On Implementing SWMR Registers from SWSR Registers in Systems with Byzantine Failures

Xing Hu
Department of Computer Science, University of Toronto, Canada

Sam Toueg
Department of Computer Science, University of Toronto, Canada

Abstract

The implementation of registers from (potentially) weaker registers is a classical problem in the theory of distributed computing. Since Lamport's pioneering work [14], this problem has been extensively studied in the context of asynchronous processes with crash failures. In this paper, we investigate this problem in the context of Byzantine process failures, with and without process signatures. In particular, we first show a strong impossibility result, namely, that there is no wait-free linearizable implementation of a 1-writer n-reader register from atomic 1-writer $(n-1)$-reader registers. In fact, this impossibility result holds even if *all the processes except the writer* are given atomic 1-writer n-reader registers, and even if we assume that the writer can only crash and *at most one* reader is subject to Byzantine failures. In light of this impossibility result, we give two register implementations. The first one implements a 1-writer n-reader register from atomic 1-writer 1-reader registers. This implementation is linearizable (under any combination of Byzantine process failures), but it is wait-free only under the assumption that the writer is correct or no reader is Byzantine – thus matching the impossibility result. The second implementation assumes process signatures; it is wait-free and linearizable under any number and combination of Byzantine process failures.

2012 ACM Subject Classification Theory of computation → Distributed computing models; Theory of computation → Distributed algorithms

Keywords and phrases distributed computing, concurrency, linearizability, shared registers

Digital Object Identifier 10.4230/LIPIcs.DISC.2022.36

Related Version *Full Version*: `https://arxiv.org/abs/2207.01470`

Funding This work was partially funded by the Natural Sciences and Engineering Research Council of Canada (Grant number: RGPIN-2014-05296).

Acknowledgements We thank Vassos Hadzilacos for his helpful comments on this paper.

1 Introduction

We consider the basic problem of implementing a single-writer *multi*-reader register from atomic single-writer *single*-reader registers in a system where processes are subject to *Byzantine failures.* In particular, (1) we give an implementation that works under some failure assumptions, and (2) we prove a matching impossibility result for the case when these assumptions do not hold. We also consider systems where processes can use unforgeable signatures, and give an implementation that works for any number of faulty processes. We now describe our motivation and results in detail.

1.1 Motivation

Implementing shared registers from weaker primitives is a fundamental problem that has been thoroughly studied in distributed computing [3, 4, 5, 8, 12, 14, 16, 17, 18, 19, 20, 21, 22].

In particular, it is well-known that in systems where processes are subject to *crash* failures, it is possible to implement a wait-free linearizable m-writer n-reader register (henceforth denoted $[m, n]$-register) from atomic 1-writer 1-reader registers (denoted $[1, 1]$-registers).

In this paper, we consider the problem of implementing *multi*-reader registers from *single*-reader registers in systems where processes are subject to *Byzantine* failures. In particular, we consider the following basic questions:

- Is there a wait-free linearizable implementation of a $[1, n]$-register from atomic $[1, 1]$-registers in systems with Byzantine processes?
- If so, under which assumption(s) such an implementation exist?

The above questions are also motivated by the growing interest in shared-memory or hybrid systems where processes are subject to Byzantine failures. For example, Cohen and Keidar [6] give f-resilient implementations of several objects (namely, *reliable broadcast*, *atomic snapshot*, and *asset transfer* objects) using atomic $[1, n]$-registers in systems with Byzantine failures where at most $f < n/2$ processes are faulty. As another example, Aguilera *et al.* use atomic $[1, n]$-registers to solve some agreement problems in hybrid systems with Byzantine process failures [1]. Moreover, Mostéfaoui *et al.* [15] prove that, in *message-passing* systems with Byzantine process failures, there is a f-resilient linearizable implementation of a $[1, n]$-register if and only if at most $f < n/3$ processes are faulty.

1.2 Description of the results

To simplify the exposition of our results, we first state them in terms of two process groups: *correct* processes that do not fail and *faulty* ones. We show that in a system with Byzantine failures the following impossibility and possibility results hold. For all $n \geq 3$:

(A) If the writer *and* some reader (even if only one of them) can be faulty, then there is no wait-free linearizable implementation of a $[1, n]$-register from atomic $[1, n-1]$-registers.

(B) If the writer *or* some readers (any number of them), but *not both*, can be faulty, then there is a wait-free linearizable implementation of a $[1, n]$-register from atomic $[1, 1]$-registers.

The case $n = 2$ is special: we give a wait-free linearizable implementation of a $[1, 2]$-register from atomic $[1, 1]$-registers that works even if the writer *and* readers can be faulty.

This simple version of the results, however, leaves several questions open. Intuitively, this is because the above results do not distinguish between the different types of faulty processes (recall that, by definition, Byzantine failures encompass all the possible failure behaviours, from simple crash to "malicious" behaviour). For example we may ask: what happens if we can assume that some processes (say the writer) are subject to crash failures *only*, while some other processes (say the readers) can fail in "malicious" ways? Is a wait-free linearizable implementation of a $[1, n]$-register from atomic $[1, 1]$-registers now possible?

Note also that the above results consider linearizability and wait-freedom (intuitively, "safety" and "liveness") as an *indivisible* requirement of a register implementation. But it can be useful to consider each requirement separately. For example, what happens if we want to implement a $[1, n]$-register with the following properties: (1) it is *always* safe (i.e., linearizable) and (2) it may lose its liveness (i.e., it may lose its wait-freedom by "blocking" some read or write operations) *only if* some specific "pattern/types" of failures occur?

To answer such questions, we now consider linearizability and wait-freedom separately, and we partition processes into *three* separate groups: (1) those that do not fail, called *correct* processes, (2) those that fail *only* by crashing, and (3) those that fail in any other way, called *malicious* processes. In systems with a mix of such processes, we prove the following:

(1) For all $n \geq 3$, there is *no* wait-free linearizable implementation I_n of a $[1, n]$-register from atomic $[1, n-1]$-registers, even if we assume that the writer can only crash and *at most one* of the readers can be malicious.

In fact, we show that this impossibility result holds even if *all* the processes except the writer are given atomic $[1, n]$-registers that all processes can read; so the writer is the *only* process that does not have an atomic $[1, n]$-register.

(2) For all $n \geq 3$, there is an implementation I_n of a $[1, n]$-register from atomic $[1, 1]$-registers such that:
- I_n is linearizable, and
- I_n is wait-free if the writer is correct or no reader is malicious.[1]

Note that this implementation guarantees *linearizability*, no matter which processes fail and how they fail (even if most processes are malicious). However, it guarantees *wait-freedom* only if the writer is correct or no reader is malicious.[2] So if the *readers* are subject to crash failures only, the implementation is wait-free *even if the writer is malicious.*
Note that the above impossibility and matching possibility results **(1)** and **(2)** imply the simpler results **(A)** and **(B)** that we stated earlier for processes that are (coarsely) characterized as either correct or faulty.

We also consider the problem of implementing a $[1, n]$-register from atomic $[1, 1]$-registers in systems where processes are subject to Byzantine failures, but they can use *unforgeable signatures.* In sharp contrast to the above results, we show that in such systems there is an implementation of $[1, n]$-register from atomic $[1, 1]$-registers that is linearizable *and* wait-free *no matter how many processes fail and how they fail.*

2 Result techniques

The techniques that we used to obtain our possibility and impossibility results (for the "no signatures" case) are also a significant contribution of this paper.

To prove the impossibility result **(1)**, one cannot use a standard partitioning argument: all the processes except the writer are given atomic $[1, n]$-registers that all processes can read, and the writer is given a $[1, n-1]$-register that all the readers except one can read; thus it is clear that the system cannot be partitioned.

So to prove this result we use an interesting *reductio ad absurdum* technique. Starting from an alleged implementation of $[1, n]$-register from $[1, n-1]$-registers, we consider a run where the implemented register is initialized to 0, the writer completes a write of 1, and then a reader reads 1. By leveraging the facts that: (1) in each step the writer can read or write only $[1, n-1]$-registers, (2) the writer may crash, (3) one of the readers may be malicious, (4) and there are at least 3 readers, we are able to successively remove every read or write step of the writer (one by one, starting from its last write operation) in a way that maintains the property that some correct reader reads 1 and at most one process in the run is malicious. As we successively remove the steps of the writer, the identity of the process that reads 1, and the identity of the process that may be malicious, keep changing. By continuing this process, we end up with a run in which the writer takes no steps, and yet a correct reader reads 1.

