Weak Failures: Definitions, Algorithms and Impossibility Results

Gadi Taubenfeld

The Interdisciplinary Center, P.O.Box 167, Herzliya 46150, Israel
tgadi@idc.ac.il

Abstract. The notion of weak failures, which should be viewed as *fractions* of traditional failures, is introduced and studied. It is known that there is no consensus algorithm using registers that can tolerate even a single crash failure. Is there a consensus algorithm using registers that can tolerate a "fraction" of a crash failure, i.e., a weak failure? It is known that there is no k-set consensus algorithm for $n > k$ processes using registers that can tolerate k crash failures. How many weak failures can a k-set consensus algorithm which uses registers tolerate? Answers to these questions follow from our general possibility and impossibility results regarding the ability to tolerate weak failures.

Keywords: Weak failures, shared memory, consensus, k-set consensus, contention.

1 Introduction

Fractions were studied by Egyptians mathematicians around 1600 B.C. However, fractions, as we use them today, didn't exist in Europe until the 17th century. It seems natural that we consider fractions also in the context of fault tolerance. Below we define, motivate and explore the new notion of weak failures. Weak failures should be viewed as *fractions* of traditional failures.

Tolerating traditional failures is always defined with respect to *all* possible executions of a given system. A system is said to tolerate t failures w.r.t. some property ϕ, if in *all* possible executions of the system in which at most t processes fails, ϕ is satisfied. When tolerating weak failures, also called fractional failures, it is only required that in *some* executions, and not necessarily in all executions, ϕ is satisfied. There are several ways for identifying the subset of executions in which ϕ should be satisfied. Below, we provide two possible definitions of weak failures.

1.1 Defining weak failures

A process is participating in an algorithm if it has executed at least one statement of that algorithm. The *point contention* of an algorithm at a given time is the maximal number of (correct and faulty) processes simultaneously participating in the algorithm. The point contention is bounded by the total number of processes.

1. An m-*failure* of *type* 1 is a failure of a process that (1) after it has failed the process executes no more steps, and (2) the failure may occur only while the point contention is *at most* m. That is, such weak failures are assumed *not to occur* once a certain predefined threshold on the level of point contention is reached.

2. An *m-failure type* 2 is a failure of a process that (1) after it has failed the process executes no more steps, and (2) the failure may occur only while the point contention is *at least* m. That is, such weak failures are assumed to happen once a certain predefined threshold on the level of point contention is reached.

When designing an algorithm for n processes, n-failures of type 1 and 1-failures of type 2 are the traditional crash failures. Thus, m-failures of type 1 where $m < n$, and m-failures of type 2 where $m > 1$, can be referred to as weak crash failures. Other types of weak failures, weak Byzantine failures, for example, can be defined similarly (but are not studied in this paper).

Considering weak failures of type 1, at first sight, it seems counterintuitive to tolerate failures in low contention environments, as the probability that a process crashes seems more likely to increase as system load increases. Below we provide motivating examples why weak failures, regardless of their type, are interesting.

We emphasize that *nothing* is preventing a process from failing. However, in the case of algorithms that tolerate weak failures of type 1 (resp. of type 2), when a process fails after (resp. before) the predefined threshold is reached, no correctness guarantees are given. It would be nice to be able to give guarantees for all the cases; unfortunately, this is not always possible.

1.2 Motivation

As already mentioned, weak failures should be viewed as *fractions* of traditional failures. This will enable us to design algorithms that can tolerate several traditional failures plus several additional 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 failures plus $t' > 0$ weak failures (of some type).

Adding the ability to tolerate weak failures to algorithms that are already designed to circumvent various impossibility results, such as the Paxos algorithm [13] and indulgent algorithms in general [10, 11], 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 weak 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 [9], one is often tempted to use a solution which guarantees no resiliency at all. We point out that there is a middle ground: tolerating weak failures (of some type) 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). Afterall, something is better than nothing.

The first type of weak failures is in particular useful in systems in which contention is usually low. The second type of weak failures 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 [9] and its generalizations [4, 12, 18].

1.3 Contributions

We have identified new types of weak failures, where failures are assumed to occur only before (type 1) or after (type 2) a specific predefined threshold on the level of contention is reached. All our technical results are for weak failures of type 1 only. From the rest of the paper, whenever we use the term weak failures, we mean weak failures of type 1, and whenever we use the term *crash m-failures*, we mean m-failures of type 1.

To illustrate the utility of the new definitions, we derive possibility and impossibility results for solving the well-known problems of consensus and k-set consensus in the presence of weak failures. The k-set consensus problem is to design an algorithm for n processes, where each process starts with an input value from some domain and must choose some participating process' input as its output. All n processes together may choose no more than k distinct output values. The 1-set consensus problem is the familiar consensus problem.

