# Atomic Register Abstractions for Byzantine-Prone Distributed Systems, Extended Version

Vincent Kowalski, Achour Mostéfaoui, Matthieu Perrin

# Atomic Register Abstractions for Byzantine-Prone Distributed Systems, Extended Version

Vincent Kowalski, Achour Mostéfaoui, and Matthieu Perrin

LS2N, Nantes Université, France

September 21, 2023

**Abstract**

The construction of the atomic register abstraction over crash-prone asynchronous message-passing systems has been extensively studied since the founding work of Attiya, Bar-Noy, and Dolev. It has been shown that $t < n/2$ (where $t$ is the maximal number of processes that may be faulty) is a necessary and sufficient requirement to build an atomic register. However, little attention has been paid to systems where faulty processes may exhibit a Byzantine behavior. This paper studies three definitions of linearizable single-writer multi-reader registers encountered in the state of the art: Read/Write registers whose `read` operations return the last written value, Read/Write-Increment registers whose `read` operations return both the last written value and the number of previously written values, and Read/Append registers whose `read` operations return the sequence of all previously written values. More specifically, it compares their computing power and the necessary and sufficient conditions on the maximum ratio $t/n$ which makes it possible to build reductions from one register to another. Namely, we prove that $t < n/3$ is necessary and sufficient to implement a Read/Write-Increment register from Read/Write registers whereas this bound is only $t < n/2$ for a reduction from a Read/Append register to Read/Write-Increment registers. Reduction algorithms meeting these bounds are also provided.

**Funding** This work was partially supported the French ANR project ByBloS (ANR-20-CE25-0002-01) and PriCLeSS (ANR-10-LABX-07-81).

## 1 Introduction

**Atomic register abstractions.** The register abstraction is the basis of the Turing machine's tape. It provides two basic operations: a write operation that allows defining a new value for the register and a read operation that returns its value. In concurrent architectures such as multi-core systems, the read/write semantics of a register is the cleanest and most easy-to-understand abstraction of shared memory and is extensively used in multi-threaded programs. In such a setting, a register that can be accessed concurrently by several processes represents a communication medium. In a message-passing system where processes may experience crash failures, Attiya, Bar-Noy, and Dolev [5] proposed the first emulation of a shared register (called ABD) for which it has been shown that $t < n/2$ (where $n$ is the total number of processes and $t$ is the maximal number of processes that may crash) is a necessary and sufficient requirement. Several algorithms have been proposed in order to enhance space or time efficiency.

According to which process is allowed to read or write a register and the significance of the returned value, several types of registers have been proposed, among which single-writer multi-reader (SWMR) and multi-writer multi-reader (MWMR) registers. Several levels of consistency can also be proposed: atomic, regular, and safe. A register is said to be atomic (or linearizable) if (a) each read or write operation appears as if it has been executed instantaneously at a single point of the timeline, between its start event and its end event, (b) no two operations appear at the same point of the timeline, and (c) a read returns the value written by the closest preceding write operation (or the initial value of the register if there is no preceding write) [17]. Reduction algorithms from one type of register to another (MWMR vs SWMR and atomic vs regular or safe) have been proposed (these registers are thus equivalent from a computability point of view).

**Byzantine-prone distributed systems.** The implementation of shared registers has been first studied in the crash failure model and then extended to Byzantine failures. A Byzantine process is a process that may deviate from its specification [22]. Byzantine faults gained interest since the work on Byzantine Fault Tolerance (BFT) [9], an implementation of a replicated state machine over a set of servers [23]. More recently, Blockchains made their appearance and a link was quickly made with the BFT approach [2]. Blockchains can be seen as eventually consistent implementations of a ledger data structure that consider in addition the semantic chaining between the different blocks and cryptography is used as a tool. The ledger itself can be seen as a particular register. In this paper, we propose to study register abstractions, the basic data structure, in distributed systems prone to Byzantine faults. Several works have addressed the design of a distributed shared storage in the client/server model prone to Byzantine failures [1, 10, 20]. A set of server processes implements a shared storage abstraction accessed by client processes. The different processes are, thus, separated into two classes and the system is not symmetric. While some servers can be Byzantine, most papers restrict the type of failure allowed to clients. [1] considers clients that can only crash, and [6] considers that clients can be Byzantine but a bounded number of times. On the other hand, [19] considered the use of signatures, and [13, 14] explored the conditions under which one can have fast reads (one-way messages) when servers never communicate with each other. Finally, [3] considers that readers can be Byzantine but not the writer.

In the context of asynchronous message-passing systems where at most $t$ processes out of the $n$ processes of the system can exhibit a Byzantine behavior, only the implementation of atomic SWMR (single-writer multi-reader) registers has been considered due to the fact that a Byzantine process can corrupt any register it can write. As Cohen and Keidar phrase it "In practice, multi-writer multi-reader (MWMR) registers are useless in a Byzantine environment as an adversary that controls the scheduler can prevent any communication between correct processes." [12]. Differently, the values written to a SWMR (single-writer multi-reader) register associated with a non-Byzantine process cannot be corrupted by a Byzantine process. As a result, [16] and [21] considered implementing an array of $n$ SWMR registers, one per process. If a register is associated with some process $p$, $p$ is the only process that can write it, while all processes can read it.