[1] That is, in every run of I_n where the writer is correct or no reader is malicious, correct processes complete all their operations.
[2] In fact it is slightly stronger than this: write operations are unconditionally "wait-free", only read operations may block if the condition is not met.

Note that this proof is reminiscent of the impossibility proof for the “Two generals’ Problem” in message-passing systems [7]. In that proof, one leverages the possibility of message losses to successively remove one message at a time. The proof given here is much more elaborate because it leverages the subtle interaction between crash *and* malicious failures that may occur at different processes.

For the matching possibility result **(2)**, we solve the problem of implementing a $[1, n]$-register from $[1, 1]$-registers with a *recursive* algorithm: intuitively, we first give an algorithm to implement a $[1, n]$-register using $[1, n-1]$-registers, rather than only $[1, 1]$-registers, and then recurse till $n = 2$. We do so because the recursive step of implementing a $[1, n]$-register using $[1, n-1]$-registers, is significantly easier than implementing a $[1, n]$-register using only $[1, 1]$-registers. This is explained in more detail in Section 5.1.

3 Model Sketch

We consider systems with asynchronous processes that communicate via single-writer registers and are subject to Byzantine failures. Recall that a single-writer n-reader register is denoted as a $[1, n]$-register; the n readers are distinct from the writer.

3.1 Process failures

A process that is subject to Byzantine failures can behave arbitrarily. In particular, it may deviate from the algorithm it is supposed to execute, or just stop this execution prematurely, i.e., crash. To distinguish between these two types of failures, we partition processes as follows:

- Processes that do not fail, i.e., *correct* processes.
- Processes that fail, i.e., *faulty* processes. Faulty processes are divided into two groups:
  - processes that just *crash*, and
  - the remaining processes, which we call *malicious*.

3.2 Atomic and implemented registers

A register is *atomic* if its read and write operations are *instantaneous* (i.e., indivisible); each read must return the value of the last write that precedes it, or the initial value of the register if no such write exists. Roughly speaking, the *implementation* of a register from a set of “base” registers is given by read/write procedures that each process can execute to read/write the implemented register; these procedures can access the given base registers (which, intuitively, may be less “powerful” than the implemented register). So each operation on an implemented register *spans an interval* that starts with an *invocation* (a procedure call) and completes with a corresponding *response* (a value returned by the procedure).

A register implementation is *wait-free* [2, 9, 13] if it guarantees that every operation invoked by a correct process completes with a response in a finite number of steps.

3.3 Linearizability of register implementations

Unless we explicitly state otherwise, all the register implementations that we consider are *linearizable* [10]. Intuitively, linearizability requires that every operation on an implemented object appears as if it took effect instantaneously at some point (the “linearization point”) in its execution interval.

As noted by [6, 15], however, the precise definition of linearizability depends on whether processes can only crash, or they can also fail in a “Byzantine way”. We now explain this for *register* implementations.

In systems with only crash failures. It is well-known that a *single-writer multi-reader* register implementation is linearizable if and only if it satisfies two simple properties. To define these properties precisely, we first define what it means for two operations to be concurrent or for one to precede the other.

▶ **Definition 1.** *Let o and o' be any two operations.*

- o precedes o' *if the response of o occurs before the invocation of o'.*
- o is concurrent with o' *if neither precedes the other.*

We say that a write operation W *immediately precedes* a read operation R if W precedes R, and there is no write operation W′ such that W precedes W′ and W′ precedes R.

Let v_0 be the *initial value* of the implemented register, and v_k be the value written by the k-th write operation of the writer w (this is well-defined because we make the standard assumption that each process applies operations sequentially).

▶ **Definition 2** (Register Linearizability)**.** *In a system with crash failures, an implementation of a $[1, n]$-register is* linearizable *if and only if it satisfies the following two properties:*

- ***Property 1** [Reading a "current" value] If a read operation R returns the value v then:*
  - *there is a write v operation that immediately precedes R or is concurrent with R, or*
  - *$v = v_0$ and no write operation precedes R.*
- ***Property 2** [No "new-old" inversion] If two read operations R and R′ return values v_k and $v_{k'}$, respectively, and R precedes R′, then $k \leq k'$.*

In systems with Byzantine failures. The above definitions do not quite work for systems with Byzantine failures. For example, it is not clear what it means for a writer w of an implemented register to "write a value v" if w is malicious, i.e., if w *deviates* from the write procedure that it is supposed to execute; similarly, if a reader r is malicious it is not clear what it means for r to "read a value v". The definition of linearizability for systems with Byzantine failures avoids the above issues by restricting the linearization requirements to processes that are *not* malicious. More precisely:

▶ **Definition 3** (Register Linearizability)**.** *In a system with Byzantine process failures, an implementation of a $[1, n]$-register is* linearizable *if and only if the following holds. If the writer is not malicious, then:*

- ***Property 1** [Reading a "current" value] If a read operation R by a process that is not malicious returns the value v then:*
  - *there is a write v operation that immediately precedes R or is concurrent with R, or*
  - *$v = v_0$ and no write operation precedes R.*
- ***Property 2** [No "new-old" inversion] If two read operations R and R′ by processes that are not malicious return values v_k and $v_{k'}$, respectively, and R precedes R′, then $k \leq k'$.*

Note that if the writer is correct or only crashes, then readers that are correct or only crash are required to read "current" values and also avoid "new-old" inversions. So in systems where faulty processes can only crash, Definition 3 reduces to Definition 2.

Cohen and Keidar were the first to define linearizability for *arbitrary* objects in systems with Byzantine failures [6], and their definition generalizes the definition of linearizability for $[1, n]$*-registers* given by Mostéfaoui *et al.* in [15]. Definition 3 is consistent with both.

We now describe the results of this paper. Because of space limitations, some of the proofs are omitted here; they can be found in [11].

4 Impossibility result

We now prove that in a system with $n+1$ Byzantine processes, if the writer *and* one of the n readers can be faulty, then there is no wait-free linearizable implementation of a $[1,n]$-register from atomic $[1,n-1]$-registers. In fact, by dividing faulty processes into those that can only crash and those that can be malicious (as defined in Section 3), we show the following stronger result.

▶ **Theorem 4.** *For all $n \geq 3$, there is* no *wait-free linearizable implementation of a $[1,n]$-register from atomic $[1,n-1]$-registers in a system with $n+1$ processes that are subject to Byzantine failures. This holds even if we assume that the writer of the implemented $[1,n]$-register can only crash and at most one reader can be malicious.*

Proof. Let $n \geq 3$. Suppose, for contradiction, that there is a wait-free linearizable implementation $\mathcal{I}$ of a $[1,n]$-register $\mathbf{R}$ from atomic $[1,n-1]$-registers, in a system where the writer w of $\mathbf{R}$ can crash and one of the n readers of $\mathbf{R}$ can be malicious.

We now construct a sequence of executions of $\mathcal{I}$ that leads to a contradiction. In all these executions, the initial value of the implemented $\mathbf{R}$ is 0, the writer w invokes only one operation into $\mathbf{R}$, namely a write of 1, and each reader reads $\mathbf{R}$ at most once (i.e., $\mathbf{R}$ is only a "one-shot" binary register). Moreover, in each of these executions the writer is not malicious (it may only crash) and there is at most one malicious reader; the other $n-1$ readers are correct. Since $\mathcal{I}$ is a linearizable register implementation and the writer of the register is not malicious, these executions of $\mathcal{I}$ must satisfy Properties 1 and 2 of Definition 3.

Let S be the following execution of $\mathcal{I}$ (see Figure 1):

- The writer w is correct.
- All the readers take no steps.
- The writer w invokes a write 1 operation on $\mathbf{R}$. Let s^0 denote the invocation step, and let t_w^0 be the time when s^0 occurs. This step is "local" to w, i.e., it does not invoke any shared register operations.
  During this write operation, w executes a sequence of steps $s^1, ..., s^m$ such that each step s^i is either the reading or the writing of an atomic $[1,n-1]$-register. Let R_i denote the register that w writes or reads in step s^i. Let t_w^i be the time when s^i occurs.
- Since $\mathcal{I}$ is a wait-free implementation and w is correct, w completes its write operation. Let s^{m+1} denote the response step, and let t_w^{m+1} be the time when s^{m+1} occurs. Like s^0, this step is also "local" to w.