It is known that, in asynchronous systems, there is no consensus algorithm for n processes using registers that can tolerate even a single crash n-failure [9, 14]. We show that, in asynchronous systems, there is a consensus algorithm for n processes, using registers, that can tolerate a single crash $(n-1)$-failure, for every $n > 1$. The above bound is tight. We show that there is no consensus algorithm for n processes, using registers, that can tolerate *two* crash $(n-1)$-failures, for every $n > 2$.

It is known that, in asynchronous systems, there is no k-set consensus algorithm for $n > k$ processes using registers that can tolerate k crash n-failures [4, 12, 18]. We show that, in asynchronous systems, for every $\ell \geq 1$, $k \geq 1$ and $n \geq 2\ell + k - 2$, there is a k-set consensus algorithm for n processes, using registers, that can tolerate $\ell + k - 2$ crash $(n-\ell)$-failures. We show that there is no k-set consensus algorithm that can tolerate $\ell + k$ crash $(n-\ell)$-failures.

Solving consensus with a single crash $(n-1)$-failure using only registers is a deceptive problem. Once you are told that it is solvable, at first glance, it may seem simple to solve. The only way to understand its tricky nature is by trying to solve it. For that reason, we suggest the readers to try to solve the problem themselves.

2 Computational model

Our model of computation consists of an asynchronous collection of n deterministic processes that communicate via atomic read/write registers. The processes have unique identifiers. Asynchrony means that there is no assumption on the relative speeds of the processes.

A register can be atomic or non-atomic. With an *atomic* register, it is assumed that operations on the register occur in some definite order. That is, reading or writing an atomic register is an indivisible action. We will consider only atomic registers. In the sequel, by *registers*, we mean *atomic* registers.

A process executes its algorithm correctly until it possibly crashes. After it has crashed, it executes no more steps. A process that crashes is said to be *faulty*; otherwise, it is *correct*. In an asynchronous system, there is no way to distinguish between a faulty and a very slow process.

In a model where participation is required, every correct process must eventually become active and execute its code. Another situation is one in which participation is not required, as is usually assumed when solving the mutual exclusion or k-exclusion problems. All the known impossibility results for consensus and k-set consensus hold even when participating is required (and hence, of course, also when participating is not required). Unless explicitly stated otherwise (i.e., when we use a known solution for k-exclusion) we will assume that *participation is required.*

3 Possibility of consensus with a single crash (*n*–1)-failure

The consensus problem is defined as follows: There are n processes where each process $i \in \{1, ..., n\}$ has an input value in_i. The requirements of the consensus problem are that there exists a decision value v such that: (1) [*Agreement* & *termination*] each non-faulty process eventually decides on v, and (2) [*Validity*] $v \in \{in_1, ..., in_n\}$.

A fundamental result in distributed computing is that it is impossible to solve consensus with a single crash failure (i.e., a single crash n-failure) [9, 14]. We consider the strongest failure type which is strictly weaker than the traditional crash failure, namely $(n-1)$-failure, and show that it is possible to solve consensus with a single crash $(n-1)$-failure.

Theorem 1. *There is a consensus algorithm for* n *processes, using registers, that can tolerate a single crash* $(n-1)$*-failure, for every* $n \geq 1$.

The above bound on the number of crash $(n-1)$-failures is tight. In Section 5, it is shown that there does not exist a consensus algorithm for n processes, using registers, that can tolerate *two* crash $(n-1)$-failures, for any $n > 2$.

Final remark before presenting the algorithm. Assume that you know how to solve consensus for two processes with a single crash 1-failure. A common approach for solving consensus for many processes in the presence of a single fault is as follows: Choose the two processes with the smallest identifiers, have them run the two-process solution, and write the result into a register. The remaining processes keep reading the register until the result appears there. We notice that such a solution for many processes guarantees to tolerate only a single crash 1-failure, but not a single crash 2-failure.

3.1 The algorithm

The code of the algorithm appears in Figure 1. In the algorithm, each process can be in one of four states, 0,1,2 or 3, as recorded in its *state* register. A process participates in three rounds (lines 2–15). At each round $round \in \{1, 2, 3\}$, the process first checks whether all the other $n-1$ processes have already written the round number $round$ into their *state* registers (lines 4–6). In case of a positive answer, the process sets the *decision* register to the maximum input value among the input values of the other $n-1$

A CONSENSUS ALGORITHM WHICH CAN TOLERATE A SINGLE CRASH $(n-1)$-FAILURE:
`Program for process` $i \in \{1,...,n\}$ `with a non-negative input` in_i`.`

