On implementing SWMR registers from SWSR registers in systems with Byzantine failures

Xing Hu    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 [13], 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.

We first prove that, without signatures, there is no wait-free linearizable implementation of a 1-writer \( n \)-reader register from atomic 1-writer 1-reader registers. In fact, we show a stronger result, namely, even under the assumption that the writer can only crash and at most one reader can be malicious, there is no linearizable implementation of a 1-writer \( n \)-reader register from atomic 1-writer \((n-1)\)-reader registers that ensures that every correct process eventually completes its operations.

In light of this impossibility result, we give two implementations of a 1-writer \( n \)-reader register from atomic 1-writer 1-reader registers that work under different assumptions. The first implementation is linearizable (under any combination of process failures), but it guarantees that every correct process eventually completes its operations only under the assumption that the writer is correct or no reader is malicious — thus matching the impossibility result. The second implementation assumes process signatures; it is bounded wait-free and linearizable under any combination of process failures.

Finally, we show that without process signatures, even if we assume that the writer is correct and at most one of the readers can be malicious, it is impossible to guarantee that every correct reader completes each read operation in a bounded number of steps.

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

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

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 [2, 8, 12, 18, 19, 21]. In particular, it is well-known that in systems where processes are subject to crash failures, it is possible to implement a \( m \)-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 it possible to implement 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 \cite{CohenKeidar2000} 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 \textit{et al.} use atomic \([1, n]\)-registers to solve some agreement problems in hybrid systems with Byzantine process failures \cite{Aquilera2019}. Moreover, Mostéfaoui \textit{et al.} \cite{Mostefaoui2020} prove that, in message-passing systems with Byzantine process failures, there is a \(f\)-resilient implementation of a \([1, n]\)-register if and only if at most \(f < n/3\) processes are faulty.

1.2 Description of the results

In this section, when we write “implementation”, we mean an implementation that is both:

(a) “safe”, i.e., it is linearizable \cite{CohenKeidar2000, Mostefaoui2020}, and (b) “live”, i.e., it ensures that every correct process eventually completes its operations (possibly under some failure assumptions).

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 matching impossibility and possibility results hold. For all \(n \geq 3\):

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

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

Note that result (A) implies that there is no wait-free implementation of a \([1, n]\)-register from atomic \([1, n-1]\)-registers.

This simple version of the results, however, leaves some questions open. One reason is because these results do not distinguish between the different types of faulty processes (recall that 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 an implementation of a \([1, n]\)-register from atomic \([1, 1]\)-registers now possible?

To answer this and similar questions, we partition processes into three separate groups:

(a) those that do not fail, called correct processes, (b) those that fail only by crashing, and (c) those that fail in any other way, called malicious processes. In systems with a mix of such process failures, we prove the following:

(1) For all \(n \geq 3\), there is no 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 every reader is given atomic \([1, n]\)-registers that it can write and all processes can read, and the writer is the only process that does not have atomic \([1, n]\)-registers.

Note that the above results consider safety and liveness as an invisible requirement of a register implementation. But it could be useful to consider each requirement separately. For example, what happens if we want to implement a \([1, n]\)-register with the following properties: (a) it is always safe (i.e., linearizable) and (b) it may lose its liveness (i.e., it may

---

1 Recall that a wait-free implementation guarantees that every correct process eventually completes its operations, regardless of the execution speeds or failures of the other processes \cite{CohenKeidar2000}.
block some read or write operations) only if some specific “pattern/types” of failures occur? We prove that in systems with a mix of process failures:

(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
- In every run of \( I_n \) where the writer is correct or no reader is malicious, every correct process completes all its operations.

So this register implementation is linearizable regardless of which processes fail and how they fail, i.e., it is always “safe”. But it guarantees “liveness” only if the writer is correct or no reader is malicious. If the writer is correct, it tolerates any number of malicious readers.

Note that (1) and (2) are matching impossibility and possibility results. They imply the simpler results (A) and (B) that we stated earlier for processes that are coarsely characterized as either correct or faulty.

If we assume that the writer is correct, the linearizable implementation of result (2) above ensures that every correct reader completes each read in a finite number of steps. This raises the question of whether, if we assume that the writer is correct, there is a linearizable implementation such that every reader completes each read in a bounded number of steps.

We prove that the answer is “No”. More precisely:

(3) For all \( n \geq 3 \), even if we assume that the writer is correct and at most one reader can be malicious, there is no linearizable implementation of a \([1, n]\)-register from atomic \([1, n−1]\)-registers that ensures that every correct reader completes every read in a bounded number of steps.

The above results are for the case that the implemented register has at least \( n = 3 \) readers. For the special case that \( n = 2 \), we give a simple implementation of a \([1, 2]\)-register from atomic \([1, 1]\)-registers that is bounded wait-free: all correct processes are guaranteed to complete their operations in a bounded number of steps regardless of which processes fail and how they fail.

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 impossibility result (1), we show that with signatures for all \( n \geq 2 \), there is an implementation of \([1, n]\)-register from atomic \([1, 1]\)-registers that is bounded wait-free.

We conclude the paper with a result about implementations from regular registers [13]. Recall that, in contrast to atomic registers, regular registers allow “new-old” inversions in the values that processes read. It is well-known that in systems with crash failures, it is easy to implement a wait-free linearizable \([1, n]\)-register from regular \([1, n]\)-registers. Here we show that in systems with Byzantine failures, such an implementation is impossible: for \( n \geq 3 \), even if we assume that the writer can only crash and at most one reader can be malicious, there is no linearizable implementation of a \([1, n]\)-register from regular \([1, n]\)-registers that ensures that every correct process eventually completes its operations.

## 2 Result techniques

The techniques that we used to obtain our main possibility and impossibility results 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.

---

2 So all processes, including the writer, are given regular registers that all the \( n \) readers can read.
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, and (4) 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 step) in a way that maintains the property that some correct reader reads 1 and at most one reader in the run is malicious. As we successively remove the steps of the writer, the identity of the reader that reads 1, and the identity of the reader 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.

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). Note that a process executes steps of a register implementation only when it executes its *own* operations on the register, i.e., only within
the intervals of these operations.

3.3 Implementation liveness properties

All the register implementations that we consider satisfy the following liveness property:

▷ Definition 1 (Termination). Every correct process completes every operation in a finite number of its own steps.

As we will see, termination may rely on some failure assumptions. For example, the register implementation that we give in Section 5.4 (Algorithm 1, Theorem 48) satisfies the Termination property under the assumption that either the writer is correct or no reader is malicious. In contrast to the Termination property, wait-freedom and bounded wait-freedom are liveness properties that do not rely on any failure assumptions.

▷ Definition 2 (Wait-freedom). Every correct process completes every operation in a finite number of its own steps, regardless of the execution speeds or failures of the other processes.

▷ Definition 3 (Bounded wait-freedom). Every correct process completes every operation in a bounded number of its own steps, regardless of the execution speeds or failures of the other processes.

3.4 Linearizability of register implementations

Roughly speaking, 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, however, the precise definition of linearizability depends on whether we assume that processes can only crash (as it was assumed in), or they can also fail in a “Byzantine way”. We now explain this for the special case of 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: intuitively, (1) every read operation reads the value written by a concurrent or immediately preceding write operation, and (2) there are no “new-old” inversions in the values read. 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 4. 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 \( w \) of the implemented register (this is well-defined because each process, including the writer, applies its operations sequentially).

▷ Definition 5 (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:

\[^3\] In a preliminary version of this paper, an implementation that satisfies the Termination property (under some failure assumption) was said to be wait-free (under this failure assumption). In particular, the register implementation given in Section 5.4 was said to be wait-free under the assumption that the writer is correct or no reader is malicious. But this use of the term “wait-free” is not conventional and can be misleading. Here we reserve the term “wait-free” for implementations that satisfy the Termination property unconditionally, as in.

\[^4\] Linearizable (implementations of) registers, however, are not equivalent to atomic registers. In fact, Golab, Higham and Woelfel have shown that with a strong adversary, some randomized algorithms that “work correctly” under the assumption that processes use atomic registers, do not work if they use linearizable register implementations instead of atomic registers.
On implementing SWMR registers from SWSR registers in systems with Byzantine failures

Property 1 [Reading a “current” value] If a read operation \( R \) returns the value \( v \) then:
1. There is a write \( v \) operation that immediately precedes \( R \) or is concurrent with \( R \), or
2. \( v = v_0 \) 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 6 (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:
1. There is a write \( v \) operation that immediately precedes \( R \) or is concurrent with \( R \), or
2. \( v = v_0 \) 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 6 reduces to Definition 5. Cohen and Keidar were the first to define linearizability for arbitrary objects in systems with Byzantine failures [5], and their definition generalizes the definition of register linearizability in such systems given by Mostéfaoui et al. in [14]. Definition 6 above (which is also for register linearizability) is consistent with both.

4 Impossibility result

We now prove that there is no wait-free linearizable implementation of a [1, \( n \)]-register from atomic [1, \( n-1 \)]-registers. In fact, we show a stronger result: even under the assumption that the writer can only crash and at most one reader can be malicious, there is no linearizable implementation of a [1, \( n \)]-register from atomic [1, \( n-1 \)]-registers that ensures that every correct process eventually completes its operations.

Theorem 7. For all \( n \geq 3 \), in a system with \( n+1 \) processes that are subject to Byzantine failures, there is no linearizable implementation of a [1, \( n \)]-register from atomic [1, \( n-1 \)]-registers that satisfies the Termination property, 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 an implementation \( I \) of a [1, \( n \)]-register \( R \) from atomic [1, \( n-1 \)]-registers that is linearizable (i.e., it satisfies the Register Linearizability property) and ensures that all correct processes complete their operations (i.e., it satisfies the Termination property), under the assumption that the writer \( w \) of \( R \) can only crash and at most one of the \( n \) readers of \( R \) can be malicious.

In [5] [14], however, processes that are subject to Byzantine failures are partitioned into only two groups, namely, correct processes and faulty processes. Thus the reader of a register that just crashes is, by definition, faulty. So, as with all other faulty processes, by the linearizability definitions in [5] [14] it is exempt from any requirement, e.g., it is allowed to read a stale value. Definition 3 avoids this by leveraging our subdivision of faulty processes into those that only crash and those that are malicious.
Figure 1 Run $A'_n$

Figure 2 Run $A_m$

Figure 3 A run with property $P_k$

Figure 4 Run $A_k$

Figure 5 Run $B_{k-1}$

Figure 6 Run $C_{k-1}$

Figure 7 Run $D_{k-1}$

Figure 8 Run $E'_{k-1}$

Figure 9 Run $F'_{k-1}$

Figure 10 Run $A_1$

Figure 11 Run $A_0$
We now construct a sequence of runs of \( I \) that leads to a contradiction. In all these runs, the initial value of the implemented \( R \) is 0, the writer \( w \) invokes only one operation into \( R \), namely a write of 1, and each reader reads \( R \) at most once (i.e., \( R \) is only a “one-shot” binary register). Moreover, in each of these runs the writer crashes (but it is not malicious) and there is at most one malicious reader; the other \( n - 1 \) readers are correct. Thus, these runs of \( I \) must satisfy the linearizability Properties 1 and 2 of Register Linearizability (Definition 6), and every correct reader must complete any read operation that it invokes.

▶ Definition 8. Let \( s \) be any step that the writer \( w \) takes when executing the implementation \( I \) of \( R \). Step \( s \) is invisible to a reader \( p \) if \( s \) is either a local step of \( w \), or the reading or the writing of an atomic \([1, n-1]\)-register that is not readable by \( p \).

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

▶ Observation 9. Let \( s \) be any step that the writer \( w \) takes when executing the implementation \( I \) of \( R \). Step \( s \) is invisible to at least one of the \( n \) readers.

Let \( A'_{m} \) be the following run of \( I \) (see Figure 1):

- The readers do not invoke any read operations, and so they take no steps.
- The writer \( w \) invokes an operation to write 1 on \( R \). By the Termination property of the implementation, it completes this operation in a finite number of steps.
- During this write operation, \( w \) takes a sequence of steps \( s^1, \ldots, s^m \) such that each \( s^i \) is either a local step, or the reading or the writing of an atomic \([1, n-1]\)-register (\( s^0 \) is the invocation step of the write operation, and \( s^m \) is the response step of this operation).
- Let \( t^w_m \) be the time when step \( s^m \) occurs.
- After the time \( t^w_m \) when \( w \) completes its write operation, \( w \) crashes.

From the run \( A'_{m} \) of \( I \), it is clear that the following run is also a run of \( I \) (see Figure 2):

- **Run \( A_{m} \):**
  - The writer \( w \) behaves exactly as in \( A'_{m} \).
  - All the readers are correct.
  - Let \( q \) be a reader such that step \( s^m \) is invisible to \( q \) (by Observation 9, this reader exists).
  - After the writer \( w \) crashes at time \( t^w_m \), \( q \) invokes a read operation on \( R \). By the Termination property of the implementation, \( q \) completes its read operation. By the linearizability properties of \( I \), this read operation on \( R \) returns 1.
  - All the other readers do not invoke any read operations, and so they take no steps.

▶ Definition 10. For every \( k, 1 \leq k \leq m \), a run of \( I \) has property \( P_k \) if the following holds:

1. Up to and including time \( t^w_k \), all processes behave exactly as in \( A_m \), that is:
   - \( w \) takes steps \( s^0, s^1, \ldots, s^k \)
   - All the readers take no steps.
2. After taking step \( s^k \) at time \( t^w_k \), \( w \) crashes before taking further steps.
3. There is a reader \( x \) that is correct such that step \( s^k \) is invisible to \( x \). After time \( t^w_k \), reader \( x \) starts and completes a read operation on \( R \) that returns 1.
4. There is a reader \( y \neq x \) that may be correct or malicious. After time \( t^w_k \), reader \( y \) may or may not take steps.
5. There is a set \( Z \) of \( n-2 \) distinct readers other than \( x \) and \( y \) that are correct and take no steps.

Note that since \( n \geq 3 \), the set \( Z \) contains at least one reader. Furthermore, all the readers that take steps do so after time \( t^w_k \).

A run of \( I \) with property \( P_k \) is shown in Figure 3. In this figure and all the subsequent ones, correct readers are in black font, while the reader that may be malicious is colored red (this reader may have taken some steps after time \( t^w_k \), but these are not shown in the figure). The “\( \not\in x \)” on top of a step \( s^i \) means that \( s^i \) is invisible to the reader \( x \). The symbol \( \not\in \) indicates where the crash of the writer \( w \) occurs.
Note that the run $A_m$ of $\mathcal{I}$ satisfies property $P_m$: the reader denoted $x$ in property $P_m$ is the reader $q$ of run $A_m$, the reader $y$ of $P_m$ is an arbitrary reader other than $q$ in $A_m$, and the set $Z$ of $P_m$ is the set of the remaining $n - 2$ readers in $A_m$. So we have:

\textbf{Observation 11.} Run $A_m$ of $\mathcal{I}$ has property $P_m$.

\textbf{Claim 12.} For every $k$, $1 \leq k \leq m$, there is a run of $\mathcal{I}$ that has property $P_k$.

\textbf{Proof.} We prove the claim by a backward induction on $k$, starting from $k = m$.

\textbf{Base Case:} $k = m$. This follows directly from Observation 11.

\textbf{Induction Step:} Let $k$ be such that $1 < k \leq m$.

- **Run $A_k$.** Suppose there is a run $A_k$ of $\mathcal{I}$ that has property $P_k$ (this is the induction hypothesis). We now show that there is a run $A_{k-1}$ of $\mathcal{I}$ that has property $P_{k-1}$.

  Since run $A_k$ of $\mathcal{I}$ satisfies $P_k$, the following holds in $A_k$ (see Figure 4):
  - Up to and including time $t^k_{w}$, all processes behave exactly as in $A_m$.
  - After taking step $s^k$ at time $t^k_{w}$, $w$ crashes before taking further steps.
  - There is a reader $q$ that is correct such that step $s^k$ is invisible to $q$. After time $t^k_{w}$, reader $q$ starts and completes a read operation on $R$ that returns 1.
  - There is a reader $p \neq q$ that may be correct or malicious. After time $t^k_{w}$, reader $p$ may or may not take steps.
  - There is a set $Z$ of $n-2$ distinct readers other than $p$ and $q$ that are correct and take no steps.

- **Run $B_{k-1}$.** From the run $A_k$ of $\mathcal{I}$ we construct the following run $B_{k-1}$ of $\mathcal{I}$ (Figure 5).

  Intuitively, $B_{k-1}$ 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”). Run $B_{k-1}$ is possible because: (1) even though $p$ may have “noticed” the removal of step $s^k$, $p$ may be malicious (all the other readers are correct in this run), and $p$ behaves exactly as in $A_k$, 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 $A_k$; so $q$ behaves as in $A_k$, and in particular $q$ reads 1 in $B_{k-1}$ as in $A_k$.

  More precisely in $B_{k-1}$:
  - All processes behave exactly as in $A_k$ up to and including time $t^k_{w}-1$.
  - After taking step $s^k-1$ at time $t^k_{w}-1$, the writer $w$ crashes before taking step $s^k$.
  - All the readers in $Z$ are correct and take no steps, exactly as in $A_k$.
  - $p$ behaves exactly as in $A_k$. This is possible because even though $p$ may have “noticed” the removal of step $s^k$, $p$ may be malicious (all the other readers are correct in this run).
  - $q$ behaves exactly as in $A_k$. In particular, after time $t^k_{w}$, $q$ starts and completes a read operation on $R$ that returns 1. This is possible because $q$ cannot distinguish between $A_k$ and $B_{k-1}$: $s^k$ is invisible to $q$, and $p$ and all the readers in $Z$ behave exactly as in $A_k$.

  Note that in $B_{k-1}$ all processes behave exactly as in $A_m$ up to and including time $t^k_{w-1}$.

  There are two cases:

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

  \textbf{Case 2:} $s^k-1$ is visible to $q$. Then, by Observation 9, $s^k-1$ is visible to $p$ or to some $r' \in Z$.

  - **Run $C'_{k-1}$.** Let $r$ be any reader in $Z$. From the run $B_{k-1}$ of $\mathcal{I}$ we construct the following run $C'_{k-1}$ of $\mathcal{I}$ (Figure 6). $C'_{k-1}$ 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 may have taken so far, and then correct reader $r \in Z$ reads 1 (this is the only value that $r$ can read, since correct $q$ previously read 1). More precisely:
    - $C'_{k-1}$ is an extension of $B_{k-1}$: all processes behave exactly as in $B_{k-1}$ up to and including the time when $q$ completes its read operation on $R$.

  These steps are not shown in Figure 4.
All the readers in $Z - \{r\}$ are correct and take no steps.

After the correct reader $q$ completes its read operation on $R$:
- $q$ takes no steps.
- $p$ resets all the atomic registers that it can write to their initial values. Process $p$ can do so because it may be malicious (all the other readers are correct in this run).

Let $t'_p$ be the time when $p$ completes all the register resettings.

Correct reader $r$ starts a read operation on $R$ after time $t'_p$. It takes no steps before this read. By the Termination property of the implementation, $r$ completes its read operation. Since $w$ is not malicious, and the read operation by correct $q$ precedes the read operation by $r$ and returns 1, by the linearizability of $I$, the read operation by correct reader $r$ also returns 1.

Note that in $C_{k-1}^r$ all processes behave exactly as in $A_m$ up to and including time $t_{w}^{k-1}$.

**Run $D_{k-1}^r$.** We can now construct the following run $D_{k-1}^r$ of $I$ (Figure 7). Intuitively, we obtain $D_{k-1}^r$ from $C_{k-1}^r$ by removing all the steps of $p$. So reader $p$ (which was malicious in $C_{k-1}^r$) is now a correct process that takes no steps. Despite this removal, $q$ behaves exactly as in $C_{k-1}^r$ because $q$ (which was correct in $C_{k-1}^r$) may now be malicious. The writer $w$ also behaves exactly as in $C_{k-1}^r$ because it cannot see the removal of $p$’s steps: they all occur after time $t_{w}^{k-1}$. Correct reader $r$ behaves exactly as in $C_{k-1}^r$ because it also cannot see the removal of $p$’s steps: in both $C_{k-1}^r$ and $D_{k-1}^r$, $r$ does not “see” any steps of $p$.

So $r$ reads 1 in $D_{k-1}^r$ as in $C_{k-1}^r$.

More precisely in $D_{k-1}^r$:
- $w$ behaves exactly as in $C_{k-1}^r$.
- All the readers in $Z - \{r\}$ are correct and take no steps, as in $C_{k-1}^r$.
- $p$ is correct and it takes no steps. So all the atomic registers that it can write retain their initial values.
- $q$ behaves exactly as in $C_{k-1}^r$. This is possible because even though $q$ may have “noticed” the removal of $p$’s steps, $q$ may be malicious (all the other readers are correct in this run).
- $r$ behaves exactly as in $C_{k-1}^r$. In particular, after time $t'_r$, reader $r$ starts and completes a read operation on $R$ that returns 1. This is possible because $r$ cannot distinguish between $C_{k-1}^r$ and $D_{k-1}^r$: $r$ cannot see the removal of $p$’s steps, and $q$ and all the readers in $Z - \{r\}$ behave exactly as in $C_{k-1}^r$.

Note that in $D_{k-1}^r$ all processes behave exactly as in $S$ up to and including time $t_{w}^{k-1}$.

If $s^{k-1}$ is invisible to reader $r$, it is clear that the run $D_{k-1}^r$ of $I$ has property $P_{k-1}$.

Recall that (1) the reader $r$ above is an arbitrary reader 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 run $D_{k-1}^r$ of $I$ has property $P_{k-1}$, as we wanted to show.

**Subcase 2b:** $s^{k-1}$ is invisible to $p$.

**Run $E_{k-1}^r$.** We construct the continuation $E_{k-1}^r$ of $D_{k-1}^r$ 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, and then correct reader $p$ starts a read operation on $R$. By the Termination property of the implementation, this read operation by $p$ must complete. Since correct $r$ previously read 1, by the linearizability of $I$, $p$ must also read 1.

More precisely in $E_{k-1}^r$:
- $E_{k-1}^r$ is an extension of the run $D_{k-1}^r$: all processes behave exactly as in $D_{k-1}^r$ up to and including the time when $r$ completes its read operation on $R$.
- All the readers in $Z - \{r\}$ are correct and take no steps, as in $D_{k-1}^r$.
- After the correct reader $r$ completes its read operation on $R$:
  - $r$ takes no steps.

If $n = 3$, then the set $Z - \{r\}$ is empty.
\( q \) resets all the atomic registers that it can write to their initial values. Process \( q \) can do so because it may be malicious (all the other readers are correct in this run).

Let \( t'_{q} \) be the time when \( q \) completes all the register resettings.

- Correct reader \( p \) starts a read operation on \( R \) after time \( t'_{q} \). It takes no steps before this read. By the Termination property of the implementation, \( p \) completes its read operation.

Since \( w \) is not malicious, and the read operation by correct \( r \) precedes the read operation by \( p \) and returns 1, by the linearizability of \( \mathcal{I} \), the read operation by correct reader \( p \) also returns 1.

Note that in \( E'_{k-1} \) all processes behave exactly as in \( A_{m} \) up to and including time \( t'_{w} \).

- Run \( F'_{k-1} \): Finally, we construct the run \( F'_{k-1} \) of \( \mathcal{I} \) by removing all the steps of \( q \) from \( E'_{k-1} \) (see Figure 9); so \( q \) (which was malicious in \( E'_{k-1} \)) is now a correct process that takes no steps. Despite this removal, \( r \) behaves exactly as in \( E'_{k-1} \) because \( r \) (which was correct in \( E'_{k-1} \)) may now be malicious. The writer \( w \) also behaves exactly as in \( E'_{k-1} \) because it cannot see the removal of \( q \)'s steps: they all occur after time \( t'_{w} \). Finally, correct \( p \) behaves exactly as in \( E'_{k-1} \) because it also cannot see the removal of \( q \)'s steps: in both \( E'_{k-1} \) and \( F'_{k-1} \), \( p \) does not “see” any steps of \( q \). So \( p \) reads 1 in \( F'_{k-1} \) as in \( E'_{k-1} \).

