A Closer Look at Fault Tolerance*

Gadi Taubenfeld†

February 22, 2017

Abstract

The traditional notion of fault tolerance requires that *all* the correct participating processes eventually terminate, and thus, is not sensitive to the *number* of correct processes that should terminate as a result of failures. Intuitively, an algorithm that in the presence of any number of faults always guarantees that all the correct processes except maybe one terminate, is more resilient to faults than an algorithm that in the presence of a single fault does not even guarantee that a single correct process ever terminates. However, according to the standard notion of fault tolerance both algorithms are classified as algorithms that can not tolerate a single fault.

To overcome this difficulty, we generalize the traditional notion of fault tolerance in a way which enables to capture more sensitive information about the resiliency of an algorithm. Then, we present several algorithms for solving classical problems which are resilient under the new notion. It is well known that, in an asynchronous systems where processes communicate either by reading and writing atomic registers or by sending and receiving messages, important problems such as, consensus, set-consensus, election, perfect renaming, implementations of a test-and-set bit, a shared stack, a swap object and a fetch-and-add object have no deterministic solutions which can tolerate even a single fault. We show that while, some of these problems have solutions which guarantee that in the presence of *any* number of faults most of the correct processes will terminate; other problems do not even have solutions which guarantee that in the presence of just *one* fault at least one correct process terminates. All our results are presented in the context of crash failures in asynchronous systems.

Keywords: Fault tolerance, crash failures, shared memory, message passing, election, test-and-set, renaming, consensus, set-consensus, stack, swap, fetch-and-add.

*A preliminary version of the results presented in this paper, has appeared in *proceedings of the 31st annual symposium on principles of distributed computing* (PODC 2012), Madeira, Portugal, July 2012.

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

1 Introduction

1.1 Motivation

According to the standard notion of fault tolerance, an algorithm is t-resilient if in the presence of up to t faults, *all* the correct processes can still complete their operations and terminate. Thus, an algorithm is *not* t-resilient, if as a result of t faults there is *some* correct process that can not properly terminate. This traditional notion of fault tolerance is not sensitive to the *number* of correct processes that may or may not complete their operations as a result of the failure of other processes.

Consider for example the renaming problem, which allows processes, with distinct initial names from a large name space, to get distinct new names from a small name space. A renaming algorithm that in the presence of any number of faults always guarantees that *most* of the correct processes, but not necessarily all, get distinct new names is clearly more resilient than a renaming algorithm that in the presence of a single fault does not guarantee that even one correct process ever gets a new name. However, using the standard notion of fault tolerance, it is not possible to compare the resiliency of such algorithms – as both are simply not even 1-resilient. This motivates us to suggest and investigate a more general notion of fault tolerance.

We generalize the traditional notion of fault tolerance by allowing a limited number of participating correct processes not to terminate in the presence of faults. Every process that do terminate is required to return a correct result. Thus, our definition guarantees safety but may sacrifice liveness (termination), for a limited number of processes, in the presence of faults. The consequences of violating liveness are often less severe than those of violating safety. In fact, there are systems that can detect and abort processes that run for too long. Sacrificing liveness for few of the processes allows us to increase the resiliency of the whole system.

1.2 Model and Basic Definitions

Our model of computation consists of an asynchronous collection of n processes that communicate either by reading and writing atomic registers or by sending and receiving messages. The processes have unique identifiers. 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.

With required participation every process must eventually execute its code. However, a more interesting and practical situation is one in which participation is not required, as is usually assumed when solving resource allocation problems. Unless explicitly stated, when the shared memory model is considered, it is assumed that participate is *not* required. When the message passing model is considered, it is assumed that participate is required, a process starts participating spontaneously or when receiving a first message.

Once a process starts participating it may fail by *crashing*. A crash failure is a failure of a process where after it has failed the process executes no more steps. All our results are presented in the context of process crash failures in asynchronous systems.

In the literature, it is common to assume that the identifiers of the n processes are integers taken from the range $\{1, ..., n\}$. However, there may be situations when there are many more identifiers than processes. For example, there might be a small number of processes, say 50, but their identifiers can be taken from the range $\{0, ..., 2^{32}\}$. In such a case identifiers cannot be easily used to index registers, and hence it is better to use symmetric algorithms.

Symmetric Algorithms: A symmetric algorithm is an algorithm in which the only way

for distinguishing processes is by comparing identifiers, which are unique. Identifiers can be written, read and compared, but there is no way of looking inside any identifier. Thus, identifiers cannot be used to index shared registers.

Designing symmetric algorithms is especially important, when the designed algorithms are intended to be used as a building blocks in an environment where the processes' name space is not known in advance. Most of the algorithms presented in this paper are symmetric.

1.3 Fault Tolerance

For the rest of the paper, n denotes the number of processes, t denotes the number of faulty processes, and $N = \{0, 1, ..., n\}$.

> **Definition:** For a given function $f : N \to N$, an algorithm is (t, f)-*resilient* if in the presence of t' faults at most $f(t')$ participating correct processes may *not* terminate their operations, for every $0 \leq t' \leq t$.

It seems that (t, f)-*resiliency* is interesting only when requiring that $f(0) = 0$. That is, in the absence of faults all the participating processes must terminate. The standard definition of t-resiliency is equivalent to (t, f)-resiliency where $f(t') = 0$ for every $0 \leq t' \leq t$. Thus, the familiar notion of *wait-freedom* is equivalent to $(n-1, f)$-resiliency where $f(t') = 0$ for every $0 \leq t' \leq n-1$. The new notion of (t, f)-resiliency is quite general, and in this paper we focus mainly on the following three levels of resiliency.

- An algorithm is *almost-t-resilient* if it is (t, f)-*resilient*, for a function f where $f(0) = 0$ and $f(t') = 1$, for every $1 \leq t' \leq t$. Thus, in the presence of any number of up to t faults, all the correct participating processes, except maybe one process must terminate.
- An algorithm is *partially-t-resilient* if it is (t, f)-*resilient*, for a function f where $f(0) = 0$ and $f(t') = t'$, for every $1 \leq t' \leq t$. Thus, in the presence of any number $t' \leq t$ faults, all the correct participating processes, except maybe t' of them must terminate.
- An algorithm is *weakly-t-resilient* if it is (t, f)-*resilient*, for a function f where $f(0) = 0$, and in the presence of any number of up to $t \geq 1$ faults, if there are *two* or more correct participating processes, then one correct participating process must terminate. (Notice that for $n = 2$, if one process fails the other one is not required to terminate.)

These new definitions are also interesting from a theoretical point of view and enable a better understanding of fault tolerance. For $n \geq 3$ and $t < n-2$, the notion of weakly-t-resiliency is strictly weaker than the notion of partially-t-resiliency. For $n \geq 3$, the notion of weakly-t-resiliency is strictly weaker than the notion of almost-t-resiliency. For $n \geq 3$ and $t \geq 2$, the notion of partially-t-resiliency is strictly weaker than the notion of almost-t-resiliency. For all n, partially-1-resiliency and almost-1-resiliency are equivalent. For $n = 2$, these three notions are equivalent.