Shared: $state[1..n]$: array of registers, ranges over $\{0,1,2,3\}$, initially all entries are 0
$input[1..n]$: arrays of registers, initial values immaterial // input values
$decision$: register, initially $\bot$ // final decision value
Local: $counter, max, round, j$: local variables, initial values immaterial

```
1  $input[i] := in_i$;
2  for $round = 1$ to $3$ do
3     $counter := 0$; $max := 0$;                                  // $state[i] = round - 1$
4     for $j = 1$ to $n$ do                                        // Am I last in this round?
5        if $state[j] \geq round$ then $counter := counter + 1$; // notice: $state[i] = round - 1$
6           if $max < input[j]$ then $max := input[j]$ fi fi od;
7     if $counter = n - 1$ then                                    // I'm last
         $decision := max$; $decide(max)$ fi;                      // decide & terminate
8     $state[i] := round$;                               // $counter < n - 1$, increment $state$
9     repeat                                             // wait until $n - 1$ processes arrive
10       $counter := 0$;
11       for $j = 1$ to $n$ do
12          if $state[j] \geq round$ then $counter := counter + 1$ fi od     // $i$ counts also itself
13    until ($counter \geq n - 1$ or $decision \neq \bot$)
14    if $decision \neq \bot$ then $decide(decision)$ fi          // decide & terminate
15 od;
                                                            // 3 rounds have been completed
16 if $counter = n - 1$ then $counter := 0$; $max := 0$;   // if $counter \neq n$, revisit round 2
17    for $j = 1$ to $n$ do                            // is some process missing from round 2?
18       if $state[j] \geq 2$ then $counter := counter + 1$;
19          if $max < input[j]$ then $max := input[j]$ fi fi od;
20    if $counter = n - 1$ then $decision := max$; $decide(max)$ fi fi;   //decide&terminate
                                                // at this point it must be that $counter = n$
21 for $j = 1$ to $n$ do                          // wait until all $n$ processes arrive to round 3
22    await ($state[j] = 3$ or $decision \neq \bot$);          // or wait for a decision to be made
23    if $decision \neq \bot$ then $decide(decision)$ fi;      // decide & terminate
24    if $max < input[j]$ then $max := input[j]$ fi od;
25 $decision := max$; $decide(max)$.                         // all have arrived; decide & terminate
```

Fig. 1. A consensus algorithm which can tolerate a single crash $(n-1)$-failure.

processes, decides on that maximum value and terminates (line 7). Otherwise, it writes the value of $round$ into its *state* register (line 8) and waits until either a decision is made or at least $n - 1$ processes, including itself, have written the round number $round$ into their single-writer *state* registers, whatever comes first (lines 9–13). In the former case, it adopts the decision value (line 14), in the latter case it completes the current round and moves on.

After completing three rounds, the process checks if there exists a process, say process j, that has not written the value 2 into its state register (lines 16–19). In case of a positive answer, it concludes that process j will never be able to reach round three, and thus, j will never set its state register to 3. This is so because j will notice that $n-1$ other processes have already set their state registers to 3, and will decide (line 7) before incrementing its state register (line 8). Thus, the process sets the *decision* register to the maximum input value among all the processes, excluding process j, decides on that maximum value and terminates (line 20).

Otherwise, if all the n processes have written the value 2 into their state registers, the process concludes that all the n processes are still active and are guaranteed not to fail. It waits until either a decision is made or until all the processes complete round three, whatever comes first (lines 21–24). In the former case, it adopts the decision value (line 23), in the latter case it decides on maximum input value among the input values of all the n processes and terminates (line 25).

3.2 Correctness proof

We say that process i is in round r if its local variable *round* equals to r.

Lemma 1. *For every $i \in \{1, ..., n\}$, when process i sets $state[i]$ to 2 (in line 8), either the point contention is already n or for some $j \neq i$, $state[j]$ will always be 0.*

Proof. If the point contention is not n when i sets $state[i]$ to 2 (in line 8), it follows that (1) by definition, some process, say j, hasn't taken any steps yet, and (2) except for process j, all the other $n-1$ processes have already incremented their *state* registers. If j is a correct process, it will eventually reach line 7 at which point its counter register will become equal to $n-1$. Thus, j will decide and terminate without ever incrementing its state register. □

Lemma 2. *If at some point in time, for every $i \in \{1, ..., n\}$ $state[i] \geq 2$, then all 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 process i has set $state[i]$ to 2 (in line 8), the point contention was already n. Thus, process i will never fail since it is assumed that no process fails once the point contention is n. □

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

1. *if process i writes into the decision register in line 7, then no other process writes into the decision register in line 7;*

2. *if process i writes into the decision register in line 7, then no other process writes into the decision register in line 25;*
3. *if process i writes into the decision register in line 20, then no other process writes into the decision register in line 25.*

Proof. Suppose process i writes into *decision* in line 7 in round $r \in \{1,2,3\}$. This means that all the other $n-1$ processes have already written r into their state registers, and hence have not written into *decision* in line 7 in round r or in previous rounds. After i writes into *decision*, it immediately terminates. Thus, i will never write a value $r' \geq r$ into its state register. Thus, for every other process, after executing the for loop in lines 4–6, the value of *counter* 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 25.