More precisely in \( F'_{k-1} \):

- \( w \) behaves exactly as in \( E'_{k-1} \).
- All the readers in \( Z - \{ r \} \) are correct and take no steps, as in \( E'_{k-1} \).
- \( q \) is correct and it takes no steps. So all the atomic registers that it can write retain their initial values.
- \( r \) behaves exactly as in \( E'_{k-1} \). This is possible because even though \( r \) may have “noticed” the removal of \( q \)'s steps, \( r \) may be malicious (all the other readers are correct in this run).
- \( p \) behaves exactly as in \( E'_{k-1} \). In particular, after time \( t'_{q} \) reader \( p \) starts and completes a read operation on \( R \) that returns 1. This is possible because \( p \) cannot distinguish between \( E'_{k-1} \) and \( F'_{k-1} \); \( p \) cannot see the removal of \( q \)'s steps, and \( r \) and all the readers in \( Z - \{ r \} \) behave exactly as in \( E'_{k-1} \).

Note that in \( F'_{k-1} \) all processes behave exactly as in \( A_{m} \) up to and including time \( t'_{w} \).

Since \( s^{k-1} \) is invisible to \( p \), it is clear that the run \( F'_{k-1} \) of \( \mathcal{I} \) has property \( P_{k-1} \).

The above concludes the proof of the Induction Step of Claim 12, we proved that, in all possible cases, there is a run of \( \mathcal{I} \) that has property \( P_{k-1} \), as we needed to show.

By the Claim 12 that we just proved, the implementation \( \mathcal{I} \) of \( R \) has a run \( A_{1} \) with property \( P_{1} \). By this property, the following holds in \( A_{1} \) (see Figure 10):

- Up to and including time \( t'_{w} \), all processes behave exactly as in \( A_{m} \).
- After taking step \( s^{1} \) at time \( t'_{w} \), \( w \) crashes before taking further steps.
- There is a reader \( q \) that is correct such that step \( s^{1} \) is invisible to \( q \). After time \( t'_{w} \), reader \( q \) starts and completes a read operation on \( R \) that returns 1.
- There is a reader \( p \neq q \) that may be correct or malicious. After time \( t'_{w} \), reader \( p \) may or may not take steps.
- There is a set \( Z \) of \( n-2 \) distinct readers other than \( p \) and \( q \) that are correct and take no steps.

From the run \( A_{1} \) of \( \mathcal{I} \) we construct the following run \( A_{0} \) of \( \mathcal{I} \) (Figure 11). Intuitively, \( A_{0} \) is the same as \( A_{1} \) except that the writer is correct and does not take any steps (i.e., \( w \) does not invoke a write 1 operation on \( R \)), but all the readers behave the same as in \( A_{1} \) and so \( q \) still reads 1.

This run of \( \mathcal{I} \) is possible because: (1) even though \( p \) may have “noticed” that \( w \) does not take the step \( s^{1} \), \( p \) may be malicious (all the other readers are correct in this run), and \( p \) behaves exactly as in \( A_{1} \), and (2) \( q \) cannot distinguish between \( A_{1} \) and \( A_{0} \) because \( s^{1} \) is invisible to \( q \), and \( p \) and all the readers in \( Z \) behave as in \( A_{1} \). So \( q \) reads 1 from \( R \) in \( A_{0} \) exactly as in \( A_{1} \). Since the initial value of the implemented register \( R \) is 0, run \( A_{0} \) of the implementation \( \mathcal{I} \) of \( R \) violates the linearizability of \( \mathcal{I} \) — a contradiction that concludes the proof of Theorem 7.
It is easy to verify that the above proof holds (without any change) even if every reader is given atomic \([1, n]\)-registers that it can write and all other processes can read, and the writer is the only process that does not have an atomic \([1, n]\)-register. Thus:

**Theorem 13.** For all \(n \geq 3\), in a system with \(n + 1\) processes that are subject to Byzantine failures, there is no linearizable implementation of a \([1, n]\)-register that satisfies Termination, even under the assumption that:
- the writer \(w\) of the implemented \([1, n]\)-register can only crash and at most one reader can be malicious, and
- \(w\) has atomic \([1, n - 1]\)-registers, and every reader has atomic \([1, n]\)-registers.

### 5 Register implementation algorithm

We now give an implementation of a \([1, n]\)-register from atomic \([1, 1]\)-registers in systems with Byzantine process failures; this implementation is linearizable, and it satisfies the Termination property 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 we define below.

**Definition 14.** A register implementation is valid if the following holds:
- It is linearizable.
- If the writer is correct or no reader is malicious, it satisfies the Termination property.

Note that, when executed in a system where processes can only crash, a valid register implementation is linearizable and “terminating” (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 is 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.

**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 some point during its execution interval. 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 \(R = \text{val}\) then . . .” means “read register \(R\) and if the value read is equal to \(\text{val}\) then . . .”

The shared registers used by the implementation are as follows:

- \(R_{s,s'}\) is an atomic \([1,1]\)-register writable by process \(s\) and readable by process \(s'\).
- \(R_{wQ}\) 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 \(q \in Q\).

**Algorithm description.** The implementation \(I_n\) of a \([1,n]\)-register from \([1,n-1]\)-registers consists of two procedures, namely \(\text{Write}(u)\) for the writer \(w\), and \(\text{Read}(u)\) for each reader \(r\) in \(\{p\} \cup Q\). To write a value \(u\), the writer \(w\) executes \(\text{Write}(u)\). If \(u\) is the \(k\)-th value written by \(w\), \(\text{Write}(u)\) first forms the unique tuple \((k, u)\) and then it calls the lower-level write procedure \(w((k,u))\) to write this tuple. Intuitively, \(\text{Write}(\cdot)\) 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 \(\text{Read}(\cdot)\), and this in turn calls a lower-level read procedure \(r_{p}(\cdot)\) that reads tuples written by \(w\). There are two versions of the procedure \(r_{p}(\cdot)\): one used when \(r = p\) and one used when \(r \in Q\). If \(r_{p}(\cdot)\) returns a tuple of the form \((j,v)\), then \(\text{Read}(\cdot)\) strips the counter \(j\) from the tuple and returns the value \(v\) as the value read (otherwise \(\text{Read}(\cdot)\) returns \(\bot\) to indicate a read failure).

Thus the lower-level procedures \(w(\cdot)\), \(r_{p}(\cdot)\), and \(r_{q}(\cdot)\) for each \(q \in Q\), are executed to write and read unique tuples of the form \((k,u)\). We now describe how these procedures work.

- To execute \(w((k,u))\), process \(w\) first writes \((\text{prepare}, \text{last_written}, (k,u))\) in the \(R_{wp}\) register that \(p\) can read, and then in the \(R_{wQ}\) register that every process in \(Q\) can read.

\[\text{If } s = s', \text{this “shared register” is actually just a local register of process } s.\]
On implementing SWMR registers from SWSR registers in systems with Byzantine failures

**Algorithm 1** Implementation $I_w$ 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 $(\text{commit},(0,u_0))$
- $R_{wQ}$: $[1,1]$-register; initially $(0,u_0)$

For all processes $q$ and $q'$ in $Q$:
- $R_{wp}$: $[1,1]$-register; initially $(0,u_0)$
- $R_{wQ}$: $[1,n-1]$-register; initially $(0,u_0)$

### IMPLEMENTED REGISTERS
- $R_{wp}$: $[1,n-1]$-register; initially $(\text{commit},(0,u_0))$
- $R_{wQ}$: $[1,n-1]$-register; initially $(0,u_0)$

#### LOCAL VARIABLES
- $c$: variable of $w$; initially 0
- $last\_written$: variable of $w$; initially $(0,u_0)$
- $previous\_k$: variable of $p$; initially 0

#### WRITE($u$):
- $\triangleright$ executed by the writer $w$
  1. $c \leftarrow c + 1$
  2. call $W((c,u))$
  3. return done

#### READ():
- $\triangleright$ 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 $\bot$

#### $W((k,u))$:
- $\triangleright$ executed by $w$ to do its $k$-th write
  8. $R_{wp} \leftarrow (\text{prepare}, last\_written, (k,u))$
  9. $R_{wQ} \leftarrow (\text{prepare}, last\_written, (k,u))$
  10. $R_{wp} \leftarrow (\text{commit}, (k,u))$
  11. $R_{wQ} \leftarrow (\text{commit}, (k,u))$
  12. $last\_written \leftarrow (k,u)$
  13. return done

#### $R_p()$:
- $\triangleright$ executed by reader $p$
  14. if $R_{wp} = (\text{commit}, (k,u))$ for some $(k,u)$ with $k \geq previous\_k$ then
     15. $R_{wp} \leftarrow (k,u)$
     16. $previous\_k \leftarrow k$
     17. return $(k,u)$
  18. else if $R_{wp} = (\text{prepare}, last\_written, \bot)$ for some last\_written then
     19. return last\_written
  20. else return $\bot$

#### $R_q()$:
- $\triangleright$ executed by any reader $q$ in $Q$
  21. if $R_{wQ} = (\text{commit}, (k,u))$ for some $(k,u)$ then
     22. return $(k,u)$
  23. else if $R_{wQ} = (\text{prepare}, last\_written, (k,u))$ for some last\_written and some $(k,u)$ then
     24. cobegin
     25. // THREAD 1
     26. repeat forever
     27. if $R_{wQ} = (\text{commit}, (k',\bot))$ for some $k' \geq k$ then
        28. return $(k,u)$
     29. else if $R_{wQ} = (\text{prepare}, \bot, (k',\bot))$ for some $k' > k$ then
        30. return $(k,u)$
     31. // THREAD 2
     32. if $R_{wQ} = (k',\bot)$ for some $k' \geq k$ then
        33. for every process $q' \in Q$ do $R_{wq'} \leftarrow (k,u)$
        34. return $(k,u)$
     35. else if $R_{wQ} = (k',\bot)$ for some $q' \in Q$ and some $k' \geq k$ then
        36. for every process $q' \in Q$ do $R_{wq'} \leftarrow (k,u)$
        37. return $(k,u)$
     38. else exit THREAD 2
     39. coend
  40. else return last\_written

$last\_written$ is the last tuple written by $w$ before $(k,u)$ (so last\_written $= (k-1,u')$ for some $u'$). Then, $w$ writes $(\text{commit}, (k,u))$ into $R_{wp}$ and then into $R_{wQ}$.
Finally, if \( p \) reads \((\text{commit}, (k, u))\) with a \( k \) at least as big as those it saw before, it returns \((k, u)\) as the tuple read (line 17), just before doing so, however, it writes \((k, u)\) 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” inversions in the tuples read.

If \( p \) reads \((\text{prepare}, \text{last} \_\text{written}, \text{–})\) (line 18), then it returns \text{last} \_\text{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 \( \bot \) (the writer is surely malicious).

- To execute \( r_q() \), process \( q \in Q \) reads \( R_{wq} \). If \( q \) reads \((\text{commit}, (k, u))\) (line 21), it just returns \((k, u)\) as the tuple read in line 22 (without “warning” other processes).

If \( q \) reads \((\text{prepare}, \text{last} \_\text{written}, (k, u))\) (line 23), then \( q \) cannot simply return \text{last} \_\text{written} as the tuple read: this is because \( p \) could have already read \((\text{commit}, (k, u))\) from \( R_{wp} \) and so \( p \) could have already read the “newer” tuple \((k, u)\) with \( R_{pq}() \). So \( q \) must determine whether to return \text{last} \_\text{written} or \((k, u)\). To do so, \( q \) forks two threads and executes them in parallel (we will explain why below).

If \( q \) does not read values of the form \((\text{prepare}, \text{last} \_\text{written}, (k, u))\) or \((\text{commit}, (k, u))\) from \( R_{wq} \), then \( q \) just returns \( \bot \) (we are surely malicious).

In THREAD 1, process \( q \) keeps reading \( R_{wq} \): if it ever reads \((\text{commit}, (k', \text{–}))\) with \( k' \geq k \), or \((\text{prepare}, \text{–}, (k', \text{–}))\) with \( k' > k \), it simply returns \((k, u)\) as the tuple read. Note that if the writer \( w \) is correct, then \( q \) cannot spin forever in this thread without returning \((k, u)\).

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 \((k, u)\).

- If \( q \) sees that \( R_{pq} \) contains a tuple at least as “new” as \((k, u)\) (line 30), then \( q \) returns \((k, u)\) as the tuple read (line 32): but before doing so, \( q \) successively writes \((k, u)\) in each register \( R_{q'q} \) 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 \((k, u)\) (line 33), then \( q \) reads \( R_{pq} \) again (line 34) (so \( q \) does not simply “trust” \( q' \) and return \((k, u)\)).

If \( q \) sees that \( R_{pq} \) contains a tuple at least as “new” as \((k, u)\) (line 34), then \( q \) successively writes \((k, u)\) to every register \( R_{q'q} \) such that \( q' \in Q \) (line 35), and it returns \((k, u)\) as the tuple read (line 36); else \( q \) exits THREAD 2 (so in this case only THREAD 1 remains).

- Finally, if \( q \) does not see that \( R_{pq} \) or any \( R_{q'q} \) contains a tuple at least as “new” as \((k, u)\) (in lines 30 and 33), then \( q \) returns \text{last} \_\text{written} (line 38).

**Why two parallel threads?** In a nutshell, this is to guarantee the Termination property of \( I_n \) in runs where the writer is correct or no reader is malicious (this property is required for the 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 that correct readers complete their 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 can also prevent a correct reader $q$ from reading any tuple in THREAD 2 as follows. When $q$ executes $R_q()$ the following can occur: (1) in line 33 $q$ sees that some $R'_{q,q}$ contains $(k', -)$ with $k' \geq k$, but (2) in line 34 $q$ sees that $R_{p,Q}$ does not contain $(k', -)$ 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 $q$ exits THREAD 2 without returning any tuple.

We now prove the correctness of the $[1,n]$-register implementation $I_n$ given in Figure 1 more precisely, we show that if the $[1,n-1]$-registers that $I_n$ uses are valid, then $I_n$ is valid (Theorem 17). Since this proof may be distracting, in a first reading of the paper a reader may want to skip this proof and go directly to Theorem 17.

**Correctness of the implementation $I_n$.**

We must show that $I_n$ is valid under the assumption that the $[1,n-1]$-register implementations that it uses, namely $R_{w,Q}$ and $R_{p,Q}$, are also valid. So in this proof we assume:

- **Assumption 1.** The implementations of the $[1,n-1]$-registers $R_{w,Q}$ and $R_{p,Q}$ that $I_n$ uses are valid.

We show that under this assumption, the implementation $I_n$ of the $[1,n]$-register is also valid, that is:

- $I_n$ is linearizable, and
- If the writer is correct or no reader is malicious, $I_n$ satisfies the Termination property.

Henceforth, we consider an arbitrary run $E$ of the implementation $I_n$ given in Figure 1.

By Assumption 1 the implemented registers $R_{w,Q}$ and $R_{p,Q}$ that $I_n$ uses are linearizable; moreover, the atomic registers that $I_n$ uses are also (trivially) linearizable. So operations on these registers appear to take effect instantaneously at some point (the “linearization point”) in their execution intervals. Therefore, without loss of generality, we can assume that in the run $E$ the operations on the registers that $I_n$ uses are sequential.

In the proof, we use the following notation (where $R$ is any atomic or implemented register used by $I_n$):

- “process $x$ reads $R = v$ in line $\ell$ of $R()$” means that process $x$ reads register $R$, this read returns the value $v$, and both occur in line $\ell$ of the read procedure $R()$.
- “process $x$ reads $R = u$ before process $y$ reads $R' = v$” means that the read operation by $x$ (which returns $u$) precedes the read operation by $y$ (which returns $v$).
- “process $x$ writes $u$ in $R$ before process $y$ writes $v$ in $R'$” means that the write $u$ operation by $x$ precedes the write $v$ operation by $y$.

We first show that $I_n$ is linearizable. Then we prove that it satisfies the Termination property if the writer is correct or no reader is malicious.

**Linearizability of $I_n$.** We consider two cases:

**Case 1:** The writer $w$ of the register implemented by $I_n$ is malicious. By Definition 6 $I_n$ is (trivially) linearizable in this case.

**Case 2:** The writer $w$ of the register implemented by $I_n$ is not malicious.

For this case, we now prove that the read and write operations of the implemented register satisfy the linearizability Properties 1 and 2 of Definition 6. In the following:

- $v_0$ is the initial value of the register that $I_n$ implements.
- For $k \geq 1$, $u_k$ denotes the $k$-th value written by $w$ using the procedure Write(). More precisely, if $w$ calls Write() with a value $u$ and this is its $k$-th call of Write(), then $u_k$ is $u$.
- $v_0$ is $(0, u_0)$.
- For $k \geq 1$, $v_k$ denotes the $k$-th value written by $w$ using the procedure W().

- **Observation 15.** For all $k \geq 0$, $v_k = (k, u_k)$.
By a slight abuse of notation:
- a write operation performed by executing the WRITE() or W() procedures with a value \( x \) is denoted \( \text{WRITE}(x) \) or \( w(x) \), respectively.
- A read operation performed by executing the READ() or R() procedures that return a value \( x \) is denoted \( \text{READ}(x) \) or \( R(x) \), respectively.

▶ Observation 16. Let \( w(v) \) be any write operation by \( w \). Then there is a \( k \geq 1 \) such that \( v = v_k \).

▶ Observation 17. Let \( R \in \{ R_{wp}, R_{wQ} \} \). If \( w \) writes \( x \) in \( R \), then \( x = (\text{COMMIT}, v_k) \) for some \( k \geq 1 \) or \( x = (\text{PREPARE}, v_k, v_{k+1}) \) for some \( k \geq 0 \).

▶ Observation 18. Suppose \( p \) is not malicious. If \( p \) reads \( R_{wp} = x \), then \( x = (\text{COMMIT}, v_k) \) or \( x = (\text{PREPARE}, v_k, v_{k+1}) \), for some \( k \geq 0 \).

▶ Observation 19. Suppose \( q \in Q \) is not malicious. If \( q \) reads \( R_{wQ} = x \), then \( x = (\text{COMMIT}, v_k) \) or \( x = (\text{PREPARE}, v_k, v_{k+1}) \), for some \( k \geq 0 \).

▶ Lemma 20. Suppose \( p \) is not malicious. Let \( R_p(v) \) be any read operation by \( p \). Then there is a \( k \geq 0 \) such that \( v = v_k \), and

\[
\begin{align*}
&\text{p reads } R_{wp} = (\text{COMMIT}, v_k) \text{ in line 14 of } R_p(v), \text{ or} \\
&\text{p reads } R_{wp} = (\text{PREPARE}, v_k, v_{k+1}) \text{ in line 15 of } R_p(v).
\end{align*}
\]

Proof. Suppose \( p \) is not malicious. Let \( R_p(v) \) be any read operation by \( p \). Note that \( p \) reads \( R_{wp} \) in \( R_p(v) \). When it does so, by Observation 18, there are two possible cases:

1. \( p \) reads \( R_{wp} = (\text{COMMIT}, v_k) \) for some \( k \geq 0 \) in line 14 of \( R_p(v) \). Then \( R_p(v) \) returns \( v_k \) in line 17 i.e., \( v = v_k \).
2. \( p \) reads \( R_{wp} = (\text{PREPARE}, v_k, v_{k+1}) \) for some \( k \geq 0 \) in line 15 of \( R_p(v) \). Then \( R_p(v) \) returns \( v_k \) in line 19 i.e., \( v = v_k \).

▶ Lemma 21. Suppose \( q \in Q \) is not malicious. Let \( R_q(v) \) be any read operation by \( q \). Then there is a \( k \geq 0 \) such that \( v = v_k \), and

\[
\begin{align*}
&\text{q reads } R_{wQ} = (\text{COMMIT}, v_k) \text{ in line 21 of } R_q(v), \\
&\text{q reads } R_{wQ} = (\text{PREPARE}, v_{k-1}, v_k) \text{ in line 22 of } R_q(v), \text{ or} \\
&\text{q reads } R_{wQ} = (\text{PREPARE}, v_k, v_{k+1}) \text{ in line 23 of } R_q(v).
\end{align*}
\]

Proof. Suppose \( q \in Q \) is not malicious. Let \( R_q(v) \) be any read operation by \( q \). Note that \( q \) reads \( R_{wQ} \) in \( R_q(v) \). When it does so, by Observation 19, there are two possible cases:

1. \( q \) reads \( R_{wQ} = (\text{COMMIT}, v_k) \) for some \( k \geq 0 \) in line 21 of \( R_q(v) \). Then \( R_q(v) \) returns \( v_k \) in line 22 i.e., \( v = v_k \).
2. \( q \) reads \( R_{wQ} = (\text{PREPARE}, v_k, v_{k+1}) \) for some \( k \geq 0 \) in line 23 of \( R_q(v) \). Then there are two sub-cases:
   a. \( R_q(v) \) returns \( v_{k+1} \) in line 27 or 29 or 36 i.e., \( v = v_{k+1} \). Let \( k' = k + 1 \). Then in this case, \( q \) reads \( R_{wQ} = (\text{PREPARE}, v_{k-1}, v_{k'}) \) in line 23 of \( R_q(v) \) and \( v = v_{k'} \).
   b. \( R_q(v) \) returns \( v_k \) in line 38 i.e., \( v = v_k \).

▶ Observation 22. Let \( R \) be any register in \( \{ R_{wp}, R_{wQ} \} \).

(1) If \( w \) writes \( \text{PREPARE}, v_{k-1}, v_k \) in \( R \) before \( w \) writes \( \text{COMMIT}, v_{k'} \) in \( R \), then \( k \leq k' \).

---

Footnote 9: For brevity, we say that “a process \( r \) reads or writes a register in line \( x \) of a \( R_r(-) \) or a \( W(-) \) operation”, if it reads or writes this register in line \( x \) of the \( R_r() \) or \( W() \) procedure executed to do this \( R_r(-) \) or \( W(-) \) operation.
On implementing SWMR registers from SWSR registers in systems with Byzantine failures

(2) If \( w \) writes \((\text{PREPARE}, v_{k-1}, v_k)\) in \( R \) before \( w \) writes \((\text{PREPARE}, v_{k'-1}, v_{k'})\) in \( R \), then \( k < k' \).
(3) If \( w \) writes \((\text{COMMIT}, v_k)\) in \( R \) before \( w \) writes \((\text{COMMIT}, v_{k'})\) in \( R \), then \( k < k' \).
(4) If \( w \) writes \((\text{COMMIT}, v_k)\) in \( R \) before \( w \) writes \((\text{PREPARE}, v_{k-1}, v_{k'})\) in \( R \), then \( k < k' \).

- **Observation 23.** Let \( R \) be any register in \( \{R_{wp}, R_{wQ}\} \). Suppose \( r \in \{p\} \cup Q \) is not malicious.

  (1) If \( w \) writes \((\text{PREPARE}, v_{k-1}, v_k)\) in \( R \) before \( r \) reads \((\text{COMMIT}, v_k')\) in \( R \), then \( k \leq k' \).
  (2) If \( w \) writes \((\text{PREPARE}, v_{k-1}, v_k)\) in \( R \) before \( r \) reads \((\text{PREPARE}, v_{k'-1}, v_{k'})\) in \( R \), then \( k \leq k' \).
  (3) If \( w \) writes \((\text{COMMIT}, v_k)\) in \( R \) before \( r \) reads \((\text{COMMIT}, v_{k'})\) in \( R \), then \( k \leq k' \).
  (4) If \( w \) writes \((\text{COMMIT}, v_k)\) in \( R \) before \( r \) reads \((\text{PREPARE}, v_{k-1}, v_{k'})\) in \( R \), then \( k < k' \).