**Defining Byzantine-tolerant registers.** Following the definition of Byzantine linearizability proposed in [12], the most natural way of specifying a register in this context would be to say that if the writer is not Byzantine, the register respects the classical specification of an atomic register, otherwise, a read operation can return any value. This register will be called the *Read/Write register*.

In order to give some significance to the registers associated with Byzantine processes, [16] and [21] give a different specification of a SWMR register. Indeed, while the specification of a register associated with a correct (non-Byzantine) process is identical to that of a classical atomic register, the specification of a register associated with a Byzantine process can be declined in different ways, depending on how the behavior of the Byzantine writer is perceived. In [16], a register keeps track of the sequence of all the values written by the writer (be it Byzantine or not) and this sequence is seen in the same way by all non-Byzantine processes. This register will be called in this paper *Read/Append register*. If the writer is correct, the sequence will correspond to the chronological sequence of values it wrote. If the writer is Byzantine, this sequence depends on the behavior of the writer, in the extreme case, the history will be reduced to the initial state (an empty sequence). In other words, a single-writer Read/Append register can be seen as a single-writer atomic ledger.

Differently, in [21], a register keeps only the last written value together with its sequence number. This will be called the *Read/Write-Increment register*. In the same way, if the writer is Byzantine, the register may be in any state as soon as it is perceived consistently by non-Byzantine processes: two correct processes that read the same register and return the same sequence number will necessarily get the same value even if the writer is Byzantine. At the extreme, the register's state will be stuck to the initial value with sequence number 0.

**Why only studying SWMR registers?** In addition to the lack of sense of multi-writer Read/Write registers discussed above, this paper only considers single-writer registers because we are only interested in memory abstractions that are equivalent to the atomic register in crash-prone systems. Indeed, when only crashes are considered, any of these register specifications can be implemented on top of any other, because writes cannot be concurrent when there is a single writing process. On the other hand, this is not the case for the multi-writer versions of the three register specifications: even in crash-prone systems, a ledger has the synchronization power of consensus and can implement a state machine; and it is easy to see that multi-writer Read/Write-

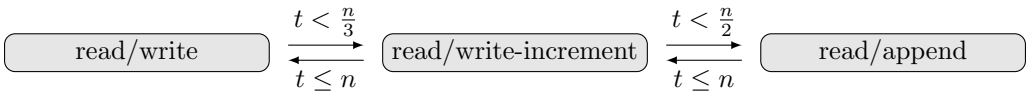



Figure 1: Conditions for implementing Byzantine-tolerant single-writer multi-reader registers

Increment registers have consensus number 2.

In crash-prone systems, basic registers are used as building blocks to construct safe synchronization algorithms (i.e. consensus, stack, queue, etc.) and the additional power is brought either by temporal properties, special hardware instructions, or by randomization. Studying and understanding the exact relationship between single-writer registers in the Byzantine framework would allow a similar approach. For example, using single-writer registers plus additional properties like randomization could help to implement Byzantine-tolerant ledgers. Although the relation of crash-prone atomic registers is well-studied, this is not the case for the Byzantine case.

Hence, this work sheds new light on the construction of Byzantine-prone asynchronous atomic registers. When one wants to implement these SWMR registers over an asynchronous Byzantine message-passing distributed system, $t < n/3$ is necessary and sufficient for all three variants [16]. This is intuitive since the three register specifications differ only in the way they deal with Byzantine writers: Moreover, it is known that in order to implement an atomic register on top of an asynchronous message-passing system prone to process crashes, readers have to write [4, 5]. So even if we only consider non-Byzantine writers, the values they write will be relayed by readers which may be Byzantine. Therefore, the correct readers must be able to distinguish the values relayed by Byzantine processes from others hence the ratio $t < n/3$.

**Contribution.** This paper investigates the relationship between the three register specifications stated above in the presence of Byzantine failures and studies under which condition one type of register can be built from another as shown in figure 1. Whereas the reduction from a Read/Write register to a Read/Write-Increment one and from a Read/Write-Increment register to a Read/Append register is straightforward by their respective definitions, it is important to note that a Read/Write-Increment register can be implemented from Read/Write registers only if $t < n/3$ and surprisingly, a Read/Append register can be reduced to a Read/Write-Increment register as soon as $t < n/2$. This shows that the sequence number mechanism in the R/WI registers is actually quite powerful, but does not close the gap with the R/A registers. This suggests that some aspects of the bad behavior of Byzantine processes are already captured by Read/Write-Increment registers and can benefit upper-layer constructions. The proposed bounds are tight, we prove on the one side that the proposed bounds are necessary, and we propose constructions that allow these reductions.

**Organization.** The remainder of this paper is organized as follows: Section 2 presents the considered distributed model. Then we investigate the relationships, on the one hand, between Read/Write registers and Read/Write-Increment registers (Section 3) and on the other hand, between Read/Write-Increment registers and Read/Append registers (Section 4). Finally, Section 5 concludes the paper.