We say that an algorithm is *almost-wait-free* if it is *almost-*$(n-1)$*-resilient*, thus, in the presence of any number of faults, all the participating correct processes, except maybe one process must terminate. We say that an algorithm is *partially-wait-free* if it is *partially-*$(n-1)$*-resilient*, thus, in the presence of any number of $t \leq n-1$ faults, all the correct participating processes, except maybe t of them must terminate. We say that an algorithm is *weakly-wait-free* if it is *weakly-*$(n-1)$*-resilient*, thus, in the presence of any number of faults, if there are two or more correct participating processes then one correct participating processes must terminate.

In an asynchronous shared memory system which supports atomic registers or in a message passing system, important problems such as consensus, set-consensus, election, perfect renaming, implementations of a test-and-set bit, a shared stack, a swap object and a fetch-and-add object, have no solutions which can tolerate even a single fault. Rather surprisingly, as we will show later, while some of these problems have solutions which satisfy almost-wait-freedom, other problems do not even have weakly-1-resilient solutions.

1.4 Contributions

New Definitions. We generalize the traditional notion of fault tolerance. Together with the technical results, the new definitions provide a deeper understanding of complexity and computability issues which are involved in the development of fault-tolerant algorithms.

Election. In this problem one or more processes independently initiate their participation in an election to decide on a leader. Each participating process should eventually output either 0 or 1 and terminate. At most one process may output 1, and in the absence of faults exactly one of the *participating* processes should output 1. The process which outputs 1 is the elected leader. It is known that there is no 1-resilient election algorithm, when processes communicate either by reading and writing atomic registers or by sending and receiving messages. We show that:

> (1) *There is an almost-wait-free symmetric election algorithm using* $\lceil \log n \rceil + 2$ *atomic registers.* (2) *There is an almost-wait-free symmetric election algorithm with* $n^2 - n$ *message complexity.*

Message complexity is the total number of message sent. The known space lower bound for election in the absence of faults is $\lceil \log n \rceil + 1$ atomic registers [38].

Test-and-set. A test-and-set bit is an object that supports two operations, called *test-and-set* and *reset*. A test-and-set operation on a single bit takes as argument a shared bit b, assigns the value 1 to b, and returns the previous value of b (which can be either 0 or 1). A reset operation takes as argument a shared registers b and writes the value 0 into b. We show that:

> (1) *There is an almost-wait-free symmetric implementation of a test-and-set bit for* n *processes using* $n+1$ *atomic registers.* (2) *Any implementation of a test-and-set bit for* n *processes using registers must use at least* n *registers, even in the absence of faults.*

It is known that in asynchronous systems where processes communicate using atomic registers there are no 1-resilient implementations of a test-and-set bit [31].

Perfect Renaming. A *perfect* renaming algorithm allows n processes with initially distinct names from a large name space to acquire distinct new names from the set $\{1, ...n\}$. A *one-shot* renaming algorithm allows each process to acquire a distinct new name just once. A *long-lived* renaming algorithm allows processes to repeatedly acquire distinct names from a small name space and release them. We show that in a shared memory model:

> (1) *There is a partially-wait-free symmetric one-shot perfect renaming algorithm using either* $n-1$ *almost-wait-free election objects* or $O(n \log n)$ *registers.* (2) *There is a partially-wait-free symmetric long-lived perfect renaming algorithm using either* $n-1$ *almost-wait-free test-and-set bits* or $O(n^2)$ *registers.*

It is known that in asynchronous systems where processes communicate either by atomic registers or by sending and receiving messages, there is no 1-resilient perfect renaming algorithm [5, 33, 42].

Fetch-and-add, swap, stack. A *fetch-and-add* object supports an operation which takes as arguments a shared register r, and a value *val*. The value of r is incremented by *val*, and the old value of r is returned. A *swap* object supports an operation which takes as arguments a shared registers and a local register and atomically exchange their values. A shared stack is a linearizable object that supports push and pop operations, by several processes, with the usual stack semantics. We show that:

> *There are partially-wait-free implementations of a fetch-and-add object, a swap object, and a stack object using atomic registers.*

The result complements the results that in asynchronous systems where processes communicate using registers there are no 1-resilient implementations of fetch-and-add, swap, and stack objects [11, 23].

Consensus and Set-consensus. The *k-set consensus* problem is to find a solution 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. We show that:

> (1) *For $n \geq 3$ and $1 \leq k \leq n-2$, there is no weakly-k-resilient k-set-consensus algorithm using either atomic registers or sending and receiving messages. In particular, for $n \geq 3$, there is no weakly-1-resilient consensus algorithm using either atomic registers or messages.* (2) *For $n \geq 3$ and $1 \leq k \leq n-2$, there is no weakly-k-resilient k-set-consensus algorithm using almost-wait-free test-and-set bits and atomic registers.*

Our results strengthen the know results that, in asynchronous systems where processes communicate either by atomic registers or by sending and receiving messages, there is no 1-resilient consensus algorithm [21, 31], and there is no k-resilient k-set-consensus algorithm [10, 24, 37].

The table below summarizes the results discussed in this paper. We use the following abbreviations: "SM" for shared memory, "MP" for message passing, "Yes" means that it is possible to solve the problem, and "No" means that it is impossible to solve the problem. In the shared memory model, the (space) complexity bounds are the number of atomic registers required. In the message passing model, the complexity bounds are the number of messages used.

Problem	Model	Results				
		Weakly wait-free	Partially wait-free	Almost wait-free	Complexity	
					upper bound	lower bound
Election	SM symmetric	Yes	Yes	Yes	$\lceil \log n \rceil + 2$	$\lceil \log n \rceil + 1$ from [38]
Election	MP symmetric	Yes	Yes	Yes	$n^2 - n$	
Test-and-set	SM symmetric	Yes	Yes	Yes	$n + 1$	n
Perfect renaming one-shot	SM symmetric	Yes	Yes		$O(n \log n)$	
Perfect renaming long-lived	SM symmetric	Yes	Yes		$O(n^2)$	
Stack, Swap, Fetch-and-add	SM asymmetric	Yes	Yes			
Consensus, Set-consensus	SM/MP	No	No	No		

2 Almost-wait-free Symmetric Election

In the leader election problem, processes do not have inputs. Each participating process should eventually output either 0 or 1 and terminate. At most one process may output the value 1, and in the absence of faults exactly one of the participating processes should output 1. The process which outputs 1 is elected as a leader. It is not require that the processes know the identity of the leader. The elected leader must be one of the participating processes, thus, there can not be an a priori leader. Notice that, in the presence of faults, it is *not* required in the above definition that a leader must eventually be elected.

In asynchronous systems where processes communicate either using atomic registers or by sending and receiving messages, election is impossible with one faulty process [21, 33, 42]. We show below that almost-wait-free symmetric election is possible in such asynchronous systems. This possibility result for election is later used for solving perfect renaming. We point out that, it follows from the results presented in Section 6 for the consensus problem, that for a stronger definition of election in which it is required that the processes know (i.e., output) the identity of the leader, even weakly-1-resilient strong-election is impossible.

2.1 Election using atomic registers

In [38], an election algorithm which is *not* weakly-1-resilient is presented. It is correct under the following assumptions: (1) processes never fail, and (2) only the elected leader is required to terminate. The election algorithm presented below, is based on the algorithm from [38].