- **Observation 24.** Let \( R \) be any register in \( \{R_{wp}, R_{wQ}\} \). Suppose \( r \) and \( r' \) are non-malicious processes in \( \{p\} \cup Q \).

  (1) If \( r \) reads \( R = (\text{PREPARE}, v_{k-1}, v_k) \) before \( r' \) reads \( R = (\text{COMMIT}, v_k') \), then \( k \leq k' \).
  (2) If \( r \) reads \( R = (\text{PREPARE}, v_{k-1}, v_k) \) before \( r' \) reads \( R = (\text{PREPARE}, v_{k'-1}, v_{k'}) \), then \( k \leq k' \).
  (3) If \( r \) reads \( R = (\text{COMMIT}, v_k) \) before \( r' \) reads \( R = (\text{COMMIT}, v_{k'}) \), then \( k \leq k' \).
  (4) If \( r \) reads \( R = (\text{COMMIT}, v_k) \) before \( r' \) reads \( R = (\text{PREPARE}, v_{k-1}, v_{k'}) \), then \( k < k' \).

**Proof of Linearizability Property 1.** We now prove that the write and read operations of the register that \( I_n \) implements satisfy Property 1 of Definition 9, i.e., processes read the “current” value of the register. To do so, we first prove this for the writes and reads of the lower-level procedures \( W() \) and \( R_{w}() \) for all readers \( r \) (Lemma 24), and then prove it for the writes and reads of the high-level procedures \( \text{WRITE}() \) and \( \text{READ}() \) (Lemma 29).

- **Lemma 25.** If \( R_r(v) \) is a read operation by a non-malicious process \( r \in \{p\} \cup Q \) then:
  - there is a \( W(v) \) operation that immediately precedes \( R_r(v) \) or is concurrent with \( R_r(v) \), or
  - \( v = v_0 \) and no \( W(-) \) operation precedes \( R_r(v) \).

**Proof.** Suppose \( r \in \{p\} \cup Q \) is not malicious. Let \( R_r(v) \) be any read operation by \( r \).

By Lemmas 20 and 21, \( v = v_k \) for some \( k \geq 0 \). We now show that:
- if \( k = 0 \) then no \( W(-) \) operation precedes \( R_r(v_k) \), and
- if \( k > 0 \) then a \( W(v_k) \) operation immediately precedes \( R_r(v_k) \) or is concurrent with \( R_r(v_k) \).

There are two cases: \( r = p \) or \( r \in Q \).

**Case 1:** \( r = p \). By Lemma 20, there are two cases:

1) \( p \) reads \( R_{wp} = (\text{COMMIT}, v_k) \) in line 14 of \( R_p(v_k) \). There are two cases:
   
   i. \( k = 0 \). Suppose, for contradiction, that there is a \( W(v) \) operation that precedes \( R_p(v_0) \). By Observation 16, \( v = v_i \) for some \( i \geq 1 \). So \( w \) writes \((\text{COMMIT}, v_i)\) into \( R_{wp} \) in line 10 of \( W(v_i) \) before \( p \) reads \( R_{wp} = (\text{COMMIT}, v_0) \) in line 14 of \( R_p(v_0) \). By Observation 23, \( i \leq 0 \) — a contradiction. So no \( W(-) \) operation precedes \( R_p(v_0) \).
   
   ii. \( k > 0 \). Then \( w \) writes \((\text{COMMIT}, v_k)\) into \( R_{wp} \) in line 10 of \( W(v_k) \) before \( p \) reads \( R_{wp} = (\text{COMMIT}, v_k) \) in \( R_p(v_k) \). So the \( W(v_k) \) operation precedes \( R_p(v_k) \) or is concurrent with \( R_p(v_k) \). We now show that if \( W(v_k) \) precedes \( R_p(v_k) \), then \( W(v_k) \) immediately precedes \( R_p(v_k) \). Suppose, for contradiction, that \( W(v_k) \) precedes \( R_p(v_k) \) but does not immediately precede \( R_p(v_k) \). Then there is a \( W(v_i) \) operation that immediately precedes \( R_p(v_k) \). Clearly, the \( W(v_k) \) operation precedes the \( W(v_i) \) operation, and so \( i < k \). Furthermore, \( w \) writes \((\text{COMMIT}, v_k)\) into \( R_{wp} \) in line 10 of \( W(v_k) \) before \( p \) reads \( R_{wp} = (\text{COMMIT}, v_k) \) in line 14 of \( R_p(v_k) \). By Observation 23, \( i \leq k \) — a contradiction. Therefore the \( W(v_k) \) operation immediately precedes \( R_p(v_k) \) or is concurrent with \( R_p(v_k) \).
2) $p$ reads $R_{wp} = (\text{prepare}, v_k, v_{k+1})$ in line 18 of $R_p(v_k)$. Then this read occurs after $w$ writes $(\text{prepare}, v_k, v_{k+1})$ in $R_{wp}$, in line 8 of the $w(v_{k+1})$ operation. Furthermore, by Observation 23(4), this read occurs before $w$ writes $(\text{commit}, v_{k+1})$ in $R_{wp}$ in line 10 of the $w(v_{k+1})$ operation. Therefore the $w(v_{k+1})$ operation is concurrent with $R_p(v_k)$. There are two cases:

i. $k = 0$. Since $w(v_1)$ is concurrent with $R_p(v_0)$, no $w(−)$ operation precedes $R_p(v_0)$.

ii. $k > 0$. Since $w(v_{k+1})$ is concurrent with $R_p(v_k)$, $w(v_k)$ immediately precedes $R_p(v_k)$ or is concurrent with $R_p(v_k)$.

Case 2: $r = q \in Q$. By Lemma 21, there are three cases:

1) $q$ reads $R_{wQ} = (\text{commit}, v_k)$ in line 21 of $R_q(v_k)$. There are two cases:

i. $k = 0$. Suppose, for contradiction, that there is a $w(v)$ operation that precedes $R_p(v_k)$. By Observation 16, $v = v_i$ for some $i \geq 1$. So $w$ writes $(\text{commit}, v_i)$ into $R_{wQ}$ in line 11 of $w(v_i)$ before $q$ reads $R_{wQ} = (\text{commit}, v_k)$ in line 21 of $R_q(v_k)$. By Observation 23(3), $i \geq 0$ — a contradiction. So no $w(−)$ operation precedes $R_q(v_k)$.

ii. $k > 0$. Then $w$ writes $(\text{commit}, v_k)$ into $R_{wQ}$ in line 11 of $w(v_k)$ before $q$ reads $R_{wQ} = (\text{commit}, v_k)$ in line 21 of $R_q(v_k)$. So the $w(v_k)$ operation precedes $R_q(v_k)$ or is concurrent with $R_q(v_k)$. We now show that if $w(v_k)$ precedes $R_q(v_k)$, then $w(v_k)$ immediately precedes $R_q(v_k)$. Suppose, for contradiction, that $w(v_k)$ precedes $R_q(v_k)$ but does not immediately precede $R_q(v_k)$. Then there is a $w(v)$ operation that immediately precedes $R_p(v_k)$. Clearly, the $w(v)$ operation precedes the $w(v_k)$ operation, and so $i > k$. Furthermore, $w$ writes $(\text{commit}, v_k)$ into $R_{wQ}$ in line 11 of $w(v_k)$ before $q$ reads $R_{wQ} = (\text{commit}, v_k)$ in line 21 of $R_q(v_k)$. By Observation 23(3), $i \leq k$ — a contradiction. Therefore the $w(v_k)$ operation immediately precedes $R_q(v_k)$ or is concurrent with $R_q(v_k)$.

2) $q$ reads $R_{wQ} = (\text{prepare}, v_{k-1}, v_k)$ in line 23 of $R_q(v_k)$. Then this read occurs after $w$ writes $(\text{prepare}, v_{k-1}, v_k)$ in $R_{wQ}$ in line 9 of the $w(v_k)$ operation. Furthermore, by Observation 23(4), this read occurs before $w$ writes $(\text{commit}, v_k)$ in $R_{wQ}$ in line 11 of the $w(v_k)$ operation. Therefore the $w(v_k)$ operation is concurrent with $R_q(v_k)$.

3) $q$ reads $R_{wQ} = (\text{prepare}, v_k, v_{k+1})$ in line 23 of $R_q(v_k)$. Then this read occurs after $w$ writes $(\text{prepare}, v_k, v_{k+1})$ in $R_{wQ}$ in line 9 of the $w(v_{k+1})$ operation. Furthermore, by Observation 23(4), this read occurs before $w$ writes $(\text{commit}, v_{k+1})$ in $R_{wQ}$ in line 11 of the $w(v_{k+1})$ operation. Therefore the $w(v_{k+1})$ operation is concurrent with $R_q(v_k)$. There are two cases:

i. $k = 0$. Since $w(v_1)$ is concurrent with $R_q(v_0)$, no $w(−)$ operation precedes $R_q(v_0)$.

ii. $k > 0$. Since $w(v_{k+1})$ is concurrent with $R_q(v_k)$, $w(v_k)$ is concurrent with $R_q(v_k)$ or immediately precedes $R_q(v_k)$.

We now prove that the write and read operations of the high-level procedures $\text{Write}(\cdot)$ and $\text{Read}(\cdot)$ satisfy Property 1 of Definition 6. By Observation 15, Lemmas 20 and 21, and the code of the procedure $\text{Read}(\cdot)$, we have:

\begin{itemize}
  \item \textbf{Observation 26.} If $\text{Read}(u)$ is an operation by a non-malicious process $r \in \{p\} \cup Q$, then $u = u_k$ for some $k \geq 0$.
  \item \textbf{Observation 27.} If $\text{Read}(u_k)$ is an operation by a non-malicious process $r \in \{p\} \cup Q$, then $r$ invokes and completes a $R_r(v_k)$ operation in $\text{Read}(u_k)$.
  \item \textbf{Observation 28.} If $\text{Write}(u_k)$ is a completed operation by $w$, then $w$ invokes and completes a $w(v_k)$ operation in $\text{Write}(u_k)$.
\end{itemize}

We now prove that the $\text{Write}(−)$ and $\text{Read}(−)$ operations satisfy Property 1 of Definition 6.
Lemma 29. [Property 1: Reading a “current” value]

If \( \text{READ}(u) \) is a read operation by a non-malicious process \( r \in \{p\} \cup Q \) then:

- there is a \( \text{WRITE}(u) \) operation that immediately precedes \( \text{READ}(u) \) or is concurrent with \( \text{READ}(u) \), or
- \( u = u_0 \) and no \( \text{WRITE}(\cdot) \) operation precedes \( \text{READ}(u) \).

Proof. Let \( \text{READ}(u) \) be any read operation by a non-malicious process \( r \in \{p\} \cup Q \). By Observation 24, \( u = u_k \) for some \( k \geq 0 \). There are two cases:

1. \( k = 0 \). Suppose, for contradiction, that a \( \text{WRITE}(u_i) \) operation precedes \( \text{READ}_r(u_0) \). Note that \( i \geq 1 \). By Observations 28 and 27 a \( \text{W}(v_i) \) operation precedes a \( \text{R}_r(v_0) \) operation. Since process \( r \) is not malicious, by Lemma 25, there is no \( \text{W}(\cdot) \) operation that precedes \( \text{R}_r(v_0) \) — a contradiction.

2. \( k > 0 \). By Observation 27 \( r \) invokes and completes a \( \text{R}_r(v_k) \) operation in \( \text{READ}(u_k) \). Since \( k > 0 \), by Lemma 25, there is a \( \text{W}(v_k) \) operation that immediately precedes \( \text{R}_r(v_k) \) or is concurrent with \( \text{R}_r(v_k) \). Let \( \text{WRITE}(u_k) \) be the operation in which \( w \) invokes the \( \text{W}(v_k) \) operation. Since \( \text{W}(v_k) \) immediately precedes \( \text{R}_r(v_k) \) or is concurrent with \( \text{R}_r(v_k) \), the \( \text{WRITE}(u_k) \) operation immediately precedes \( \text{READ}(u_k) \) or is concurrent with \( \text{READ}(u_k) \).

Proof of linearizability Property 2. We now prove that the write and read operations of the register that \( I_p \) implements satisfy Property 2 of Definition \( [\text{I}] \). i.e., we prove that there are no “new-old” inversions in the values that processes read. To do so, we first prove this for the writes and reads of the lower-level procedures \( \text{W}(\cdot) \) and \( \text{R}_r(\cdot) \) for all readers \( r \) (Lemma 29), and then prove it for the writes and reads of the high-level procedures \( \text{WRITE}(\cdot) \) and \( \text{READ}(\cdot) \) (Lemma 30).

We first show that there are no “new-old” inversions in the consecutive reads of process \( p \).

Lemma 30. Suppose \( p \) is not malicious. If \( \text{R}_p(v_k) \) and \( \text{R}_p(v_{k'}) \) are read operations by \( p \), and \( \text{R}_p(v_k) \) precedes \( \text{R}_p(v_{k'}) \), then \( k \leq k' \).

Proof. Suppose \( p \) is not malicious. Let \( \text{R}_p(v_k) \) and \( \text{R}_p(v_{k'}) \) be read operations by \( p \) such that \( \text{R}_p(v_k) \) precedes \( \text{R}_p(v_{k'}) \). By Lemma 29, the following occurs:

1. \( p \) reads \( R_{wp} = (\text{COMMIT}, v_k) \) in line 11 of \( \text{R}_p(v_k) \), or
2. \( p \) reads \( R_{wp} = (\text{PREPARE}, v_k, v_{k+1}) \) in line 18 of \( \text{R}_p(v_k) \),

before the following occurs:

1. \( p \) reads \( R_{wp} = (\text{COMMIT}, v_{k'}) \) in line 14 of \( \text{R}_p(v_{k'}) \), or
2. \( p \) reads \( R_{wp} = (\text{PREPARE}, v_{k'}, v_{k'+1}) \) in line 18 of \( \text{R}_p(v_{k'}) \).

So there are four possible cases:

1. \( p \) reads \( R_{wp} = (\text{COMMIT}, v_k) \) in \( \text{R}_p(v_k) \) before \( p \) reads \( R_{wp} = (\text{COMMIT}, v_{k'}) \) in \( \text{R}_p(v_{k'}) \).
2. \( p \) reads \( R_{wp} = (\text{COMMIT}, v_k) \) in \( \text{R}_p(v_k) \) before \( p \) reads \( R_{wp} = (\text{PREPARE}, v_{k'}, v_{k'+1}) \) in \( \text{R}_p(v_{k'}) \).
3. \( p \) reads \( R_{wp} = (\text{PREPARE}, v_k, v_{k+1}) \) in \( \text{R}_p(v_k) \) before \( p \) reads \( R_{wp} = (\text{COMMIT}, v_{k'}) \) in \( \text{R}_p(v_{k'}) \).
4. \( p \) reads \( R_{wp} = (\text{PREPARE}, v_k, v_{k+1}) \) in \( \text{R}_p(v_k) \) before \( p \) reads \( R_{wp} = (\text{PREPARE}, v_{k'}, v_{k'+1}) \) in \( \text{R}_p(v_{k'}) \).

To prove that there are no “new-old” inversions between the reads of \( p \) and those of any reader \( q \in Q \), and also between the reads of any pair of readers \( q, q' \in Q \), we first make some straightforward observations that are clear from the code of \( I_p \). We first note that the counters of the tuples in registers \( R_{pQ} \) and \( R_{qq'} \) do not decrease.
Observation 31. Suppose $p$ is not malicious. If $p$ writes $v_k$ in $R_{pq}$ before $p$ writes $v_{k'}$ in $R_{pq}$, then $k \leq k'$.

Observation 32. Suppose $p$ and $q \in Q$ are not malicious. If $p$ writes $v_k$ in $R_{pq}$ before $q$ reads $R_{pq} = v_{k'}$, then $k \leq k'$.

Observation 33. Suppose $q \in Q$ is not malicious. For all processes $q' \in Q$, if $q$ writes $v_k$ in $R_{qq'}$ before $q$ reads $v_{k'}$ in $R_{qq'}$, then $k \leq k'$.

Observation 34. Suppose $q \in Q$ and $q' \in Q$ are not malicious. If $q$ writes $v_k$ in $R_{qq'}$ before $q'$ reads $R_{qq'} = v_{k'}$, then $k \leq k'$.

The following observations relate the counters of the tuples that $w$ successively writes in registers $R_{wp}$ and $R_{wQ}$.

Observation 35. (1) If $w$ writes $(\text{PREPARE}, v_{k-1}, v_k)$ in $R_{wp}$ before $w$ writes $(\text{PREPARE}, v_{k'-1}, v_{k'})$ in $R_{wQ}$, then $k \leq k'$.

(2) If $w$ writes $(\text{PREPARE}, v_{k-1}, v_k)$ in $R_{wp}$ before $w$ writes $(\text{COMMIT}, v_{k'})$ in $R_{wQ}$, then $k \leq k'$.

(3) If $w$ writes $(\text{COMMIT}, v_k)$ in $R_{wp}$ before $w$ writes $(\text{PREPARE}, v_{k'-1}, v_{k'})$ in $R_{wQ}$, then $k < k'$.

(4) If $w$ writes $(\text{COMMIT}, v_k)$ in $R_{wp}$ before $w$ writes $(\text{COMMIT}, v_{k'})$ in $R_{wQ}$, then $k \leq k'$.

(5) If $w$ writes $(\text{PREPARE}, v_{k-1}, v_k)$ in $R_{wQ}$ before $w$ writes $(\text{PREPARE}, v_{k'-1}, v_{k'})$ in $R_{wQ}$, then $k < k'$.

(6) If $w$ writes $(\text{PREPARE}, v_{k-1}, v_k)$ in $R_{wQ}$ before $w$ writes $(\text{COMMIT}, v_{k'})$ in $R_{wQ}$, then $k \leq k'$.

(7) If $w$ writes $(\text{COMMIT}, v_k)$ in $R_{wQ}$ before $w$ writes $(\text{PREPARE}, v_{k'-1}, v_{k'})$ in $R_{wQ}$, then $k < k'$.

(8) If $w$ writes $(\text{COMMIT}, v_k)$ in $R_{wQ}$ before $w$ writes $(\text{COMMIT}, v_{k'})$ in $R_{wQ}$, then $k < k'$.

The next observations relate the counters of the tuples that $p$ and processes $q \in Q$ read from $R_{wp}$ and $R_{wQ}$, respectively.

Observation 36. Suppose $p$ and $q \in Q$ are not malicious.

(1) If $p$ reads $R_{wp} = (\text{PREPARE}, v_{k-1}, v_k)$ before $q$ reads $R_{wQ} = (\text{PREPARE}, v_{k'-1}, v_{k'})$, then $k \leq k'$.

(2) If $p$ reads $R_{wp} = (\text{PREPARE}, v_{k-1}, v_k)$ before $q$ reads $R_{wQ} = (\text{COMMIT}, v_{k'})$, then $k - 1 \leq k'$.

(3) If $p$ reads $R_{wp} = (\text{COMMIT}, v_k)$ before $q$ reads $R_{wQ} = (\text{PREPARE}, v_{k'-1}, v_{k'})$, then $k \leq k'$.

(4) If $p$ reads $R_{wp} = (\text{COMMIT}, v_k)$ before $q$ reads $R_{wQ} = (\text{COMMIT}, v_{k'})$, then $k \leq k'$.

(5) If $q$ reads $R_{wQ} = (\text{PREPARE}, v_{k-1}, v_k)$ before $p$ reads $R_{wp} = (\text{PREPARE}, v_{k'-1}, v_{k'})$, then $k \leq k'$.

(6) If $q$ reads $R_{wQ} = (\text{PREPARE}, v_{k-1}, v_k)$ before $p$ reads $R_{wp} = (\text{COMMIT}, v_{k'})$, then $k \leq k'$.

(7) If $q$ reads $R_{wQ} = (\text{COMMIT}, v_k)$ before $p$ reads $R_{wp} = (\text{PREPARE}, v_{k'-1}, v_{k'})$, then $k + 1 \leq k'$.

(8) If $q$ reads $R_{wQ} = (\text{COMMIT}, v_k)$ before $p$ reads $R_{wp} = (\text{COMMIT}, v_{k'})$, then $k \leq k'$.

Now we prove that there is no “new-old” inversion for a read by $p$ that precedes a read by a process $q \in Q$.

Lemma 37. If $R_p(v_k)$ and $R_q(v_{k'})$ are read operations by non-malicious processes $p$ and $q \in Q$ respectively, and $R_p(v_k)$ precedes $R_q(v_{k'})$, then $k \leq k'$.

Proof. Suppose processes $p$ and $q \in Q$ are not malicious. Let $R_p(v_k)$ and $R_q(v_{k'})$ be read operations by $p$ and $q$ respectively, such that $R_p(v_k)$ precedes $R_q(v_{k'})$. By Lemmas 20 and 21, the following occurs:
1. \( p \) reads \( R_{wp} = (\text{commit}, v_k) \) in line 14 of \( R_p(v_k) \), or
2. \( p \) reads \( R_{wp} = (\text{prepare}, v_k, v_{k+1}) \) in line 18 of \( R_p(v_k) \)

before the following occurs:

1. \( q \) reads \( R_{wQ} = (\text{commit}, v_{k'}^q) \) in line 21 of \( R_q(v_{k'}) \), or
2. \( q \) reads \( R_{wQ} = (\text{prepare}, v_{k-1}, v_k) \) in line 23 of \( R_q(v_k) \), or
3. \( q \) reads \( R_{wQ} = (\text{prepare}, v_{k'}, v_{k'+1}) \) in line 25 of \( R_q(v_{k'}) \).

So there are six possible cases:

1. \( p \) reads \( R_{wp} = (\text{commit}, v_k) \) in \( R_p(v_k) \) before \( q \) reads \( R_{wQ} = (\text{commit}, v_{k'}) \) in \( R_q(v_k) \).
   
   By Observation 36(1), \( k \leq k' \).
2. \( p \) reads \( R_{wp} = (\text{commit}, v_k) \) in \( R_p(v_k) \) before \( q \) reads \( R_{wQ} = (\text{prepare}, v_{k-1}, v_k) \) in \( R_q(v_k) \).
   
   By Observation 36(3), \( k \leq k' \).
3. \( p \) reads \( R_{wp} = (\text{commit}, v_k) \) in \( R_p(v_k) \) before \( q \) reads \( R_{wQ} = (\text{prepare}, v_{k'}, v_{k'+1}) \) in \( R_q(v_{k'}) \).
   
   By Observation 36(4), \( k \leq k' \).
4. \( p \) reads \( R_{wp} = (\text{prepare}, v_k, v_{k+1}) \) in \( R_p(v_k) \) before \( q \) reads \( R_{wQ} = (\text{commit}, v_{k'}) \) in \( R_q(v_{k'}) \).
   
   By Observation 36(2), \( (k+1) - 1 \leq k' \). So \( k \leq k' \).
5. \( p \) reads \( R_{wp} = (\text{prepare}, v_k, v_{k+1}) \) in \( R_p(v_k) \) before \( q \) reads \( R_{wQ} = (\text{prepare}, v_{k'-1}, v_{k'}) \) in \( R_q(v_k) \).
   
   By Observation 36(4), \( k \leq k' \).
6. \( p \) reads \( R_{wp} = (\text{prepare}, v_k, v_{k+1}) \) in \( R_p(v_k) \) before \( q \) reads \( R_{wQ} = (\text{prepare}, v_{k'}, v_{k'+1}) \) in \( R_q(v_{k'}) \).
   
   By Observation 36(5), \( k+1 \leq k' \). So \( k \leq k' \).

Now we prove that there is no “new-old” inversion for a read by a process \( q \in Q \) that precedes a read by \( p \).