# 2 Computing Model

We consider the classical Byzantine-prone asynchronous shared-memory model.

**Computing entities.** The system is made up of $n$ sequential processes, denoted $p_1, p_2, \ldots, p_n$. These processes are asynchronous in the sense that each process progresses at its own speed, which can be arbitrary and may vary along any execution, and remains always unknown to the other processes. Each process $p_i$ has access to its own identifier $i$ it can use in the code.

**Failure model.** A Byzantine process is a process that behaves arbitrarily [18, 22]: it may start in an arbitrary state, stop executing at any time (this behavior is called a crash), perform arbitrary state transitions, attempt to communicate arbitrary or different values to different processes, etc. Among the $n$ processes of the system, it is supposed that at most $t$ can exhibit a Byzantine behavior in any given execution. A Byzantine process is also called a *faulty* process and a process that commits no failure (i.e., a non-Byzantine process) is also called a *correct* process.

**Communication model.** The processes communicate by invoking operations on a collection of shared objects. Since efficiency is not a central issue of this paper, we consider that processes have access to as many shared objects as necessary. We consider only linearizable (atomic) $t$-resilient shared objects, as defined thereafter. $t$-resiliency is a classical liveness condition stating that all operations invoked by correct processes terminate provided there are at most $t$ faulty processes [7].

Following [12], Definition 1 defines Byzantine linearizability in terms of admitted distributed histories, depending on a given object defined by a sequential specification (in our case, the register specifications already discussed). A distributed history (or simply history when it is clear from context) is an abstraction of a distributed execution, composed of a sequence of atomic steps taken by the processes. The steps taken by correct processes can be either 1) the invocation and response events of the operations specified by a given algorithm, 2) executions of atomic operations on shared objects of the underlying model, or 3) local computations. We only accept well-formed histories, in which correct processes alternate invocation and response events. In other word, a correct process cannot return from an operation it has not yet invoked, and cannot invoke an operation before it has returned from its previous invoked operation.

More precisely, Byzantine linearizability is defined as an extension of linearizability [15], such that only the operations of correct processes must be correctly specified, whereas the histories of Byzantine processes can be re-interpreted into any local history, so that the full history respects the specification of the object. In particular, operations correctly performed by Byzantine processes may be either recognized, ignored or reinterpreted as a different correct operation by the correct processes. A consequence of this definition is that no process can alter the semantics of an operation invoked on a shared object. A Byzantine process can invoke or not an operation independently from its code. However, if Byzantine processes attempt to alter the internal implementation of the object, correct processes will agree on some sequence of operations that were invoked, and the effect of these operations will conform to the specification of the object. This notion of linearizability is close to the one defined in [11] for the special case of the ledger data structure.

**Definition 1 (Byzantine Linearizability)** *A history $H$ is* linearizable *with respect to an object $O$ if there exists a sequential history $H'$ (called a linearization of $H$) such that (1) after removing some pending operations from $H$ and completing others by adding matching responses, it contains the same invocations and responses as $H$, (2) if an operation $o$ returns before an operation $o'$ starts in $H$, then $o$ appears before $o'$ in $H'$, and (3) $H'$ satisfies $O$'s sequential specification.*

*A history $H$ is Byzantine linearizable with respect to an object $O$ if there exists a history $H'$ linearizable with respect to $O$, such that $H'|_{correct} = H|_{correct}$ (where $H|_{correct}$ denotes the history $H$ where only the operations done by correct processes are considered).*

*We say that an object is Byzantine linearizable, or simply linearizable if all of its executions are Byzantine linearizable.*

**The three register abstractions.** This paper considers three variations of the classical SWMR register, whose sequential specification is defined below and illustrated in Figure 2:

**Read/Write (R/W) registers** offer two operations: a `write` operation, only accessible to the writing process, that does not return any value, and a `read` operation accessible to all processes, that returns the last written value, or the initial value $\perp$ if no value was written.

**Read/Write-Increment (R/WI) registers** offer two operations: a `write-incr` operation, only accessible to the writing process, that does not return any value, and a `read` operation accessible to all processes, that returns a pair $x = \langle x.\mathsf{value}, x.\mathsf{count}\rangle$, such that $x.\mathsf{value}$ is the last written value (similar to R/W registers), and $x.\mathsf{count}$ is the number of times the operation `write-incr` was called by the writer. If no value was ever written, the `read` operation returns $\langle \perp, 0\rangle$.

Remark that the definition of Byzantine linearizability implies that, if two processes read the same count field, then they must read the same value field as well.

**Read/Append (R/A) registers** offer two operations: an `append` operation, only accessible to the writing process, that does not return any value, and a `read` operation accessible to all processes, that returns the ordered sequence of all written values on that register since the beginning of the execution.

The initial empty sequence is denoted by $\varepsilon$, and the concatenation of a sequence $l$ and a value $v$ is denoted by $l \oplus v$. Given a sequence $l$ and an index $s \in \mathbb{N}$, let us denote by $l[s-1]$

the $s^{\text{th}}$ value in $l$ (i.e. the first index is 0). Given two sequences $l$ and $l'$, let $l \subseteq l'$ denote the fact that $l$ is a prefix of $l'$.