▶ **Definition 5.** *For all i, $0 \leq i \leq m+1$, the step s^i of the writer w is* invisible *to a reader x if: (1) s^i is the invocation step s^0, (2) s^i is the response step s^{m+1}, (3) s^i is the reading of an atomic register, or (4) s^i is the writing to an atomic register that is not readable by x.*

Since there are n readers, and the registers that w uses are atomic $[1,n-1]$-registers, every write by w into one of these registers is invisible to one of the readers. So:

▶ **Observation 6.** *For all $0 \leq k \leq m+1$, step s^k is invisible to at least one of the n readers.*

▶ **Definition 7.** *For every k, $0 \leq k \leq m+1$, an execution of $\mathcal{I}$ has property P_k if the following holds:*

1. *The writer w behaves exactly as in S up to and including time t_w^k; then it crashes and takes no steps after time t_w^k. So, w executes steps $s^0, s^1, \ldots, s^k$ and then crashes.*
2. *There is a reader x that is correct and such that:*
   - *Step s^k is invisible to x.*
   - *After time t_w^k, process x starts and completes a read operation on $\boldsymbol{R}$ that returns* 1.
3. *There is a set Z of $n-2$ distinct readers that are correct and take no steps.*

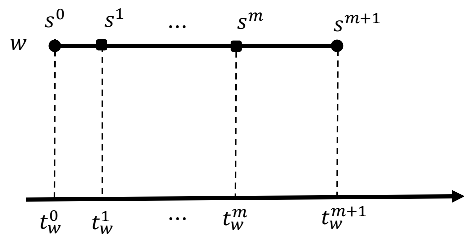
s^0 s^1 … s^m s^{m+1}
w
t_w^0 t_w^1 … t_w^m t_w^{m+1}

Figure 1 Execution S.

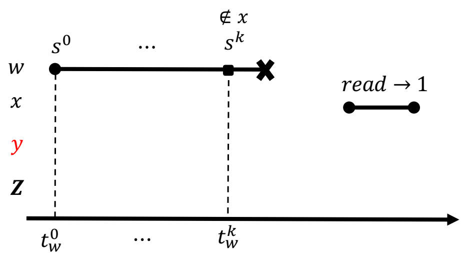

Figure 2 An execution with property P_k.

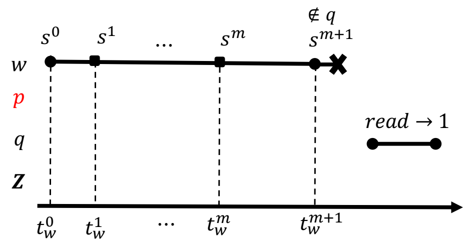
$\notin q$
s^0 s^1 … s^m s^{m+1}
w p q Z
$read \to 1$
t_w^0 t_w^1 … t_w^m t_w^{m+1}

Figure 3 Execution A_{m+1}.

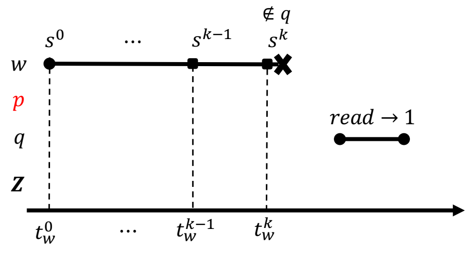

Figure 4 Execution A_k.

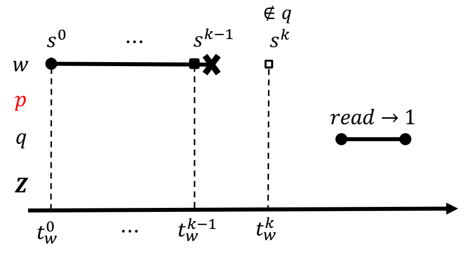

Figure 5 Execution B_{k-1}.

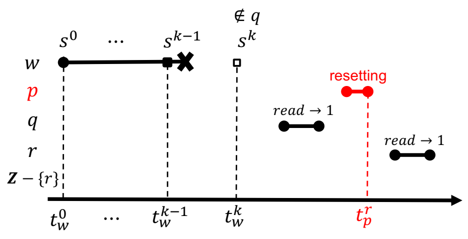

Figure 6 Execution C_{k-1}^r.

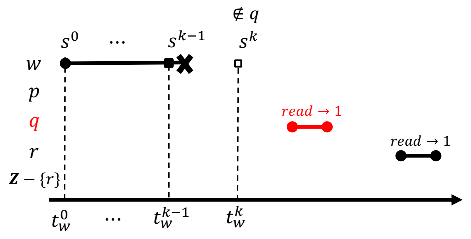

Figure 7 Execution D_{k-1}^r.

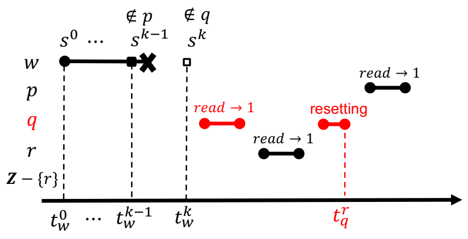

Figure 8 Execution E_{k-1}^r.

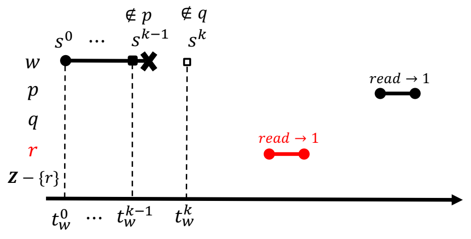

Figure 9 Execution F_{k-1}^r.

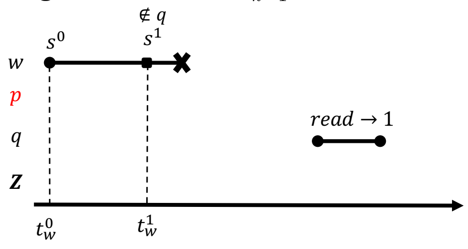

Figure 10 Execution A_1.

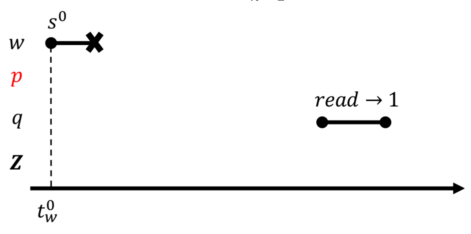

Figure 11 Execution A_0.

Note that since $n \geq 3$, the set Z is not empty. Also note that while the property P_k requires the $n-1$ readers in $\{x\} \cup Z$ to be correct, P_k does not restrict the behavior of the remaining reader; in particular, it may be correct or malicious, and it may or may not take steps.

An execution of $\mathcal{I}$ with property P_k is shown in Figure 2. In this figure and all the subsequent ones, correct readers are in black font, while the reader that may be malicious is colored red; the steps that this process may have taken are not shown in the figure. The "$\notin x$" on top of a step s^i means that s^i is invisible to the reader x. The symbol ✖ indicates where the crash of the writer w occurs.

▷ Claim. For every k, $0 \leq k \leq m+1$, there is an execution of $\mathcal{I}$ that has property P_k.

Proof. We prove the claim by a backward induction on k, starting from $k = m+1$.

Base Case. $k = m+1$. Consider the following execution denoted A_{m+1} (Figure 3):

- The writer w behaves as in execution S up to and including time t_w^{m+1}; then it crashes.
- A reader q is correct. After time t_w^{m+1}, q starts a read operation on $\mathbf{R}$. Since $\mathcal{I}$ is a wait-free implementation, q completes its read operation. Since w is not malicious, and the write operation by w immediately precedes the read operation by q, by the linearizability of $\mathcal{I}$, the read operation by q returns 1.
- There is a set Z of $n-2$ readers that are correct and take no steps, exactly as in S.
- p is the remaining reader.

Since s^{m+1} is a response step, it is invisible to q. So it is clear that A_{m+1} has property P_{m+1}.

Induction Step. Let k be such that $1 \leq k \leq m+1$. Suppose there is an execution A_k of $\mathcal{I}$ that has property P_k (this is the induction hypothesis). We now show that there is an execution A_{k-1} of $\mathcal{I}$ that has property P_{k-1}. We consider two cases, namely, $k > 1$ and $k = 1$.