**Lemma 38.** If \( R_q(v_k) \) and \( R_p(v_{k'}) \) are read operations by non-malicious processes \( q \in Q \) and \( p \) respectively, and \( R_q(v_k) \) precedes \( R_p(v_{k'}) \), then \( k \leq k' \).

**Proof.** Suppose processes \( q \in Q \) and \( p \) are not malicious. Let \( R_q(v_k) \) and \( R_p(v_{k'}) \) be two read operations by \( q \) and \( p \) respectively, such that \( R_q(v_k) \) precedes \( R_p(v_{k'}) \). By Lemmas 20 and 21 the following occurs:

1. \( q \) reads \( R_{wQ} = (\text{commit}, v_k) \) in line 21 of \( R_q(v_k) \), or
2. \( q \) reads \( R_{wQ} = (\text{prepare}, v_{k-1}, v_k) \) in line 23 of \( R_q(v_k) \), or
3. \( q \) reads \( R_{wQ} = (\text{prepare}, v_{k'}, v_{k'+1}) \) in line 25 of \( R_q(v_{k'}) \)

before the following occurs:

1. \( p \) reads \( R_{wp} = (\text{commit}, v_k) \) in line 14 of \( R_p(v_k) \), or
2. \( p \) reads \( R_{wp} = (\text{prepare}, v_{k'}, v_{k'+1}) \) in line 18 of \( R_p(v_{k'}) \).

So there are six possible cases:

1. \( q \) reads \( R_{wQ} = (\text{commit}, v_k) \) in \( R_q(v_k) \) before \( p \) reads \( R_{wp} = (\text{commit}, v_{k'}) \) in \( R_p(v_{k'}) \).
   
   By Observation 38(4), \( k \leq k' \).
2. \( q \) reads \( R_{wQ} = (\text{commit}, v_k) \) in \( R_q(v_k) \) before \( p \) reads \( R_{wp} = (\text{prepare}, v_{k'}, v_{k'+1}) \) in \( R_p(v_{k'}) \).
   
   By Observation 36(7), \( k+1 \leq k' + 1 \). So \( k \leq k' \).
3. \( q \) reads \( R_{wQ} = (\text{prepare}, v_{k-1}, v_k) \) in \( R_q(v_k) \) before \( p \) reads \( R_{wp} = (\text{commit}, v_{k'}) \) in \( R_p(v_{k'}) \).
   
   By Observation 36(6), \( k \leq k' \).
Lemma 39. Finally, we prove that there are no “new-old” inversions between the reads of processes in $X$.

4. $q$ reads $R_{wQ} = \text{(prepare, } v_{k-1}, v_k) \text{ in } R_q(v_k)$ before $p$ reads $R_{wp} = \text{(prepare, } v_{k'}, v_{k'+1}) \text{ in } R_p(v_{k'}).$ By Observation 36(5), $k \leq k' + 1$.
   
i. $k < k' + 1$. Then $k \leq k'$.
   
ii. $k = k' + 1$. We now show that this case is impossible. Since $k = k' + 1$, $p$ reads $R_{wp} = \text{(prepare, } v_{k-1}, v_k) \text{ in } R_p(v_{k-1}),$ and $R_p(v_{k-1}) \text{ returns in line 19}$ since $q$ reads $R_{wQ} = \text{(prepare, } v_{k-1}, v_k) \text{ in line 23}$ of $R_q(v_k), R_q(v_k) \text{ returns in line 27 or 29}$. We now consider each one of these cases.
   
a. $R_q(v_k) \text{ returns in line 27}$. Then $q$ reads $R_{wQ} = \text{(commit, } v_r) \text{ for some } \ell \geq k \text{ in line 26}$ of $R_q(v_k).$ Since $R_q(v_k) \text{ precedes } R_p(v_{k-1}), q \text{ reads } R_{wQ} = \text{(commit, } v_r) \text{ before } p \text{ reads } R_{wp} = \text{(prepare, } v_{k-1}, v_k) \text{ in } R_p(v_{k-1}).$ By Observation 36(7), $\ell < k$ — a contradiction.
   
b. $R_q(v_k) \text{ returns in line 29}$. So $q$ read $R_{wQ} = \text{(prepare, } - , v_r) \text{ for some } \ell > k \text{ in line 28}$ of $R_q(v_k).$ Since $R_q(v_k) \text{ precedes } R_p(v_{k-1}), q \text{ reads } R_{wQ} = \text{(prepare, } - , v_r) \text{ in } R_q(v_k) \text{ before } p \text{ reads } R_{wp} = \text{(prepare, } v_{k-1}, v_k) \text{ in } R_p(v_{k-1}).$ By Observation 36(5), $\ell \leq k$ — a contradiction.
   
c. $R_q(v_k) \text{ returns in line 32 or 36}$. Then $q$ reads $R_{qQ} = v_r \text{ for some } \ell \geq k \text{ in line 30}$ or 34 of $R_q(v_k).$ So $p$ writes $v_r$ to $R_{QQ}$ in line 15 of some $R_p(-)$ operation before $q$ reads $R_{QQ}$ in $R_q(v_k).$ Thus, $p$ reads $R_{wp} = \text{(commit, } v_r) \text{ in line 14}$ before $q$ reads $R_{QQ}$ in $R_q(v_k).$ Since $R_q(v_k)$ precedes $R_p(v_{k-1}), q$ read $R_{QQ}$ in $R_q(v_k)$ before $p$ reads $R_{wp} = \text{(prepare, } v_{k-1}, v_k) \text{ in } R_p(v_{k-1}).$ So $p$ read $R_{wp} = \text{(commit, } v_r) \text{ before } p \text{ reads } R_{wp} = \text{(prepare, } v_{k-1}, v_k) \text{ in } R_p(v_{k-1}).$ By Observation 36(7), $\ell < k$ — a contradiction.

5. $q$ reads $R_{wQ} = \text{(prepare, } v_{k'}, v_{k'+1}) \text{ in } R_q(v_k)$ before $p$ reads $R_{wp} = \text{(commit, } v_{k'}) \text{ in } R_p(v_{k'})$. By Observation 36(5), $k + 1 \leq k'$. So $k \leq k'$.

6. $q$ reads $R_{wQ} = \text{(prepare, } v_{k'}, v_{k'+1}) \text{ in } R_q(v_k)$ before $p$ reads $R_{wp} = \text{(prepare, } v_{k'}, v_{k'+1}) \text{ in } R_p(v_{k'}).$ By Observation 36(5), $k + 1 \leq k' + 1$. So $k \leq k'$.

Finally, we prove that there are no “new-old” inversions between the reads of processes in $Q$.

\textbf{Lemma 39.} If $R_q(v_k)$ and $R_{q'}(v_{k'})$ are read operations by non-malicious processes $q \in Q$ and $q' \in Q$ respectively, and $R_q(v_k) \text{ precedes } R_{q'}(v_{k'}),$ then $k \leq k'$.

\textbf{Proof.} Suppose processes $q \in Q$ and $q' \in Q$ are not malicious. Let $R_q(v_k)$ and $R_{q'}(v_{k'})$ be read operations by $q$ and $q'$ respectively, such that $R_q(v_k) \text{ precedes } R_{q'}(v_{k'}).$ By Lemma 21 the following occurs:

1. $q$ reads $R_{wQ} = \text{(commit, } v_k) \text{ in line 21}$ of $R_q(v_k),$ or
2. $q$ reads $R_{wQ} = \text{(prepare, } v_{k-1}, v_k)$ in line 23 of $R_q(v_k),$ or
3. $q$ reads $R_{wQ} = \text{(prepare, } v_k, v_{k'+1}) \text{ in line 23}$ of $R_q(v_k)$

before the following occurs:

1. $q'$ reads $R_{wQ} = \text{(commit, } v_{k'}) \text{ in line 21}$ of $R_{q'}(v_{k'}),$ or
2. $q'$ reads $R_{wQ} = \text{(prepare, } v_{k'-1}, v_{k'}) \text{ in line 23}$ of $R_{q'}(v_{k'}),$ or
3. $q'$ reads $R_{wQ} = \text{(prepare, } v_{k'}, v_{k'+1}) \text{ in line 23}$ of $R_{q'}(v_{k'}).$

So there are nine possible cases:

1. $q$ reads $R_{wQ} = \text{(commit, } v_k) \text{ in } R_q(v_k)$ before $q' \text{ reads } R_{wQ} = \text{(commit, } v_{k'}) \text{ in } R_{q'}(v_{k'}).$
   
   By Observation 24(3), $k \leq k'$.

2. $q$ reads $R_{wQ} = \text{(commit, } v_k) \text{ in } R_q(v_k)$ before $q' \text{ reads } R_{wQ} = \text{(prepare, } v_{k'-1}, v_{k'}) \text{ in } R_{q'}(v_{k'}).$
   
   By Observation 24(4), $k < k'$. So $k \leq k'$.

3. $q$ reads $R_{wQ} = \text{(commit, } v_k) \text{ in } R_q(v_k)$ before $q' \text{ reads } R_{wQ} = \text{(prepare, } v_{k'}, v_{k'+1}) \text{ in } R_{q'}(v_{k'}).$
   
   By Observation 24(4), $k < k' + 1$. So $k \leq k'$.

4. $q$ reads $R_{wQ} = \text{(prepare, } v_{k-1}, v_k) \text{ in } R_q(v_k)$ before $q' \text{ reads } R_{wQ} = \text{(commit, } v_{k'}) \text{ in } R_{q'}(v_{k'}).$
   
   By Observation 24(1), $k \leq k'$. 

"
5. \( q \) reads \( R_{wQ} = (\text{PREPARE}, v_{k-1}, v_k) \) in \( R_q(v_k) \) before \( q' \) reads \( R_{wQ} = (\text{PREPARE}, v_{k'-1}, v_{k'}) \) in \( R_q'(v_{k'}). \) By Observation 24, \( k \leq k'. \)
6. \( q \) reads \( R_{wQ} = (\text{PREPARE}, v_{k-1}, v_k) \) in \( R_q(v_k) \) before \( q' \) reads \( R_{wQ} = (\text{PREPARE}, v_{k'}, v_{k'+1}) \) in \( R_q'(v_{k'}). \) By Observation 24, \( k \leq k' + 1. \)

i. \( k < k' + 1. \) Then \( k \leq k'. \)

ii. \( k = k' + 1. \) We now show that this case is impossible. Since \( k = k' + 1, q' \) reads \( R_{wQ} = (\text{PREPARE}, v_{k-1}, v_k) \) in \( R_q'(v_{k-1}) \), and \( R_q'(v_{k-1}) \) returns in line 33 So \( q' \) reads \( R_{q'} \) in line 33 of \( R_q(v_{k-1}) \) before \( R_q'(v_{k-1}) \) returns in line 33. Since \( q \) reads \( R_{wQ} = (\text{PREPARE}, v_{k-1}, v_k) \) in \( R_q(v_k) \), \( R_q(v_k) \) returns in line 27, 29, 32 or 36. We now consider each one of these cases.

a. \( R_q(v_k) \) returns in line 27. Then \( q \) reads \( R_{wQ} = (\text{COMMIT}, v_{\ell}) \) for some \( \ell \geq k \) in line 26 of \( R_q(v_k) \). Since \( R_q(v_k) \) precedes \( R_q'(v_{k-1}) \), \( q \) reads \( R_{wQ} = (\text{COMMIT}, v_{\ell}) \) in \( R_q'(v_{k-1}) \) before \( q' \) reads \( R_{wQ} = (\text{PREPARE}, v_{k-1}, v_k) \) in \( R_q'(v_{k-1}) \). By Observation 24, \( \ell < k \) — a contradiction.

b. \( R_q(v_k) \) returns in line 29. Then \( q \) reads \( R_{wQ} = (\text{PREPARE}, \ldots, v_{\ell}) \) for some \( \ell > k \) in line 28 of \( R_q(v_k) \). Since \( R_q(v_k) \) precedes \( R_q'(v_{k-1}) \), \( q \) reads \( R_{wQ} = (\text{PREPARE}, \ldots, v_{\ell}) \) in \( R_q(v_{k-1}) \) before \( q' \) reads \( R_{wQ} = (\text{PREPARE}, v_{k-1}, v_k) \) in \( R_q'(v_{k-1}) \). By Observation 24, \( \ell < k \) — a contradiction.

c. \( R_q(v_k) \) returns in line 32 or 36. Then \( q \) writes \( v_k \) in \( R_{wQ} \) in line 31 or 33 of \( R_q(v_k) \). Since \( R_q(v_k) \) precedes \( R_q'(v_{k-1}) \), \( q \) writes \( v_k \) in \( R_q'(v_{k-1}) \) before \( q' \) reads \( R_{wQ} \) in line 33 of \( R_q'(v_{k-1}) \). Thus, by Observation 34, \( q' \) reads \( R_{wQ} = v_k \) for some \( \ell \geq k \) in line 33 of \( R_q'(v_{k-1}) \). So \( R_q'(v_{k-1}) \) returns in line 36 rather than in line 38 — a contradiction.

7. \( q \) reads \( R_{wQ} = (\text{PREPARE}, v_{k}, v_{k+1}) \) in \( R_q(v_k) \) before \( q' \) reads \( R_{wQ} = (\text{COMMIT}, v_{\ell'}) \) in \( R_q'(v_{\ell'}) \). By Observation 24, \( k + 1 \leq k'. \) So \( k \leq k'. \)
8. \( q \) reads \( R_{wQ} = (\text{PREPARE}, v_{k}, v_{k+1}) \) in \( R_q(v_k) \) before \( q' \) reads \( R_{wQ} = (\text{PREPARE}, v_{k'-1}, v_{k'}) \) in \( R_q'(v_{k'}). \) By Observation 24, \( k + 1 \leq k'. \) So \( k \leq k'. \)
9. \( q \) reads \( R_{wQ} = (\text{PREPARE}, v_{k}, v_{k+1}) \) in \( R_q(v_k) \) before \( q' \) reads \( R_{wQ} = (\text{PREPARE}, v_{k'}, v_{k'+1}) \) in \( R_q'(v_{k'}). \) By Observation 24, \( k + 1 \leq k' + 1. \) So \( k \leq k'. \)

We now prove that the writes and reads of the lower-level procedures \( W() \) and \( R_r() \) for all readers \( r \) satisfy Property 2 of Definition 6.

**Lemma 40.** Let \( R_q(v_k) \) and \( R_{r'}(v_{k'}) \) be any read operations by some non-malicious processes \( r \) and \( r' \) in \( \{ p \} \cup Q. \) If \( R_q(v_k) \) precedes \( R_{r'}(v_{k'}) \) then \( k \leq k'. \)

**Proof.** Immediate from Lemmas 30, 37, 38, 39.

Finally, we prove that the write and read operations of the high-level procedures \( \text{WRITE}() \) and \( \text{READ}() \) satisfy Property 2 of Definition 6.

**Lemma 41.** **[Property 2: No “new-old” inversion]**
Let \( \text{READ}(u_k) \) and \( \text{READ}(u_{k'}) \) be any read operations by some non-malicious processes \( r \) and \( r' \) in \( \{ p \} \cup Q. \) If \( \text{READ}(u_k) \) precedes \( \text{READ}(u_{k'}) \) then \( k \leq k'. \)

**Proof.** Let \( \text{READ}(u_k) \) and \( \text{READ}(u_{k'}) \) be any read operations by some non-malicious processes \( r \) and \( r' \) in \( \{ p \} \cup Q. \) Suppose \( \text{READ}(u_k) \) precedes \( \text{READ}(u_{k'}) \). By Observation 27 in the \( \text{READ}(u_k) \) and \( \text{READ}(u_{k'}) \) operations, processes \( r \) and \( r' \) invoke and complete a \( R_r(v_k) \) and \( R_{r'}(v_{k'}) \) operation, respectively. Since \( \text{READ}(u_k) \) precedes \( \text{READ}(u_{k'}) \), \( R_r(v_k) \) precedes \( R_{r'}(v_{k'}). \) By Lemma 10, \( k \leq k'. \)

By Lemmas 29 and 41, the \( \text{WRITE}(-) \) and \( \text{READ}(-) \) operations of the register implementation \( I_n \) satisfy the linearizability Properties 1 and 2 of Definition 6. Therefore:

**Theorem 42.** For all \( n \geq 2, \) the implementation \( I_n \) is linearizable.
Termination of $I_n$. We now prove the Termination property of the implementation $I_n$. As in the previous section, we assume that the implementations of the registers $R_{wQ}$ and $R_{pQ}$ that $I_n$ uses are valid (so they are linearizable).

Note that if $w$ is malicious, it could write (commit, $\langle k, - \rangle$) followed by (commit, $\langle k', - \rangle$) in $R_{wp}$ such that $k' < k$. To prevent $p$ from “acting on” a commit tuple that is out of order, $p$ remembers in the variable previous $\_ k$ the value $k$ of the last (commit, $\langle k, - \rangle$) tuple that it accepted. Using this variable in the guard of line 14 ensures that the tuples that $p$ writes in $R_{pQ}$ in line 15 have non-decreasing values of $k$ even if the writer $w$ is malicious. So if $p$ is not malicious, correct processes that read $R_{pQ}$, read tuples $\langle k, - \rangle$ with non-decreasing values of $k$. More precisely:

- **Observation 43.** Suppose $p, q \in Q$, and $q' \in Q$ are not malicious. If $q$ reads $R_{pQ} = \langle k, - \rangle$ before $q'$ reads $R_{pQ} = \langle k', - \rangle$, then $k \leq k'$.

- **Theorem 44.** For all $n \geq 2$, the implementation $I_n$ satisfies the Termination property if the writer is correct or no reader is malicious.

**Proof.** We must show that if the writer is correct or no reader is malicious, then every correct process completes each operation that it invokes in a finite number of steps. If the writer $w$ is correct, it is clear from the code of the write procedures WRITE() and W() that $w$ completes every WRITE() invocation with a response. If the reader $p$ is correct, it is also clear from the code of the procedures READ() and $R_p()$ that $p$ completes every READ() invocation with a response. Let $q$ be a correct process in $Q$. It remains to show that if the writer is correct or no reader is malicious then $q$ completes every READ() invocation with a response. Consider any execution of READ() by $q$. Note that in line 4 of READ() process $q$ calls $R_q()$, and if $R_q()$ returns a response then READ() also returns a response. We now show that $R_q()$ returns a response. To do so we first show the following.

- **Claim 45.** If the writer $w$ is correct, then $q$ does not loop forever in Thread 1 (lines 25-29).

**Proof.** Suppose, for contradiction, that the writer $w$ is correct but $q$ loops forever in lines 25-29. Thus, $q$ reads $R_{wQ} = \langle \text{prepare, last}_\text{written}, (k, u) \rangle$ for some last $\_ \text{written}$ and some $(k, u)$ in line 23. Since $w$ is correct: (1) $w$ previously wrote $\langle \text{prepare, last}_\text{written}, (k, u) \rangle$ into $R_{wQ}$, and (2) $w$ eventually writes (commit, $\langle k, u \rangle$) into $R_{wQ}$. Furthermore, by Observation 22 if $w$ writes into $R_{wQ}$ after writing (commit, $\langle k, u \rangle$) into $R_{wQ}$, then it writes (commit, $\langle k', - \rangle$) or (commit, $\langle k', - \rangle$) into $R_{wQ}$ with $k' > k$. Since $q$ spins forever in the loop of lines 25-29, $q$ reads $R_{wQ}$ infinitely many times. From the above, it is clear that eventually $q$ reads $R_{wQ} = \langle \text{commit, } k', - \rangle$ for some $k' \geq k$ in lines 26 or $R_{wQ} = \langle \text{prepare, } -, (k', -) \rangle$ for some $k' > k$ in lines 28 and then $q$ exits the loop by returning a tuple in lines 27 or 29 — a contradiction.

- **Claim 46.** Suppose no reader is malicious. Then (1) $q$ does not block in Thread 2, and (2) if $q$ evaluates the condition of line 34 in Thread 2, then $q$ finds that this condition holds.

**Proof.** Suppose no reader is malicious. So in particular no reader in $Q$ is malicious. Thus, since the implementation of $R_{pQ}$ is valid, $q$’s read operations of $R_{pQ}$ in lines 30 and 31 do not block. So $q$ cannot block in Thread 2.

Suppose $q$ evaluates the condition of line 34 in Thread 2. Since $q$ reaches line 34 in Thread 2, process $q$ previously read: (1) $R_{wQ} = \langle \text{prepare, last}_\text{written}, (k, u) \rangle$ for some last $\_ \text{written}$ and some $(k, u)$ in line 23 and (2) $R_{q'} = \langle k', - \rangle$ for some $q' \in Q$ and some $k' \geq k$ in line 33. Thus, since reader $q'$ is not malicious, $q'$ writes $\langle k', - \rangle$ into $R_{q'}$ before $q$ reached line 34. Note that $q'$ can write $\langle k', - \rangle$ into $R_{q'}$ only in line 31 or 33 in some execution of $R_{q'}()$ by $q'$. Before doing so, $q'$ must have read $R_{pQ} = \langle k', - \rangle$ in line 30 or 34 for some $k'' \geq k'$ (in that execution of $R_{q'}()$). So this reading of $\langle k', - \rangle$ from register $R_{pQ}$ occurred before $q$ reached line 34. Thus, by Observation 43 when $q$ reads $R_{pQ}$ in line 34...
On implementing SWMR registers from SWSR registers in systems with Byzantine failures

We now prove that if the writer is correct or no reader is malicious, then the execution of \( R_q() \) by the correct process \( q \) returns. Suppose, for contradiction, that: (1) writer is correct or no reader is malicious, but (2) the execution of \( R_q() \) by \( q \) does not return. So \( q \) does not return in line 22 or line 10 of \( R_q() \). Thus \( q \) enters the cobegin-coend section of the code of \( R_q() \), and it executes THREAD 1 and THREAD 2 in parallel.

Consider the forever loop in THREAD 1. From the code of \( R_q() \), it is clear that \( q \) stops executing this loop if and only if either \( q \) returns a value in lines 27 or 29 of this loop, or \( q \) exits \( R_q() \) altogether by returning some value in THREAD 2. Thus, since the execution of \( R_q() \) by \( q \) does not return, \( q \) loops forever in THREAD 1.

By Claim 45, this implies that the writer \( w \) is not correct. So, by the assumption (1) on process failures, no reader is malicious. By Claim 46, \( q \) does not block inside THREAD 2. By the code of THREAD 2, either \( q \) returns a value in line 32 or 38 of THREAD 2, or \( q \) reaches line 34 and evaluates the condition in this line. In the latter case, by Claim 46, the condition in line 34 evaluates to true, and so \( q \) returns a value in line 36 of THREAD 2. So in all cases, \( q \) exits \( R_q() \) by returning a value in THREAD 2 — a contradiction to assumption (2).

By Theorems 42 and 44, we have that if the implementations of \( R_{wQ} \) and \( R_{pQ} \) are valid (Assumption 1), then the implementation \( I_n \) is also valid. So we have:

\[ \text{For all } n \geq 2, I_n \text{ is a valid implementation of a } [1,n]-\text{register from implemented } [1,n-1]-\text{registers and atomic } [1,1]-\text{registers, provided that the implementations of the } [1,n-1]-\text{registers that it uses (namely, } R_{wQ} \text{ and } R_{pQ} \text{) are also valid.} \]

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

We now prove that in a system with Byzantine process failures, there is an implementation of a [1, n]-register from atomic [1, 1]-registers that is linearizable (always) and satisfies the Termination property if the writer or any number of readers, but not both, can fail. This matches the impossibility result given by Theorem 7 in Section 4. More precisely:

\[ \text{For all } n \geq 2, \text{ in a system of } n+1 \text{ processes that are subject to Byzantine failures, there is an implementation } I_n \text{ of a } [1,n]-\text{register from atomic } [1,1]-\text{registers such that:} \]

\[ \text{\begin{itemize} \item } I_n \text{ is linearizable.} \item \text{if the writer is correct or no reader is malicious, } I_n \text{ satisfies the Termination property.} \end{itemize}} \]

**Proof.** We must show that for all \( n \geq 2 \), there is a valid implementation \( I_n \) of a [1, n]-register from atomic [1, 1]-registers. We prove this by induction on \( n \).