Suppose process i writes into *decision* in line 20. This means that there exists a process, say process j, that has not written the value 2 into its state register, at the time when process i checked $state[j]$ in line 18. Although process 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 j will reach $n-1$ when j will execute 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 22 "**await** ($state[j] = 3$ or $decision \neq \bot$)", the waiting may terminate only because $decision \neq \bot$. Thus, no process will ever reach and execute line 25. □

Lemma 4. *For every two processes i and j,*

1. *if i writes the value v into the decision register in line 7, and j writes the value v' into the decision register in line 20, then $v = v'$.*
2. *if i writes the value v into the decision register in line 20, and j writes the value v' into the decision register also in line 20, then $v = v'$.*
3. *if i writes the value v into the decision register in line 25, and j writes the value v' into the decision register also in line 25, then $v = v'$.*

Proof.

1. Assume that i writes the value v into the decision register in line 7, and j writes the value v' into the decision register in line 20. When i terminates, the value of its state register is either 0,1 or 2. In the first two cases (0 and 1), the value of *max* that j computes in line 19, does not depend on the input value of i, and hence $v = v'$. Consider the case that when i terminates (line 7), the value of its state register is 2. Thus, when i terminates, the values of all the other $n-1$ state registers must be 3. When j starts executing the for-loop in line 17, the value of the state registers of $n-1$ processes must be 3. Thus, i and j set their *max* registers (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 i. Thus, $v = v'$.
2. Assume that i writes the value v into the decision register in line 20, and j writes the value v' into the decision register also in line 20. When i started executed the for-loop in line 17, the value of the state register of *exactly* one process, say process k, was less than 2. Similarly, when j started executed the for-loop in line 17, the

value of the state register 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, i and j set their *max* registers (in line 19) to the same value, since they choose the maximum input value from the same $n-1$ input values. Thus, $v = v'$.

3. Assume that i writes the value v into the decision register in line 25, and j writes the value v' into the decision register in line 25. Both i and j set their *max* registers (line 24) to the same value, since they choose the maximum input value from the set of all n input values. Thus, $v = v'$. □

Theorem 2 (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, they write the same value. Also, whenever a process writes into the decision register, this written value is the input of a participating process. Each correct process decides only on a value written into the decision register. □

Theorem 3 (termination). *In the presence of at most a single crash $(n-1)$-failure, every correct process eventually terminates.*

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 await statement in line 22. In both places, whenever a process needs to wait, it continuously examines the value of the decision register, and if it finds out that $decision \neq \perp$ it decides on the value written in *decision* and terminates. Thus, we can conclude that: if some process terminates then every correct process will eventually terminate.

So, let us assume, by contradiction, that no correct process ever terminates. There are at least $n-1$ correct processes. At least $n-1$ correct process will execute the for loop in lines 1–15 with $round = 1$. They all will eventually execute the assignment in line 8, setting their state registers to 1. Thus, each correct process with $round = 1$, will eventually exit the repeat loop in lines 9–14, and will move to round two. By a similar argument, each correct process will eventually complete rounds two and three (i.e., will complete the for loop in lines 2–15).

A process reaches the for loop in lines 21–24, only if its local *counter* register 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 21–24 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 await statement on line 22. However, this is not possible since the value of the state register of each process which reaches line 22 must be 3. Thus, all the waiting processes in line 22, will be able to proceed beyond the await statement and terminate. A contradiction. □

Remark: It is tempting to simplify the algorithm by deleting the shared register *decision*, and removing all the read and write accesses to it. In such an algorithm a process decides only on the maximum value it has computed. Such a simplification would make the algorithm incorrect, as a process in round 2 may get stuck forever in the repeat-until loop in lines 9 13. This will happen if some process decides in round 1 (and terminates) while another process fails in round 1.

4 Possibility of *k*-set consensus with *l*+*k*–2 crash (*n*–*l*)-failures

The k-set consensus problem is to design an algorithm for n processes, where each process starts with an input value from some domain and must choose some participating process' input as its output. All n processes together may choose no more than k distinct output values. The 1-set consensus problem is the familiar consensus problem.