Case $k > 1$. Since execution A_k of $\mathcal{I}$ satisfies P_k, the following holds in A_k (see Figure 4):

- The writer w behaves as in execution S up to and including time t_w^k; then it crashes.
- There is a reader q that is correct such that step s^k is invisible to q. After time t_w^k, q starts and completes a read operation on $\mathbf{R}$ that returns 1.
- There is a set Z of $n-2$ readers that are correct and take no steps, exactly as in S.
- p is the remaining reader.

Then the following execution B_{k-1} of $\mathcal{I}$ also exists (Figure 5): B_{k-1} is exactly like A_k except that w crashes just before taking step s^k (so B_{k-1} is just A_k with the step s^k "removed").

B_{k-1} is possible because: (1) even though p may have "noticed" the removal of step s^k, p may be malicious (all other readers are correct in this execution), and (2) q cannot distinguish between A_k and B_{k-1} because s^k is invisible to q, and p and all the readers in Z behave as in A_k.

Since $k > 1$, A_k has a step $s^{k-1} \neq s^0$. There are two cases:

Case 1. s^{k-1} *is invisible to* q. Then B_{k-1} is an execution of $\mathcal{I}$ that has the property P_{k-1}, as we wanted to show.

Case 2. s^{k-1} *is visible to* q. Then s^{k-1} is invisible to p or to some reader in Z.

Let r be *any* process in Z. We construct the execution C_{k-1}^r of $\mathcal{I}$ shown in Figure 6: C_{k-1}^r is a continuation of B_{k-1} where, after the correct reader q reads 1, malicious p wipes out any trace of the write steps that it has taken so far, and then a correct process $r \in Z$ reads 1 (this is the only value that r can read, since correct q previously read 1). More precisely:

- C^r_{k-1} is an extension of B_{k-1}.
- After the correct reader q completes its read operation on $\mathbf{R}$, q takes no steps.
- All the readers in $Z-\{r\}$ are correct and take no steps[3].
- After q completes its read operation, p resets all the atomic registers that it can write to their initial values. Process p can do so because it may be malicious (all other readers are correct in this execution). Let t^r_p be the time when p completes all the register resettings.
- A correct reader r starts a read operation on $\mathbf{R}$ after time t^r_p. It takes no steps before this read. Since $\mathcal{I}$ is a wait-free implementation, r completes its read operation. Since w is not malicious and the read operation by correct q returns 1 and precedes the read operation by r, by the linearizability of $\mathcal{I}$, the read operation by r returns 1.

We can now construct the following execution D^r_{k-1} of $\mathcal{I}$ (Figure 7). D^r_{k-1} is obtained from C^r_{k-1} by removing all the steps of p. Despite this removal, q behaves the same as in C^r_{k-1} because q is now malicious. Correct r also behaves as in C^q_{k-1} because it cannot see the removal of p's steps: in both C^r_{k-1} and D^r_{k-1}, r does not "see" any steps of p. More precisely in D^r_{k-1}:

- w behaves exactly as in C^r_{k-1}.
- p is correct and takes no steps. So all its registers retain their initial value.
- All the readers in $Z-\{r\}$ are correct and take no steps as in C^r_{k-1}.
- q behaves the same as in C^r_{k-1}. This is possible because even though q may have "noticed" the removal of p's steps, q may be malicious (all other readers are correct in this execution).
- After possibly malicious q "reads" 1, the correct reader r starts and completes a read operation on $\mathbf{R}$. Since r cannot see the removal of p's steps, and q and all the readers in $Z-\{r\}$ behave the same as in C^r_{k-1}, r cannot distinguish between D^r_{k-1} and C^r_{k-1}. So the read operation by r returns 1 as in C^r_{k-1}.

Note that if s^{k-1} is invisible to process r, then the execution D^r_{k-1} of $\mathcal{I}$ has property P_{k-1}.

Recall that (1) the process r above is an *arbitrary* process in Z, and (2) s^{k-1} is invisible to p or to some reader $r' \in Z$. So there are two cases:

Subcase 2a. *s^{k-1} is invisible to some reader $r' \in Z$.* In the above we proved that the execution $D^{r'}_{k-1}$ of $\mathcal{I}$ has property P_{k-1}, as we wanted to show.

Subcase 2b. *s^{k-1} is invisible to p.* In this case we construct the continuation E^r_{k-1} of D^r_{k-1} shown in Figure 8: after r reads 1, malicious process q wipes out any trace of the write steps that it has taken so far (by reinitializing its registers), and then correct process p applies a read operation to $\mathbf{R}$. By wait freedom, this read operation by p must complete. Since w is not malicious and correct r previously read 1, by linearizability, this read operation by p must return 1.

Finally, we construct the execution F^r_{k-1} of $\mathcal{I}$ by removing all the steps of q from E^r_{k-1} (see Figure 9); so q (which was malicious in E^r_{k-1}) is now a correct process that takes no steps. Despite this removal, r behaves the same as in E^r_{k-1} because r (which was correct in E^r_{k-1}) may now be malicious. Moreover, correct p also behaves as in E^r_{k-1} because it cannot see the removal of q's steps: in both E^r_{k-1} and F^r_{k-1}, p does not "see" any steps of q. So the read operation by p returns 1 as in E^r_{k-1}.

Note that, since s^{k-1} is invisible to p, F^r_{k-1} is an execution of $\mathcal{I}$ that has property P_{k-1}.

[3] If $n = 3$, then the set $Z-\{r\}$ is empty.

Case $k = 1$. By the induction hypothesis, there is an execution A_1 as follows (Figure 10):

- The writer w behaves exactly as in S up to and including time t^1_w; then it crashes.
- After time t^1_w, a correct reader q starts and completes a read operation on $\mathbf{R}$ that returns 1. Furthermore, s^1 is invisible to q.
- There is a set Z of $n - 2$ readers that are correct and take no steps.
- p is the remaining reader.

Then the following execution A_0 of $\mathcal{I}$ also exists (Figure 11): A_0 is like A_1 except that w crashes just before taking step s^1 (so A_0 is just A_1 with the step s^1 "removed"). A_0 is possible because: (1) even though p may have "noticed" the removal of step s^1, p may be malicious (all other readers are correct in this execution), and (2) q cannot distinguish between A_0 and A_1 because s^1 is invisible to q, and p and all the readers in Z behave as in A_1. Since s^0 is an invocation step, it is invisible to q. It is now easy to see that execution A_0 of $\mathcal{I}$ has property P_0, as we wanted to show. ◁

By the claim that we just proved, implementation $\mathcal{I}$ has an execution A_0 with property P_0. By this property, in A_0 process w crashes immediately after the *invocation step* s^0 of its write 1 operation, and some correct reader x later reads the value 1. Since the invocation step s^0 is *invisible* to all the readers (because it does not involve writing any of the shared registers), there is an execution of A'_0 of $\mathcal{I}$ where: (1) w does not take any step at all (so it is not malicious), and (2) a correct reader x reads 1 exactly as in A_0 (because no reader can distinguish between A_0 and A'_0). This execution A'_0 of $\mathcal{I}$ violates the linearizability of $\mathcal{I}$. ◀

It is easy to verify that the above proof holds (without any change) even if all the readers have atomic $[1, n]$-registers that they can write and all processes can read. Thus:

▶ **Theorem 8.** *For all $n \geq 3$, there is* no *wait-free linearizable implementation of a $[1, n]$-register in a system of $n + 1$ processes that are subject to Byzantine failures such that:* *ss*

- *the writer w of the implemented $[1, n]$-register has atomic $[1, n - 1]$-registers, and every reader has atomic $[1, n]$-registers, and*
- *w can only crash and at most one reader can be malicious.*

5 Register implementation algorithm

We now give an implemention of a $[1, n]$-register from atomic $[1, 1]$-registers in systems with Byzantine process failures; this implementation is linearizable, and it is wait-free provided the writer of the register *or* any number of the readers *but not both* can be faulty. More precisely, it is a *valid* implementation as defined below.

▶ **Definition 9.** *A register implementation is* valid *if it satisfies the following:*

- It is linearizable.
- It is wait-free if the writer is correct or no reader is malicious.