Thanks to Byzantine linearizability, the sequences returned to different reads must be consistent, i.e. one must be a prefix of the other.

```
1 initial state is
2 |  value ← ⊥;
3 operation write (v) invoked by p_i is
4 |  value ← v;
5 operation read () invoked by any p_j is
6 |  return value;
```

(a) Read/write register REG[i]

```
1 initial state is
2 |  value ← ⊥;
3 |  count ← 0;
4 operation write-incr(v) invoked by p_i is
5 |  value ← v;
6 |  count ← count + 1;
7 operation read() invoked by any p_j is
8 |  return ⟨value, count⟩;
```

(b) Read/write-inc register REG[i]

```
1 initial state is
2 |  log ← ε.
3 operation append(v) invoked by p_i is
4 |  log ← log ⊕ v.
5 operation read() invoked by any p_j is
6 |  return log.
```

(c) Read/append register REG[i]

Figure 2: Sequential specification of the different registers

More precisely, we consider that processes have access to as many arrays of $n$ SWMR registers as necessary. The $i$-th entry of the array is associated with process $p_i$. This means that $p_i$ is the only process that can write a value in this register while other processes can only read it.

**Remark.** If a process is supposed to write only once in its register, a Byzantine process $p_w$ can write a first value that will be read by some processes $p_i$ and then write a second value before some other processes $p_j$ reads the register. If the Byzantine process uses a R/W register, $p_i$ and $p_j$ may read different values. If the considered register is a R/WI one, $p_j$ knows that the writing process is Byzantine because the value it reads is associated with a count $= 2$. Finally, if one uses a R/A register, all sequences read by correct processes will contain the very same first value even though the Byzantine process appended other values. It is clear that the different registers offer different information on the behavior of Byzantine processes.

**Notation.** Let $R$ denote a type of SWMR register through which processes can communicate. The acronym $\mathcal{BASM}_{n,t}[R]$ is used to denote the $n$-process asynchronous system where up to $t$ processes may exhibit Byzantine behavior and communication is through as many instances of $R$ as necessary. $\mathcal{BASM}_{n,t}[R, C]$ denotes $\mathcal{BASM}_{n,t}[R]$ enriched with the condition $C$ on $t$ and $n$.

**A characterization of Byzantine linearizability for Read/Append registers.** In order to simplify the proofs of subsequent algorithms, Proposition 2 defines four properties that characterize linearizable R/A registers. Clearly, these properties are verified by any linearizable R/A register.

**Proposition 2 (Linearizability for Read/Append registers)** *Let $H$ be a distributed history of a Read/Append register object that verifies the following properties.*

**Validity:** *If a* `read` *operation performed by a correct process returns* log*, and if the writing process is correct, then for all $s \in \{1, ..., |log|\}$,* $\log[s-1]$ *is the $s^{th}$ value written. (By indistinguishability with the scenario where the writer crashes before the end of the read, this implies that the $s^{th}$ write started before the read completed).*

**Read after write:** *if a read done by a correct process starts after the $s^{th}$ write of a correct process completes, then the read cannot return a sequence containing less than $s$ values.*

**Inclusion:** *let $r_i$ and $r_j$ be two* `read` *operations, done by correct processes, that return respectively $log_i$ and $log_j$. Then $log_i$ is a prefix of $log_j$, or $log_j$ is a prefix of $log_i$.*

**Read after read:** *let $r_i$ and $r_j$ be two* `read` *operations, done by correct processes, that return respectively $log_i$ and $log_j$. If $r_i$ completes before $r_j$ starts, then $log_i$ is a prefix of $log_j$.*

*Then $H$ is Byzantine linearizable with respect to the Read/Append register.*

**Proof 3** Proof when the writing process is correct. *Let us consider the history $H' = H|_{correct}$, i.e. $H'$ is $H$ in which the reads of Byzantine processes were removed. Clearly, $H'|_{correct} = H|_{correct}$. We will prove that $H'$ is linearizable.*

*For each operation $o$ of $H'$, we define the timestamp $ts(o)$ of $o$ as follows. If $o$ is the $s^{th}$ write operation, then $ts(o) = s$. If $o$ is a read operation that returns $log$, then $ts(o) = |log|$. We also define the binary relation $\rightarrow$ between operations as $o_1 \rightarrow o_2$ if either 1) $o_1$ returned before $o_2$ was started (denoted by $o_1 \rightarrow_1 o_2$), or 2) $ts(o_1) < ts(o_2)$ (denoted by $o_1 \rightarrow_2 o_2$), or 3) $o_1$ is a write, $o_2$ is a read, and $ts(o_1) = ts(o_2)$ (denoted by $o_1 \rightarrow_3 o_2$).*