Another fundamental result in distributed computing is that for $1 \leq k \leq n-1$, it is impossible to solve k-set consensus in the presence of k crash failures (i.e., k crash n-failures) for n processes [4, 12, 18]. We show that it is possible to solve k-set consensus in the presence of $\ell+k-2$ crash $(n-\ell)$-failures. In particular, it is possible to solve k-set consensus in the presence of k crash $(n-2)$-failures. The possibility result presented below *does not* imply the result stated in Theorem 1.

Theorem 4. *For every $\ell \geq 1$, $k \geq 1$ and $n \geq 2\ell + k - 2$, there is a k-set consensus algorithm for n processes, using registers, that can tolerate $\ell + k - 2$ crash $(n-\ell)$-failures.*

The following algorithm solves k-set consensus and tolerates $\ell + k - 2$ crash $(n-\ell)$-failures. In Section 5, we show that there is no k-set consensus algorithm that can tolerate $\ell + k$ crash $(n-\ell)$-failures. The question whether it is possible to solve k-set consensus in the presence of $\ell + k - 1$ crash $(n-\ell)$-failures, is an interesting open problem.

4.1 The algorithm

In the implementation below we use a shared object called *atomic snapshot* which can be wait-free implemented from registers [1, 3]. A snapshot object consists of a set of $m > 1$ components, each capable of storing value. Processes can perform two different types of operations: UPDATE any individual component or instantaneously (atomically) SCAN the entire collection to obtain the values of all the components. So, for an atomic snapshot object S, $S.update(i, v)$ writes v to the i^{th} component, and $S.scan$ returns a snapshot of all m components.

A *single-writer* atomic snapshot object is a restricted version in which there are the same number of processes as components and only process $i \in \{1, ..., n\}$ can UPDATE the i^{th} component. Let $A[1..n]$ be an array of n registers and S be a single-writer atomic snapshot object. Then, the assignment $A := S.scan$ atomically sets $A[i]$ to the value of the i^{th} component of S, for each $i \in \{1, ..., n\}$. It is often easier to design fault-tolerant algorithms for asynchronous systems and prove them correct if one can think of the shared memory as a snapshot object, rather than as a collection of individual registers.

The algorithm also makes use of a single one-shot mutual exclusion object and a single one-shot $(k-1)$-exclusion object. The *k-exclusion* problem, which is a natural generalization of the mutual exclusion problem, is to design an algorithm which guarantees that up to k processes and no more are permitted to be in their critical sections

A k-SET CONSENSUS ALGORITHM WHICH CAN TOLERATE $\ell+k-2$ CRASH $(n-\ell)$-FAILURES:
`Program for process` $i \in \{1, ..., n\}$ `with a non-negative input` in_i`.`

```
Constants: $\ell$, $k$: positive integers
Shared:    $Flag$: single-writer atomic snapshot object, ranges over $\{0,1\}$, initially all entries are 0
           $decision$: register, initially $\perp$                          // final decision value
           $EX[1]$: one-shot mutual exclusion object                        // used for election
           $EX[2]$: one-shot $(k-1)$-exclusion object
Local:     $lflag[1..n]$: array of variables, ranges over $\{0,1\}$, initial values immaterial
           $counter, group, j$: variables, initial values immaterial

1   $Flag.update(i,1)$;                                                   // announce participation
2   repeat                                  // wait until at least $n-\ell-k+2$ processes participate
3     $lflag := Flag.scan$; $counter := 0$;
4     for $j = 1$ to $n$ do if $lflag[j] = 1$ then $counter := counter + 1$ fi od       // counting
5   until $counter \geq n-\ell-k+2$;
6   if $counter \leq n-\ell$ then $group := 2$ else $group := 1$ fi
7   participate in $EX[group]$ and in parallel continuously check if $decision \neq \perp$
8     if at any point $decision \neq \perp$ then decide($decision$) fi;          // decide & terminate
9     if you enter the critical section of $EX[group]$
10    then $decision := in_i$; decide($in_i$) fi.                                // decide & terminate
```

Fig. 2. A k-set consensus algorithm which can tolerate $\ell+k-2$ crash $(n-\ell)$-failures.

simultaneously. A solution is required to withstand the slow-down or even the crash (fail by stopping) of up to $k-1$ of processes. For $k=1$, the 1-exclusion problem is exactly the mutual exclusion problem. The simpler *one-shot* version assumes that a process may try to access its critical section at most once. It is well known that, for any $k \geq 1$, k-exclusion can be solved using registers only, even when it is assumed that participation is *not* required [2, 16]. We assume that the reader is familiar with the definition of the k-exclusion problem. A formal definition of the k-exclusion problem is given in the Appendix.