Note that, when executed in a system where processes can only crash, a valid register implementation is linearizable and wait-free (unconditionally).

5.1 Some difficulties to overcome

Note that in a system with Byzantine process failures, implementing a $[1, n]$-register from $[1, 1]$-registers is non-trivial, even if the writer can only crash. To see this, we now illustrate some of the issues that arise. First note that with $[1, 1]$-registers the writer cannot *simultaneously* inform all the readers about a new write. So different readers may have different views of whether there is a write in progress: some readers may not see it, some readers may see it as still in progress, while other readers may see it as having completed. Thus readers must communicate with each other to avoid "new-old" inversions in the values they read. With non-Byzantine failures, readers can easily coordinate their reads because they can trust the information they pass to each other. With Byzantine failures, however, readers cannot blindly trust what other readers tell them.

For example, suppose a reader q is aware that a write v operation is in progress (say because the writer w directly "told" q about it via the register that they share). To avoid a "new-old" inversion, q checks whether any other reader q' has already read v (because it is possible that from q''s point of view, the write of v already completed). Suppose some q' "warns" q that it has already read the new value v, and so q also reads v. But what if q' is malicious and "lied" to q (and only to q) about having read v? Note that q may be the *only* correct reader currently aware that the write of v is in progress (say because w is slow). Now suppose that a reader q'' that is *not* aware of the write of v also wants to read: if q'' reads the old value of the register this creates a "new-old" inversion with the newer value v that q previously read; but if q'' reads v because q warns q'' that it had read v, then q'' may be reading a value v *that was never written by the correct writer* w: q itself could be malicious and could have "lied" about reading v!

The above is only one of many possible scenarios illustrating why it is not easy to implement a $[1, n]$-register from $[1, 1]$-registers when some readers can be malicious, even if the writer itself is not malicious.

5.2 A recursive solution

To simplify this task, we do not *directly* implement a $[1, n]$-register using *only* $[1, 1]$-registers. Instead, we first give an implementation I_n of a $[1, n]$-register that uses some $[1, n-1]$-registers together with some $[1, 1]$-registers. Then, by replacing the $[1, n-1]$-registers with I_{n-1} implementations, we get an implementation of the $[1, n]$-register that uses some $[1, n-2]$-registers and some $[1, 1]$-registers. By recursing down to $n = 2$, this gives an implementation of the $[1, n]$-register that uses only $[1, 1]$-registers. In other words, we can implement a $[1, n]$-register from $[1, 1]$-registers with a *recursive* construction that gradually reduces the number of readers of the base registers that it uses (all the way down to 1). We now describe this recursive implementation and prove its correctness.

5.3 Implementing a [1,n]-register from [1,n-1]-registers

Algorithm 1 gives an implementation I_n of a $[1, n]$-register that is writable by a process w and readable by every process in $\{p\} \cup Q$, where p is an arbitrary reader and all remaining $n - 1$ readers are in Q. We distinguish p from the other readers in Q because p and $q \in Q$ use different procedures for reading the implemented $[1, n]$-register. I_n uses two kinds of registers: *atomic* $[1, 1]$-registers and *implemented* $[1, n-1]$-registers. We will show that I_n is *valid* under the assumption that the $[1, n-1]$-register implementations that it uses are also *valid* (and therefore *linearizable*).

Notation. Recall that if R is an atomic register, all operations applied to R are instantaneous, whereas if R is an implemented register, each operation spans an interval of time, from an invocation to a response. However, since we assume that the $[1, n-1]$-register implementations that I_n uses are valid and therefore *linearizable*, we can think of each operation on an implemented $[1, n-1]$-register as being atomic, i.e., as if it takes effect instantaneously at

Algorithm 1 Implementation I_n of a $[1, n]$-register writable by (an arbitrary) process w and readable by the n processes in $\{p\} \cup Q$, for $n \geq 2$. It uses two $[1, n-1]$-registers and some $[1, 1]$-registers.

```
ATOMIC REGISTERS
    $R_{wp}$: $[1,1]$-register; initially $(\text{COMMIT}, \langle 0, u_0\rangle)$
    For every processes $q, q' \in Q$:
        $R_{qq'}$: $[1,1]$-register; initially $\langle 0, u_0\rangle$
IMPLEMENTED REGISTERS
    $R_{wQ}$: $[1, n-1]$-register; initially $(\text{COMMIT}, \langle 0, u_0\rangle)$
    $R_{pQ}$: $[1, n-1]$-register; initially $\langle 0, u_0\rangle$

LOCAL VARIABLES
    $c$: variable of $w$; initially 0
    $last\_written$: variable of $w$; initially $\langle 0, u_0\rangle$
    $previous\_k$: variable of $p$; initially 0
```

```
WRITE($u$):                              ▷ executed by the writer $w$
1:      $c \leftarrow c + 1$
2:      call W($\langle c, u\rangle$)
3:      return done

READ():                                  ▷ executed by any reader $r$ in $\{p\} \cup Q$
4:      call R$_r$()
5:      if this call returns some tuple $\langle k, u\rangle$ then
6:          return $u$
7:      else return $\perp$
```

```
W($\langle k, u\rangle$):                              ▷ executed by $w$ to do its $k$-th write
8:      $R_{wp} \leftarrow (\text{PREPARE}, last\_written, \langle k, u\rangle)$
9:      $R_{wQ} \leftarrow (\text{PREPARE}, last\_written, \langle k, u\rangle)$
10:     $R_{wp} \leftarrow (\text{COMMIT}, \langle k, u\rangle)$
11:     $R_{wQ} \leftarrow (\text{COMMIT}, \langle k, u\rangle)$
12:     $last\_written \leftarrow \langle k, u\rangle$
13:     return done
R$_p$():                                                ▷ executed by reader $p$
14:     if $R_{wp} = (\text{COMMIT}, \langle k, u\rangle)$ for some $\langle k, u\rangle$ with $k \geq previous\_k$ then
15:         $R_{pQ} \leftarrow \langle k, u\rangle$
16:         $previous\_k \leftarrow k$
17:         return $\langle k, u\rangle$
18:     elseif $R_{wp} = (\text{PREPARE}, last\_written, -)$ for some $last\_written$ then
19:         return $last\_written$
20:     else return $\perp$
R$_q$():                                                ▷ executed by any reader $q \in Q$
21:     if $R_{wQ} = (\text{COMMIT}, \langle k, u\rangle)$ for some $\langle k, u\rangle$ then
22:         return $\langle k, u\rangle$
23:     elseif $R_{wQ} = (\text{PREPARE}, last\_written, \langle k, u\rangle)$ for some $last\_written$ and some $\langle k, u\rangle$ then
24:         cobegin
            // THREAD 1
25:             repeat forever
26:                 if $R_{wQ} = (\text{COMMIT}, \langle k', -\rangle)$ for some $k' \geq k$ then
27:                     return $\langle k, u\rangle$
28:                 if $R_{wQ} = (\text{PREPARE}, -, \langle k', -\rangle)$ for some $k' > k$ then
29:                     return $\langle k, u\rangle$
            // THREAD 2
30:             if $R_{pQ} = \langle k', -\rangle$ for some $k' \geq k$ then
31:                 for every process $q' \in Q$ do $R_{qq'} \leftarrow \langle k, u\rangle$
32:                 return $\langle k, u\rangle$
33:             elseif $R_{q'q} = \langle k', -\rangle$ for some $q' \in Q$ and some $k' \geq k$ then
34:                 if $R_{pQ} = \langle k', -\rangle$ for some $k' \geq k$ then
35:                     for every process $q' \in Q$ do $R_{qq'} \leftarrow \langle k, u\rangle$
36:                 return $\langle k, u\rangle$
37:             else return $last\_written$
38:         coend
39:     else return $\perp$
```

some point during its execution interval [10]. Thus to read or write a register R we use the same notation, irrespective of whether R is atomic or implemented. In particular, in our implementation algorithm (shown in Figure 1) we use the following notation:

- "$R \leftarrow v$" denotes the operation that writes v into R.
- "**if** $R = \mathit{val}$ **then** ..." means "read register R and if the value read is equal to *val* then ..."

The shared registers used by the implementation are as follows:

- $R_{rr'}$ is an atomic $[1, 1]$-register writable by process r and readable by process r'.[4]
- R_{wQ} is an implemented $[1, n-1]$-register writable by w and readable by every $q \in Q$.
- R_{pQ} is an implemented $[1, n-1]$-register writable by p and readable by every $q \in Q$.