Theorem 2.1 *There is an almost-wait-free symmetric election algorithm using* $\lceil \log n \rceil + 2$ *atomic registers.*

The algorithm below is for n processes each with a unique identifier taken from some (possibly infinite) set which does not include 0. The algorithm uses the shared registers *turn* and *done* and the array of registers V with $\lceil \log n \rceil$ entries indexed 1 through $\lceil \log n \rceil$. All these registers are initially 0. Also, for each process, the local variables *level* and j are used. We denote by *e.turn*, *e.done* and *e.V*[*] the shared registers of the specific election algorithm (object) named e. This should simplify

the construction of algorithms that use election as a basic building block.

AN ALMOST-WAIT-FREE SYMMETRIC ELECTION: `process p's program.`

```
function election (e: object_name) return:value in {0, 1};          /* access election object e */
1   e.turn := p;
2   for level := 1 to ⌈log n⌉ do
3       repeat
4           if e.done = 1 then return(0) fi;                          /* not the leader */
5           if e.turn ≠ p then
6               for j := 1 to level − 1 do if e.V[j] = p then e.V[j] := 0 fi od;
7               return(0) fi                                          /* not the leader */
8       until e.V[level] = 0;
9       e.V[level] := p;
10      if e.turn ≠ p then
11          for j := 1 to level do if e.V[j] = p then e.V[j] := 0 fi od;
12          return(0) fi                                              /* not the leader */
13  od;
14  e.done := 1; return(1).                                           /* leader! */
end_function
```

A process becomes the leader if it manages to write its id into all the registers during the period that *e.turn* equals its id. Any process that notices that *e.turn* is no longer equals its id, gives up on becoming the leader, and erases any write it has made (lines 6 & 11).

The register *done* is used to guarantee that in the absence of faults, all the participating processes will terminate. In the absence of faults, the elected leader will set *done* to 1, enabling all the other processes to observe it and terminate. Without the register *done*, we are back to the solution from [38] which, even in the absence of faults, only guarantees that the elected leader terminates.

There are runs of the algorithm in which every process manages to set $\lceil \log n \rceil$ registers before discovering that another process has modified *e.turn*, and as a result has to set back to 0 some of the registers before terminating. Proving the correctness of the algorithm is rather challenging, due to the existence of such runs.

In [38], it has been proven that, even in the absence of faults, any election algorithm for n processes must use at least $\lceil \log n \rceil + 1$ registers. (This lower bound holds even for non-symmetric algorithms.) Thus, our algorithm which uses $\lceil \log n \rceil + 2$ registers, provides an almost tight space upper bound.

2.2 A correctness proof for the election algorithm

The proof of the election algorithm is an adaptation of the proof for the algorithm from [38] which guarantees that only the leader terminates, and is correct only in the absence of faults. The fact that our election algorithm uses $\lceil \log n \rceil + 2$ atomic registers is obvious from inspecting the algorithm.

Theorem 2.2 (liveness) *In the absence of faults, at least one leader is elected.*

Proof: First we observe that there is no finite run in which all the n processes return 0 and terminate. This is so, because in each run, the last process to write into *turn* never returns 0, unless some other process sets *done* to 1 and returns 1.

Assume to the contrary that no leader is elected. Let r be an infinite run with no faults where no leader is elected, and let p be the last processes to write to *turn* in run r. Let q be the process with the highest value of *level* when p writes to *turn*. At some point q will notice that $turn \neq q$, and will set back to 0, all the entries of the array V which equal to q. Repeat this argument with the new highest process. Thus, any entry of the array V which process p may wait on, will eventually be set back to 0, enabling p to proceed until it is elected. A contradiction. ∎

We say that a process is at level k, when the value of its private *level* register is k. We say that a group of processes P have noticed *together* that $V[k] = 0$, if each process in P: (1) is at level k, and (2) has notice that $V[k] = 0$ (when executing the until statement in line 8), before any other process in P has written $V[k]$ (by executing the assignment in line 9).

Lemma 2.3 *For any $k \in \{1, ..., \lceil \log n \rceil\}$ and for any group of processes P, if the processes P have noticed together that $V[k] = 0$ then at most one process in P can either (1) continue to level $k + 1$ or (2) change any register other than $V[k]$.*

Proof: Assume that a set of processes, denoted P, are at level k, and they have noticed together that $V[k] = 0$. One of these processes, say $p \in P$, must be the last to update *turn*. If $k = 1$, each process in $P - \{p\}$ will notice that *turn* is different from its id (line 10), possibly write 0 into $V[1]$ (line 11), and return 0 (line 12).

Assume $k > 1$. To reach level k, each process in P must have seen in all the levels smaller then k that *turn* is equal to its id. Thus, *before* p has set *turn* to its id, each of the other processes in P, must have seen in all the levels smaller then k that *turn* is equal to its id.

Since the processes in P have noticed together that $V[k] = 0$, by definition, it must be the case that before any of the processes in $P - \{p\}$ could execute the assignment at line 9, p has already set $V[1], ..., V[k - 1]$ to its id. This implies that by the time each process in $P - \{p\}$ executes the statement in line 9, the following two conditions hold: (1) *turn* is different from its id, and (2) the values of the registers $V[1], ..., V[k - 1]$ are all different from its id. Thus, by the time each process in $P - \{p\}$ executes the if statement in line 10 it finds out that *turn* is different from its id, possibly writes 0 into $V[k]$ (line 11), and returns 0 (line 12), without a need to write 0 to any of the registers $V[1], ..., V[k - 1]$. Process p, may continue to level $k + 1$ or itself notices that $turn \neq p$ and sets some or all of the registers $V[1], ..., V[k - 1]$ to 0, but it is the only process, among the processes in P, that may set any shared register other than $V[k]$. ∎

Theorem 2.4 (safety) *At most one leader is elected.*

For proving the theorem, an accounting system of credits is used. Initially, the number of credits is $2n - 1$. New credits can not be created during the execution of the algorithm. The credit system ensures that a process acquires exactly 2^{k-1} credits before it can reach level k. Being elected is equivalent to reaching level $\log n + 1$. Thus, the credit system ensures that a process must acquire $2^{\log n+1-1} = n$ credits before it can be elected. Once a process is elected, it may not release any of its credits. Thus, it is not possible for two processes to get elected.

Without loss of generality it is assumed that n, the number of processes, is a power of 2. Initially, each process holds 1 credit, and each register $V[k]$ where $1 \leq k \leq \log n$ holds 2^{k-1} credits. Thus, the total number of credits is $n + \sum_{k=1}^{\log n} 2^{k-1} = 2n - 1$. As a results of an operation taken by a process credits may be transferred from a register to a process and vice versa, and between processes. We list below 4 rules which capture all possible operations by processes and their effect:

1. **No transfer of credits:** No credits are transferred when a process (1) checks the value of a register, (2) writes into *turn*, or (3) executes a *return* statement.

2. **Transferring credits between a register and a process:** When a process writes its id into register $V[k]$, changing $V[k]$'s value from 0 to its id, 2^{k-1} credits are transferred from $V[k]$ to that process. When a process writes 0 into register $V[k]$ which does not already holding 0, 2^{k-1} credits are transferred to $V[k]$ from that process.

   Remark: This is the only rule for transferring credits between a register and a process. Initially, $V[k]$ holds 2^{k-1} credits, so the first time $V[k]$'s value changes, it has enough credits to transfer. Before any subsequent transfer from $V[k]$ to a process, its value has to be set back to 0, and each time this happens $V[k]$ gets back 2^{k-1} credits. So, $V[k]$ always has enough credits to transfer to a process that changes $V[k]$'s value from 0 to its id (line 9).