The code of the algorithm appears in Figure 2. The first step of each process i is to set the i^{th} component *Flag* to 1 (line 1). Then, process i repeatedly takes a snapshot of the *Flag* object (line 3) and each time it takes a snapshot, it sets *counter* to the number of Flag components which are set to 1 (line 4). Processes i continues to take snapshots until $counter \geq n-\ell-k+2$ (line 5). Since at most $\ell+k-2$ processes may fail, each correct process must eventually notice that $counter \geq n-\ell-k+2$. Next, process i participates in either $EX[1]$ or $EX[2]$ depending on the current value of *group* (line 7). If, at any point, process i notices that $decision \neq \perp$ it decides on the value of *decision* (line 8). If it enters its critical section, it sets *decision* to its input value and decides on that value.

4.2 Correctness proof

We notice that the *maximum* value of the *counter* of a process is when the process exits the repeat-loop, and at that point, this value is at least $n-\ell-k+2$. We use the notation $counter.p$ to denote the local counter variable of process p. As stated in Theorem 4, it is assumed below that: $\ell \geq 1$, $k \geq 1$ and $n \geq 2\ell+k-2$.

Lemma 5. *For every $m \in \{n-\ell-k+2, ..., n\}$, the maximum value of the counter of at most m processes is at most m.*

Proof. Assume to the contrary that for some $m \in \{n-\ell-k+2, ..., n\}$, the maximum value of the counter of more than m processes is at most m. Let P denote the set of these processes. Let $p \in P$ be the last process, among the processes in P, to update *Flag* in line 1. Since $|P| > m$, when p takes a snapshot it must notice that at least $m+1$ components of *Flag* are already set to 1. A contradiction. □

Lemma 6. *At least ℓ processes do not participate in $EX[2]$. Thus, at least ℓ processes participate in $EX[1]$ or fail.*

Proof. A process, say p, may participate in $EX[2]$, only if on exit of the repeat-loop, $counter.p \leq n-\ell$. Thus, by Lemma 5, at most $n-l$ processes participate in $EX[2]$, which implies that at least ℓ processes do not participate in $EX[2]$. This implies that, at least ℓ processes participate in $EX[1]$ or fail. □

Lemma 7. *If a process participates in $EX[1]$ then this process cannot fail.*

Proof. For a process, say p, to participate in $EX[1]$, it must be that on exit of the repeat-loop, $counter.p > n-\ell$. This implies that when p exits the repeat-loop, the point contention is at least $n-\ell+1$. Since the only type of failures is crash $(n-\ell)$-failures, p will not fail once it exits the repeat-loop and starts participating in $EX[1]$. □

Lemma 8. *There is at least one correct process.*

Proof. Since at most $\ell+k-2$ processes may fail, the number of correct processes is at least $n-(\ell+k-2)$. It is assumed (in the statement of Theorem 4) that $n \geq 2\ell+k-2$. Thus, the number of correct processes is at least $(2\ell+k-2)-(\ell+k-2)=\ell$. Since it is assumed that $\ell \geq 1$, there is at least one correct process. □

Theorem 5 (termination). *In the presence of at most $\ell+k-2$ crash $(n-\ell)$-failures, every correct process eventually terminates.*

Proof. First we notice that no correct process will get stuck forever in the repeat-loop (lines 2-5). Since at most $\ell+k-2$ processes may fail, each correct process must eventually notice that at least $n-\ell-k+2$ of the components of *Flag* are set to 1 and will exit the repeat-loop (lines 2-5).

By Lemma 7, if a process participates in $EX[1]$ then this process *cannot* fail. Thus, if some process participates in $EX[1]$, eventually some process will write its input value into *decision* letting all the other correct processes terminate.

So, let's assume that no process participates in $EX[1]$. This means that each one of the n processes either participates in $EX[2]$ or fails. Thus, by Lemma 8, at least one correct process participates in $EX[2]$. Also, by Lemma 6, at least ℓ processes which do not participate in $EX[2]$ fail, and since at most $\ell + k - 2$ processes may fail, we conclude that at most $k - 2$ of the processes that participate in $EX[2]$ may fail.

Since (1) $EX[2]$ is a one-shot $(k - 1)$-exclusion object (that, by definition, can tolerate $k - 2$ crash n-failures), (2) there exists a correct process which participates in $EX[2]$ (i.e., this process never fails), and (3) at most $k - 2$ of the processes which participate in $EX[2]$ fail, it follows that some correct process which participates in $EX[2]$, will eventually enter its critical section write its input value into *decision*, letting all the other correct processes terminate. □

Theorem 6 (k-agreement & validity). *All the participating processes decide on at most k different values, and each one of these decision values is the input of a participating process.*

Proof. There are exactly one one-shot mutual exclusion object and one one-shot $(k-1)$-exclusion object. In $EX[1]$ at most one process enters its critical section, and writes its input value into the decision register. In $EX[2]$ at most $k - 1$ processes enter their critical sections and write their input value into the decision register. Thus, at most k different values are written into *decision*. Also, whenever a process writes into the decision register, this written value is its input. Each correct process decides only on a value written into the decision register. □

5 Impossibility results

A natural question to ask next is, for a given k and ℓ, what is the maximum number of crash $(n - \ell)$-failures that can be tolerated by a k-set consensus algorithm? An *initial failure* of a given process is a failure which happens before the process has taken any steps.