Description. The implementation I_n of a $[1, n]$-register from $[1, n-1]$-registers consists of two procedures, namely WRITE() for the writer w, and READ() for each reader r in $\{p\} \cup Q$. To write a value u, the writer w executes WRITE(u). If u is the k-th value written by w, WRITE(u) first forms the unique tuple $\langle k, u\rangle$ and then it calls the lower-level write procedure W($\langle k, u\rangle$) to write this tuple. Intuitively, WRITE() tags the values that it writes with a counter value to make them unique and to indicate in which order they are written.

To read a value, a reader $r \in \{p\} \cup Q$ calls READ(), and this in turn calls a lower-level read procedure R$_r$() that reads tuples written by W(). There are two version of the procedure R$_r$(): one used when $r = p$ and one used when $r \in Q$. If R$_r$() returns a tuple of the form $\langle j, v\rangle$, then READ() strips the counter j from the tuple and returns the value v as the value read (otherwise READ() returns $\perp$ to indicate a read failure).

Thus the lower-level procedures W(), R$_p$(), and R$_q$() for each $q \in Q$, are executed to write and read unique tuples of the form $\langle k, u\rangle$. We now describe how these procedures work.

- To execute W($\langle k, u\rangle$), process w first writes (PREPARE, *last_written*, $\langle k, u\rangle$) in the R_{wp} register that p can read, and then in the R_{wQ} register that every process in Q can read; *last_written* is the *last* tuple written by w before $\langle k, u\rangle$ (so *last_written* $= \langle k-1, u'\rangle$ for some u'). Then, w writes (COMMIT, $\langle k, u\rangle$) into R_{wp} and then into R_{wQ}.
- To execute R$_p$(), process p reads R_{wp} (line 14). If p reads (COMMIT, $\langle k, u\rangle$) with a k at least as big as those it saw before, it returns $\langle k, u\rangle$ as the tuple read (line 17); just before doing so, however, it writes $\langle k, u\rangle$ in the R_{pQ} register that every process $q \in Q$ can read (line 15): intuitively, this is to "warn" them that p read a "new" tuple, to help avoid "new-old" inversion in the tuples read.
  If p reads (PREPARE, *last_written*, $\langle k, u\rangle$) (line 18), then it returns *last_written* as the tuple read (without giving any "warning" about this to processes in Q).
  If p reads anything else from R_{wp}, then it returns $\perp$ (the writer is surely malicious).
- To execute R$_q$(), process $q \in Q$ reads R_{wQ}. If q reads (COMMIT, $\langle k, u\rangle$) (line 21), it just returns $\langle k, u\rangle$ as the tuple read in line 22 (without "warning" other processes).
  If q reads (PREPARE, *last_written*, $\langle k, u\rangle$) (line 23), then q *cannot* simply return *last_written* as the tuple read: this is because p could have already read (COMMIT, $\langle k, u\rangle$) from R_{wp} and so p could have already read the "newer" tuple $\langle k, u\rangle$ with R$_p$(). So q must determine whether to return *last_written* or $\langle k, u\rangle$. To do so, q forks two threads and executes them in parallel (we will explain why below).[5]

[4] If $r = r'$, this "shared register" is actually just a local register of process r.
[5] If q does not read values of the form (PREPARE, *last_written*, $\langle k, u\rangle$) or (COMMIT, $\langle k, u\rangle$) from R_{wQ}, then w is surely malicious, and q just returns $\perp$ in line 39.

In THREAD 1, process q keeps reading R_{wQ}: if it ever reads (COMMIT, $\langle k', -\rangle$) with $k' \geq k$, or (PREPARE, $-$, $\langle k', -\rangle$) with $k' > k$, it simply returns $\langle k, u\rangle$ as the tuple read. Note that *if the writer w is correct*, then q *cannot* spin forever in this thread without returning $\langle k, u\rangle$.

In THREAD 2, process q first reads the register R_{pQ} to see whether p "warned" processes in Q that it read a tuple at least as "new" as $\langle k, u\rangle$.

- If q sees that R_{pQ} contains a tuple at least as "new" as $\langle k, u\rangle$ (line 30), then q returns $\langle k, u\rangle$ as the tuple read (line 32); but before doing so, q successively writes $\langle k, u\rangle$ in each register $R_{qq'}$ such that $q' \in Q$ (line 31): intuitively, this is to "warn" each process in Q that q read this "new" tuple.
- Otherwise, q reads every $R_{q'q}$ register to avoid a new-old inversion with any tuple read by any process $q' \in Q$: if q sees that some $R_{q'q}$ contains a tuple at least as "new" as $\langle k, u\rangle$ (line 33), then q reads R_{pQ} *again* (line 34) (so q does *not* simply "trust" q' and return $\langle k, u\rangle$!). If q sees that R_{pQ} contains a tuple at least as "new" as $\langle k, u\rangle$ (line 34), then q returns $\langle k, u\rangle$ as the tuple read (line 36); and before doing so q successively writes $\langle k, u\rangle$ to every register $R_{qq'}$ such that $q' \in Q$ (line 35).
- Finally, if q does not see that R_{pQ} or $R_{qq'}$ contain a tuple at least as "new" as $\langle k, u\rangle$ (in lines 30 and 33), then q returns *last_written* (line 37).

Why two parallel threads? In a nutshell, this is to guarantee the wait-freedom of I_n in runs where the writer is correct or no reader is malicious. This is required for our implementation to be valid. It turns out that:

(A) if only THREAD 1 is executed, then a faulty writer can block correct readers even if no reader is malicious, and

(B) if only THREAD 2 is executed, then malicious readers can block correct readers from returning any value in this thread even if the writer is correct.

But if the writer is correct or no reader is malicious, we can show that every read operation by a correct reader is guaranteed to complete with a return value *in one of the two threads*.

It is easy to see why a faulty writer (even one that just crashes) may block a correct reader in THREAD 1. We now explain how malicious readers may impede correct readers in THREAD 2.

In THREAD 2 readers must read R_{pQ} at least once (in line 30). Recall that (a) R_{pQ} is an *implemented* $[1, n-1]$-register, and (b) we are *only* assuming that this implementation is *valid*. In particular, if the writer p of R_{pQ} crashes *and* some readers of R_{pQ} are malicious, the implementation of R_{pQ} does *not* guarantee the wait-freedom of its read operations. In other words, if p crashes *and* some readers of R_{pQ} are malicious, a correct reader q may block while trying to read R_{pQ}!

Malicious readers may also prevent a correct reader q from reading any tuple in THREAD 2 as follows. When q executes $\text{R}_q()$ the following can occur: (1) in line 33, q sees that some $R_{q'q}$ contains $\langle k', -\rangle$ with $k' \geq k$, but (2) in line 34 q sees that R_{pQ} does *not* contain $\langle k', -\rangle$ with $k' \geq k$. We can show that this can occur *only if* at least one of p or q' is malicious. Note that if (1) and (2) indeed occur, then q terminates THREAD 2 *without returning any tuple* (because the **if** of line 34 does not have a corresponding **else**).

The correctness of the implementation I_n given by Algorithm 1 is stated in Theorem 10. The proof of this theorem is given in [11].

▶ **Theorem 10.** *For all $n \geq 2$, I_n is an implementation of a $[1, n]$-register from implemented $[1, n-1]$-registers and atomic $[1, 1]$-registers. I_n is valid if the implementations of the $[1, n-1]$-registers that it uses (namely, R_{wQ} and R_{pQ}) are valid.*

It's worth noting that for the case $n = 2$, there is a simple implementation ${I_2}'$ that is stronger than the I_2 implementation given by Algorithm 1: in contrast to I_2, ${I_2}'$ is *unconditionally* wait-free. The implementation ${I_2}'$ is given by Algorithm 3 in Appendix A. Note that Algorithm 3 is a simple version of Algorithm 1: the set of readers Q now contains only one process q, and so preventing new-old inversions is much easier.