**Base Case.** Let \( n = 2 \). Consider the implementation \( I_2 \) of Theorem 47. 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 \( I_2 \) of a [1, 2]-register from atomic [1, 1]-registers.

**Induction Step.** Let \( n > 2 \). Suppose there is a valid implementation \( I_{n-1} \) of a [1, n-1]-register that uses only atomic [1,1]-registers. We must show there is a valid implementation \( I_n \) of a [1, n]-register that uses only atomic [1,1]-registers.

By Theorem 47, there is an implementation \( I_n \) of a [1, n]-register that uses:

1. two implemented [1, n-1]-registers (namely, 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 \( I_{n-1} \) (\( I_{n-1} \) exists by our induction hypothesis). This gives an implementation \( I_n \) of a [1, n]-register that uses only atomic
Implementation $I_n$ of a $[1,n]$-register writable by process $w$ and readable by a set $P$ of $n$ processes in a system with unforgeable signatures. $I_n$ uses atomic $[1,1]$-registers.

### Local variables
- $c$: variable of $w$; initially $0$.
- $tuples$: variable of each $p$ in $P$; initially $\emptyset$.

### Atomic Registers

For all processes $i$ and $j$ in $\{w\} \cup P$:
- $R_{ij}$: atomic $[1,1]$-register; initially $\langle 0, u_0 \rangle_w$.

#### WRITE($u$):
- $c \leftarrow c + 1$
- call $\mathcal{W}(\langle c, u \rangle_w)$
- return done

#### READ($p$):
- $R_{pi} \leftarrow \langle k, u \rangle_w$
- return $u$
- else return $\bot$

#### $\mathcal{W}(\langle k, u \rangle_w)$:
- for every process $i \in P$ do
  - $R_{wi} \leftarrow \langle k, u \rangle_w$
- return done

#### $\mathcal{R}(\cdot)$:
- for every process $i \in \{w\} \cup P$ do
  - if $R_{ip} = \langle \ell, \text{val} \rangle_w$ for some $\langle \ell, \text{val} \rangle$ validly signed by $w$ then
    - $tuples \leftarrow \langle 0 \rangle \cup \{ \langle \ell, \text{val} \rangle_w \}$
    - $\langle k, u \rangle_w \leftarrow \text{tuple} \langle \ell, \text{val} \rangle_w$ with maximum sequence number $\ell$ in $tuples$
  - for every process $i \in P$ do
    - $R_{pi} \leftarrow \langle k, u \rangle_w$
- return $\langle k, u \rangle_w$

[1,1]-registers (because $I_{n-1}$ uses only atomic [1,1]-registers). Since the implementations of $R_{wQ}$ and $R_{pQ}$ are valid, $I_n$ is valid.

For the special case that $n = 2$ (i.e., there are only two readers), there is a simple implementation $I'_2$ that is stronger than the implementation $I_2$ given by Theorem 48, in contrast to $I_2$, which satisfies Termination if the writer is correct or no reader is malicious. $I'_2$ satisfies Termination unconditionally; in other words $I'_2$ is wait-free, and in fact it is bounded wait-free (Definition 5).

> **Theorem 49.** The implementation $I'_2$ (given by Algorithm 3 in Appendix A) is a bounded wait-free linearizable implementation of a $[1,2]$-register from atomic $[1,1]$-registers.

### 6 Register implementation for systems with digital signatures

We now consider systems where processes are subject to Byzantine failures, but they can use unforgeable signatures. Algorithm 2 gives a wait-free linearizable implementation $I''_n$ of a $[1,n]$-register that is writable by process $w$ and readable by a set $P$ of $n$ processes. This implementation tolerates any combination and number of faulty processes, and it works as follows.

To write $u$, the writer $w$ calls WRITE($u$). In this procedure, $w$ adds a sequence number $k$ to form a tuple $\langle k, u \rangle$, then it signs $\langle k, u \rangle$ with $w$ (the signed tuple is denoted $\langle k, u \rangle_w$), and finally it executes the lower-level write procedure $w(\langle k, u \rangle_w)$. It is worth noting that in this algorithm, the writer $w$ is the only process that signs values.

To read a value, a reader $p \in P$ calls READ(). This procedure calls a lower-level read procedure $r()$ that reads signed tuples written by the $w()$ procedure. If $r()$ returns a tuple of the form $\langle k, u \rangle_w$ for some $k$ and $u$, READ() strips the signature $w$ and sequence number $k$ from the tuple, and then it returns the value $u$ as the value read (otherwise READ() returns $\bot$ to indicate a read failure).
On implementing SWMR registers from SWSR registers in systems with Byzantine failures

The lower-level procedures \( w() \) and \( r() \) work as follows (in these procedures, \( R_{ij} \) denotes an atomic \([1,1]\)-register that is writable by process \( i \) and readable by process \( j \)):

- To execute \( w((k,u)_w) \), the writer \( w \) simply writes \((k,u)_w\) in \( R_{wi} \) for every process \( i \in P \).
- To execute \( r() \), a reader \( p \in P \) first reads the \([1,1]\)-register \( R_{ip} \) for every process \( i \in \{w\} \cup P \) to form the set \textit{tuples} of all the tuples validly signed by \( w \) that it reads. Then \( p \) selects the tuple \((k,u)_w\) with the maximum sequence number \( k \) in \textit{tuples}, and returns this tuple; but before doing so \( p \) writes \((k,u)_w\) into the \([1,1]\)-register \( R_{pi} \) for every reader \( i \in P \) to notify them that it read \((k,u)_w\).

We now prove that the implementation \( T^*_n \) given by Algorithm 2 is wait-free and linearizable.

**Wait-freedom.** This is trivial: the code of Algorithm 2 does not contain any loop or wait statement, so every call to the \texttt{Write()} and \texttt{Read()} procedures by any correct process terminates with a return value in a bounded number of its own steps. Thus:

- **Observation 50.** For all \( n \geq 2 \), the implementation \( T^*_n \) is bounded wait-free.

**Linearizability.** To prove that the implementation \( T^*_n \) given by Algorithm 2 is linearizable, we must show that if the writer \( w \) of the register implemented by \( T^*_n \) is not malicious then Properties 1 and 2 of Definition 6 hold. So for the rest of this section we assume that the writer \( w \) is not malicious, and henceforth we omit to repeat this assumption in our observations, lemmas, and theorem.

As in the previous section:

- \( u_0 \) is the initial value of the register that Algorithm 2 implements.
- For \( k \geq 1 \), \( u_k \) denotes the \( k \)-th value written by \( w \) using the procedure \texttt{Write()}.
  
  More precisely, if \( w \) calls \texttt{Write()} with a value \( u \) and this is its \( k \)-th call of \texttt{Write()}, \( u_k \) is \( u \).
- \( u_0 \) is \((0,u_0)_w\).
- For \( k \geq 1 \), \( u_k \) denotes the \( k \)-th value written by \( w \) using the procedure \texttt{W}().

We first prove the linearizability Properties 1 and 2 of Definition 6 are satisfied by the writes and reads of the lower-level procedures \texttt{W}() and \texttt{R}() (Lemmas 60 and 64), and then prove they are also satisfied by the writes and reads of the high-level procedures \texttt{Write()} and \texttt{Read()} (Lemmas 65 and 66).

- **Observation 51.** For all \( k \geq 0 \), \( v_k = (k,u_k)_w \).

- **Observation 52.** Let \( w(v) \) be any write operation by \( w \). Then there is a \( k \geq 1 \) such that \( v = v_k \).

Note that a correct reader enters a value \( v \) into its set \textit{tuples} only if \( v \) is a tuple \((\ell,\text{val})\) validly signed by \( w \), i.e., \((\ell,\text{val})_w\). Since \( w \) is not malicious, and signatures are unforgeable, it must be that \((\ell,\text{val})_w\) is \( v_\ell \). So we have:

- **Observation 53.** For every non-malicious reader \( p \in P \), if \( v \in \text{tuples} \) then \( v = v_k \) for some \( k \geq 0 \).

- **Observation 54.** For every non-malicious reader \( p \in P \), if \( r(v) \) is an operation by \( p \), then \( v = v_k \) for some \( k \geq 0 \). Furthermore, if \( k > 0 \) then \( w \) invokes \( w(v_k) \) before \( r(v_k) \) returns. So the operation \( w(v_k) \) precedes \( r(v_k) \) or is concurrent with \( r(v_k) \).

From Observation 53 and lines 15-17 of the procedure \texttt{R}(), if a non-malicious reader \( p \in P \) writes a value \( v \) in \( R_{pi} \), then \( v = v_k \) for some \( k \geq 0 \). Furthermore, By Observation 52 if the non-malicious writer \( w \) writes a value \( v \) in \( R_{wi} \), then \( v = v_k \) for some \( k \geq 1 \). So:

- **Observation 55.** Suppose a process \( i \in \{w\} \cup P \) is not malicious. For every process \( j \in \{w\} \cup P \), if \( i \) writes \( v \) in \( R_{ij} \), then \( v = v_k \) for some \( k \geq 0 \).

- **Observation 56.** Suppose processes \( i \in \{w\} \cup P \) and \( j \in \{w\} \cup P \) are not malicious. If \( j \) reads \( R_{ij} = v \), then \( v = v_k \) for some \( k \geq 0 \).
Observation 57. Suppose a process \( i \in \{w\} \cup P \) is not malicious. For every process \( j \in \{w\} \cup P \), if \( i \) writes \( v_k \) in \( R_{ij} \) before \( i \) writes \( v_{k'} \) in \( R_{ij} \), then \( k \leq k' \).

Observation 58. Suppose processes \( i \in \{w\} \cup P \) and \( j \in \{w\} \cup P \) are not malicious. If \( i \) writes \( v_k \) in \( R_{ij} \) before \( j \) reads \( R_{ij} = v_{k'} \), then \( k \leq k' \).

Lemma 59. Suppose a reader \( p \in P \) is not malicious. If a \( w(v_k) \) operation precedes a \( R(v_{k'}) \) operation by \( p \), then \( k' \geq k \).

Proof. Suppose a \( w(v_k) \) operation precedes a \( R(v_{k'}) \) operation by some non-malicious reader \( p \). So \( w \) writes \( v_k \) into \( R_{wp} \) in line 9 of \( w(v_k) \) before \( p \) reads \( R_{wp} \) in line 13 of \( R(v_{k'}) \). Note that \( k \geq 1 \). By Observations 56 and 58, \( p \) reads \( R_{wp} = v_{k''} \) for some \( k'' \geq k \) in line 13 of \( R(v_{k'}) \). Then \( p \) adds \( v_{k''} \) to tuples in line 14 of \( R(v_{k'}) \). By line 15 of \( R(v_{k'}) \), \( v_{k'} \) is the tuple with the maximum sequence number in tuples and so \( k' \geq k'' \). Since \( k'' \geq k \), \( k' \geq k \).

Lemma 60. If \( R(v) \) is an operation by a non-malicious reader \( p \in P \) then
- there is a \( w(v) \) operation that immediately precedes \( R(v) \) or is concurrent with \( R(v) \), or
- \( v = v_0 \) and no \( W(-) \) operation precedes \( R(v) \).

Proof. Suppose \( p \in P \) is not malicious. Let \( R(v) \) be any read operation by \( p \). By Observation 54, \( v = v_k \) for some \( k \geq 0 \). There are two cases:
Case \( k = 0 \). Suppose, for contradiction, that there is a \( w(v) \) operation that precedes \( R(v_0) \). By Observation 52, \( v = v_i \) for some \( i \geq 1 \). Since \( w(v_i) \) precedes \( R(v_0) \), by Lemma 59, \( i \leq 0 \) — a contradiction. So no \( W(-) \) operation precedes \( R(v_0) \).
Case \( k > 0 \). By Observation 54, the operation \( w(v_k) \) precedes \( R(v_k) \) or is concurrent with \( R(v_k) \). We now show that if \( w(v_k) \) precedes \( R(v_k) \), then \( w(v_k) \) immediately precedes \( R(v_k) \). Suppose, for contradiction, that \( w(v_k) \) precedes \( R(v_k) \) but does not immediately precede \( R(v_k) \). Then there is a \( w(v_i) \) operation that immediately precedes \( R(v_k) \). Clearly, the \( w(v_k) \) operation operation precedes the \( w(v_i) \) operation, and so \( i > k \). Since \( w(v_i) \) precedes \( R(v_k) \), by Lemma 59, \( i \leq k \) — a contradiction. Therefore the \( w(v_k) \) operation immediately precedes \( R(v_k) \) or is concurrent with \( R(v_k) \).

By Observations 51 and 54 and the code of procedure READ():

Observation 61. If READ(\( u \)) is an operation by a non-malicious process \( p \in P \), then \( u = u_k \) for some \( k \geq 0 \).

Observation 62. If READ(\( u_k \)) is an operation by a non-malicious process \( p \in P \), then \( p \) invokes and completes a \( R(v_k) \) operation in READ(\( u_k \)).

Observation 63. If WRITE(\( u_k \)) is a completed operation by \( w \), then \( w \) invokes and completes a \( w(v_k) \) operation in WRITE(\( u_k \)).

Lemma 64. [Property 1: Reading a “current” value]
If READ(\( u \)) is an operation by a non-malicious process \( p \in P \) then:
- there is a \( \text{WRITE}(u) \) operation that immediately precedes \( \text{READ}(u) \) or is concurrent with \( \text{READ}(u) \), or
- \( u = u_0 \) and no \( \text{WRITE}(-) \) operation precedes \( \text{READ}(u) \).

Proof. Let READ(\( u \)) be any read operation by a non-malicious process \( p \in P \). By Observation 51, \( u = u_k \) for some \( k \geq 0 \). There are two cases:
Case \( k = 0 \). Suppose, for contradiction, that a \( \text{WRITE}(u_i) \) operation precedes \( \text{READ}(u_0) \). Note that \( i \geq 1 \). By Observations 62 and 63, a \( w(v_i) \) operation precedes a \( R(v_0) \) operation. Since process \( p \) is not malicious, by Lemma 60, there is no \( W(-) \) operation that precedes \( R(v_0) \) — a contradiction.
Case \( k > 0 \). By Observation 62, \( p \) invokes and completes a \( R(v_k) \) operation in READ(\( u_k \)). Since \( k > 0 \), \( v_k \neq v_0 \). So, by Lemma 60, there is a \( w(v_k) \) operation that immediately precedes
On implementing SWMR registers from SWSR registers in systems with Byzantine failures

Consider a system where processes are subject to Byzantine failures and for all \( n \geq 2 \), \( I_n^s \) is a bounded wait-free linearizable implementation of a \([1,n]\)-register from atomic \([1,1]\)-registers.

### 7 Register implementations with bounded termination

The linearizable register implementations given in Section 8 (Algorithm 2, Theorem 68) and in Appendix A (Algorithm 3, Theorem 49) guarantee that every correct process completes every operation in a bounded number of steps (regardless of which processes fail or how they fail). In contrast, the linearizable register implementation given in Section 5.4 (Algorithm 1, Theorem 48) satisfies the Termination property, namely every correct process completes every operation in a finite number of steps, and it does so under the assumption that the writer of the register is correct or no reader is malicious. This raises the question of whether, under the same failure assumption, there is a register implementation that satisfies the following stronger termination property:

**Definition 69 (Bounded Termination).** Every correct process completes every operation in a bounded number of its steps.

It turns out that the answer is “No”, even if we assume that the writer is not faulty and at most one of the readers can fail. More precisely:

**Theorem 70.** For all \( n \geq 3 \), in a system with \( n + 1 \) processes that are subject to Byzantine failures, there is no linearizable implementation of a \([1,n]\)-register from atomic \([1,n-1]\)-registers that satisfies the Bounded Termination property, even if we assume that the writer of the implemented \([1,n]\)-register is correct and at most one reader can be malicious.
This is in sharp contrast to Theorem 48 which implies that, if we assume that the writer is correct, there is a linearizable implementation of a \([1, n]\)-register from atomic \([1, 1]\)-registers that satisfies the Termination property and tolerates any number of malicious readers. Thus, Theorems 48 and 70 imply that there is an inherent difference between achieving Termination and achieving Bounded Termination in systems with Byzantine failures.

To prove Theorem 70, it is easy to modify the impossibility proof of Theorem 7 given in Section 4 as we now explain. First note that in the successive runs that we construct in the proof of Theorem 7, we leverage the fact that a correct reader that starts a read operation cannot wait for the writer to complete a concurrent write, even if the reader is aware that this write is in progress: in our runs, the writer actually crashes so such waiting is not possible. So a correct reader must complete its read operation even if the writer stops taking steps. However, if the writer is assumed to be correct and we require “only” Termination, a reader that is aware that a write operation is in progress can wait for the writer to complete this operation to determine what value to read (so the impossibility proof breaks down in this case, as it should). But if we require the register implementation to satisfy Bounded Termination, then we are back to a situation where a reader must complete its operation without waiting for the writer to take steps. So the proof of Theorem 70 can use this fact exactly as the proof of Theorem 7 does.

Thus, we can modify the proof of Theorem 7 to obtain one for Theorem 70 as follows. Roughly speaking, whenever the writer \(w\) crashes in the proof of Theorem 7, the writer is correct but just pauses in the proof of Theorem 70: the writer later resumes taking steps and completes its write operation, but it does so only after the readers complete their read operations. Since the delayed steps of the writer are not seen by the readers, they behave as in the proof of Theorem 7.

More precisely, in the successive runs that we construct in the proof:
- If, in the proof of Theorem 7, the writer \(w\) crashes after taking a step \(s^k\) and before taking further steps,
- then, in the proof of Theorem 70, the correct writer \(w\) temporarily stops taking steps after taking a step \(s^k\) and before taking further steps.

The readers behave the same in the corresponding runs of both proofs. After the reads by correct readers are completed, the writer resumes taking steps and completes its operation. Since the proof of Theorem 70 is mostly a verbatim repetition of the proof of Theorem 7, we relegate it to Appendix B.

8 Implementations from regular registers

In a seminal work [13], Lamport considered the problem of implementing “atomic” registers from regular registers, in systems where processes may crash. Recall that, as with an atomic register, a regular register ensures that a reader reads the “current” value of the register, but in contrast to an atomic register, a regular register allows “new-old” inversions in the values read. In other words, a regular register must satisfy only Property 1 of the register linearizability Definition 5.

We now consider this problem for systems where processes are subject to Byzantine failures. To do so, we must first define what it means for a register implementation to be an implementation of a regular register in systems where processes can be malicious. Intuitively, we require that if the writer is not malicious then non-malicious readers must read the “current” value of the register (this is Property 1 of Definition 6):

- **Definition 71 (Register Regularity).** In a system with Byzantine process failures, an implementation of a \([1, n]\)-register is a regular register implementation if and only if, when the writer is not malicious, the following property holds:
  - [Reading a “current” value] If a read operation \(R\) by a process that is not malicious returns...
On implementing SWMR registers from SWSR registers in systems with Byzantine failures

An impossibility result. Recall that, by Theorem 4, in a system with Byzantine failures there is no linearizable implementation of a \([1, n]−\text{register}\) from atomic \([1, n−1]−\text{registers}\) that satisfies the Termination property. This raises the following question: What happens if all processes, including the writer, are given regular \([1, n]−\text{registers}\) instead of atomic \([1, n−1]−\text{registers}\)? Note that these regular registers can be read by all the \(n\) readers, as in the desired register implementation, but they are “only” regular. This question is answered by the following:

\[ \text{Theorem 72. For all } n \geq 3, \text{ in a system with } n+1 \text{ processes that are subject to Byzantine failures, there is no linearizable implementation of a } [1, n]−\text{register from regular } [1, n]−\text{registers that satisfies the Termination property, even if we assume that the writer of the implemented } [1, n]−\text{register can only crash and at most one reader can be malicious.} \]

The above impossibility result is in sharp contrast to a corresponding possibility result in the case of systems with only crash failures: in such systems it is easy to implement a wait-free linearizable \([1, n]−\text{register}\) from regular \([1, n]−\text{registers}\).

Proof. Let \(n \geq 3\). Consider a system with \(n+1\) processes with Byzantine failures.

\[ \text{\triangleright Claim 73. For every process } x, \text{ there is a wait-free implementation of a regular } [1, n]−\text{register } R_{xn}, \text{ writable by } x \text{ and readable by the other } n \text{ processes, from atomic } [1, 1]−\text{registers}.} \]

Proof. The implementation of \(R_{xn}\) is very simple. For each reader \(p\), the writer \(x\) has an atomic \([1, 1]−\text{register } R_{xp}\) that \(x\) can write and \(p\) can read.

- To write a value \(v\) into \(R_{xn}\), the writer \(x\) successively writes \(v\) into \(R_{xp}\) for every reader \(p\).
- To read a value from \(R_{xn}\), a reader \(p\) reads register \(R_{xp}\) and returns the value read.

It is clear that if the writer \(x\) is not malicious, then any non-malicious reader \(p\) that reads \(R_{xn}\), reads the value written into \(R_{xn}\) by a write operation by \(x\) that immediately precedes the read of \(R_{xn}\) or is concurrent with this read; more precisely, this implementation of \(R_{xn}\) satisfies the property of regular registers, namely, Property 1 of Definition 71.

Let \(w\) be any process. Assume \(w\) can only crash and at most one of the remaining \(n\) processes can be malicious. For contradiction, suppose that using regular \([1, n]−\text{registers}\) there is an implementation \(I_n\) of a \([1, n]−\text{register}\), writable by \(w\) and readable by the other \(n\) processes, such that: (1) \(I_n\) is linearizable, and (2) \(I_n\) satisfies the Termination property.

By Claim 73, every regular \([1, n]−\text{register}\) used by \(I_n\) has a wait-free implementation from atomic \([1, 1]−\text{registers}\). Thus, by replacing every regular \([1, n]−\text{register}\) used by \(I_n\) with its corresponding wait-free implementation, we obtain an implementation \(I'_n\) of a \([1, n]−\text{register}\), writable by \(w\) and readable by the other \(n\) processes, from atomic \([1, 1]−\text{registers}\). It is clear that like \(I_n\): (1) \(I'_n\) is linearizable, and (2) \(I'_n\) satisfies the Termination property. Therefore \(I'_n\) contradicts Theorem 7.

9 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, without signatures, there is no wait-free linearizable implementation of a \([1, n]−\text{register}\) from atomic \([1, n−1]−\text{registers}\). In fact, we showed a stronger result,

\[ \text{Recall that } v_0 \text{ is the initial value of the implemented register.} \]
namely, even under the assumption that the writer can only crash and at most one reader can be malicious, there is no linearizable implementation of a $[1, n]$-register from atomic $[1, n-1]$-registers that ensures that every correct process eventually completes its operations. In light of this strong impossibility result, we gave an implementation of a $[1, n]$-register from atomic $[1, 1]$-registers that is “safe” (i.e., it is linearizable) under any combination of Byzantine process failures, but it is “live” (i.e., it ensures that every correct process eventually completes its operations) only under the assumption that the writer is correct or no reader is malicious; this matches the impossibility result.

If we assume that the writer is correct, with the above implementation (which tolerates any number of malicious readers) every reader completes each read in a finite number of steps. We showed that is impossible to ensure they do so in a bounded number of steps, even if we make the additional assumption that at most one reader can be malicious.

In sharp contrast with the above results, for the case that processes can use signatures, we gave a bounded wait-free linearizable implementation of a $[1, n]$-register from atomic $[1, 1]$-registers which does not rely on any failure assumptions.

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. [14] prove that one can implement a linearizable $f$-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 [5] show that if $f < n/2$, one can use atomic $[1, n]$-registers to get a linearizable $f$-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 [14], 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 [5, 14]) require every correct process to help the execution of every operation, even the operations of other processes.