*Let us prove that $\rightarrow$ is cycle-free. Indeed, suppose there is a cycle $o_1 \rightarrow o_2 \rightarrow \ldots \rightarrow o_k = o_1$. Since none of the cases contains reflexivity, the cycle contains at least two different operations. Moreover, let us notice that $o_1 \rightarrow o_2$ implies $ts(o_1) \leq ts(o_2)$: this is true by definition for $\rightarrow_2$ and $\rightarrow_3$, and this is implied by the Validity, Read after read and Read after Writes properties for $\rightarrow_1$. Hence, all operations in the cycle have the same timestamp, and they are not compared by $\rightarrow_2$. Since $\rightarrow_1$ itself is a partial order, it means there is a write operation $w$ and a read operation $r$ such that $w \rightarrow_3 r$. Consequently, there is a write operation $w'$ and a read operation $r'$ such that $r' \rightarrow w'$, which is only possible if $r' \rightarrow_1 w'$. In other words, a read returns a value that has not yet been written, which is prevented by the Validity property.*

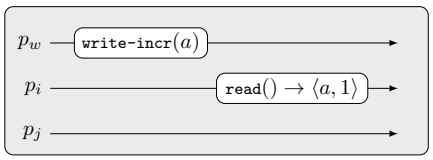
*Finally, the reflexive and transitive closure of $\rightarrow$ can be extended into a total order $\prec$ that respects real time thanks to $\rightarrow_1$, and that respects the sequential specification of the Read/Append register: by $\rightarrow_2$, the* `read` *operations are ordered by size of the returned sequences, hence by inclusion by the Inclusion property, and the sequences are composed of written values by the Validity property;* `append` *operations are ordered before the* `read` *operations that include their values in their return sequence by $\rightarrow_1$, and after the others by $\rightarrow_2$. Hence, $H'$ is linearizable, so $H$ is Byzantine linearizable, which concludes the proof when the writer is correct.*

Proof when the writing process is Byzantine. *Let us consider the history $H' = H|_{correct}$, i.e. $H'$ only contains the reads done by the correct processes in $H$. As is the case of a correct writer, we define the binary relation $\rightarrow$ between two read operations $o_1$ and $o_2$, that return respectively $log_1$ and $log_2$, as $o_1 \rightarrow o_2$ if either 1) $o_1$ was terminated before $o_2$ was started (denoted by $o_1 \rightarrow_1 o_2$), or 2) $log_1$ is a strict prefix of $log_2$ (denoted by $o_1 \rightarrow_2 o_2$). The Read after read property implies that $\rightarrow$ is cycle-free, so it can be extended into a total order $\prec$.*

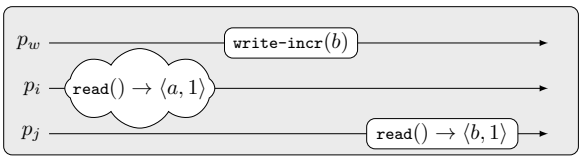
*Thanks to the Inclusion property and the definition of $\rightarrow_2$, for all operations $o_1$ and $o_2$ that return respectively $log_1$ and $log_2$, if $o_1 \prec o_2$, then $log_1$ is a prefix of $log_2$. Since $\prec$ contains $\rightarrow_2$, it is possible to build a linearizable extension $H''$ of $H'$ by adding write operations corresponding to the read values, in the order in which they are read, and just before the time where they were read first. Hence, $H''$ is linearizable and $H''|_{correct} = H|_{correct}$, so $H$ is Byzantine linearizable, which concludes the proof.*

# 3 From R/W registers to R/WI registers

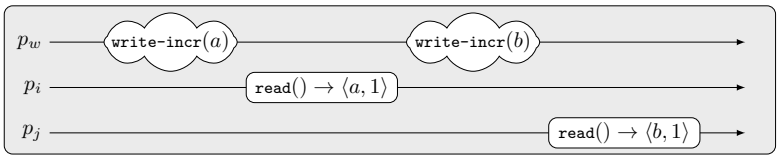
This section shows that the hypothesis $t < \frac{n}{3}$ is necessary, and sufficient, to implement a R/WI register on top of R/W registers. More precisely, Section 3.1 proves that $t < \frac{n}{3}$ is an upper bound on the number Byzantine tolerated by any such reduction algorithm, and then Section 3.2 presents an algorithm whose resilience is optimal.




(a) Situation $S_1$: $p_j$ is Byzantine, but does nothing to prevent $p_i$ from reading $\langle a, 1\rangle$




(b) Situation $S_2$: $p_i$ is Byzantine, but cannot prevent $p_j$ from reading $\langle b, 1\rangle$




(c) Situation $S_3$: $p_w$ is Byzantine, and can force $p_i$ and $p_j$ to read different values

Figure 3: Illustration of the scenarios from the proof of Theorem 4, with $n = 3$ and $t = 1$

## 3.1 An upper bound on resilience

This section proves that the maximal resilience of any implementation of a linearizable Read/Write-Increment register in a system where processes communicate only through Read/Write registers, is at most $t \geq \frac{n}{3}$.

