

- The second special case, when $k = n$ implies that consensus can be solved for n processes in an asynchronous read/write shared memory system in the presence of any number of 0-constrained crash failures. This result, together with Observation 1 and Observation 2, implies the celebrated result by E.W. Dijkstra (from 1965), which originated the field of distributed computing, that mutual exclusion can be solved for n processes in an asynchronous read/write shared memory fault-free system where (process) participation is not required [1].

It is shown in [2, 17], among other results, that consensus can be solved (1) despite a single process crash if this crash occurs before the number of participating processes bypasses $\lambda = n - 1$; and (2) despite $k - 1$ process crashes, where $k > 1$, if these crashes occur before the number of participating processes bypasses $\lambda = n - k$.The main question left open in [2, 17] is whether this possibility result is tight.

Our main result, as stated in Theorem 1, shows that the answer to this open question is negative and proves a new stronger result which is shown to be tight. Furthermore, two cumbersome and complicated consensus algorithms were presented to prove the above results [2, 17]. These algorithms are based on totally different design principles, and the following question was posed as an open problem in [2]: "Does it exist a non-trivial generic consensus algorithm that can be instantiated for any value of $k \geq 1$?"[2]. Our result answers this second question positively.

To prove the if direction part in the proof of Theorem 1, a rather simple and elegant consensus algorithm is presented. This new algorithm is based on two underlying (read/write implementable)

[2]"Non-trivial generic" means here that the algorithm must not be a case statement with different sub-algorithms for different values of k.

objects, namely a crash-tolerant adopt-commit object [5] and a not-crash-tolerant deadlock-free acquire-restricted mutex object (a mutex object without a release operation [15, 16]). We show that the proposed algorithm is optimal in the λ-constrained crash failures model.

Finally, contention-related crash failures were also investigated in [3] in a model where processes communicate by accessing shared objects which are computationally stronger than atomic read/write registers.

2 The Consensus Algorithm

This section proves the "if direction" of Theorem 1.

Theorem 2 (If direction) *For every $0 \leq k \leq n$, an algorithm exists that solves consensus for n processes in the presence of f $(n-k)$-constrained crash failures **if** $f \leq k$.*

To prove this theorem, we present below a consensus algorithm tolerating k λ-constrained failures, where $\lambda = n - k$. The processes, denoted p_1, p_2, ..., p_n, execute the same code. It is assumed that proposed values are integers and that the default value $\perp$ is greater than any integer.

2.1 Shared and local objects used by the algorithm

Shared objects. The processes cooperate through the following shared objects (which can be built on top of asynchronous read/write systems, the first one in the presence of any number of crashes, the second one in failure-free systems, but as we will see, the access to this object will be restricted to correct processes only).

- $INPUT[1..n]$ is an array of atomic single-writer multi-reader registers. It is initialized to $[\perp, ..., \perp]$. $INPUT[i]$ 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 $\perp$ (a value that cannot be proposed).
- AC is an adopt/commit object. This object, which can be built in asynchronous read/write systems prone to any number of process crashes, was introduced in [5]. It provides the processes with a single operation (that a process can invoke only once) denoted ac_propose(). This operation takes a value as an 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 $\langle \texttt{adopt}, v\rangle$.

  Let us remark that if, initially, a process executes solo ac_propose(v), it returns the value v, and, if any, all later all invocations of ac_propose() will return v. The same occurs if (initially) a set of processes invoke ac_propose() with the same value v: the adopt/commit object will always return v. Wait-free implementation of the adopt-commit object are described in [5, 13].

- ARM is a one-shot *acquire-restricted* deadlock-free mutex object, i.e., a mutex object that provides the processes with a single operation denoted acquire() (i.e., a mutex object without release() operation). One-shot means that a process can invoke acquire() at most once.

  Let us observe that as there is no release() operation, only one process can return from its invocation of acquire(). The other processes that invoked acquire() never terminate their acquire() operation. The ARM object will be used to elect a process when needed in specific circumstances.

  As we will see, the proposed consensus algorithm allows only correct processes to invoke the acquire() operation. So any algorithm implementing a failure-free deadlock-free mutex algorithm (or a read/write-based leader election algorithm) can be used [15]. Such space efficient algorithms exist, that use only log n atomic read/write registers [16].

Local objects. Each process p_i manages four local variables denoted $input_i[1..n]$, val_i, res_i and tag_i. Their initial values are irrelevant.