   A process at level k may changes $V[k]$'s value back to 0 at most once (line 11). Under the assumption, which we justify later, that the credit system ensures that a process acquires exactly 2^{k-1} credits before it can reach level k, and that these 2^{k-1} credits are not used for something else, a process always has enough credits to transfer to $V[k]$ if it changes $V[k]$'s value to 0.

3. **Transferring credits between processes when moving to an upper level:** Let P be a maximal[1] set of processes that have noticed together that $V[k] = 0$. By Lemma 2.3, at most one process from P can continue to level $k + 1$. Assume process $p \in P$ continues to level $k + 1$. We consider two cases:

   - At level k, when executing line 9, process p changes $V[k]$'s value from 0 to its id. By rule 2, 2^{k-1} credits are transferred from $V[k]$ to p. Thus, p has 2^k credits available, 2^{k-1} credits from reaching level k, plus 2^{k-1} credits from $V[k]$, giving p the total of 2^k credits it needs for level $k + 1$.
   - At level k, when executing line 9, process p does not change $V[k]$'s value from 0 to its id. This implies that there must be another process $q \in P$ that, before p has executed line 9, was the last process to change $V[k]$'s value back to 0. By rule 2, 2^{k-1} credits are transferred from $V[k]$ to q. Thus, q has 2^k credits available, 2^{k-1} credits from reaching level k, plus 2^{k-1} credits from $V[k]$. In this case, immediately after p executes line 9, 2^{k-1} credits are transferred from q to p, leaving q with 2^{k-1} credits and giving p the total of 2^k credits (2^{k-1} credits from reaching level k, plus 2^{k-1} credits from q) it needs for level $k + 1$.

   By Lemma 2.3, each process in P (including q) that does not continue to the level $k + 1$ can only execute $V[k] := 0$ (line 11), transferring (by Rule 2) to $V[k]$ the 2^{k-1} credits it has by getting this far, if it succeeds in changing $V[k]$'s value back to 0.

4. **Transferring credits between processes without moving to an upper level:** Let P be a maximal set of processes that have noticed together that $V[k] = 0$, and assume that no process in P continues to level $k + 1$. By Lemma 2.3, at most one process in P, say process p, can change any register other than $V[k]$. We consider two cases:

   - At level k, when executing line 9, process p changes $V[k]$'s value from 0 to its id. By rule 2, 2^{k-1} credits are transferred from $V[k]$ to p, Thus, p has 2^k credits available, 2^{k-1} credits from reaching level k, plus 2^{k-1} credits from $V[k]$.

[1] A set of processes P is maximal with respect to property ϕ, if (1) P satisfies ϕ, and (2) there is not set Q, such that $P \subset Q$ and Q satisfies ϕ.

- At level k, when executing line 9, process p does not change $V[k]$'s value from 0 to its id. This implies that there must be another process $q \in P$ that, before p has executed line 9, was the last process to change $V[k]$'s value back to 0. By rule 2, 2^{k-1} credits are transferred from $V[k]$ to q, giving q a total of 2^k credits. In this case, immediately after p executes line 9, 2^{k-1} credits are transferred from q to p, leaving q with 2^{k-1} credits, and giving p a total of 2^k credits (2^{k-1} credits from reaching level k, plus 2^{k-1} credits from q).

  Setting to 0 every variable from $V[1]$ to $V[k]$ accounts for $2^k - 1$ credits (i.e., $\sum_{i=1}^{k} 2^{i-1} = 2^k - 1$), so (in both cases) p has enough credits and no new credits should be created by p when it sets to 0 multiple registers.

  By Lemma 2.3, each process in $P - \{p\}$ (including q) can only execute $V[k] := 0$ (line 11), transferring (by Rule 2) to $V[k]$ the 2^{k-1} credits it has by getting this far, if it succeeds in changing $V[k]$'s value back to 0.

Given the above description of the accounting system, we can now justify the following two claims made earlier:

1. *A process acquires exactly 2^{k-1} credits before it can reach level k.*
   This is proven by induction on the level k. For $k = 1$, the claim follows immediately from the fact that initially each process has one credit. We assume that the claim holds for level k and prove that it also holds for level $k + 1$. By Rule 3, before process p moves to level $k+1$, it gets additional 2^{k-1} credits either from $V[k]$ or from another process. Thus, p has 2^k credits available, 2^{k-1} credits by the induction hypothesis (from reaching level k) plus 2^{k-1} as explained above, giving p the total of 2^k credits it needs for level $k + 1$.

2. *No new credits are created.*
   As already explained in Rule 2, $V[k]$ always has enough credits to transfer to a process that changes $V[k]$'s value from 0 to its id. Furthermore, since the credit system ensures that a process acquires exactly 2^{k-1} credits before it can reach level k, and since these 2^{k-1} credits are not used for something else, a process at level k always has enough credits to transfer to $V[k]$ if it changes $V[k]$'s value to 0.

   By Lemma 2.3, at most one process in a maximal set of processes P that have noticed together that $V[k] = 0$, say process p, can change any register other than $V[k]$. By Rule 4, p has 2^k credits available, 2^{k-1} credits from reaching level k plus 2^{k-1} from either $V[k]$ or from another process. As already explained in Rule 4, setting to 0 every variable from $V[1]$ to $V[k]$ accounts for $2^k - 1$ credits (i.e., $\sum_{i=1}^{k} 2^{i-1} = 2^k - 1$), so (in both cases) p has enough credits and no new credits should be created by p when it sets to 0 multiple registers.

As already mentioned, initially, the number of credits is $2n - 1$. No new credits are created, and a process must acquire n credits before it can be elected. Once a process is elected, it may not release any of its credits. Thus, it is not possible for two processes to get elected. ∎

Theorem 2.5 (almost-wait-freedom) *In the absence of faults, every participating process eventually terminates. In the presence of faults, every correct participating process, except maybe one, eventually terminates.*

Proof: Once a leader is elected and returns, all correct participating processes will eventually find out that $done = 1$ and terminate. In particular, in the absence of faults, since by Theorem 2.2 at least one leader is eventually elected, all the participating processes will terminate. Also, regardless of the number of faults, a correct process which is not the last to write into *turn*, will eventually either (1) notices this fact, returns 0 and terminates or (2) it will reach line 14, return 1 (i.e., be elected) and terminates. Thus, in the presence of faults, only the last correct process to write into *turn* may be blocked (as a result of a faulty process not setting one of the entries of V back to 0), in which case no leader will be elected. ∎

2.3 Election in a message passing system

We present a simple election algorithm in which the process with the maximum identifer is elected.

Theorem 2.6 *There is an almost-wait-free symmetric election algorithm with $n^2 - n$ message complexity.*

Proof: In the algorithm each process sends its identifer to every other process, and collects, through the messages seen, identifiers of other processes. As soon as a process collects an identifer which is bigger than itself it returns 0. If a process collects the identifiers of all the other $n - 1$ processes, and finds out that it is the process with the maximum identifer, it returns 1. In the code below $my.id$ refers to the identifier of the process executing the algorithm, and $message.val$ refers to the value of the message received. Each process has a local *counter* variable which is initially set to 0.