In contrast, with object implementations in shared-memory systems, a common assumption is that processes that do not have ongoing operations take no steps; so a process that executes an operation cannot count on getting help from any process that is not currently executing its own operation.

**Acknowledgments**

We thank Vassos Hadzilacos for his helpful comments on this paper. This work was partially funded by the Natural Sciences and Engineering Research Council of Canada (Grant number: RGPIN-2014-05296).
On implementing SWMR registers from SWSR registers in systems with Byzantine failures

References

1. Marcos K. Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra Marathe, and Igor Zablotski. The impact of RDMA on agreement. In Proceedings of the 38th Annual ACM Symposium on Principles of Distributed Computing, PODC ’19, page 409–418, 2019.

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

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

4. James E. Burns and Gary L. Peterson. Constructing multi-reader atomic values from non-atomic values. In Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing, PODC ’87, page 222–231, 1987.

5. 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 ’21, pages 18:1–18:18, 2021.

6. Wojciech Golab, Lisa Higham, and Philipp Woelfel. Linearizable implementations do not suffice for randomized distributed computation. In Proceedings of the Forty-Third Annual ACM Symposium on Theory of Computing, STOC ’11, page 373–382, 2011.

7. Jim Gray. Notes on data base operating systems. In Operating Systems, An Advanced Course, page 393–481, 1978.

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

9. Maurice Herlihy. Wait-free synchronization. ACM Trans. Program. Lang. Syst., 13(1):124–149, 1991.

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.

11. Xing Hu and Sam Toueg. On Implementing SWMR Registers from SWSR Registers in Systems with Byzantine Failures. In 36th International Symposium on Distributed Computing, DISC ’22, pages 36:1–36:19, 2022.

12. Amos Israeli and Amnon Shaham. Optimal multi-writer multi-reader atomic register. In Proceedings of the 11th Annual ACM Symposium on Principles of Distributed Computing, PODC ’92, page 71–82, 1992.

13. Leslie Lamport. On interprocess communication Parts I–II. Distributed Computing, 1(2):77–101, 1986.

14. 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, 05 2017. [doi:10.1007/s00224-016-9699-8]

15. Richard Newman-Wolfe. A protocol for wait-free, atomic, multi-reader shared variables. In Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing, PODC ’87, page 232–248, 1987.

16. Gary L. Peterson. Concurrent reading while writing. ACM Trans. Program. Lang. Syst., 5(1):46–55, jan 1983.

17. 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, SFCS ’87, page 383–392, 1987.

18. Ambuj K. Singh, James H. Anderson, and Mohamed G. Gouda. The elusive atomic register revisited. In Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing, PODC ’87, page 206–221, 1987.

19. K. Vidyasankar. Converting Lamport’s regular register to atomic register. Inf. Process. Lett., 28(6):287–290, aug 1988.

20. K. Vidyasankar. A very simple construction of 1-writer multireader multivalued atomic variable. Inf. Process. Lett., 37(6):323–326, mar 1991.

21. Paul M. B. Vitanyi and Baruch Awerbuch. Atomic shared register access by asynchronous hardware. In 27th Annual Symposium on Foundations of Computer Science, SFCS ’86, pages 233–243, 1986.
Algorithm 3 Implementation $I'_2$ of a $[1,2]$-register writable by $w$ and readable by $p$ and $q$. $I_2$ uses atomic $[1,1]$-registers.

**ATOMIC REGISTERS**

- $R_{wp}$: $[1,1]$-register; initially (COMMIT, $⟨0, u_0⟩$)
- $R_{rq}$: $[1,1]$-register; initially (COMMIT, $⟨0, u_0⟩$)
- $R_{pq}$: $[1,1]$-register; initially $(0, u_0)$

**LOCAL VARIABLES**

- $c$: variable of $w$; initially $0$
- $last\_read$: variable of $q$ initially $(0, u_0)$
- $last\_written$: variable of $w$; initially $(0, u_0)$

**WRITE($u$):** ▶ executed by the writer $w$

1. $c ← c + 1$
2. call $w((c, u))$
3. return done

**READ():** ▶ executed by any reader $r ∈ \{p, q\}$

4. call $r\_r()$
5. if this call returns some tuple $(k, u)$ then
   6. return $u$
   7. else return ⊥

**W($(k, u)$):** ▶ executed by $w$ to do its $k$-th write

8. $R_{wp} ← (\text{PREPARE, last\_written, } ⟨k, u⟩)$
9. $R_{rq} ← (\text{PREPARE, last\_written, } ⟨k, u⟩)$
10. $R_{pq} ← (\text{COMMIT, } ⟨k, u⟩)$
11. $R_{wp} ← (\text{COMMIT, } ⟨k, u⟩)$
12. last\_written ← $⟨k, u⟩$
13. return done

**R$_p()$:** ▶ executed by reader $p$

14. if $R_{wp} = (\text{COMMIT, } ⟨k, u⟩)$ for some $(k, u)$ then
15. $R_{pq} ← ⟨k, u⟩$
16. return $(k, u)$
17. else if $R_{wp} = (\text{PREPARE, last\_written, } −)$ for some last\_written then
18. return last\_written
19. else return ⊥

**R$_q()$:** ▶ executed by reader $q$

20. if $R_{rq} = (\text{COMMIT, } ⟨k, u⟩)$ for some $(k, u)$ then
21. return $(k, u)$
22. else if $R_{rq} = (\text{PREPARE, last\_written, } ⟨k, u⟩)$ for some last\_written and some $(k, u)$ then
23. if $R_{pq} = ⟨k′, −⟩$ for some $k′ ≥ k$ then
24. last\_read ← $⟨k, u⟩$
25. return $(k, u)$
26. else last\_read = $(k′, −)$ and some $k′ ≥ k$ then
27. return $(k, u)$
28. else
29. return last\_written
30. else return ⊥

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 (Section 5.3): $I'_2$ 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. We now prove the correctness of $I'_2$.

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 its own steps. Thus:

**Observation 74.** The implementation $I'_2$ is bounded wait-free.

The proof that $I'_2$ is linearizable is in many parts similar (or even identical) to the proof that the register implementation $I_n$ is linearizable (Theorem 42 in Section 5.3). It is given here for completeness.
To prove that $I'_2$ is linearizable, we consider two cases:

**CASE 1:** The writer $w$ of the register implemented by $I'_2$ is malicious. By Definition 6, $I'_2$ is (trivially) linearizable in this case.

**CASE 2:** The writer $w$ of the register implemented by $I'_2$ is not malicious.

For this case, we now prove that the read and write operations of the implemented register satisfy the linearizability Properties 1 and 2 of Definition 6.

In the following:

- $u_0$ is the initial value of the register that $I'_2$ implements.
- For $k \geq 1$, $u_k$ denotes the $k$-th value written by $w$ using the procedure WRITE(). More precisely, if $w$ calls WRITE() with a value $u$ and this is its $k$-th call of WRITE(), then $u_k$ is $u$.
- $v_0$ is $\langle 0, u_0 \rangle$.
- For $k \geq 1$, $v_k$ denotes the $k$-th value written by $w$ using the procedure W().

**Observation 75.** For all $k \geq 0$, $v_k = \langle k, u_k \rangle$.

**Observation 76.** Let $W(v)$ be any write operation by $w$. Then there is a $k \geq 1$ such that $v = v_k$.

**Observation 77.** Let $R \in \{R_{wp}, R_{wq}\}$. If $w$ writes $x$ in $R$, then $x = (\text{COMMIT}, v_k)$ for some $k \geq 1$ or $x = (\text{PREPARE}, v_k, v_{k+1})$ for some $k \geq 0$.

**Observation 78.** Suppose $p$ is not malicious. If $p$ reads $R_{wp} = x$, then $x = (\text{COMMIT}, v_k)$ or $x = (\text{PREPARE}, v_k, v_{k+1})$, for some $k \geq 0$.

**Observation 79.** Suppose $q$ is not malicious. If $q$ reads $R_{wq} = x$, then $x = (\text{COMMIT}, v_k)$ or $x = (\text{PREPARE}, v_k, v_{k+1})$, for some $k \geq 0$.

**Lemma 80.** Suppose $p$ is not malicious. Let $R_p(v)$ be any read operation by $p$. Then there is a $k \geq 0$ such that $v = v_k$, and

- If $p$ reads $R_{wp} = (\text{COMMIT}, v_k)$ in line 14 of $R_p(v)$, then $p$ reads $R_{wp} = (\text{PREPARE}, v_k, v_{k+1})$ in line 17 of $R_p(v)$.

**Proof.** Suppose $p$ is not malicious. Let $R_p(v)$ be any read operation by $p$. Note that $p$ reads $R_{wp}$ in $R_p(v)$. When it does so, by Observation 78, there are two possible cases:

1. $p$ reads $R_{wp} = (\text{COMMIT}, v_k)$ for some $k \geq 0$ in line 14 of $R_p(v)$. Then $R_p(v)$ returns $v_k$ in line 16, i.e., $v = v_k$.
2. $p$ reads $R_{wp} = (\text{PREPARE}, v_k, v_{k+1})$ for some $k \geq 0$ in line 17 of $R_p(v)$. Then $R_p(v)$ returns $v_k$ in line 18, i.e., $v = v_k$.

**Lemma 81.** Suppose $q$ is not malicious. Let $R_q(v)$ be any read operation by $q$. Then there is a $k \geq 0$ such that $v = v_k$, and

- If $q$ reads $R_{wq} = (\text{COMMIT}, v_k)$ in line 20 of $R_q(v)$,
- If $q$ reads $R_{wq} = (\text{PREPARE}, v_{k-1}, v_k)$ in line 22 of $R_q(v)$, or
- If $q$ reads $R_{wq} = (\text{PREPARE}, v_k, v_{k+1})$ in line 22 of $R_q(v)$.

**Proof.** Suppose $q$ is not malicious. Let $R_q(v)$ be any read operation by $q$. Note that $q$ reads $R_{wq}$ in $R_q(v)$. When it does so, by Observation 79, there are two possible cases:

---

11 For brevity, we say that “a process $r$ reads or writes a register in line $x$ of a $R_r(-)$ or a $W(-)$ operation”, if it reads or writes this register in line $x$ of the $R_r()$ or $W()$ procedure executed to do this $R_r(-)$ or $W(-)$ operation.
1. \( q \) reads \( R_{wq} = (\text{commit}, v_k) \) for some \( k \geq 0 \) in line 29 of \( R_q(v) \). Then \( R_q(v) \) returns \( v_k \) in line 21, i.e., \( v = v_k \).
2. \( q \) reads \( R_{wq} = (\text{prepare}, v_k, v_{k+1}) \) for some \( k \geq 0 \) in line 22 of \( R_q(v) \). Then there are two subcases:
   a. \( R_q(v) \) returns \( v_{k+1} \) in line 25 or 27, i.e., \( v = v_{k+1} \). Let \( k' = k + 1 \). Then in this case, \( q \) reads \( R_{wq} = (\text{prepare}, v_{k'-1}, v_{k'}) \) in line 22 of \( R_q(v) \) and \( v = v_{k'} \).
   b. \( R_q(v) \) returns \( v_k \) in line 29, i.e., \( v = v_k \).

**Observation 82.** Let \( R \) be any register in \( \{R_{wp}, R_{wq}\} \).

1. If \( w \) writes \((\text{prepare}, v_{k-1}, v_k) \) in \( R \) before \( w \) writes \((\text{commit}, v_k) \) in \( R \), then \( k \leq k' \).
2. If \( w \) writes \((\text{prepare}, v_{k-1}, v_k) \) in \( R \) before \( w \) writes \((\text{prepare}, v_{k'-1}, v_{k'}) \) in \( R \), then \( k < k' \).
3. If \( w \) writes \((\text{commit}, v_k) \) in \( R \) before \( w \) writes \((\text{commit}, v_{k'}) \) in \( R \), then \( k < k' \).
4. If \( w \) writes \((\text{commit}, v_k) \) in \( R \) before \( w \) writes \((\text{prepare}, v_{k'-1}, v_{k'}) \) in \( R \), then \( k < k' \).

**Observation 83.** Let \( R \) be any register in \( \{R_{wp}, R_{wq}\} \). Suppose \( r \in \{p, q\} \) is not malicious.

1. If \( w \) writes \((\text{prepare}, v_{k-1}, v_k) \) in \( R \) before \( r \) reads \((\text{commit}, v_{k'}) \) in \( R \), then \( k < k' \).
2. If \( w \) writes \((\text{prepare}, v_{k-1}, v_k) \) in \( R \) before \( r \) reads \((\text{prepare}, v_{k'-1}, v_{k'}) \) in \( R \), then \( k < k' \).
3. If \( w \) writes \((\text{commit}, v_k) \) in \( R \) before \( r \) reads \((\text{commit}, v_{k'}) \) in \( R \), then \( k < k' \).
4. If \( w \) writes \((\text{commit}, v_k) \) in \( R \) before \( r \) reads \((\text{prepare}, v_{k'-1}, v_{k'}) \) in \( R \), then \( k < k' \).

**Observation 84.** Let \( R \) be any register in \( \{R_{wp}, R_{wq}\} \). Suppose \( r \) and \( r' \) are non-malicious processes in \( \{p, q\} \).

1. If \( r \) reads \( R = (\text{prepare}, v_{k-1}, v_k) \) before \( r' \) reads \( R = (\text{commit}, v_{k'}) \), then \( k < k' \).
2. If \( r \) reads \( R = (\text{prepare}, v_{k-1}, v_k) \) before \( r' \) reads \( R = (\text{prepare}, v_{k'-1}, v_{k'}) \), then \( k < k' \).
3. If \( r \) reads \( R = (\text{commit}, v_k) \) before \( r' \) reads \( R = (\text{commit}, v_{k'}) \), then \( k < k' \).
4. If \( r \) reads \( R = (\text{commit}, v_k) \) before \( r' \) reads \( R = (\text{prepare}, v_{k'-1}, v_{k'}) \), then \( k < k' \).

**Proof of Linearizability Property 1.** We now prove that the write and read operations of the register that \( T_2 \) implements satisfy Property 1 of Definition 6, i.e., processes read the “current” value of the register. To do so, we first prove this for the writes and reads of the lower-level procedures \( w() \) and \( R_r() \) for all readers \( r \in \{p, q\} \) (Lemma 85), and then prove it for the writes and reads of the high-level procedures \( \text{WRITE}() \) and \( \text{READ}() \) (Lemma 89).

**Lemma 85.** If \( R_r(v) \) is a read operation by a non-malicious process \( r \in \{p, q\} \) then:

- there is a \( w(v) \) operation that immediately precedes \( R_r(v) \) or is concurrent with \( R_r(v) \), or
- \( v = v_0 \) and no \( w(-) \) operation precedes \( R_r(v) \).

Proof. Let \( R_r(v) \) be any read operation by a non-malicious process \( r \in \{p, q\} \). By Lemmas 80 and 81, \( v = v_k \) for some \( k \geq 0 \). We now show that:

- if \( k = 0 \) then no \( w(-) \) operation precedes \( R_r(v_k) \), and
- if \( k > 0 \) then a \( w(v_k) \) operation immediately precedes \( R_r(v_k) \) or is concurrent with \( R_r(v_k) \).

There are two cases: \( r = p \) or \( r = q \).

- Case 1: \( r = p \). By Lemma 80, there are two cases:
  1) \( p \) reads \( R_{wp} = (\text{commit}, v_k) \) in line 14 of \( R_p(v_k) \). There are two cases:
i. $k = 0$. Suppose, for contradiction, that there is a $w(v)$ operation that precedes $r_p(v_0)$. By Observation 76, $v = v_i$ for some $i \geq 1$. So $w$ writes (commit, $v_i$) into $R_{wp}$ in line 10 of $w(v_i)$ before $p$ reads $R_{wp} = (\text{commit}, v_0)$ in line 14 of $r_p(v_k)$. By Observation 83[3], $i \leq 0$ — a contradiction. So no $w(\cdot)$ operation precedes $r_p(v_0)$.

ii. $k > 0$. Then $w$ writes (commit, $v_k$) into $R_{wp}$ in line 10 of $w(v_k)$ before $p$ reads $R_{wp} = (\text{commit}, v_k)$ in $r_p(v_k)$. So the $w(v_k)$ operation precedes $r_p(v_k)$ or is concurrent with $r_p(v_k)$. We now show that if $w(v_k)$ precedes $r_p(v_k)$, then $w(v_k)$ immediately precedes $r_p(v_k)$. Suppose, for contradiction, that $w(v_k)$ precedes $r_p(v_k)$ but does not immediately precede $r_p(v_k)$. Then there is a $w(v_i)$ operation that immediately precedes $r_p(v_k)$. Clearly, the $w(v_k)$ operation precedes the $w(v_i)$ operation, and so $i > k$. Furthermore, $w$ writes (commit, $v_i$) into $R_{wp}$ in line 10 of $w(v_i)$ before $p$ reads $R_{wp} = (\text{commit}, v_k)$ in line 14 of $r_p(v_k)$. By Observation 83[3], $i \leq k$ — a contradiction. Therefore the $w(v_k)$ operation immediately precedes $r_p(v_k)$ or is concurrent with $r_p(v_k)$.

2) $p$ reads $R_{wp} = (\text{prepare}, v_{k+1})$ in line 17 of $r_p(v_k)$. Then this read occurs after $w$ writes (prepare, $v_{k+1}$) in $R_{wp}$ in line 8 of the $w(v_{k+1})$ operation. Furthermore, by Observation 83[4], this read occurs before $w$ writes (commit, $v_{k+1}$) in $R_{wp}$ in line 10 of the $w(v_{k+1})$ operation. Therefore the $w(v_{k+1})$ operation is concurrent with $r_p(v_k)$.

There are two cases:

i. $k = 0$. Since $w(v_1)$ is concurrent with $r_p(v_0)$, no $w(\cdot)$ operation precedes $r_p(v_0)$.

ii. $k > 0$. Since $w(v_{k+1})$ is concurrent with $r_p(v_k)$, $w(v_k)$ immediately precedes $r_p(v_k)$ or is concurrent with $r_p(v_k)$.

Case 2: $r = q$. By Lemma 81, there are three cases:

1) $q$ reads $R_{wq} = (\text{commit}, v_k)$ in line 20 of $r_q(v_k)$. There are two cases:

i. $k = 0$. Suppose, for contradiction, that there is a $w(v)$ operation that precedes $r_q(v_0)$. By Observation 76, $v = v_i$ for some $i \geq 1$. So $w$ writes (commit, $v_i$) into $R_{wq}$ in line 11 of $w(v_i)$ before $q$ reads $R_{wq} = (\text{commit}, v_k)$ in line 20 of $r_q(v_k)$. By Observation 83[3], $i \leq 0$ — a contradiction. So no $w(\cdot)$ operation precedes $r_q(v_0)$.

ii. $k > 0$. Then $w$ writes (commit, $v_k$) into $R_{wq}$ in line 11 of $w(v_k)$ before $q$ reads $R_{wq} = (\text{commit}, v_k)$ in line 20 of $r_q(v_k)$. So the $w(v_k)$ operation precedes $r_q(v_k)$ or is concurrent with $r_q(v_k)$. We now show that if $w(v_k)$ precedes $r_q(v_k)$, then $w(v_k)$ immediately precedes $r_q(v_k)$. Suppose, for contradiction, that $w(v_k)$ precedes $r_q(v_k)$ but does not immediately precede $r_q(v_k)$. Then there is a $w(v_i)$ operation that immediately precedes $r_q(v_k)$. Clearly, the $w(v_k)$ operation precedes the $w(v_i)$ operation, and so $i > k$. Furthermore, $w$ writes (commit, $v_i$) into $R_{wq}$ in line 11 of $w(v_i)$ before $q$ reads $R_{wq} = (\text{commit}, v_k)$ in line 20 of $r_q(v_k)$. By Observation 83[3], $i \leq k$ — a contradiction. Therefore the $w(v_k)$ operation immediately precedes $r_q(v_k)$ or is concurrent with $r_q(v_k)$.

2) $q$ reads $R_{wq} = (\text{prepare}, v_{k+1})$ in line 22 of $r_q(v_k)$. Then this read occurs after $w$ writes (prepare, $v_{k+1}$) in $R_{wq}$ in line 9 of the $w(v_{k+1})$ operation. Furthermore, by Observation 83[4], this read occurs before $w$ writes (commit, $v_{k+1}$) in $R_{wq}$ in line 11 of the $w(v_{k+1})$ operation. Therefore the $w(v_{k+1})$ operation is concurrent with $r_q(v_k)$.

There are two cases:

i. $k = 0$. Since $w(v_1)$ is concurrent with $r_q(v_0)$, no $w(\cdot)$ operation precedes $r_q(v_0)$.

ii. $k > 0$. Since $w(v_{k+1})$ is concurrent with $r_q(v_k)$, $w(v_{k+1})$ is concurrent with $r_q(v_k)$ or immediately precedes $r_q(v_k)$.
We now prove that the write and read operations of the high-level procedures \texttt{WRITE()} and \texttt{READ()} satisfy Property 1 of Definition [6].

By Observation [75], Lemmas [80] and [81], and the code of the procedure \texttt{READ()}, we have:

\begin{itemize}
  \item \textbf{Observation 86.} If \texttt{READ}\,(u) is an operation by a non-malicious process \( r \in \{p,q\} \), then \( u = u_k \) for some \( k \geq 0 \).
  
  \item \textbf{Observation 87.} If \texttt{READ}\,(u_k) is an operation by a non-malicious process \( r \in \{p,q\} \), then \( r \) invokes and completes a \( r\_v(v_k) \) operation in \texttt{READ}\,(u_k).
  
  \item \textbf{Observation 88.} If \texttt{WRITE}\,(u_k) is a completed operation by \( w \), then \( w \) invokes and completes a \( w(v_k) \) operation in \texttt{WRITE}\,(u_k).
\end{itemize}

We now prove that the \texttt{WRITE}(-) and \texttt{READ}(-) operations satisfy Property 1 of Definition [6].

\begin{itemize}
  \item \textbf{Lemma 89.} [Property 1: Reading a “current” value]

  If \texttt{READ}\,(u) is a read operation by a non-malicious process \( r \in \{p,q\} \) then:

  \begin{itemize}
    \item there is a \texttt{WRITE}\,(u) operation that immediately precedes \texttt{READ}\,(u) or is concurrent with \texttt{READ}\,(u), or
    
    \item \( u = u_0 \) and no \texttt{WRITE}(-) operation precedes \texttt{READ}\,(u).
  \end{itemize}

  \textbf{Proof.} Let \texttt{READ}\,(u) be any read operation by a non-malicious process \( r \in \{p,q\} \). By Observation [86], \( u = u_k \) for some \( k \geq 0 \). There are two cases:

  (1) \( k = 0 \). Suppose, for contradiction, that a \texttt{WRITE}\,(u_i) operation precedes \texttt{READ}\_r\,(u_0).

   Note that \( i \geq 1 \). By Observations [88] and [57], a \( w(v_i) \) operation precedes a \( r\_v(v_0) \) operation. Since process \( r \) is not malicious, by Lemma [55], there is no \( w(-) \) operation that precedes \( r\_v(v_0) \) — a contradiction.

  (2) \( k > 0 \). By Observation [57], \( r \) invokes and completes a \( r\_v(v_k) \) operation in \texttt{READ}\,(u_k).

   Since \( k > 0 \), by Lemma [85], there is a \( w(v_k) \) operation that immediately precedes \( r\_v(v_k) \) or is concurrent with \( r\_v(v_k) \). Let \texttt{WRITE}\,(u_k) be the operation in which \( w \) invokes the \( w(v_k) \) operation. Since \( w(v_k) \) immediately precedes \( r\_v(v_k) \) or is concurrent with \( r\_v(v_k) \), the \texttt{WRITE}\,(u_k) operation immediately precedes \texttt{READ}\,(u_k) or is concurrent with \texttt{READ}\,(u_k).
\end{itemize}

\textbf{Proof of linearizability Property 2.} We now prove that the write and read operations of the register that \( T'_v \) implements satisfy Property 2 of Definition [6] i.e., we prove that there are no “new-old” inversions in the values that processes read. To do so, we first prove this for the writes and reads of the lower-level procedures \( w() \), \( r\_p() \), and \( r\_s() \), and then prove it for the writes and reads of the high-level procedures \texttt{WRITE()} and \texttt{READ()} (Lemma [104]).