Theorem 7. *For every $\ell \geq 0$, $k \geq 1$ and $n > \ell + k$, there is no k-set consensus algorithm for n processes, using registers, that can tolerate k crash $(n - \ell)$-failures and ℓ initial failures.*

Proof. Assume to the contrary that for some $\ell \geq 0$, $k \geq 1$ and $n > \ell + k$, there is an algorithm that can tolerate k crash $(n - \ell)$-failures and ℓ initial failures. Let $m = n - \ell$. Since we can always remove ℓ processes assuming that they always fail initially, it implies that there is a k-set consensus algorithm for m processes, where $m > k$, using registers, that can tolerate k crash m-failures. However, this is known to be impossible [4, 12, 18]. □

Since, for $m \geq 1$ a crash m-failure is strictly stronger (i.e, more severe) type of a failure than initial failure, an immediate corollary of Theorem 7 is that:

Corollary 1. *For every $\ell \geq 0$, $k \geq 1$ and $n > 2\ell + k$, there is no k-set consensus algorithm for n processes, using registers, that can tolerate $\ell + k$ crash $(n - \ell)$-failures.*

In the statement of Corollary 1, it is assumed that $n > 2\ell + k$, since in the context of $\ell + k$ crash $(n - \ell)$-failures, it makes sense to assume that $\ell + k$ is at most $n - \ell$. For the special case of consensus (i.e., 1-set consensus), we get that:

Corollary 2. *For every $0 \leq \ell < n/2$, there is no consensus algorithm for n processes, using registers, that can tolerate $\ell + 1$ crash $(n - \ell)$-failures.*

We have shown earlier (Theorem 1) that there is a consensus algorithm for n processes, using registers, that can tolerate a *single* crash $(n-1)$-failure. It follows from Corollary 2 that, there is no consensus algorithm that can tolerate *two* crash $(n - 1)$-failures.

6 Related work

Extensions of the notion of fault tolerance, which are different from those considered in this paper, were proposed in [5]. In [5], a precise way is presented to characterize adversaries by introducing the notion of disagreement power: the biggest integer k for which the adversary can prevent processes from agreeing on k values when using registers only; and it is shown how to compute the disagreement power of an adversary.

In [20], 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 [21].

The consensus problem was formally defined in [15]. The impossibility result that there is no consensus algorithm that can tolerate even a single crash failure was first proved for the asynchronous message-passing model in [9], and later has been extended for the shared memory model with atomic registers in [14]. The impossibility result that, for $1 \leq k \leq n - 1$ there is no k-resilient k-set-consensus algorithm for n processes using atomic registers is from [4, 12, 18].

The mutual exclusion problem was first stated and solved for n processes by Dijkstra in [6]. Numerous solutions for the problem have been proposed since it was first introduced in 1965. Dozens of interesting mutual exclusion algorithms and lower bounds are described in details in [17, 19]. The ℓ-exclusion problem, which generalizes the mutual exclusion problem, was first defined and solved in [8, 7]. Several papers have proposed ℓ-exclusion algorithms for solving the problem using atomic read/write registers satisfying various progress properties (see for example, [2, 16]).

7 Discussion

We have provided a new perspective on the relationship between failures and contention. From the computability point of view, this new perspective allows us to derive "fine-grained" analysis of the limit in computability for consensus and set consensus. That is, to illustrate the utility of the new definitions of weak failures, we have derived possibility and impossibility results for the well-known basic problems of consensus and k-set consensus. The definitions together with our technical results indicate that there is an interesting area of fault tolerance that deserves further investigation.

Two specific interesting open problems are: (1) Is the following generalization of Theorem 1 correct: There is a k-set consensus algorithm for n processes, using registers, that can tolerate k crash $(n-1)$-failures, for every $n > k \geq 1$; (2) Is the following generalization of Theorem 4 correct: For every $\ell \geq 0$, $k \geq 1$ and $n \geq 2\ell + k - 1$, there is a k-set consensus algorithm for n processes, using registers, that can tolerate $\ell + k - 1$ crash $(n - \ell)$-failures.