ALMOST-WAIT-FREE SYMMETRIC ELECTION ALGORITHM:
program for a process with identifier $my.id$.

```
1 send my.id to all the other processes;
2 each time a message is received do
3    if my.id < message.val then return(0) else counter := counter + 1 fi;
4    if counter = n − 1 then return(1) fi                          /* leader! */
5 od
```

Clearly, in the absence of faults exactly one process is elected and it is always the process with the maximum identifier. In the presence of faults, only the correct participating process with the maximum identifier *among* the currently participating processes may not terminate, all the other processes will get a message from it, return 0 and terminate. The message complexity is $n^2 - n$, since each process sends one message to each other process. ∎

3 Almost-wait-free Symmetric Test-and-set Bit

We show that n registers are necessary and $n + 1$ registers are sufficient for implementing a single almost-wait-free test-and-set bit using registers for n processes. A test-and-set bit supports two atomic operations, called *test-and-set* and *reset*. A test-and-set operation takes as argument a shared bit b, assigns the value 1 to b, and returns the previous value of b (which can be either 0 or 1). A reset operation takes as argument a shared bit b and writes the value 0 into b.

The *sequential specification* of an object specifies how the object behaves in sequential runs, that is, in runs when its operations are applied sequentially. The sequential specification of a test-and-set bit is quite simple. In sequential runs, the first test-and-set operation returns 0, a test-and-set operation that happens immediately after a reset operation also returns 0, and all other test-and-set operations return 1. We require that, although operations of processes may overlap, each operation should appear to take effect instantaneously. In particular, operations that do not overlap should take effect in their "real-time" order. This correctness requirement is called *linearizability* [25].

3.1 An upper bound

The algorithm below is for n processes each with a unique identifier taken from some (possibly infinite) set which does not include 0. It makes use of exactly n registers which are long enough to store a process identifier and one atomic bit. The algorithm is based on the symmetric mutual exclusion algorithm presented in [38].

Theorem 3.1 *There is an almost-wait-free symmetric algorithm which implements a test-and-set bit using atomic registers. The algorithm is for n processes and uses $n+1$ atomic registers.*

The algorithm uses a register, called *turn*, to indicate who has priority to return 1, $n-1$ *lock* registers to ensure that at most one process will return 1 between resets, and a bit, called *winner*, to indicate whether some process already returned 1. Initially the values of all these shared registers are 0. In addition, each process has a private boolean variable, called *locked*. We denote by *b.turn*, *b.winner* and *b.lock*[*] the shared registers for the implementation of a specific test-and-set *bit*, named b.

AN ALMOST-WAIT-FREE SYMMETRIC TEST-AND-SET BIT: `process` p`'s program.`

```
function test-and-set (b:bit) return:value in {0,1};                    /* access bit b */
1   if b.turn ≠ 0 then return(0) fi;                                    /* lost */
2   b.turn := p;
3   repeat
4         for j := 1 to n − 1 do                                        /* get locks */
5               if b.lock[j] = 0 then b.lock[j] := p fi od
6         locked := 1;
7         for j := 1 to n − 1 do                                        /* have all locks? */
8               if b.lock[j] ≠ p then locked := 0 fi od;
9   until b.turn ≠ p or locked = 1 or b.winner = 1;
10  if b.turn ≠ p or b.winner = 1 then
11        for j := 1 to n − 1 do                                        /* lost, release locks */
12              if b.lock[j] = p then b.lock[j] := 0 fi od
13        return(0) fi;
14  b.winner := 1; return(1).                                           /* wins */
end_function

function reset (b:bit);                                                 /* access bit b */
1   b.winner := 0; b.turn := 0;                                         /* release locks */
2   for j := 1 to n − 1 do
3         if b.lock[j] = p then b.lock[j] := 0 fi od.
end_function
```

In the test-and-set operation, a process, say p, initially checks whether *b.turn* $\neq 0$, and if so returns 0. Otherwise, p takes priority by setting *b.turn* to p, and attempts to obtain all the $n-1$ locks by setting them to p. This prevents other processes that also saw $b.turn = 0$ and set *b.turn* to their ids from entering. That is, if p obtains all the locks before the other processes set *b.turn*, they will not be able to get any of the locks since the values of the locks are not 0. Otherwise, if p sees $b.turn \neq p$ or $b.winner = 1$, it will release the locks it holds, allowing some other process to proceed, and will return 0. In the reset operation, p sets *b.turn* to 0, so the other processes can proceed, and releases all the locks it currently holds.

3.2 A correctness proof for the test-and-set algorithm

We prove that our implementation is linearizable w.r.t. the sequential specification of a test-and-set bit mentioned earlier. For that it is enough to prove the following theorems. We say that run is *well structured*, if in that run a reset operation may be initiated only by a process that its last operation (before applying the reset operation) is a test-and-set operation which has returned 0. We say that a process is a *winner* in a given finite run, if the *last* completed operation of that process in the run is a test-and-set operation which has returned 0.

Theorem 3.2 (safety) *There is at most one winner in any well structured run.*

Proof: Assume some process p is a winner. We show that no other process can become a winner before p preforms a reset operation. When process p last accessed *turn* and the $n-1$ locks, the value of each of these n shared registers was p. Any other process has to set all the $n-1$ locks and see *turn* set to its value for it to become a winner. But a process always checks a lock before writing it, and can only change one lock which has been already set (and not released yet) by some other process. So if all the n shared registers have the value p, and each of the remaining $n-1$ processes can overwrite at most one such register, at least one shared register must still hold the value p, preventing processes other than p from becoming winners. ∎

We say that a pending test-and-set operations is *potentially successful* if no process has become a winner since the operation was issued.

Theorem 3.3 (liveness) *In the absence of faults, at least one process will eventually become a winner, in any given run with potentially successful pending test-and-set operations.*

Proof: Assume to the contrary that no process will become a winner. Since no process becomes a winner, *turn* is not set back to 0, and hence *turn* must eventually have a nonzero value, say p, and this value will not change thereafter. Every participating process other than p will eventually notice $turn = p$, it will release the locks it holds, will return 0 and thereafter will not update any other registers because *turn* is not zero. At this point, since process p always finds $turn = p$, nothing is preventing process p from getting all the locks and becoming a winner. A contradiction. ∎

Theorem 3.4 (almost-wait-freedom) *In the absence of faults, every participating process (i.e, pending operation) eventually returns. In the presence of faults, every correct participating process, except maybe one, eventually returns.*

Proof: Once some process becomes the winner and returns 1, as long as the winner does not initiate a reset operation, all correct participating processes will eventually find out that $done = 1$ and return 0. In particular, in the absence of faults, since by Theorem 3.3 at least one process will eventually become the winner, all the participating processes will return. Also, regardless of the number of faults, a correct process which is not the last to write *turn*, will eventually either notice this fact and return 0 or becomes the winner and returns 1. Thus, in the presence of faults, only the last process to write *turn* may be blocked. ∎

3.3 A lower bound

We show that the $n + 1$ space upper bound is almost tight.