2.2 An informal description of the algorithm

We present below the algorithm for process p_i. Recall that there are at most k λ-constrained crash failures, where $\lambda = n - k$.

1. p_i first deposits its proposed value in_i in $INPUT[i]$.
2. p_i repeatedly reads the $INPUT[1..n]$ array until $INPUT[1..n]$ contains at least $n - k$ entries different from their initial value $\perp$. Because at most k processes may crash, and the process participation assumption, this loop statement eventually terminates.
3. p_i computes the smallest value deposited in the array $INPUT[1..n]$ and sets val_i to that value.
4. p_i champions the value in val_i for it to be decided. To this end, it uses the underlying wait-free adopt/commit object; namely, it invokes AC.ac_propose(val_i) from which it obtains a pair $\langle tag_i, res_i \rangle$.
5. Once p_i's invocation of the adopt-commit object terminates, there are two possible cases,
   - if $tag_i = \texttt{commit}$, due to the weak agreement property of the object AC, no value different from res_i can be decided. Consequently, p_i writes res_i in the shared register DEC and returns res_i as the agreed upon consensus value, and terminates.
   - if $tag_i = \texttt{adopt}$, p_i continues to the next step below.
6. Notice that if p_i arrives here, it must be the case that process participation is above $n - k$, and hence no process will fail from that point in time. So, p_i continually checks whether $DEC \neq \perp$ and, in parallel, starts participating in the single-shot *mutex* object.
7. If p_i finds out that $DEC \neq \perp$, it returns the value of DEC as the agreed-upon consensus value and terminates.
8. If p_i enters the critical section, it writes res_i in the shared register DEC, returns res_i as the agreed-upon consensus value, and terminates.

Notice that if process p_i terminates in step 5, and process p_j terminates in step 8, then, due to the weak agreement property of the object AC it must be the case that $res_i = res_j$.

2.3 A formal description and correctness proof

Algorithm 1 describes the behavior of a process p_i. The statement $\mathsf{return}(v)$ returns the value v to the invoking process and terminates its execution of the algorithm. The idea that underlies the design of this algorithm is pretty simple, namely:

- Failure-prone part: Exploitation of the *participating processes* assumption to benefit from the adopt-commit object AC (Lines 1-5) and try to decide from it.
- Failure-free part: Exploitation of the λ*-constrained failures* assumption (Lines 6-8) to ensure that, if the adopt-commit object does not allow processes to decide, the decision will be obtained from the acquire-restricted mutex object, whose invocations occur in a failure-free context (crashes can no longer occur when processes access ARM).

operation $\mathsf{propose}(in_i)$ **is**
(1) $INPUT[i] \leftarrow in_i$;
(2) **repeat** $input_i[1..n] \leftarrow$ asynchronous non-atomic reading of $INPUT[1..n]$
until $\big(input_i[1..n]$ contains at most k $\perp\big)$ **end repeat**;
(3) $val_i \leftarrow \mathsf{min}\big($values deposited in $input_i[1..n]\big)$;
(4) $\langle tag_i, res_i \rangle \leftarrow AC.\mathsf{ac\_propose}(val_i)$;
(5) **if** $(tag_i = \mathtt{commit})$ **then** $DEC \leftarrow res_i$; $\mathsf{return}(DEC)$ **end if**;
(6) Launch in parallel the local thread T;
(7) $\mathsf{wait}\big(DEC \neq \perp\big)$; $\mathsf{kill}(T)$; $\mathsf{return}(DEC)$.

thread T **is**
(8) $ARM.\mathsf{acquire}()$; **if** $DEC = \perp$ **then** $DEC \leftarrow res_i$ **end if**.

Algorithm 1: Consensus tolerating k λ-constrained failures, where $\lambda = n - k$

Lemma 1 (Validity) *A decided value is a proposed value.*

Proof A process decides either on Line 5 or 7. Whatever the line, it decides the value of the shared register DEC, which was previously assigned a value that has been deposited in a local variable res_i (Line 5 or 8). The only place where a local variable res_i is updated is Line 4, and it follows from the validity property of the adopt-commit object that this value is the proposed value val_j of some process p_j. Since val_j is the minimum value seen by p_j in $INPUT$ (Line 3) that contains only the input values of the processes (and maybe some $\perp$ values that are, by definition, greater than any input variables), val_j contains the proposed value of some process. $\Box_{Lemma\ 1}$