**Theorem 4** *It is impossible to implement a linearizable Read/Write-Increment register in the model $\mathcal{BASM}_{n,t}[R/W]$, when $n \geq 3$ and $t \geq \frac{n}{3}$.*

**Proof 5** *Let us assume that there exists an algorithm, A, which implements linearizable R/WI registers using a collection of R/W registers, even when $t \geq \frac{n}{3}$. We are going to show (proof by contradiction) that there exists an execution allowed by A and that is not linearizable.*

*Let us consider a system made up of $n \geq 3$ processes, and let $t \geq \frac{n}{3}$. We can partition the set of processes into three non-empty sub-sets $W$, $I$ and $J$, whose size is at most $t$. Let us pick three processes $p_w \in W$ (the writing process), $p_i \in I$ and $p_j \in J$ (two reading processes), and let us consider three situations in which A is used to implement a R/WI register $x$ that can be written by $p_w$ only. These situations are represented in Figure 3 and described thereafter.*

**$S_1$:** *In the first scenario (Figure 3a), the processes in $J$ are Byzantine and do not take any step during the execution. All other processes are correct. Initially, Process $p_w$ writes $a$ in $x$. The* `write-incr` *terminates because $|J| \leq t$ processes are Byzantine. Then, Process $p_i$ reads $x$. Similarly, the* `read` *terminates, and returns $\langle a, 1\rangle$ because $x$ is linearizable.*

**$S_2$:** *In the second scenario (Figure 3b), only the processes of $I$ are Byzantine. At first, Byzantine processes are not quiet, but they simulate their behavior in $S_1$. At this point, all shared registers in which processes of $I$ can write are the same as after the reading of $\langle a, 1\rangle$ by $p_i$. In a second stage, Byzantine processes become quiet and $p_w$ writes $b$ in $x$. As previously, the* `write-incr` *must terminate because $|I| \leq t$ processes are Byzantine. In a third stage, $p_j$ reads $x$. The* `read` *terminates and returns $\langle b, 1\rangle$ because $x$ is linearizable and the correct process $p_w$ indeed only wrote one value, which was $b$.*

**$S_3$:** *In the last scenario (Figure 3c), processes of $W$ are Byzantine, and the processes of $J$ are initially slow. Initially, $p_w$ behaves correctly and writes $a$ in $x$, then Process $p_i$ reads $x$. So far, the situation is the same as $S_1$, so $p_i$ gets $\langle a, 1\rangle$ as the result of its read. For the rest of the executions, processes in $I$ become too slow to play any role.*

*Then, all processes in $W$ write, in their respective registers, the values that were contained in their registers in $S_2$ after $p_w$ wrote $b$, and keep simulating the behavior they had in $S_2$.*

*Finally, $p_j$ reads $x$. Notice that, at this point, $S_3$ and $S_2$ are indistinguishable to all processes in $J$: in both situations, all registers written by processes in $W$ contain the value resulting from a write of $b$, all registers written by processes in $I$ contain the value resulting from a write of $a$, all processes in $W$ respond accordingly to a write of $b$, and all processes in $I$ are quiet. Therefore, $p_j$ must return $\langle b, 1\rangle$.*

*Remark that, in $S_3$, two correct processes $p_i$ and $p_j$ have read respectively $\langle a, 1\rangle$ and $\langle b, 1\rangle$ in $x$. This violates linearizability, so A cannot exist.*

## 3.2 A resilience-optimal algorithm

Algorithm 1 presents an implementation of a R/A register in the model $\mathcal{BASM}_{n,t}[\text{R/W}, t < \frac{n}{3}]$. Since R/WI registers can be trivially implemented from R/A registers, this proves that R/WI registers can be implemented in $\mathcal{BASM}_{n,t}[\text{R/W}, t < \frac{n}{3}]$ as well. The writing process is denoted by $p_w$.

**Shared memory and local variables.** The $n$ processes share three variables called ENDORSE, APPROVE and CONFIRM, defined as follows.

- ENDORSE$[0...][1..n]$ is an infinite array of arrays of $n$ SWMR atomic R/W registers such that, for any $s \in \mathbb{N}$ and $i \in \{1, ..., n\}$, ENDORSE$[s][i]$ can only be written by $p_i$, is initialized to a value $\perp$ that cannot be appended to the R/A register and eventually contains $p_i$'s opinion on what the $s^{\text{th}}$ value appended is.
- Similarly, APPROVE$[0...][1..n]$ is an infinite array of arrays of $n$ SWMR atomic R/W registers, initialized to $\perp$ as well. A correct process $p_i$ only writes, in APPROVE$[s][i]$, a value that it has previously read in more than $\frac{n+t}{2}$ cells of ENDORSE$[s]$.
- CONFIRM$[1..n]$ is an array of $n$ SWMR atomic R/W registers such that for any $i \in \{1, ..., n\}$, CONFIRM$[i]$ can only be written by $p_i$ and contains sequences of appended values that have already been read by $p_i$ in more than $2t$ cells of APPROVE$[s]$.

Besides these three shared variables, each process $p_i$ maintains five local variables:

- $log_i$ is a sequence that represents the current state of the shared R/A register seen by $p_i$;
- $count_i$ is an integer that represents the number of appended values;
- $endorse_i[1..n]$ is an array of size $n$, that stores a local copy of ENDORSE$[|log_i|]$ by $p_i$;
- $approve_i[1..n]$ is an array of size $n$, that stores a local copy of APPROVE$[|log_i|]$ by $p_i$;
- $confirm_i[1..n]$ is an array of size $n$, that stores a local copy of CONFIRM$[|log_i|]$ by $p_i$.