▶ **Theorem 11.** *The implementation ${I_2}'$ (given by Algorithm 3 in Appendix A) is a wait-free linearizable implementation of a $[1, 2]$-register from atomic $[1, 1]$-registers.*

5.4 Implementing a [1,n]-register from atomic [1,1]-registers

We now show our main "possibility" result: in a system with Byzantine process failures, there is an implementation of a $[1, n]$-register from atomic $[1, 1]$-registers that is linearizable and wait-free provided that the writer *or* any number of readers, but *not both*, can fail. In fact we show the following stronger result:

▶ **Theorem 12.** *For all $n \geq 2$, in a system of $n + 1$ processes that are subject to Byzantine failures, there is a* valid *implementation $\mathcal{I}_n$ of a $[1, n]$-register from atomic $[1, 1]$-registers.*

Proof. We show Theorem 12 by induction on n.

Base Case. Let $n = 2$. Consider the implementation I_2 of Theorem 10. Since $n = 2$, the set Q now contains only one process. So each register R_{wQ} and R_{pQ} in I_2 can be implemented directly by an *atomic* $[1, 1]$-register. Since these are *valid* implementations of R_{wQ} and R_{pQ}, there is a valid implementation $\mathcal{I}_2$ of a $[1, 2]$-register from atomic $[1, 1]$-registers.[6]

Induction Step. Let $n > 2$. Suppose there is a valid implementation $\mathcal{I}_{n-1}$ of a $[1, n-1]$-register that uses only atomic $[1, 1]$-registers. We must show there is a valid implementation $\mathcal{I}_n$ of a $[1, n]$-register that uses only atomic $[1, 1]$-registers.

By Theorem 10, there is an implementation I_n of a $[1, n]$-register that uses:

1. two implemented $[1, n-1]$-registers (namely, of registers R_{wQ} and R_{pQ}), and
2. some atomic $[1, 1]$-registers

such that I_n is valid if the implementations of the $[1, n-1]$-registers R_{wQ} and R_{pQ} are valid. Implement R_{wQ} and R_{pQ} in I_n using the valid implementation $\mathcal{I}_{n-1}$ ($\mathcal{I}_{n-1}$ exists by our induction hypothesis). This gives an implementation $\mathcal{I}_n$ of a $[1, n]$-register that uses only atomic $[1, 1]$-registers (because $\mathcal{I}_{n-1}$ uses only atomic $[1, 1]$-registers). Since the implementations of R_{wQ} and R_{pQ} are valid, $\mathcal{I}_n$ is valid. ◀

Since $\mathcal{I}_n$ is valid, it is linearizable (no matter which processes fail and how); and it is wait-free *provided the writer is correct or no reader is malicious.* This matches the impossibility result given by Theorem 4 in Section 4.

[6] To show that $\mathcal{I}_2$ exists, we could also use the wait-free and linearizable implementation ${I_2}'$ mentioned in Theorem 11.

Algorithm 2 Implementation $\mathcal{I}_s$ of a $[1,n]$-register writable by process w and readable by a set P of n processes in a system with unforgeable signatures. $\mathcal{I}_s$ uses atomic $[1,1]$-registers.

```
ATOMIC REGISTERS
    For every processes $i$ and $j$:
        $R_{ij}$: atomic $[1,1]$-register; initially $\langle 0, u_0\rangle_w$.

LOCAL VARIABLES
    $c$: variable of $w$; initially 0
    $tuples$: variable of each $p \in P$; initially $\emptyset$.

WRITE($u$):                          ▷ executed by the writer $w$
1:   $c \leftarrow c + 1$
2:   call W($\langle c, u\rangle_w$)
3:   return done

READ():                              ▷ executed by any reader $p$ in $P$
4:   call R()
5:   if this call returns some tuple $\langle k, u\rangle_w$ then
6:       return $u$
7:   else return $\bot$

W($\langle k, u\rangle_w$):                        ▷ executed by $w$ to do its $k$-th write
8:   for every process $i \in P$ do
9:       $R_{wi} \leftarrow \langle k, u\rangle_w$                    ▷ $\langle k, u\rangle$ signed by $w$
10:  return done

R():                                 ▷ executed by any reader $p \in P$
11:  $tuples \leftarrow \emptyset$
12:  for every process $i \in \{w\} \cup P$ do
13:      if $R_{ip} = \langle \ell, val\rangle_w$ for some $\langle \ell, val\rangle$ validly signed by $w$ then
14:          $tuples \leftarrow tuples \cup \{\langle \ell, val\rangle_w\}$
15:  $\langle k, u\rangle_w \leftarrow$ tuple $\langle \ell, val\rangle_w$ with maximum sequence number $\ell$ in $tuples$
16:  for every process $i \in P$ do
17:      $R_{pi} \leftarrow \langle k, u\rangle_w$
18:  return $\langle k, u\rangle_w$
```

6 Implementation for systems with digital signatures

Algorithm 2 gives a linearizable and wait-free implementation $\mathcal{I}_s$ of a $[1,n]$-register that is writable by process w and readable by a set P of n processes. $\mathcal{I}_s$ uses unforgeable signatures of processes (actually only w does) and *atomic* $[1,1]$-registers between each pair of processes.

As in Algorithm 1, to write a value u the writer w first adds a counter k to form a tuple $\langle k, u\rangle$. It then signs $\langle k, u\rangle$, and the signed tuple is denoted $\langle k, u\rangle_w$. As before, the actual write and read operations are done by lower-level procedures W() and R(), which work as follows:

- To execute W($\langle k, u\rangle_w$), the writer w simply writes $\langle k, u\rangle_w$ in R_{wi} for every process i.
- To execute R(), the process p first reads the $[1,1]$-register R_{ip} of every process i to collect a set *tuples* of the tuples with valid signature of w. Then p selects the tuple $\langle k, u\rangle_w$ with maximum sequence number k in *tuples*, and return this tuple; but before doing so p writes $\langle k, u\rangle_w$ into every $[1,1]$-register R_{pi} to notify every process i that it read this tuple.

The correctness of the implementation $\mathcal{I}_s$ given by Algorithm 2 is stated in Theorem 13. The proof of this theorem is given in [11].

▶ **Theorem 13.** *Consider a system where processes are subject to Byzantine failures and can use unforgeable signatures. For every $n \geq 2$, $\mathcal{I}_s$ is a wait-free linearizable implementation of a $[1,n]$-register from atomic $[1,1]$-registers that tolerates any number of faulty processes.*

7 Concluding remarks

The implementation of registers from weaker registers is a basic problem in distributed computing that has been extensively studied in the context of processes with crash failures. In this paper, we investigated this problem in the context of Byzantine processes failures, with and without process signatures. We first proved that there is no wait-free linearizable implementation of a $[1, n]$-register from atomic $[1, n-1]$-registers. In fact, we showed that this impossibility holds even if every process except the writer can use atomic $[1, n]$-registers, and even under the assumption that the writer can only crash and *at most one* reader can be malicious. This is in sharp contrast to the situation in systems with crash failures only, where there *is* a wait-free linearizable implementation of a $[1, n]$-register even from *safe* $[1, 1]$-registers [14].

In light of this strong impossibility result, we gave an implementation of a $[1, n]$-register from atomic $[1, 1]$-registers that is linearizable (intuitively, "safe") under any combination of Byzantine process failures, but is wait-free (intuitively, "live") only under the assumption that the writer is correct or no reader is malicious; this matches the impossibility result. We also gave an implementation that uses process signatures, and is wait-free and linearizable under any number and combination of Byzantine process failures.

Perhaps surprisingly, none of the above results refers to a ratio of faulty vs. correct processes, such as $n/3$ or $n/2$, that we typically encounter in results that involve Byzantine processes. For example, Mostéfaoui *et al.* [15] prove that one can implement an f-resilient $[1, n]$-register in *message-passing* systems with Byzantine process failures if and only if $f < n/3$. As an other example, Cohen and Keidar [6] show that if $f < n/2$, one can use atomic $[1, n]$-registers to get f-resilient implementations of reliable broadcast, atomic snapshot, and asset transfer objects in systems with Byzantine process failures.

It is worth noting that, since atomic $[1, 1]$-registers can simulate message-passing channels, one can use the f-resilient implementation of a $[1, n]$-register for *message-passing* systems given in [15], to obtain an *f-resilient* implementation of a $[1, n]$-register using atomic $[1, 1]$-registers. But f-resilient implementations (such as the ones given in [6, 15]) require *every* correct process to help the execution of *every* operation, even the operations of *other* processes. In contrast, with wait-free object implementations in shared-memory systems, processes that do not have ongoing operations take no steps.