Lemma 2 *If, when a process p_i exits at Line 2 at time t, at least $n - k + 1$ entries of $INPUT$ are different from $\perp$, then p_i is a correct process and no more crash occurs after time t.*

Proof If there is a time t at which at least $n - k + 1$ entries of $INPUT$ are different from $\perp$, it follows that the number of participating processes is greater than $n - k$. It then follows from the λ-constrained crash failures no process crashes after time t and p_i is a correct process. $\Box_{Lemma\ 2}$

Lemma 3 *If a process p_i executes Line 6, it is a correct process.*

Proof Let p_i be a process that executes Line 6. If at least $n-k+1$ entries of $INPUT[1..n]$ were different from $\perp$ when p_i exited Line 2, it follows from Lemma 2 that p_i is a correct process. So, let us consider the case where, when p_i exited Line 2, exactly $n-k$ entries of $INPUT[1..n]$ were different from $\perp$.

Recall that by obligation property, if the processes that invoke ac_propose() propose the same input value v, only the pair $\langle \texttt{commit}, v \rangle$ can be returned. Thus, since process p_i did not obtain $tag_i = \texttt{commit}$ at Line 4, it must be that some other process proposed, at Line 4, a value different than the value proposed by p_i. This implies that the minimum value computed by p_i at Line 3, is (1) different than the minimum value computed by some other process, say process p_j, at Line 3, and (2) that process p_j computed this minimum value at Line 3 before p_i reached Line 6.

Consequently, the set (of size $n-k$) of non-$\perp$ entries in $input_i$ at the time when p_i has exited Line 2 must be different than the set (of size at least $n-k$) of non-$\perp$ entries in $input_j$ at the time when p_j has exited Line 2, from which it follows that when the last of p_i and p_j exited Line 4, there were at least $n-k+1$ participating processes. Thus, by Lemma 2, p_i is a correct process. $\Box_{Lemma\ 3}$

Lemma 4 (Termination) *Every correct process decides.*

Proof Correct processes are required to participate, and there are no more than k crashes (model assumption). Thus, at least $n-k$ processes eventually write their input value into $INPUT$ and, thus, no process remains stuck in the loop at Line 2.

Since the adopt-commit object is wait-free, the invocation of AC.ac_propose(val_i) at Line 4 always terminates. If a correct process p_i obtains the pair $\langle \texttt{commit}, v \rangle$ when it invokes AC.ac_propose(val_i) at Line 4, it assigns $v \neq \perp$ to the shared register DEC and then decides. Any process that obtains the tag adopt will later decide at Line 7.

When no process p_i obtains the pair $\langle \texttt{commit}, v \rangle$ at Line 5, or when every process that obtains a pair $\langle \texttt{commit}, v \rangle$ crashes before updating the shared register DEC, DEC will not be updated at Line 5.

In such a case by Lemma 3, every correct process launches in parallel its local thread T (Line 6). By the deadlock-freedom property of ARM, some process, say process p_k, will eventually enter the critical section (Line 8). Process p_k then assigns $v \neq \perp$ to DEC. Again, all other processes will be able to decide with their threads T (Line 7). $\Box_{Lemma\ 4}$

Lemma 5 (Agreement) *No two processes decide different values.*

Proof We consider two cases. The first is when some process p_i obtains the pair $\langle \texttt{commit}, v \rangle$ from the invocation of AC.ac_propose(in_i) at Line 4. In this case, due to the weak agreement property of the adopt-commit object, all the processes that return from this invocation obtain a pair $\langle -, v \rangle$. It follows that the local variables res_j of every correct process p_j contains v. As only the content of the shared variable DEC or a local variable res_j can be decided by p_j, only the value v can be decided.

The second case is when no process p_i obtains the pair $\langle \texttt{commit}, - \rangle$. In this case, when a process p_i decides, this occurs at Line 7. By Lemma 3, p_i is correct (and also all the processes that cross Line 6 are correct) and launched its local thread T. So, only correct processes launch their threads T. Due to the deadlock-freedom property of mutex, one and only one of them, say process p_j, terminates its invocation of ARM.acquire() and imposes rec_j as the decided value. $\Box_{Lemma\ 5}$

Theorem 2 follows from Algorithm 1, Lemma 1, Lemma 4, and Lemma 5.