Observation 3.5 (lower bound) *Even in the absence of faults, any implementation of a test-and-set bit for n processes using atomic registers must use at least n atomic registers.*

Proof: In [15, 17], it is proven that any deadlock-free mutual exclusion algorithm for n processes must use at least n shared registers. On the other hand, it is trivial to implement a deadlock-free mutual exclusion algorithm for n processes using a single test-and-set bit, say x, as follows: A process first keeps on accessing x until, in one atomic step, it succeeds to change x from 0 to 1. Then, the process can safely enter its critical section. The exit code is to reset x to 0. It is trivial to show that the algorithm satisfies mutual exclusion and is deadlock-free. The result follows. ∎

4 Partially-wait-free Symmetric Perfect Renaming

A *renaming* algorithm allows processes with initially distinct initial names from a large name space to acquire distinct new names from a small name space. A *perfect* renaming algorithm allows n processes with initially distinct names from a large name space to acquire distinct new names from the set $\{1, ...n\}$. A *one-shot* renaming algorithm allows each process to acquire a distinct new name just once. A *long-lived* renaming algorithm allows processes to repeatedly acquire distinct names and release them (however, once a process acquires a new name it must first release it before trying to acquire another one).

It is well known that, in asynchronous systems where processes communicate by reading and writing atomic registers there is no 1-resilient perfect renaming algorithm [5, 33, 42]. Contrary to this impossibility result, we show that there is a partially-wait-free perfect renaming algorithm in such a shared memory systems. A *partially-wait-free* renaming algorithm, should guarantee that t failures, where $1 \leq t \leq n-1$, may prevent at most t correct participating processes from acquiring new names.

Theorem 4.1 *There is a partially-wait-free symmetric one-shot perfect renaming algorithm using either $n - 1$ almost-wait-free election objects or $O(n \log n)$ registers.*

Proof: First we present an algorithm which uses $n - 1$ almost-wait-free election objects. The election objects are indexed 1,2,....,$n - 1$. Each process scans the objects, in order, starting with object number 1. At each step, the process applies the election operation, and either: moves to the next object if it is not elected in object $i < n - 1$, stops if it is being elected, or stops if it not elected

in object $n-1$. The process is assigned either the name equal to the index of the object on which its election operation has succeeded, or n if it is not elected in all $n-1$ objects. Notice that at most $n-i+1$ processes may participate in object i, for $1 \leq i \leq n-1$. Thus, by Theorem 2.1, the almost-wait-free, election object indexed i, where $1 \leq i \leq n-1$, can be implemented using $\lceil \log(n-i+1) \rceil + 2$ atomic registers. Thus, the number of registers used are at most:

$$3(n-1) + \sum_{i=2}^{n} \log i = 3(n-1) + \log n! = O(n \log n).$$

The result follows. ∎

Theorem 4.2 *There is a partially-wait-free symmetric long-lived perfect renaming algorithm using either $n-1$ almost-wait-free test-and-set bits or $O(n^2)$ atomic registers.*

Proof: First we present an algorithm which uses $n-1$ almost-wait-free test-and-set bit bits. The bits have initial values 0, and are indexed 1,2,....,$n-1$. Each process scans the bits, in order, starting with bit number 1. At each step, the process applies a *test-and-set* operation, and either: moves to the next bit if the returned value is 1 in bit $i < n-1$, stops when the returned value is 0, or stops if the returned value is 1 in bit $n-1$. The process is assigned the name equal to the index of the bit on which its (last) *test-and-set* operation returned 0, or n if the returned value is 1 in all $n-1$ bits. A process which is assigned the name i can later release this name by applying a *reset* operation to the i'th bit setting its value back to 0. A process which is assigned the name n doesn't have to access any shared bit to release the name n. At most $n-i+1$ processes may concurrently access the bit indexed i, for $1 \leq i \leq n-1$. Thus, by Theorem 3.1, the bit indexed i, where $1 \leq i \leq n-1$, can be implemented using $n-i+2$ registers. Thus, the number of registers used are:

$$\sum_{i=2}^{n}(i+1) = \frac{n^2+3n-4}{2}.$$

The result follows. ∎

5 Partially-wait-free fetch-and-add, swap, and stack objects

A *fetch-and-add* object supports one operation, which takes as arguments a shared register r, and a value *val*. The value of r is incremented by *val*, and the old value of r is returned. A *swap* object supports one operation, which takes as arguments a shared registers and a local register and atomically exchange their values. A concurrent *stack* is a linearizable object that supports push and pop operations, by several processes, with the usual stack semantics. A *sequential* process is a process that has at most one pending operation at any given time.

Lemma 5.1 *Assume that there is a wait-free implementation for n sequential processes of an object o using wait-free test-and-set bits and atomic registers. Then, there is a partially-wait-free implementation for n processes of o using atomic registers only.*

Proof: Let A be a wait-free implementation for n sequential processes of an object o using k wait-free test-and-set bits, denoted $TS_1, ..., TS_k$ and registers. Let A' be the implementation A where each wait-free test-and-set bit TS_i, where $i \in \{1, ..., k\}$, is replaced with an implementation of an almost-wait-free test-and-set bit, denoted $TS_i^{'}$, from atomic registers. Recall that by Theorem 3.1, it is possible to implement almost-wait-free test-and-set bit using atomic registers.

While executing A', a failure of a process, say p, may prevent at most one correct process from completing its operation in A'. To see that, consider two cases: (1) the failure occurred during the execution of an operation, by p, on some almost-wait-free test-and-set bit $TS_i^{'}$ ($i \in \{1, ..., K\}$), and (2) the failure occurred while p did not execute an operation on some almost-wait-free test-and-set bit. In the first case, by definition of almost-wait-freedom, the failure of p may prevent at most one correct process from completing its operation on $TS_i^{'}$ and has no effect on the other processes. In particular, the failure of p has no effect on processes executing an operation on $TS_j^{'}$, where $i \neq j$. In the second case, since the original algorithm A is a wait-free, such a failure would not prevent any correct process from completing its operation in A'.

Thus, a failure of t processes may prevent at most t correct processes from completing their operations. This implies that A' is a partially-wait-free implementation of o using almost-wait-free test-and-set bits and registers. The result follows. ∎

Theorem 5.2 *There are partially-wait-free implementations for n processes of a fetch-and-add object, a swap object, and a stack object using atomic registers only.*

Proof: In [3], a class of shared objects, called Common2, were defined. Each object in Common2 is known to have a wait-free implementation from registers together with any other object in Common2, for an arbitrary number of sequential processes. Commonly used primitives such as test-and-set, fetch-and-add, swap, and stack are in Common2 [2, 3]. Thus, any of the objects in Commom2 has a wait-free implemention using registers and wait-free test-and-set bits, for arbitrary number of sequential processes. (The implementations presented in [3] are not symmetric.) This last observation together with Lemma 5.1 implies that there are partially-wait-free implementations for n processes of a fetch-and-add object, a swap object, and a stack object using atomic registers only. ∎

6 Impossibility Results for Consensus and Set-consensus

The *k-set consensus* problem is to find a solution 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 for n processes.

The consensus and set-consensus problems belong to a class of problems called *colorless tasks*. Colorless tasks (also called convergence tasks [11]) allow a process to adopt an input or output value of any other participating process, so the task can be defined in terms of input and output sets instead of vectors.

For proving the following lemma we need to assume a model where participation is required. Recall that with required participation every process must eventually execute its code.