We first show that there are no “new-old” inversions in the consecutive reads of process \( p \).

\begin{itemize}
  \item \textbf{Lemma 90.} Suppose \( p \) is not malicious. If \( r\_p(v_k) \) and \( r\_p(v_k') \) are read operations by \( p \), and \( r\_p(v_k) \) precedes \( r\_p(v_k') \), then \( k \leq k' \).
\end{itemize}

\textbf{Proof.} Suppose \( p \) is not malicious. Let \( r\_p(v_k) \) and \( r\_p(v_k') \) be read operations by \( p \) such that \( r\_p(v_k) \) precedes \( r\_p(v_k') \). By Lemma [80], the following occurs:

1. \( p \) reads \( R_{wp} = \{\text{COMMIT}, v_k\} \) in line 14 of \( r\_p(v_k) \), or
2. \( p \) reads \( R_{wp} = \{\text{PREPARE}, v_k, v_{k+1}\} \) in line 17 of \( r\_p(v_k) \)

before the following occurs:

1. \( p \) reads \( R_{wp} = \{\text{COMMIT}, v_k'\} \) in line 14 of \( r\_p(v_k') \), or
2. \( p \) reads \( R_{wp} = \{\text{PREPARE}, v_{k'}, v_{k'+1}\} \) in line 17 of \( r\_p(v_k') \).
So there are four possible cases:
1. $p$ reads $R_{wp} = (\text{COMMIT}, v_k)$ in $R_p(v_k)$ before $p$ reads $R_{wp} = (\text{COMMIT}, v_{k'})$ in $R_p(v_{k'})$.
   By Observation 84(1), $k \leq k'$.
2. $p$ reads $R_{wp} = (\text{COMMIT}, v_k)$ in $R_p(v_k)$ before $p$ reads $R_{wp} = (\text{PREPARE}, v_{k'}, v_{k'+1})$ in $R_p(v_{k'})$. By Observation 84(1), $k < k' + 1$. So $k \leq k'$.
3. $p$ reads $R_{wp} = (\text{PREPARE}, v_k, v_{k'+1})$ in $R_p(v_k)$ before $p$ reads $R_{wp} = (\text{COMMIT}, v_{k'})$ in $R_p(v_{k'})$. By Observation 84(1), $k + 1 \leq k'$. So $k \leq k'$.
4. $p$ reads $R_{wp} = (\text{PREPARE}, v_k, v_{k'+1})$ in $R_p(v_k)$ before $p$ reads $R_{wp} = (\text{PREPARE}, v_{k'}, v_{k'+1})$ in $R_p(v_{k'}).$ By Observation 84(1), $k + 1 \leq k'$. So $k \leq k'$.

To prove that there are no “new-old” inversions between the reads of $p$ and $q$, and also between the reads of $q$ itself, we first make some straightforward observations that are clear from the code of $T_2$. We first note that the counters of the tuples in the register $R_{pq}$ do not decrease.

► Observation 91. Suppose $p$ is not malicious. If $p$ writes $v_k$ in $R_{pq}$ before $p$ writes $v_{k'}$ in $R_{pq}$, then $k \leq k'$.

► Observation 92. Suppose $p$ and $q$ are not malicious. If $p$ writes $v_k$ in $R_{pq}$ before $q$ reads $R_{pq} = v_{k'}$, then $k \leq k'$.

The following observations relate the counters of the tuples that $w$ successively writes in registers $R_{wp}$ and $R_{wq}$.

► Observation 93. (1) If $w$ writes $(\text{PREPARE}, v_{k-1}, v_k)$ in $R_{wp}$ before $w$ writes $(\text{PREPARE}, v_{k'-1}, v_{k'})$ in $R_{wq}$, then $k \leq k'$.
(2) If $w$ writes $(\text{PREPARE}, v_{k-1}, v_k)$ in $R_{wp}$ before $w$ writes $(\text{COMMIT}, v_{k'})$ in $R_{wq}$, then $k \leq k'$.
(3) If $w$ writes $(\text{COMMIT}, v_k)$ in $R_{wp}$ before $w$ writes $(\text{PREPARE}, v_{k'-1}, v_{k'})$ in $R_{wq}$, then $k < k'$.
(4) If $w$ writes $(\text{COMMIT}, v_k)$ in $R_{wp}$ before $w$ writes $(\text{COMMIT}, v_{k'})$ in $R_{wq}$, then $k \leq k'$.
(5) If $w$ writes $(\text{PREPARE}, v_{k-1}, v_k)$ in $R_{wq}$ before $w$ writes $(\text{PREPARE}, v_{k-1}, v_{k'})$ in $R_{wq}$, then $k < k'$.
(6) If $w$ writes $(\text{PREPARE}, v_{k-1}, v_k)$ in $R_{wq}$ before $w$ writes $(\text{COMMIT}, v_{k'})$ in $R_{wq}$, then $k \leq k'$.
(7) If $w$ writes $(\text{COMMIT}, v_k)$ in $R_{wq}$ before $w$ writes $(\text{PREPARE}, v_{k-1}, v_{k'})$ in $R_{wq}$, then $k < k'$.
(8) If $w$ writes $(\text{COMMIT}, v_k)$ in $R_{wq}$ before $w$ writes $(\text{COMMIT}, v_{k'})$ in $R_{wq}$, then $k < k'$.

The next observations relate the counters of the tuples that $p$ and $q$ read from $R_{wp}$ and $R_{wq}$, respectively.

► Observation 94. Suppose $p$ and $q$ are not malicious.
(1) If $p$ reads $R_{wp} = (\text{PREPARE}, v_{k-1}, v_k)$ before $q$ reads $R_{wq} = (\text{PREPARE}, v_{k'-1}, v_{k'})$, then $k \leq k'$.
(2) If $p$ reads $R_{wp} = (\text{PREPARE}, v_{k-1}, v_k)$ before $q$ reads $R_{wq} = (\text{COMMIT}, v_{k'})$, then $k - 1 \leq k'$.
(3) If $p$ reads $R_{wp} = (\text{COMMIT}, v_k)$ before $q$ reads $R_{wq} = (\text{PREPARE}, v_{k'-1}, v_{k'})$, then $k \leq k'$.
(4) If $p$ reads $R_{wp} = (\text{COMMIT}, v_k)$ before $q$ reads $R_{wq} = (\text{COMMIT}, v_{k'})$, then $k \leq k'$.
(5) If $q$ reads $R_{wq} = (\text{PREPARE}, v_{k-1}, v_k)$ before $p$ reads $R_{wp} = (\text{PREPARE}, v_{k'-1}, v_{k'})$, then $k \leq k'$.
(6) If $q$ reads $R_{wq} = (\text{PREPARE}, v_{k-1}, v_k)$ before $p$ reads $R_{wp} = (\text{COMMIT}, v_{k'})$, then $k \leq k'$.
(7) If $q$ reads $R_{wq} = (\text{COMMIT}, v_k)$ before $p$ reads $R_{wp} = (\text{PREPARE}, v_{k'-1}, v_{k'})$, then $k + 1 \leq k'$.
(8) If $q$ reads $R_{wq} = (\text{COMMIT}, v_k)$ before $p$ reads $R_{wp} = (\text{COMMIT}, v_{k'})$, then $k \leq k'$. 
Now we prove that there is no “new-old” inversion for a read by p that precedes a read by q.

Lemma 95. If \( R_p(v_k) \) and \( R_q(v_{k'}) \) are read operations by non-malicious processes p and q respectively, and \( R_p(v_k) \) precedes \( R_q(v_{k'}) \), then \( k \leq k' \).

Proof. Suppose processes p and q are not malicious. Let \( R_p(v_k) \) and \( R_q(v_{k'}) \) be read operations by p and q respectively, such that \( R_p(v_k) \) precedes \( R_q(v_{k'}) \). By Lemmas 80 and 81, the following occurs:

1. p reads \( R_{wp} = (\text{commit}, v_k) \) in line 14 of \( R_p(v_k) \), or
2. p reads \( R_{wp} = (\text{prepare}, v_k, v_{k+1}) \) in line 17 of \( R_p(v_k) \) before the following occurs:
   - q reads \( R_{wq} = (\text{commit}, v_{k'}) \) in line 20 of \( R_q(v_{k'}) \), or
   - q reads \( R_{wq} = (\text{prepare}, v_{k'-1}, v_{k'}) \) in line 22 of \( R_q(v_{k'}) \), or
   - q reads \( R_{wq} = (\text{prepare}, v_{k'}, v_{k'+1}) \) in line 22 of \( R_q(v_{k'}) \).

So there are six possible cases:

1. p reads \( R_{wp} = (\text{commit}, v_k) \) in \( R_p(v_k) \) before q reads \( R_{wq} = (\text{commit}, v_{k'}) \) in \( R_q(v_{k'}) \). By Observation 94, \( k \leq k' \).
2. p reads \( R_{wp} = (\text{commit}, v_k) \) in \( R_p(v_k) \) before q reads \( R_{wq} = (\text{prepare}, v_{k'-1}, v_{k'}) \) in \( R_q(v_{k'}) \). By Observation 94, \( k \leq k' \).
3. p reads \( R_{wp} = (\text{commit}, v_k) \) in \( R_p(v_k) \) before q reads \( R_{wq} = (\text{prepare}, v_{k'}, v_{k'+1}) \) in \( R_q(v_{k'}) \). By Observation 94, \( k \leq k' + 1 \).
   - k = \( k' + 1 \). Then \( k \leq k' \).
   - k = \( k' + 1 \). We now show that this case is impossible. Since \( k = k' + 1 \), q reads \( R_{wq} = (\text{prepare}, v_{k'-1}, v_{k'}) \) before \( R_{wq} = (\text{commit}, v_k) \) in \( R_q(v_{k-1}) \). By Observation 92, \( k \leq k' \).
4. p reads \( R_{wp} = (\text{prepare}, v_{k}, v_{k+1}) \) in \( R_p(v_k) \) before q reads \( R_{wq} = (\text{commit}, v_k) \) in \( R_q(v_{k'}) \). By Observation 94, \( (k+1) - 1 \leq k' \). So \( k \leq k' \).
5. p reads \( R_{wp} = (\text{prepare}, v_{k}, v_{k+1}) \) in \( R_p(v_k) \) before q reads \( R_{wq} = (\text{prepare}, v_{k'-1}, v_{k'}) \) in \( R_q(v_{k'}) \). By Observation 94, \( k \leq k' \).
6. p reads \( R_{wp} = (\text{prepare}, v_{k}, v_{k+1}) \) in \( R_p(v_k) \) before q reads \( R_{wq} = (\text{prepare}, v_{k'}, v_{k'+1}) \) in \( R_q(v_{k'}) \). By Observation 94, \( k + 1 \leq k' + 1 \). So \( k \leq k' \).

Now we prove that there is no “new-old” inversion for a read by q that precedes a read by p.

Lemma 96. If \( R_q(v_k) \) and \( R_p(v_{k'}) \) are read operations by non-malicious processes q and p respectively, and \( R_q(v_k) \) precedes \( R_p(v_{k'}) \), then \( k \leq k' \).

Proof. Suppose processes q and p are not malicious. Let \( R_q(v_k) \) and \( R_p(v_{k'}) \) be two read operations by q and p respectively, such that \( R_q(v_k) \) precedes \( R_p(v_{k'}) \). By Lemmas 80 and 81, the following occurs:

1. q reads \( R_{wq} = (\text{commit}, v_k) \) in line 20 of \( R_q(v_k) \), or
2. q reads \( R_{wq} = (\text{prepare}, v_{k-1}, v_k) \) in line 22 of \( R_q(v_k) \), or
3. q reads \( R_{wq} = (\text{prepare}, v_k, v_{k+1}) \) in line 22 of \( R_q(v_k) \) before the following occurs:
   - p reads \( R_{wp} = (\text{commit}, v_{k'}) \) in line 14 of \( R_p(v_{k'}) \), or
   - p reads \( R_{wp} = (\text{prepare}, v_{k'}, v_{k'+1}) \) in line 17 of \( R_p(v_{k'}) \).

So there are six possible cases:
On implementing SWMR registers from SWSR registers in systems with Byzantine failures

1. $q$ reads $R_{uq} = (\text{commit}, v_k)$ in $R_q(v_k)$ before $p$ reads $R_{wp} = (\text{commit}, v_{k'})$ in $R_p(v_{k'})$.

By Observation 94[5], $k \leq k'$.

2. $q$ reads $R_{uq} = (\text{commit}, v_k)$ in $R_q(v_k)$ before $p$ reads $R_{wp} = (\text{prepare}, v_{k'}, v_{k'+1})$ in $R_p(v_{k'})$.

By Observation 94[7], $k + 1 \leq k' + 1$. So $k \leq k'$.

3. $q$ reads $R_{uq} = (\text{prepare}, v_{k-1}, v_k)$ in $R_q(v_k)$ before $p$ reads $R_{wp} = (\text{commit}, v_{k'})$ in $R_p(v_{k'})$.

By Observation 94[6], $k \leq k'$.

4. $q$ reads $R_{uq} = (\text{prepare}, v_{k-1}, v_k)$ in $R_q(v_k)$ before $p$ reads $R_{wp} = (\text{prepare}, v_{k'}, v_{k'+1})$ in $R_p(v_{k'})$.

By Observation 94[8], $k \leq k' + 1$.

i. $k = k' + 1$. We now show that this case is impossible. Since $k = k' + 1$, $p$ reads $R_{wp} = (\text{prepare}, v_{k-1}, v_k)$ in $R_p(v_{k-1})$, and $R_p(v_{k-1})$ returns in line 18.

Note that $q$ reads $R_{pq}$ in line 23 of $R_q(v_k)$.

> Claim 97. Process $p$ writes $v_{\ell}$ into $R_{pq}$ with some $\ell \geq k$ before $q$ reads $R_{pq}$ in $R_q(v_k)$.

Proof. Since $q$ reads $R_{uq} = (\text{prepare}, v_{k-1}, v_k)$ in line 22 of $R_q(v_k)$, $R_q(v_k)$ returns in line 25 or 27. So there are two cases:

a. $R_q(v_k)$ returns in line 25. Then $q$ reads $R_{pq} = v_{\ell}$ for some $\ell \geq k$ in line 23 of $R_q(v_k)$. Thus, since $k \geq 1$ and $R_{pq}$ is initialized to $v_0$, $p$ wrote $v_{\ell}$ into $R_{pq}$ before $q$ reads $R_{pq}$ in $R_q(v_k)$.

b. $R_q(v_k)$ returns in line 27. Then $q$ reads $\text{last_read} = v_{\ell'}$ for some $\ell' \geq k$ in line 26 of $R_q(v_k)$. Thus, since $k \geq 1$ and $\text{last_read}$ is initialized to $v_0$, $q$ wrote $v_{\ell'}$ into $\text{last_read}$ in line 24 of some $R_q(v_{\ell'})$ operation that precedes $R_q(v_k)$. So $q$ reads $R_{pq} = v_{\ell'}$ for some $\ell' \geq k$ in line 23 of this $R_q(v_{\ell'})$ operation that precedes $R_q(v_k)$. Thus $p$ wrote $v_{\ell'}$ into $R_{pq}$ before $q$ reads $R_{pq}$ in $R_q(v_k)$.

From the code of $R_p()$ it is clear that if $p$ writes $v_{\ell'}$ to $R_{pq}$ (this can occur only in line 13 of some $R_p(-)$ operation) then $p$ previously reads $R_{wp} = (\text{commit}, v_{\ell'})$ (in line 13 of that $R_p(-)$ operation). Thus, by Claim 97, $p$ reads $R_{wp} = (\text{commit}, v_{\ell'})$ with $\ell \geq k$ before $q$ reads $R_{pq}$ in $R_q(v_k)$. Since $R_q(v_k)$ precedes $R_p(v_{k-1})$, $p$ reads $R_{wp} = (\text{commit}, v_{\ell'})$ before $p$ reads $R_{wp} = (\text{prepare}, v_{k-1}, v_k)$ in $R_p(v_{k-1})$.

By Observation 94[9], $\ell < k$ — a contradiction.

5. $q$ reads $R_{uq} = (\text{prepare}, v_{k'}, v_{k'+1})$ in $R_q(v_k)$ before $p$ reads $R_{wp} = (\text{commit}, v_{k'})$ in $R_p(v_{k'})$.

By Observation 94[6], $k + 1 \leq k'$. So $k \leq k'$.

6. $q$ reads $R_{uq} = (\text{prepare}, v_{k'}, v_{k'+1})$ in $R_q(v_k)$ before $p$ reads $R_{wp} = (\text{prepare}, v_{k'}, v_{k'+1})$ in $R_p(v_{k'})$.

By Observation 94[5], $k + 1 \leq k' + 1$. So $k \leq k'$.

Finally, we show that there are no “new-old” inversions in the successive reads of $q$.

To do so, we first observe that the counters of the tuples in the variable $\text{last_read}$ of $q$ do not decrease. To see this, note that if $q$ writes $v_k$ in $\text{last_read}$ (this occurs in line 24 of $R_q(v_k)$) then $q$ previously read $R_{uq} = (\text{prepare}, -, v_k)$ (in line 22 of $R_q(v_k)$). So, by Observation 84[3], we have:

> Observation 98. Suppose $q$ is not malicious. If $q$ writes $v_k$ in $\text{last_read}$ before $q$ writes $v_{k'}$ in $\text{last_read}$, then $k \leq k'$.

> Observation 99. Suppose $q$ is not malicious. If $q$ writes $v_k$ in $\text{last_read}$ before $q$ reads $\text{last_read} = v_k$, then $k \leq k'$.

> Observation 100. Suppose $q$ is not malicious. If $q$ reads $\text{last_read} = v_k$ before $q$ reads $\text{last_read} = v_{k'}$, then $k \leq k'$.

> Lemma 101. If $R_q(v_k)$ and $R_q(v_{k'})$ are read operations by non-malicious process $q$, and $R_q(v_k)$ precedes $R_q(v_{k'})$, then $k \leq k'$. 
Proof. Suppose process $q$ is not malicious. Let $R_q(v_k)$ and $R_q(v_{k'})$ be read operations by $q$, such that $R_q(v_k)$ precedes $R_q(v_{k'})$. By Lemma 51, the following occurs:
1. $q$ reads $R_{wq} = \langle \text{COMMIT}, v_k \rangle$ in line 20 of $R_q(v_k)$, or
2. $q$ reads $R_{wq} = \langle \text{PREPARE}, v_{k-1}, v_k \rangle$ in line 22 of $R_q(v_k)$, or
3. $q$ reads $R_{wq} = \langle \text{PREPARE}, v_k, v_{k+1} \rangle$ in line 22 of $R_q(v_k)$

before the following occurs:
1. $q$ reads $R_{wq} = \langle \text{COMMIT}, v_{k'} \rangle$ in line 20 of $R_q(v_{k'})$, or
2. $q$ reads $R_{wq} = \langle \text{PREPARE}, v_{k'-1}, v_{k'} \rangle$ in line 22 of $R_q(v_{k'})$, or
3. $q$ reads $R_{wq} = \langle \text{PREPARE}, v_{k'}, v_{k'+1} \rangle$ in line 22 of $R_q(v_{k'})$.

So there are nine possible cases:
1. $q$ reads $R_{wq} = \langle \text{COMMIT}, v_k \rangle$ in $R_q(v_k)$ before $q$ reads $R_{wq} = \langle \text{COMMIT}, v_{k'} \rangle$ in $R_q(v_{k'})$. By Observation 51, $k \leq k'$.
2. $q$ reads $R_{wq} = \langle \text{COMMIT}, v_k \rangle$ in $R_q(v_k)$ before $q$ reads $R_{wq} = \langle \text{PREPARE}, v_{k-1}, v_k \rangle$ in $R_q(v_k)$. By Observation 51, $k < k'$. So $k \leq k'$.
3. $q$ reads $R_{wq} = \langle \text{COMMIT}, v_{k'} \rangle$ in $R_q(v_{k'})$ before $q$ reads $R_{wq} = \langle \text{PREPARE}, v_k, v_{k'+1} \rangle$ in $R_q(v_k)$. By Observation 51, $k < k' + 1$. So $k \leq k'$.
4. $q$ reads $R_{wq} = \langle \text{PREPARE}, v_{k-1}, v_k \rangle$ in $R_q(v_k)$ before $q$ reads $R_{wq} = \langle \text{COMMIT}, v_{k'} \rangle$ in $R_q(v_{k'})$. By Observation 51, $k \leq k'$.
5. $q$ reads $R_{wq} = \langle \text{PREPARE}, v_{k-1}, v_{k'} \rangle$ in $R_q(v_k)$ before $q$ reads $R_{wq} = \langle \text{PREPARE}, v_{k'-1}, v_{k'} \rangle$ in $R_q(v_{k'})$. By Observation 51, $k \leq k'$.
6. $q$ reads $R_{wq} = \langle \text{PREPARE}, v_{k-1}, v_{k'} \rangle$ in $R_q(v_k)$ before $q$ reads $R_{wq} = \langle \text{PREPARE}, v_{k'}, v_{k'+1} \rangle$ in $R_q(v_{k'})$. By Observation 51, $k < k' + 1$.
   i. $k < k' + 1$. Then $k \leq k'$.
   ii. $k = k' + 1$. We now show that this case is impossible. Since $k = k' + 1$, $q$ reads $R_q = \langle \text{PREPARE}, v_{k-1}, v_{k'} \rangle$ in $R_q(v_{k-1})$, and $R_q(v_{k'})$ returns in line 26. Note that $q$ reads last_read in line 25 of $R_q(v_{k-1})$ before $R_q(v_{k-1})$ returns in line 29.

> Claim 102. Process $q$ reads last_read = $v_{\ell}$ for some $\ell \geq k$ in line 25 of $R_q(v_{k-1})$.

Proof. Since $q$ reads $R_{wq} = \langle \text{PREPARE}, v_{k-1}, v_k \rangle$ in $R_q(v_k)$, $R_q(v_k)$ returns in line 25 or 27. There are two cases:
   a. $R_q(v_k)$ returns in line 25 of $R_q(v_k)$. So $q$ writes $v_k$ in last_read in line 24 of $R_q(v_k)$. Since $R_q(v_k)$ precedes $R_q(v_{k-1})$, by Observations 99 when $q$ reads last_read in line 26 of $R_q(v_{k-1})$, $q$ reads last_read = $v_{\ell}$ for some $\ell \geq k$.
   b. $R_q(v_k)$ returns in line 27 of $R_q(v_k)$. So $q$ reads last_read = $v_{\ell'}$ with some $\ell' \geq k$ in line 26 of $R_q(v_k)$. Since $R_q(v_k)$ precedes $R_q(v_{k-1})$, by Observations 100 when $q$ reads last_read in line 26 of $R_q(v_{k-1})$, $q$ reads last_read = $v_{\ell'}$ for some $\ell' \geq k$.