3 Optimality of the Algorithm

This section proves the "only if direction" of Theorem 1. We point out that the impossibility result we give below was essentially already presented in [2, 17]. Our proof is an adaptation of the proof from [2, 17].

Theorem 3 (Only if direction) *For every $0 \leq k \leq n$, an algorithm exists that solves consensus for n processes in the presence of f $(n-k)$-constrained crash failures* ***only if*** *$f \leq k$.*

Proof To prove the only if direction, we have to show that, in the context of process participation and λ-constrained crash failures, with $\lambda = n - k$, there is no read/write registers-based algorithm that solves consensus while tolerating $(k+1)$ λ-constrained crash failures. To this end, assume to the contrary that for some k such that $n > k+1$, and $\lambda = n-k$ that there is a read/write-based algorithm A that tolerates $k+1$ λ-constrained crash failures.

Given an execution of A, let us remove any set of k processes by assuming they crashed initially. It then follows from the contradiction assumption that algorithm A solves consensus in a system of $n' = n - k$ processes. However, in a system of $n' = n - k$ processes, the number of participating processes is always smaller or equal to n', from which follows that, in such an execution, n'-constrained crash failures are crashes that occur at any time, i.e., these crashes are not constrained by some timing assumption. It follows that A may be used to generate a read/write-based consensus algorithm for $n' - k$ processes that tolerates one crash failure that can occur at any time. This contradicts the known impossibility of consensus in the presence of asynchrony and even a single crash failure, presented in [4, 10]. $\square_{Theorem\ 3}$

4 Discussion

Better sooner than later in general. There are many reasons why it is better for failures to occur sooner rather than later. For example, identifying failures early in the software development life cycle helps save valuable time and resources. When failures are detected early in, the necessary actions can be taken promptly to mitigate or address the issue. Early failures offer a chance to iterate and optimize, increasing the chances of success in subsequent attempts. It also provides ample time to recover and redirect efforts toward alternative solutions.

Better sooner than later in this article. In this article, we have identified yet another reason why it is better for failures to occur sooner rather than later: in the context of asynchronous distributed algorithms, more failures can be tolerated when it is a priori known that they may occur earlier in the computation. That is, we have demonstrated a tradeoff between the number of failures that can be tolerated and the information about how early they may occur. In the two extreme cases, if failures may occur only initially, then both mutual exclusion and consensus can be solved in the presence of any number of (initial) failures; while when failures may occur at any time, then it is impossible to solve these problems even in the presence of a single (any time) failure. More generally, for every $0 \leq k \leq n$, if it is known that failures may occur only before the number of participating processes bypasses a predefined threshold that equals $n-k$, then it is possible to solve consensus for n processes in the presence of up to k failures, but not in the presence of $k+1$ failures.

On simplicity. The proposed algorithm is simple. This does not mean that the problem was simple! As correctness, simplicity is a first class citizen property. Simplicity, as it captures the essence of a problem, makes its understanding easier. As said by A. Perlis (the very first Turing Award), "*Simplicity does not precede complexity, but follows it.*" [12].

Finally, let us notice that the following question has recently been addressed in [14]: *Are consensus and mutex the same problem?* It is worth noticing that the present paper adds a new relation linking mutex and consensus when considering the notions of participating processes and failure timing.

References

[1] Dijkstra E.W., Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569 (1965)

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

[3] Durand A., Raynal M., and Taubenfeld T., Reaching agreement in the presence of contention-related crash failures. *Theoretical Computer Science*, 966-967:12 pages (2023)

[4] 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)

[5] 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)

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

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

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

[9] Lamport L., Shostak R. and Pease M., The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3)-382-401 (1982)

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

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

[12] Perlis A., Epigrams on Programming. *ACM Sigplan*, 17(9):7-13 (1982)

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

[14] Raynal M., Mutual exclusion vs consensus: both sides of the same coin? *Bulletin of the European Association of Theoretical Computer Science (EATCS)*, Vol.140, 14 pages (2023)

[15] Raynal M. and Taubenfeld G., A visit to mutual exclusion in seven dates. *Theoretical Computer Science*, 919:47-65 (2022)

[16] Styer E. and Peterson G.L., Tight bounds for shared memory symmetric mutual exclusion problems. *Proc. 6th ACM Symposium on Principles of Distributed Computing (PODC)*, ACM Press, pp. 117-191 (1989)

[17] 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)

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