Lemma 6.1 *Assume a model where participation is required, $n \geq 3$ and $t \leq n-2$. When processes communicate either by reading and writing atomic registers or by sending and receiving messages, for any colorless task T: there is a weakly-t-resilient algorithm which solves T if and only if there is a t-resilient algorithm which solves T.*

Proof: Let A be a weakly-t-resilient algorithm using atomic registers which solves T. We use A to implement a t-resilient algorithm, called A', which uses atomic registers and solves T. An additional shared register called *output* is used, which has initial value $\perp$. Every process executes as in A, and before it terminates it writes its output into *output*. During its execution of A, a process also continuously checks whether *output* $\neq \perp$, and in case the test is positive, it adopts the value of *output* as its own output value and terminates. Since participation is required, $n \geq 3$ and $t \leq n-2$, one correct process will eventually terminate. Once one correct process writes its output into *output*, it is guaranteed that each participating correct will eventually either terminates according its code in A, or will notice that *output* $\neq \perp$, and terminates. The resulting algorithm is A'. Proving the other direction is trivial. The proof for the case where communication is by sending and receiving messages is almost the same. Instead of writing to *output*, a process sends its decision to everyone before terminating. Each process that receives a message with such a decision value, decides on that value, sends it to everyone and terminates. ∎

The following results hold for a model where participation is required, and thus also hold for a model where participation is not required.

Theorem 6.2 *For $n \geq 3$, there is no weakly-1-resilient consensus algorithm using either reading and writing atomic registers or sending and receiving messages.*

Proof: The proof follows from Lemma 6.1 and the known result that there is no 1-resilient consensus algorithm using either reading and writing atomic registers or sending and receiving messages [21, 31]. This known impossibility result was proved for a model where participation is required and thus also trivially holds for a model where participation is not required. ∎

Theorem 6.3 *For $n \geq 3$ and $1 \leq k \leq n-2$, there is no weakly-k-resilient k-set-consensus algorithm using either reading and writing atomic registers or sending and receiving messages.*

Proof: The proof follows from Lemma 6.1 and the known result the there for $1 \leq k \leq n-1$, is no k-resilient k-set-consensus algorithm for n processes using atomic registers [10, 24, 37]. The impossibility result for the message passing model follows immediately from the one for the shared memory model. This known impossibility result was proved for a model where participation is required and thus also trivially holds for a model where participation is not required. ∎

Corollary 6.4 *For $n \geq 3$ and $1 \leq k \leq n-2$, there is no weakly-k-resilient k-set-consensus algorithm using almost-wait-free test-and-set bits and atomic registers.*

Proof: The proof follows immediately from Theorem 3.1 and Theorem 6.3. ∎

7 Related Work

The mutual exclusion problem was first stated and solved for n processes by Dijkstra in [20]. Numerous solutions for the problem have been proposed since it was first introduced in 1965. Because of its importance and as a result of new hardware and software developments, new solutions to the problem are being designed all the time. In [15, 17], Burns and Lynch have shown that any deadlock-free mutual exclusion algorithm for n processes using registers must use at least n shared registers, even when *multi-writer* registers are allowed. Most modern processor architectures support some form of "read-modify-write" interprocess synchronization such as test-and-set. Results regarding solving mutual exclusion using test-and-set bits can be found in [4, 13, 14, 28, 35]. Dozens of interesting mutual exclusion algorithms and lower bounds are described in details in [36, 39].

The election problem (sometimes called the one shot mutual exclusion problem) is a special case of the mutual exclusion problem where only one process is allowed enter once its critical section. This process is the elected leader. A symmetric election algorithm is presented in [38], for n processes, which uses only three atomic registers, in the absence of failures. The three registers solution is correct in a model where participation is required. The authors also showed that when participation is not required, $\lceil \log n \rceil + 1$ registers are necessary and sufficient for solving the election problem, in the absence of failures. In addition, in [38], Styer and Peterson have proved that n registers are necessary and sufficient for deadlock-free symmetric mutual exclusion, while $n + 2\lceil \log n \rceil + 1$ registers are sufficient for starvation-free symmetric mutual exclusion. Finally, they proved that $2n - 1$ registers are necessary and sufficient for *memoryless* starvation-free symmetric mutual exclusion. We use some key ideas from [38], in our implementations of an almost-wait-free election object and an almost-wait-free test-and-set bit.

The impossibility result that there are no election algorithm and no perfect renaming algorithm that can tolerate a single crash failure was first proved for the asynchronous message-passing model in [6, 33], and later has been extended for the shared memory model in [42]. The one-shot renaming problem was first solved for message-passing systems [6], and later for shared memory systems [9]. In [16] a long-lived wait-free renaming algorithm was presented. The ℓ-assignment algorithm presented in [16], can be used as an an optimal name space long-lived renaming algorithm with exponential step complexity. Several of the many papers on renaming are [1, 7, 8, 12, 18, 22, 24, 27, 32].

The consensus problem was formally defined in [34]. 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 [21], and later has been extended for the shared memory model with atomic registers, in [31]. 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 [10, 24, 37].

Extensions of the notion of fault tolerance, which are different from those considered in this paper, were proposed recently in[19] where, by introducing new adversaries and new progress conditions, 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. The ability to solve consensus under various symmetric and asymmetric progress conditions was studied in [26, 40].

In [41], the author has considered new types of weak crash failures, where failures may occur only until a certain predefined threshold on the level of contention is reached. The utility of the new definitions, was illustrated by deriving possibility and impossibility results for the well-known basic

problems of consensus and k-set consensus. Weak crash failures should be viewed as *fractions* of traditional crash failures, and enable to design algorithms that can tolerate several (traditional) crash failures plus several additional weak crash failures. Furthermore, adding the ability to tolerate weak crash failures to algorithms that are already designed to circumvent various impossibility results for traditional crash failures, such as the Paxos algorithm [30], can make such algorithms even more robust against possible failures.

A comprehensive discussion of wait-free synchronization is given in [23]. In [3], a class of objects called Common2 is defined. Each object in Common2 has a wait-free implementation from registers together with any other object in Common2, for arbitrary number of processes. Commonly used objects such as test-and-set, fetch-and-add, swap, and stack are in Common2 [2, 3]. In [25], the related notion of a non-blocking is introduced. It guarantees that some correct process with a pending operation, will always be able to complete its operation in a finite number of its own steps regardless of the execution speed of other processes. For one-shot objects (like election, consensus, one-shot renaming) wait-freedom and non-blocking are the same.

8 Discussion

We have refined the traditional notion of *t-resiliency* by defining the finer grained notion of *(t, f)-resiliency*. In particular, we have extended the investigation of fault-tolerance by presenting several new notions: weakly-t-resiliency, partially-t-resiliency and almost-t-resiliency.

In the traditional notion of *t-resiliency* it is assumed that failures are *uniform*: processes are equally probable to fail, and failure of one process does not affect the reliability of the other processes. As discussed in [29], in real systems, failures may be correlated because of software or hardware features shared by subsets of processes. Our new resiliency notions can be defined similarly also for such *non-uniform* failure models, and it would be interesting to extend our results to cover such failure models.

All our results are presented in the context of crash failures in asynchronous systems, it would be interesting to consider also other types of failures such as omission failures and Byzantine failures, and to consider synchronous systems. Another interesting direction would be to extend the results for other objects. In particular, is there an almost-wait-free (or even a weakly-wait-free) implementation of a shared *queue* object from registers? 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 definition is another interesting direction.

Several other questions are left open. We have presented a symmetric almost-wait-free implementation of a test-and-set bit from registers. Are there similar symmetric almost-wait-free implementations for perfect renaming, stack, swap and fetch-and-add objects from registers? In case that there is no almost-wait-free perfect renaming, what is the smallest m for which there is a solution for almost-wait-free renaming in which a process always gets a distinct name in the range $\{1, ..., m\}$? Finally, are there implementations which are more space, time or message efficient than the implementations presented?