References

1 Marcos K. Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra Marathe, and Igor Zablotchi. The impact of RDMA on agreement. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC 2019, Toronto, ON, Canada, July 29 - August 2, 2019*, PODC '19, pages 409–418, 2019. doi:10.1145/3293611.3331601.

2 James H Anderson and Mohamed G Gouda. *The virtue of patience: Concurrent programming with and without waiting.* Citeseer, 1990.

3 Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *Journal of the ACM*, 42(1):124–142, 1995. doi:10.1145/200836.200869.

4 B. Bloom. Constructing two-writer atomic registers. *IEEE Trans. Comput.*, 37(12):1506–1514, December 1988.

5 James E. Burns and Gary L. Peterson. Constructing multi-reader atomic values from non-atomic values. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '87, pages 222–231, 1987. doi:10.1109/12.9729.

6 Shir Cohen and Idit Keidar. Tame the Wild with Byzantine Linearizability: Reliable Broadcast, Snapshots, and Asset Transfer. In *35th International Symposium on Distributed Computing (DISC 2021)*, volume 209, pages 18:1–18:18, 2021. doi:10.4230/LIPIcs.DISC.2021.18.

7 Jim Gray. Notes on data base operating systems. In *Operating Systems, An Advanced Course*, pages 393–481, 1978. doi:10.1007/3-540-08755-9_9.

8 S. Haldar and K. Vidyasankar. Constructing 1-writer multireader multivalued atomic variables from regular variables. *J. ACM*, 42(1):186–203, January 1995. doi:10.1145/200836.200871.

9 Maurice P. Herlihy. Impossibility and universality results for wait-free synchronization. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, PODC '88, pages 276–290, 1988. doi:10.1145/62546.62593.

10 Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990. doi:10.1145/78969.78972.

11 Xing Hu and Sam Toueg. On implementing swmr registers from swsr registers in systems with byzantine failures, 2022. doi:10.48550/arXiv.2207.01470.

12 Amos Israeli and Amnon Shaham. Optimal multi-writer multi-reader atomic register. In *Proceedings of the Eleventh Annual ACM Symposium on Principles of Distributed Computing*, PODC '92, pages 71–82, 1992. doi:10.1145/135419.135435.

13 Prasad Jayanti, Tushar Deepak Chandra, and Sam Toueg. Fault-tolerant wait-free shared objects. *J. ACM*, 45(3):451–500, May 1998. doi:10.1145/278298.278305.

14 Leslie Lamport. On interprocess communication Parts I–II. *Distributed Computing*, 1(2):77–101, 1986. doi:10.1007/BF01786227.

15 Achour Mostéfaoui, Matoula Petrolia, Michel Raynal, and Claude Jard. Atomic read/write memory in signature-free byzantine asynchronous message-passing systems. *Theory of Computing Systems*, 60, May 2017. doi:10.1007/s00224-016-9699-8.

16 Richard Newman-Wolfe. A protocol for wait-free, atomic, multi-reader shared variables. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '87, pages 232–248, 1987. doi:10.1145/41840.41860.

17 Gary L. Peterson. Concurrent reading while writing. *ACM Trans. Program. Lang. Syst.*, 5(1):46–55, January 1983. doi:10.1109/SFCS.1986.11.

18 Gary L. Peterson and James E. Burns. Concurrent reading while writing ii: The multi-writer case. In *28th Annual Symposium on Foundations of Computer Science, Los Angeles, California, USA, 27-29 October 1987*, SFCS '87, pages 383–392, 1987. doi:10.1109/SFCS.1987.15.

19 Ambuj K. Singh, James H. Anderson, and Mohamed G. Gouda. The elusive atomic register revisited. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '87, pages 206–221, 1987. doi:10.1145/41840.41858.

20 K. Vidyasankar. Converting Lamport's regular register to atomic register. *Inf. Process. Lett.*, 28(6):287–290, August 1988. doi:10.1016/0020-0190(88)90175-5.

21 K. Vidyasankar. A very simple construction of 1-writer multireader multivalued atomic variable. *Inf. Process. Lett.*, 37(6):323–326, March 1991. doi:10.1016/0020-0190(91)90149-C.

22 Paul M. B. Vitanyi and Baruch Awerbuch. Atomic shared register access by asynchronous hardware. In *27th Annual Symposium on Foundations of Computer Science (sfcs 1986)*, pages 233–243, 1986.

A A wait-free linearizable implementation of a [1,2]-register from atomic [1,1]-registers.

Algorithm 3 gives a *wait-free* linearizable implementation ${I_2}'$ of a $[1, 2]$-register from atomic $[1, 1]$-registers. This algorithm is a simpler version of Algorithm 1 for the valid implementation I_n of a $[1, n]$-register: ${I_2}'$ has only two readers, namely p and q, so preventing new-old inversions among readers is easier. In contrast to Algorithm 1, the code of Algorithm 3 has no parallel threads.

Since the code of Algorithm 3 does not contain any loop or wait statement, it is clear that every call to the WRITE() and READ() procedures by any correct process terminates with a return value in a bounded number of steps; i.e., the implementation ${I_2}'$ is wait-free. The proof that it is a *linearizable* implementation is given in [11]. So we have:

▶ **Theorem 11.** *The implementation ${I_2}'$ (given by Algorithm 3 in Appendix A) is a wait-free linearizable implementation of a $[1, 2]$-register from atomic $[1, 1]$-registers.*

Algorithm 3 Implementation ${I_2}'$ of a $[1, 2]$-register writable by w and readable by p and q. ${I_2}'$ uses some $[1, 1]$-registers.

```
ATOMIC REGISTERS
   $R_{wp}$: $[1,1]$-register; initially (COMMIT, $\langle 0, u_0 \rangle$)
   $R_{wq}$: $[1,1]$-register; initially (COMMIT, $\langle 0, u_0 \rangle$)
   $R_{pq}$: $[1,1]$-register; initially $\langle 0, u_0 \rangle$

LOCAL VARIABLES
   $c$: variable of $w$; initially 0
   $last\_written$: variable of $w$; initially $\langle 0, u_0 \rangle$
   $last\_read$: variable of $q$ initially $\langle 0, u_0 \rangle$

WRITE($u$):                                   ▷ executed by the writer $w$
     $c \leftarrow c + 1$ s
     call W($\langle c, u \rangle$)
    return done

READ():                                       ▷ executed by any reader $r \in \{p, q\}$
    call R$_r$()
    if this call returns some tuple $\langle k, u \rangle$ then
        return $u$
    else return $\perp$

W($\langle k, u \rangle$):                    ▷ executed by $w$ to do its $k$-th write
     $R_{wp} \leftarrow$ (PREPARE, $last\_written$, $\langle k, u \rangle$)
     $R_{wq} \leftarrow$ (PREPARE, $last\_written$, $\langle k, u \rangle$)
     $R_{wp} \leftarrow$ (COMMIT, $\langle k, u \rangle$)
     $R_{wq} \leftarrow$ (COMMIT, $\langle k, u \rangle$)
    $last\_written \leftarrow \langle k, u \rangle$
    return done

R$_p$():                                      ▷ executed by reader $p$
    if $R_{wp}$ = (COMMIT, $\langle k, u \rangle$) for some $\langle k, u \rangle$ then
        $R_{pq} \leftarrow \langle k, u \rangle$
        return $\langle k, u \rangle$
    elseif $R_{wp}$ = (PREPARE, $last\_written$, $-$) for some $last\_written$ then
        return $last\_written$
    else return $\perp$

R$_q$():                                      ▷ executed by reader $q$
     if $R_{wq}$ = (COMMIT, $\langle k, u \rangle$) for some $\langle k, u \rangle$ then
        return $\langle k, u \rangle$
     elseif $R_{wq}$ = (PREPARE, $last\_written$, $\langle k, u \rangle$) for some $last\_written$ and some $\langle k, u \rangle$ then
        if $R_{pq} = \langle k', - \rangle$ for some $k' \geq k$ then
            $last\_read \leftarrow \langle k, u \rangle$
            return $\langle k, u \rangle$
        elseif $last\_read = \langle k', - \rangle$ and some $k' \geq k$ then
            return $\langle k, u \rangle$
        else
            return $last\_written$
     else return $\perp$
```