**Notations.** Let $v$ be a value, $l$ a sequence of values, and $s \in \mathbb{N}$. We say that a process $p_i$ *endorses* $v$ as the $(s+1)^{\text{th}}$ value (or simply *endorses* $v$ when $s$ is immaterial) if $p_i$ writes $v$ in ENDORSE$[s][i]$. Similarly, we say that $p_i$ *approves* $v$ as the $(s+1)^{\text{th}}$ value (or simply *approves* $v$) if $p_i$ writes $v$ in APPROVE$[s][i]$, and that $p_i$ *confirms* $l$ if $p_i$ writes $l$ in CONFIRM$[i]$. We also say that $p_i$ *confirms* $v$ as the $(s+1)^{\text{th}}$ value (or simply *confirms* $v$) if there exists a sequence $l$ of size $s$ such that $p_i$ confirms $l \oplus v$. Finally, we say that $p_i$ *logs* $v$ as the $(s+1)^{\text{th}}$ value (or simply *logs* $v$) when $p_i$ executes $log_i \leftarrow log_i \oplus v$ in Line 18, with $|log_i| = s$.

**Description of the algorithm.** Algorithm 1 implements a confirmation mechanism similar to the protocol for reliable broadcast proposed by Bracha and Toueg in 1985 [8]. In order to write its $(s+1)^{\text{th}}$ value $v$, the writing process $p_w$ writes $v$ in ENDORSE$[s][w]$ (Line 2) and then waits until it has logged $v$ as its $(s+1)^{\text{th}}$ value, which will happen after enough correct processes have confirmed the write by calling the procedure `synch`() (which they do regularly thanks to Line 8).

When any process $p_i$ invokes `synch`(), it first updates its local copy of the shared variables CONFIRM, APPROVE and ENDORSE (Lines 9-11), and then it checks the five following conditions to help progress on an agreement on the written values.

- The first time $p_i$ reads a non-$\perp$ value in ENDORSE$[s][w]$, it endorses this value by writing it to ENDORSE$[s][i]$ (Line 12).
- If some value $v$ has been endorsed by more than $\frac{n+t}{2}$ processes, i.e. by a majority of the correct processes, $p_i$ approves this value by writing it to APPROVE$[s][i]$ (Line 14).

  Remark that two correct processes cannot approve different values. However, at this stage, it is still possible that some correct process approves a value, and other correct processes do not approve any value.
- If some value $v$ has been approved by at least $t+1$ different processes, i.e. by at least one correct process, $p_i$ trusts this correct process and approves the value as well (Line 15).

**operation** append($v$) *invoked by* $p_w$ **is**
1 $count_w \leftarrow |log_w|$;
2 ENDORSE$[count_w][w]$.write($v$);
3 **while** $|log_w| \leq count_w$ **do** synch();

**operation** read() *invoked by any* $p_i$ **is**
4 synch();
5 $count_i \leftarrow \max\{c \in \mathbb{N} : |\{j : |confirm_i[j]| \geq c\}| > t\}$;
6 **while** $|log_i| < count_i$ **do** synch();
7 **return** $log_i$;

**background task** $T()$ *repeatedly executed by all* $p_i$ **is**
8 synch();

**procedure** synch() *invoked by any* $p_i$ **is**
9 **for** $j$ **from** 1 **to** $n$ **do** $endorse_i[j] \leftarrow$ ENDORSE$[|log_i|][j]$.read();
10 **for** $j$ **from** 1 **to** $n$ **do** $approve_i[j] \leftarrow$ APPROVE$[|log_i|][j]$.read();
11 **for** $j$ **from** 1 **to** $n$ **do** $confirm_i[j] \leftarrow$ CONFIRM$[j]$.read();
12 **if** $endorse_i[i] = \perp \wedge \exists v \neq \perp : endorse_i[w] = v$ **then** ENDORSE$[|log_i|][i]$.write($v$);
13 **if** $approve_i[i] = \perp$ **then**
14 **if** $\exists v \neq \perp : |\{j : endorse_i[j] = v\}| > \frac{n+t}{2}$ **then** APPROVE$[|log_i|][i]$.write($v$);
15 **if** $\exists v \neq \perp : |\{j : approve_i[j] = v\}| > t$ **then** APPROVE$[|log_i|][i]$.write($v$);
16 **else if** $confirm_i[i] = log_i$ **then**
17 **if** $\exists v \neq \perp : |\{j : approve_i[j] = v\}| > 2t$ **then** CONFIRM$[i]$.write($log_i \oplus v$);
18 **else if** $\exists v \neq \perp : |\{j : confirm_i[j] \supseteq log_i \oplus v\}| > 2t$ **then** $log_i \leftarrow log_i \oplus v$;

**Algorithm 1:** Implementation of a R/A register in the model $\mathcal{BASM}_{n,t}[\text{R/W}, t < \frac{n}{3}]$

- If some value $v$ has been approved by at least $2t + 1$ processes, i.e at least $t + 1$ correct processes, $p_i$ confirms the value by appending it at the end of CONFIRM$[i]$ (Line 17).