All our results are presented in the context of weakening the notion of crash failures in asynchronous systems. It would be interesting to consider also other types of weak failures such as weak omission failures or weak Byzantine failures and to consider synchronous systems. Another interesting direction would be to extend the results to objects other than atomic registers and to consider problems other than consensus and set-consensus. We have assumed that the number of processes is finite and known, it would be interesting to consider also the case of unbounded concurrency. Considering failure detectors in the context of the new definitions is another interesting direction.

References

1. Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit. Atomic snapshots of shared memory. *Journal of the ACM*, 40(4):873–890, 1993.
2. Y. Afek, D. Dolev, E. Gafni, M. Merritt, and N. Shavit. 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. J. H. Anderson. Composite registers. *Distributed Computing*, 6(3):141–154, 1993.
4. E. Borowsky and E. Gafni. Generalizecl FLP impossibility result for t-resilient asynchronous computations. In *Proc. 25th ACM Symp. on Theory of Computing*, pages 91–100, 1993.
5. C. Delporte-Gallet, H. Fauconnier, R. Guerraoui, and A. Tielmanns. The disagreement power of an adversary. *Distributed Computing*, 24(3):137–147, 2011.
6. E.W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, 1965.
7. M.J. Fischer, N. A.Lynch, J.E. Burns, and A. Borodin. Distributed FIFO allocation of identical resources using small shared space. *ACM Trans. on Programming Languages and Systems*, 11(1):90–114, January 1989.
8. M.J. Fischer, N.A. Lynch, J.E. Burns, and A. Borodin. Resource allocation with immunity to limited process failure. In *Proc. 20th IEEE Symp. on Foundations of Computer Science*, pages 234–254, October 1979.
9. M.J. Fischer, N.A. Lynch, and M.S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.
10. R. Guerraoui. Indulgent algorithms. In *Proc. 19th ACM Symp. on Principles of Distributed Computing*, pages 289–298, 2000.
11. R. Guerraoui and M. Raynal. The information structure of indulgent consensus. *IEEE Transactions on Computers*, 53(4):453–466, 2004.
12. M. P. Herlihy and N. Shavit. The topological structure of asynchronous computability. *Journal of the ACM*, 46(6):858–923, July 1999.
13. L. Lamport. The part-time parliament. *ACM Trans. on Computer Systems*, 16(2):133–169, May 1998.
14. M.C. Loui and H. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. *Advances in Computing Research*, 4:163–183, 1987.

15. M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, 1980.
16. G.L. Peterson. Observations on ℓ-exclusion. In *28th annual allerton conference on communication, control and computing*, pages 568–577, October 1990.
17. M. Raynal. *Algorithms for mutual exclusion*. The MIT Press, 1986. Translation of: Algorithmique du parallélisme, 1984.
18. M. Saks and F. Zaharoglou. Wait-free k-set agreement is impossible: The topology of public knowledge. *SIAM Journal on Computing*, 29, 2000.
19. G. Taubenfeld. *Synchronization Algorithms and Concurrent Programming*. Pearson / Prentice-Hall, 2006. ISBN 0-131-97259-6, 423 pages.
20. G. Taubenfeld. A closer look at fault tolerance. In *Proc. 31st ACM Symp. on Principles of Distributed Computing*, pages 261–270, 2012.
21. G. Taubenfeld, S. Katz, and S. Moran. Initial failures in distributed computations. *International Journal of Parallel Programming*, 18(4):255–276, 1989.

A A formal definition of the *k*-exclusion problem

The *k-exclusion* problem, which is a natural generalization of the mutual exclusion problem is to design a protocol which guarantees that up to k processes and no more may simultaneously access identical copies of the same non-sharable resource when there are several competing processes. That is, k processes are permitted to be in their critical section simultaneously. A solution is required to withstand the slow-down or even the crash (fail by stopping) of up to $k-1$ of processes. For $k = 1$, the 1-exclusion problem is the exactly mutual exclusion problem.

To illustrate the k-exclusion problem, consider the case of buying a ticket for a bus ride. Here a resource is a seat on the bus, and the parameter k is the number of available seats. In the k-exclusion problem, a passenger needs only to make sure that there is some free seat on the bus, but not to reserve a particular seat.

More formally, it is assumed that each process is executing a sequence of instructions in an infinite loop. The instructions are divided into four continuous sections of code: the *remainder, entry, critical section* and *exit*. The k-exclusion problem is to write the code for the *entry code* and the *exit code* in such a way that the following basic requirements are satisfied.

- *k-exclusion:* No more than k processes are at their critical section at the same time.
- *k-deadlock-freedom:* If strictly fewer than k processes fail (are delayed forever) then if a process is trying to enter its critical section, then some process, not necessarily the same one, eventually enters its critical section.

The k-deadlock-freedom requirement may still allow "starvation" of individual processes. It is possible to consider stronger progress requirements which do not allow starvation.