Acknowledgements: I wish to thank the three anonymous referees for their constructive suggestions and corrections.

References

[1] Y. Afek, H. Attiya, A. Fouren, G. Stupp, and D. Touitou. Long-lived renaming made adaptive. In *Proc. 18th ACM Symp. on Principles of Distributed Computing*, pages 91–103, May 1999.

[2] Y. Afek, E. Gafni, and A. Morrison. Common2 extended to stacks and unbounded concurrency. In *Proc. 25th ACM Symp. on Principles of Distributed Computing*, pages 218–227, 2006.

[3] Y. Afek, E. Weisberger, and H. Weisman. A completeness theorem for a class of synchronization objects. In *Proc. 12th ACM Symp. on Principles of Distributed Computing*, pages 159–170, 1993.

[4] J. H. Anderson and M. Moir. Using k-exclusion to implement resilient, scalable shared objects. In *Proc. 14th ACM Symp. on Principles of Distributed Computing*, pages 141–150, August 1994.

[5] H. Attiya, A. Bar-Noy, D. Dolev, D. Koller, D. Peleg, and R. Reischuk. Achievable cases in an asynchronous environment. In *Proc. 28th IEEE Symp. on Foundations of Computer Science*, pages 337–346, October 1987.

[6] H. Attiya, A. Bar-Noy, D. Dolev, D. Koller, D. Peleg, and R. Reischuk. Renaming in an asynchronous environment. *Journal of the Association for Computing Machinery*, 37(3):524–548, July 1990.

[7] H. Attiya and A. Fouren. Polynomial and adaptive long-lived $(2k-1)$-renaming. In *Proc. 14th International Symp. on Distributed Computing: Lecture Notes in Computer Science 1914*, pages 149–163, October 2000.

[8] H. Attiya and A. Fouren. Algorithms adapting to point contention. *Journal of the ACM*, 50(4):144–468, 2003.

[9] A. Bar-Noy and D. Dolev. Shared memory versus message-passing in an asynchronous distributed environment. In *Proc. 8th ACM Symp. on Principles of Distributed Computing*, pages 307–318, 1989.

[10] 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.

[11] E. Borowsky, E. Gafni, N. A. Lynch, and S. Rajsbaum. The BG distributed simulation algorithm. *Distributed Computing*, 14(3):127–146, 2001.

[12] A. Brodsky, F. Ellen, and P. Woelfel. Fully-adaptive algorithms for long-lived renaming. *Distributed Computing*, 24(2):119–134, 2011.

[13] J. E. Burns, M. J. Fischer, P. Jackson, N. A. Lynch, and G. L. Peterson. Shared data requirements for implementation of mutual exclusion using a test-and-set primitive. In *Proc. of the International Conf. on Parallel Processing*, pages 79–87, August 1978.

[14] J. E. Burns, P. Jackson, N. A. Lynch, M. J. Fischer, and G. L. Peterson. Data requirements for implementation of N-process mutual exclusion using a single shared variable. *Journal of the Association for Computing Machinery*, 29(1):183–205, 1982.

[15] J.E. Burns and A.N. Lynch. Mutual exclusion using indivisible reads and writes. In *18th annual allerton conference on communication, control and computing*, pages 833–842, October 1980.

[16] J.E. Burns and G.L. Peterson. The ambiguity of choosing. In *Proc. 8th ACM Symp. on Principles of Distributed Computing*, pages 145–158, August 1989.

[17] J.N. Burns and N.A. Lynch. Bounds on shared-memory for mutual exclusion. *Information and Computation*, 107(2):171–184, December 1993.

[18] A. Castaneda, S. Rajsbaum, and M. Raynal. The renaming problem in shared memory systems: An introduction. *Computer Science Review*, 5(3):229–251, 2011.

[19] C. Delporte-Gallet, H. Fauconnier, R. Guerraoui, and A. Tielmanns. The disagreement power of an adversary. In *Proc. 28th ACM Symp. on Principles of Distributed Computing*, pages 288–289, 2009.

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

[21] 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.

[22] E. Gafni, M. Merritt, and G. Taubenfeld. The concurrency hierarchy, and algorithms for unbounded concurrency. In *Proc. 20th ACM Symp. on Principles of Distributed Computing*, pages 161–169, August 2001.

[23] M. P. Herlihy. Wait-free synchronization. *ACM Trans. on Programming Languages and Systems*, 13(1):124–149, January 1991.

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

[25] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. on Programming Languages and Systems*, 12(3):463–492, 1990.

[26] D. Imbs, M. Raynal, and G. Taubenfeld. On asymmetric progress conditions. In *Proc. 29th ACM Symp. on Principles of Distributed Computing*, pages 55–64, 2010.

[27] M. Inoue, S. Umetani, T. Masuzawa, and H. Fujiwara. Adaptive long-lived $O(k^2)$-renaming with $O(k^2)$ steps. In *15th international symposium on distributed computing*, 2001.

[28] E. Kushilevitz and M. O. Rabin. Randomized mutual exclusion algorithms revisited. In *Proc. 11th ACM Symp. on Principles of Distributed Computing*, pages 275–283, 1992.

[29] P. Kuznetsov. Understanding non-uniform failure models. *Distributed computing column of the Bulletin of the European Association for Theoretical Computer Science (BEATCS)*, 106:54–77, 2012.

[30] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, July 1998.

[31] M.C. Loui and H. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. *Advances in Computing Research*, 4:163–183, 1987.

[32] M. Moir and J. H. Anderson. Wait-free algorithms for fast, long-lived renaming. *Science of Computer Programming*, 25(1):1–39, October 1995.

[33] S. Moran and Y. Wolfstahl. Extended impossibility results for asynchronous complete networks. *Information Processing Letters*, 26(3):145–151, 1987.

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

[35] G. L. Peterson. New bounds on mutual exclusion problems. Technical Report TR68, University of Rochester, February 1980 (Corrected, Nov. 1994).

[36] M. Raynal. *Algorithms for mutual exclusion*. The MIT Press, 1986. Translation of: Algorithmique du parallélisme, 1984.

[37] M. Saks and F. Zaharoglou. Wait-free k-set agreement is impossible: The topology of public knowledge. *SIAM Journal on Computing*, 29, 2000.

[38] E. Styer and G. L. Peterson. Tight bounds for shared memory symmetric mutual exclusion problems. In *Proc. 8th ACM Symp. on Principles of Distributed Computing*, pages 177–191, August 1989.

[39] G. Taubenfeld. *Synchronization Algorithms and Concurrent Programming*. Pearson / Prentice-Hall, 2006. ISBN 0-131-97259-6, 423 pages.

[40] G. Taubenfeld. The computational structure of progress conditions. In *24th international symposium on distributed computing (DISC 2010)*, September 2010. *LNCS 6343* Springer Verlag 2010, 221–235.

[41] G. Taubenfeld. Brief Announcement: Computing in the presence of weak crash failures. In: *Proc. 35th ACM Symp. on Principles of Distributed Computing* (PODC '16), pages 349–351, July 2016.

[42] G. Taubenfeld and S. Moran. Possibility and impossibility results in a shared memory environment. *Acta Informatica*, 33(1):1–20, 1996.