By Claim 102 and the code of $R_q()$, it is clear that $R_q(v_{k-1})$ returns $v_k$ in line 27 — a contradiction.
7. $q$ reads $R_{wq} = \langle \text{PREPARE}, v_k, v_{k+1} \rangle$ in $R_q(v_k)$ before $q$ reads $R_{wq} = \langle \text{COMMIT}, v_{k'} \rangle$ in $R_q(v_{k'})$. By Observation 51, $k < k'$. So $k \leq k'$.
8. $q$ reads $R_{wq} = \langle \text{PREPARE}, v_k, v_{k+1} \rangle$ in $R_q(v_k)$ before $q$ reads $R_{wq} = \langle \text{PREPARE}, v_{k'}, v_k \rangle$ in $R_q(v_{k'})$. By Observation 51, $k < k'$. So $k \leq k'$.
9. $q$ reads $R_{wq} = \langle \text{PREPARE}, v_k, v_{k+1} \rangle$ in $R_q(v_k)$ before $q$ reads $R_{wq} = \langle \text{PREPARE}, v_{k'}, v_{k'+1} \rangle$ in $R_q(v_{k'})$. By Observation 51, $k < k'$. So $k \leq k'$.

We now prove that the writes and reads of the lower-level procedures $w()$, $R_p()$, and $R_q()$ satisfy Property 2 of Definition 6.

> Lemma 103. Let $R_r(v_k)$ and $R_r'(v_{k'})$ be any read operations by non-malicious processes $r$ and $r'$ in $\{p, q\}$. If $R_r(v_k)$ precedes $R_r'(v_{k'})$ then $k \leq k'$. 
Proof. Immediate from Lemmas 90, 95, 96, 101.

Finally, we prove that the write and read operations of the high-level procedures WRITE() and READ() satisfy Property 2 of Definition 6.

Lemma 104. [Property 2: No “new-old” inversion]
Let READ(u_k) and READ(u_{k'}) be any read operations by non-malicious processes in \{p, q\}. If READ(u_k) precedes READ(u_{k'}) then k \leq k'.

Proof. Let READ(u_k) and READ(u_{k'}) be any read operations by non-malicious processes r and r' in \{p, q\}. Suppose READ(u_k) precedes READ(u_{k'}). By Observation 87 in the READ(u_k) and READ(u_{k'}) operations, processes r and r' invoke and complete a R_r(v_k) and R_r'(v_{k'}) operation, respectively. Since READ(u_k) precedes READ(u_{k'}), R_r'(v_{k'}) precedes R_{r'}(v_k). By Lemma 103, k \leq k'.

By Lemmas 89 and 104, the WRITE(−) and READ(−) operations of the register implementation \mathcal{I}_2' satisfy the linearizability Properties 1 and 2 of Definition 6, so \mathcal{I}_2' is linearizable. By Observation 72, \mathcal{I}_2' is also bounded wait-free. Thus:

Theorem 49. The implementation \mathcal{I}_2' (given by Algorithm \ref{alg:1} in Appendix A) is a bounded wait-free linearizable implementation of a [1, 2]-register from atomic [1, 1]-registers.

Bounded Termination: impossibility proof

We now prove that in a system with n + 1 Byzantine processes, there is no linearizable implementation of a [1, n]-register from atomic [1, n − 1]-registers that satisfies the Bounded Termination property even if we assume that only the readers can be faulty, and at most one of them can fail. More precisely:

Theorem 70. For all n ≥ 3, in a system with n + 1 processes that are subject to Byzantine failures, there is no linearizable implementation of a [1, n]-register from atomic [1, n − 1]-registers that satisfies the Bounded Termination property, even if we assume that the writer of the implemented [1, n]-register is correct and at most one reader can be malicious.

Proof. Let n ≥ 3. Suppose, for contradiction, that there is an implementation \mathcal{I}_B of a [1, n]-register \mathcal{R} from atomic [1, n − 1]-registers that is linearizable (i.e., it satisfies the Register Linearizability property) and satisfies the Bounded Termination property in a system where the writer w of \mathcal{R} is correct and at most one of the n readers of \mathcal{R} can be malicious.

We now construct a sequence of runs of \mathcal{I}_B that leads to a contradiction. In all these runs, the initial value of the implemented \mathcal{R} is 0, the writer w invokes only one operation into \mathcal{R}, namely a write of 1, and each reader reads \mathcal{R} at most once (i.e., \mathcal{R} is only a “one-shot” binary register).

In all these runs: (a) the writer is correct and (b) there is at most one malicious reader (the other n − 1 readers are correct). Thus, these runs of \mathcal{I}_B must satisfy the linearizability Properties 1 and 2 of Register Linearizability (Definition 6), and Bounded Termination (Definition 59), i.e., every correct reader must complete any read operation that it invokes in a bounded number of steps.

Definition 105. Let s be any step that the writer w takes when executing the implementation \mathcal{I}_B of \mathcal{R}. Step s is invisible to a reader p if s is either a local step of w, or the reading or the writing of an atomic [1, n − 1]-register that is not readable by p.

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

Observation 106. Let s be any step that the writer w takes when executing the implementation \mathcal{I}_B of \mathcal{R}. Step s is invisible to at least one of the n readers.
Let $A_m$ be the following run of $I_B$ (see Figure 12):

- The writer $w$ and all the readers are correct.
- The writer $w$ invokes an operation to write 1 on $R$. By the Bounded Termination property of $I_B$, $w$ completes this operation.

During this write operation, $w$ takes a sequence of steps $s^1, \ldots, s^m$ such that each $s^i$ is either a local step, or the reading or the writing of an atomic $[1, n - 1]$-register ($s^0$ is the invocation step of the write operation, and $s^m$ is the response step of this operation).

Let $t_w^i$ be the time when step $s^i$ occurs.
- After taking the step $s^m$ at time $t_w^m$, the writer $w$ stops taking steps (it has completed its write operation on $R$).
On implementing SWMR registers from SWSR registers in systems with Byzantine failures

Let $q$ be a reader such that step $s^m$ is invisible to $q$ (by Observation 106, this reader exists).

After the time $t^m_{w}$, correct reader $q$ invokes a read operation on $R$. By the Bounded Termination property of $\mathcal{I}_B$, $q$ completes its read operation. By the linearizability properties of $\mathcal{I}_B$, this read operation on $R$ returns $1$.

All the other readers take no steps.

Definition 107. For every $k$, $1 \leq k \leq m$, a run of $\mathcal{I}_B$ has property $P_k$ if the following holds:

1. Up to and including time $t^k_{w}$, all processes behave exactly as in $A_m$, that is:
   - $w$ takes steps $s^0, s^1, \ldots, s^k$
   - All the readers take no steps.
2. After taking the step $s^k$ at time $t^k_{w}$, the correct writer $w$ behaves as follows:
   - If $k = m$, $w$ stops taking steps: it has completed its write operation on $R$.
   - If $k < m$, $w$ temporarily stops taking steps.
3. There is a reader $x$ that is correct such that step $s^k$ is invisible to $x$. After time $t^k_{w}$, reader $x$ starts and completes a read operation on $R$ that returns $1$.
4. There is a reader $y \neq x$ that may be correct or malicious. After time $t^k_{w}$, reader $y$ may or may not take steps.
5. There is a set $Z$ of $n-2$ distinct readers other than $x$ and $y$ that are correct and take no steps.
6. If $k < m$, after the reader $x$ reads 1 from $R$, the correct writer $w$ resumes taking steps and completes its write operation on $R$.

Note that since $n \geq 3$, the set $Z$ contains at least one reader. Furthermore, all the readers that take steps do so after time $t^k_{w}$.

A run of $\mathcal{I}_B$ with property $P_k$ is shown in Figure 13. In this figure and all the subsequent ones, correct readers are in black font, while the reader that may be malicious is colored red (this reader may have taken some steps after time $t^k_{w}$, but these are not shown in the figure). The “$x$” on top of a step $s^i$ means that $s^i$ is invisible to the reader $x$.

Note that the run $A_m$ of $\mathcal{I}_B$ satisfies property $P_m$: the reader denoted $x$ in property $P_m$ is the reader $q$ of run $A_m$, the reader $y$ of $P_m$ is an arbitrary reader other than $q$ in $A_m$, and the set $Z$ of $P_m$ is the set of the remaining $n-2$ readers in $A_m$. So we have:

Observation 108. Run $A_m$ of $\mathcal{I}_B$ has property $P_m$.

Claim 109. For every $k$, $1 \leq k \leq m$, there is a run of $\mathcal{I}_B$ that has property $P_k$.

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

Base Case: $k = m$. This follows directly from Observation 108.

Induction Step: Let $k$ be such that $1 < k \leq m$.

- Run $A_k$. Suppose there is a run $A_k$ of $\mathcal{I}_B$ that has property $P_k$ (this is the induction hypothesis). We now show that there is a run $A_{k-1}$ of $\mathcal{I}_B$ that has property $P_{k-1}$.

Since run $A_k$ of $\mathcal{I}_B$ satisfies $P_k$, the following holds in $A_k$ (see Figure 14):

- Up to and including time $t^k_{w}$, all processes behave exactly as in $A_m$.
- After taking the step $s^k$ at time $t^k_{w}$, the correct writer $w$ behaves as follows:
  - If $k = m$, $w$ stops taking steps: it has completed its write operation on $R$.
  - If $k < m$, $w$ temporarily stops taking steps.
- There is a reader $q$ that is correct such that step $s^k$ is invisible to $q$. After time $t^k_{w}$, reader $q$ starts and completes a read operation on $R$ that returns $1$.
- There is a reader $p \neq q$ that may be correct or malicious. After time $t^k_{w}$, reader $p$ may or may not take steps.\footnote{These steps are not shown in Figure 14}
There is a set $Z$ of $n-2$ distinct readers other than $p$ and $q$ that are correct and take no steps.

If $k < m$, after $q$ reads 1 from $R$, the correct writer $w$ resumes taking steps and completes its write operation on $R$.

Run $B_{k-1}$. From the run $A_k$ of $I_B$ we construct the following run $B_{k-1}$ of $I_B$ (Figure 15).

Intuitively, $B_{k-1}$ is the same as $A_k$ except that: (a) after taking step $s_{k-1}$ at time $t_{w}^{k-1}$, the writer $w$ temporarily stops taking steps, and (b) $w$ resumes taking steps only after the reader $q$ completes its read of 1. This run is possible because even though $p$ may have “noticed” that $w$ “pauses” after taking step $s_{k-1}$, $p$ may be malicious (all the other readers are correct in this run), and $p$ behaves exactly as in $A_k$, and (2) $q$ cannot distinguish between $A_k$ and $B_{k-1}$ because step $s_{k}$ is invisible to $q$, and $p$ and all the readers in $Z$ behave as in $A_k$; so $q$ behaves as in $A_k$, and in particular $q$ reads 1 in $B_{k-1}$ as in $A_k$. After $q$ reads 1, $w$ completes its write operation on $R$.

More precisely in $B_{k-1}$:

- All processes behave exactly as in $A_k$ up to and including time $t_{w}^{k-1}$.
- After taking step $s_{k-1}$ at time $t_{w}^{k-1}$, $w$ temporarily stops taking steps.
- All the readers in $Z$ are correct and take no steps, exactly as in $A_k$.
- $p$ behaves exactly as in $A_k$. This is possible because even though $p$ may have “noticed” that $w$ temporarily stops taking steps after step $s_{k-1}$, $p$ may be malicious (all the other readers are correct in this run).
- $q$ behaves exactly as in $A_k$. In particular, after time $t_{w}^{k}$, $q$ starts and completes a read operation on $R$ that returns 1. This is possible because $q$ cannot distinguish between $A_k$ and $B_{k-1}$: $s_{k}$ is invisible to $q$, and $p$ and all the readers in $Z$ behave exactly as in $A_k$.
- After $q$ reads 1 from $R$, the correct writer $w$ resumes taking steps and completes its write operation on $R$.

Note that in $B_{k-1}$ all processes behave exactly as in $A_m$ up to and including time $t_{w}^{k-1}$.

There are two cases:

**Case 1**: $s_{k-1}$ is invisible to $q$. Then $B_{k-1}$ is a run of $I_B$ that has the property $P_{k-1}$, as we wanted to show.

**Case 2**: $s_{k-1}$ is visible to $q$. Then, by Observation 106, $s_{k-1}$ is invisible to $p$ or to some $r' \in Z$.

- Run $C'_{k-1}$. Let $r$ be any reader in $Z$. From the run $B_{k-1}$ of $I_B$ we construct the following run $C'_{k-1}$ of $I_B$ (Figure 16). $C'_{k-1}$ is the same as $B_{k-1}$ up to the time when $q$ completes its read operation on $R$. After the correct reader $q$ reads 1, malicious process $p$ wipes out any trace of the write steps that it may have taken so far, and then correct reader $r \in Z$ starts a read operation on $R$. By the Bounded Termination property of $I_B$, this read operation by $r$ must complete (without waiting for the correct writer $w$ to complete its write operation). Since $q$ previously read 1, by the linearizability of $I_B$, $r$ also reads 1. After $r$ reads 1, $w$ completes its write operation on $R$.

More precisely in $C'_{k-1}$:

- All processes behave exactly as in $B_{k-1}$ up to and including the time when $q$ completes its read operation on $R$.
- All the readers in $Z - \{r\}$ are correct and take no steps.
- After the correct reader $q$ completes its read operation on $R$:
  - $q$ takes no steps.
  - $p$ resets all the atomic registers that it can write to their initial values. Process $p$ can do so because it may be malicious (all the other readers are correct in this run).
  - Let $t'_p$ be the time when $p$ completes all the register resettings.
  - Correct reader $r$ starts a read operation on $R$ after time $t'_p$. It takes no steps.

---

13 Even though $r$ “knows” that $w$ is correct and so $w$ will eventually take all the steps necessary to complete its write operation, $r$ cannot wait for them: this would violate the Bounded Termination property of $I_B$.

14 If $n = 3$, then the set $Z - \{r\}$ is empty.
On implementing SWMR registers from SWSR registers in systems with Byzantine failures

before this read. By the Bounded Termination property of \( I_B \), \( r \) completes its read operation (without waiting for correct \( w \) to resume taking its steps). Since \( w \) is correct, and the read operation by correct \( q \) precedes the read operation by \( r \) and returns 1, by the linearity of \( I_B \), the read operation by correct reader \( r \) also returns 1.

After \( r \) reads 1 from \( R \), the correct writer \( w \) resumes taking steps and completes its write operation on \( R \).

Note that in \( C_{k-1}^r \) all processes behave exactly as in \( A_m \) up to and including time \( t_w^{k-1} \).

- **Run \( D_{k-1}^r \).** We can now construct the following run \( D_{k-1}^r \) of \( I_B \) (Figure 17). Intuitively, we obtain \( D_{k-1}^r \) from \( C_{k-1}^r \) by removing all the steps of \( p \). So reader \( p \) (which was malicious in \( C_{k-1}^r \)) is now a correct process that takes no steps. Despite the removal of \( p \)'s steps, \( q \) behaves exactly as in \( C_{k-1}^r \) because \( q \) (which was correct in \( C_{k-1}^r \)) may now be malicious.

Up to and including time \( t_w^{k-1} \), the writer \( w \) also behaves exactly as in \( C_{k-1}^r \) because it cannot see the removal of \( p \)'s steps: they all occur after time \( t_w^{k-1} \). Correct reader \( r \) behaves exactly as in \( C_{k-1}^r \) because it also cannot see the removal of \( p \)'s steps: in both \( C_{k-1}^r \) and \( D_{k-1}^r \), \( r \) does not “see” any steps of \( p \). So \( r \) reads 1 in \( D_{k-1}^r \) as in \( C_{k-1}^r \). After \( r \) reads 1, \( w \) completes its write operation on \( R \).

More precisely in \( D_{k-1}^r \):

- After taking step \( s^{k-1} \) at time \( t_w^{k-1} \), \( w \) temporarily stops taking steps, as in \( C_{k-1}^r \).
- All the readers in \( Z - \{ r \} \) are correct and take no steps, as in \( C_{k-1}^r \).
- \( p \) is correct and it takes no steps. So all the atomic registers that it can write retain their initial values.
- \( q \) behaves exactly as in \( C_{k-1}^r \). This is possible because even though \( q \) may have “noticed” the removal of \( p \)'s steps, \( q \) may be malicious (all the other readers are correct in this run).
- \( r \) behaves exactly as in \( C_{k-1}^r \). In particular, after time \( t_w^r \) reader \( r \) starts and completes a read operation on \( R \) that returns 1. This is possible because \( r \) cannot distinguish between \( C_{k-1}^r \) and \( D_{k-1}^r \): \( r \) cannot see the removal of \( p \)'s steps, and \( q \) and all the readers in \( Z - \{ r \} \) behave exactly as in \( C_{k-1}^r \).

After \( r \) reads 1 from \( R \), the correct writer \( w \) resumes taking steps and completes its write operation on \( R \).

Note that in \( D_{k-1}^r \) all processes behave exactly as in \( A_m \) up to and including time \( t_w^{k-1} \).

If \( s^{k-1} \) is invisible to reader \( r \), it is clear that the run \( D_{k-1}^r \) of \( I_B \) has property \( P_{k-1} \).

Recall that (1) the reader \( r \) above is an arbitrary reader 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 run \( D_{k-1}^r \) of \( I_B \) has property \( P_{k-1} \), as we wanted to show.

- **Subcase 2b:** \( s^{k-1} \) is invisible to \( p \).

- **Run \( E_{k-1}^r \).** From the run \( D_{k-1}^r \) of \( I_B \) we construct the following run \( E_{k-1}^r \) of \( I_B \) (Figure 18). \( E_{k-1}^r \) is the same as \( D_{k-1}^r \) up to the time when \( r \) completes its read operation on \( R \). After \( r \) reads 1, malicious process \( q \) wipes out any trace of the write steps that it may have taken so far, and then correct reader \( p \) starts a read operation on \( R \). By the Bounded Termination property of \( I_B \), this read operation by \( p \) must complete (without waiting for the correct writer \( w \) to complete its write operation). Since \( r \) previously read 1, by the linearity of \( I_B \), \( p \) also reads 1. After \( p \) reads 1, \( w \) completes its write operation on \( R \).

More precisely in \( E_{k-1}^r \):

- All processes behave exactly as in \( D_{k-1}^r \) up to and including the time when \( r \) completes its read operation on \( R \).
- All the readers in \( Z - \{ r \} \) are correct and take no steps, as in \( D_{k-1}^r \).
- After the correct reader \( r \) completes its read operation on \( R \):
  - \( r \) takes no steps.
  - \( q \) resets all the atomic registers that it can write to their initial values. Process \( q \) can do so because it may be malicious (all the other readers are correct in this run).
Let \( t^r_q \) be the time when \( q \) completes all the register resettings.

- Correct reader \( p \) starts a read operation on \( R \) after time \( t^r_q \). It takes no steps before this read. By the Bounded Termination property of \( \mathcal{I}_B \), \( p \) completes its read operation (without waiting for correct \( w \) to resume taking its steps). Since \( w \) is correct, and the read operation by correct \( r \) precedes the read operation by \( p \) and returns 1, by the linearizability of \( \mathcal{I}_B \), the read operation by correct reader \( p \) also returns 1.

- After \( p \) reads 1 from \( R \), the correct writer \( w \) resumes taking steps and completes its write operation on \( R \).

Note that in \( E^r_{k-1} \) all processes behave exactly as in \( A_m \) up to and including time \( t^w_{k-1} \).

- \( \textbf{Run } F^r_{k-1} \). Finally, we construct the run \( F^r_{k-1} \) of \( \mathcal{I}_B \) by removing all the steps of \( q \) from \( E^r_{k-1} \) (see Figure 10). So \( q \) (which was malicious in \( E^r_{k-1} \)) is now a correct process that takes no steps. Despite the removal of \( q \)’s steps, \( r \) behaves exactly as in \( E^r_{k-1} \) because \( r \) (which was correct in \( E^r_{k-1} \)) may now be malicious. Up to and including time \( t^w_{k-1} \), the writer \( w \) also behaves exactly as in \( E^r_{k-1} \) because it cannot see the removal of \( q \)’s steps: they all occur after time \( t^w_{k-1} \). Correct reader \( p \) behaves exactly as in \( E^r_{k-1} \) because it also 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 \( p \) reads 1 in \( F^r_{k-1} \) as in \( E^r_{k-1} \). After \( p \) reads 1, \( w \) completes its write operation on \( R \).

More precisely in \( F^r_{k-1} \):

- After taking step \( s^{k-1} \) at time \( t^w_{k-1} \), \( w \) temporarily stops taking steps, as in \( E^r_{k-1} \).
- All the readers in \( Z - \{r\} \) are correct and take no steps, as in \( E^r_{k-1} \).
- \( q \) is correct and it takes no steps. So all the atomic registers that it can write retain their initial values.

- \( r \) behaves exactly as in \( E^r_{k-1} \). This is possible because even though \( r \) may have “noticed” the removal of \( q \)’s steps, \( r \) may be malicious (all the other readers are correct in this run).

- \( p \) behaves exactly as in \( E^r_{k-1} \). In particular, after time \( t^r_q \), reader \( p \) starts and completes a read operation on \( R \) that returns 1. This is possible because \( p \) cannot distinguish between \( E^r_{k-1} \) and \( F^r_{k-1} \); \( p \) cannot see the removal of \( q \)’s steps, and \( r \) and all the readers in \( Z - \{r\} \) behave exactly as in \( E^r_{k-1} \).

- After \( p \) reads 1 from \( R \), the correct writer \( w \) resumes taking steps and completes its write operation on \( R \).

Note that in \( F^r_{k-1} \) all processes behave exactly as in \( A_m \) up to and including time \( t^w_{k-1} \).

Since \( s^{k-1} \) is invisible to \( p \), it is clear that the run \( F^r_{k-1} \) of \( \mathcal{I}_B \) has property \( P_{k-1} \).

The above concludes the proof of the Induction Step of Claim 109; we proved that, in all possible cases, there is a run of \( \mathcal{I}_B \) that has property \( P_{k-1} \), as we needed to show. □

By the Claim 109 that we just proved, the implementation \( \mathcal{I}_B \) of \( R \) has a run \( A_1 \) with property \( P_1 \). By this property, the following holds in \( A_1 \) (see Figure 20):

- Up to and including time \( t^1_w \), all processes behave exactly as in \( A_m \).

- After taking the step \( s^1 \) at time \( t^1_w \), the correct writer \( w \) temporarily stops taking steps.

- There is a reader \( q \) that is correct such that step \( s^1 \) is invisible to \( q \). After time \( t^1_w \), reader \( q \) starts and completes a read operation on \( R \) that returns 1.

- There is a reader \( p \neq q \) that may be correct or malicious. After time \( t^1_w \), reader \( p \) may or may not take steps.

- There is a set \( Z \) of \( n-2 \) distinct readers other than \( p \) and \( q \) that are correct and take no steps.

- After \( q \) reads 1 from \( R \), the correct writer \( w \) resumes taking steps and completes its write operation on \( R \).

From the run \( A_1 \) of \( \mathcal{I}_B \) we construct the following run \( A_0 \) of \( \mathcal{I}_B \) (Figure 21). Intuitively, \( A_0 \) is the same as \( A_1 \) except that the correct writer \( w \) does not take any steps (i.e., \( w \) does not invoke a write 1 operation on \( R \)), but all the readers behave the same as in \( A_1 \) and so \( q \) still reads 1. This run of \( \mathcal{I}_B \) is possible because: (1) even though \( p \) may have “noticed” that \( w \) does not take any steps, \( p \) may be malicious (all the other readers are correct in this run), and \( p \) behaves exactly as in \( A_1 \), and (2) \( q \) cannot distinguish between \( A_1 \) and \( A_0 \) because \( s^1 \)
On implementing SWMR registers from SWSR registers in systems with Byzantine failures

is invisible to $q$, and $p$ and all the readers in $Z$ behave as in $A_1$. So $q$ reads 1 from $R$ in $A_0$ exactly as in $A_1$. Since the initial value of the implemented register $R$ is 0, run $A_0$ of the implementation $I_B$ of $R$ violates the linearizability of $I_B$ — a contradiction that concludes the proof of Theorem 70.

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 110.** For all $n \geq 3$, in a system with $n + 1$ processes that are subject to Byzantine failures, there is no linearizable implementation of a $[1,n]$-register that satisfies Bounded Termination, even under the assumption that:

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