  Remark that if the condition of Line 17 is true for some process, then the condition of Line 15 will always remain true (for the same $s$) even if $t$ Byzantine processes change their approved value. Hence, all correct processes will eventually confirm the value.

- If some value $v$ has been confirmed by more than $2t$ different processes (correct or not), $p_i$ logs $v$ (Line 18). Recall that $l \supseteq l'$ denotes the fact that $l'$ is a prefix of $l$.

When any process $p_i$ reads the Read/Append register, it first invokes synch() to update its local copies of the shared variables (Line 4), then it computes the number $count_i$ of values that have been confirmed by more than $t$ processes (Line 5) and waits until all these values have been confirmed by at least $2t$ processes (Line 6). This ensures the predicate AIC$(log_i)$ defined by Definition 6: all these values have been confirmed by at least $t + 1$ correct processes, which will remain true in the future even if $t$ Byzantine processes change their mind. Then, $p_i$ returns the sequence composed of these values (Line 7) knowing that these values will also be returned by all subsequent reads.

**Definition 6 (Add In Confirm)** *For all sequences of values $l$, let us define the predicate* AIC$(l)$ *as follows:* AIC$(l)$ *is verified if, and only if, there exists at least $t + 1$ correct processes $p_j$ such that $l$ is a prefix of* CONFIRM$[j]$.

## 3.3 Correctness of the Algorithm

We now prove the correctness of Algorithm 1.

**Lemma 7 (Approved value)** *Two correct processes cannot approve different $s^{th}$ values.*

**Proof 8** *Suppose some correct processes $p_i$ (resp. $p_j$) writes $v_i$ (resp. $v_j$) in* APPROVE$[s]$*, for some $s$. Since this can happen on Line 15 only if $p_i$ (resp. $p_j$) has read $v_i$ (resp. $v_j$) from a correct process in* APPROVE$[s]$*, some correct process wrote $v_i$ (resp. $v_j$) in* APPROVE$[s]$ *on Line 14. By the condition on Line 14, more than $\frac{n+t}{2}$ processes endorsed $v_i$ (resp. $v_j$). Therefore, at least $t + 1$ processes endorsed both values on Line 12, among which there is one correct process. This contradicts the condition $endorse_i[i] = \perp$ of Line 12.*

**Lemma 9 (Confirmed value)** *If a correct process $p_i$ logs $v$ on Line 18, then it confirmed $log_i \oplus v$.*

**Proof 10** *Suppose a correct process $p_i$ writes $log_i \oplus v$ on Line 18. The condition on Line 16 was false, so $p_i$ wrote $log_i \oplus v'$ to* CONFIRM$[i]$ *on Line 17, for some $v'$ that was approved by a correct process. By the condition on Line 18, some correct process confirmed $v$ on Line 17, so some correct process approved $v$. Since $v$ and $v'$ were both approved by a correct process, $v = v'$ by Lemma 7.*

**Lemma 11 (Logged value)** *If the writing process $p_w$ is correct and some correct process $p_i$ logs $v$ as its $s^{th}$ value, then $v$ is the $s^{th}$ value written by $p_w$.*

**Proof 12** *When $p_i$ logs $v$ as its $s^{th}$ value (Line 18), $log_i \oplus v$ has already been confirmed by at least $2t + 1$ processes, hence by some correct process in Line 17. Therefore, $v$ was approved by at least $2t + 1$ processes, on Line 14 or 15. Remark that the first correct process that approved $v$ could not have done so on Line 15 because, at that moment, it was approved by at most $t$ Byzantine processes. Hence, some correct process approved $v$ on Line 14, after $v$ had been endorsed by more than $\frac{n+t}{2} \geq 2t$ processes. Some correct process among them endorsed $v$ on Line 12. By Line 12, $v'$ is the value $v$ that $p_w$ wrote on Line 2 during its $s^{th}$ invocation of* `append`.

**Lemma 13 (Stability of aic)** *If* AIC$(l)$ *is true for some $l$ at some time, then it remains true forever.*

**Proof 14** *Suppose that $l$ is a prefix of* CONFIRM$[i]$ *for some correct process $p_i$. If* CONFIRM$[i]$ *is a prefix of $log_i$, then it will remain true because $p_i$ only appends values at the end of $log_i$, to overwrite either $log_i$ or* CONFIRM$[i]$*. Otherwise, $p_i$ already executed Line 17, but not yet Line 18, and by Lemma 9, it can only write* CONFIRM$[i]$ *in $log_i$. Hence, $l$ remains a prefix of* CONFIRM$[i]$ *forever, and the same is true for all correct processes.*

**Lemma 15 (Inclusion for aic)** *If there exist $l$ and $l'$ such that the condition* AIC$(l)$ *is true at some time, and* AIC$(l')$ *is true at some time, then $l$ is a prefix of $l'$ or $l'$ is a prefix of $l$.*

**Proof 16** *By Lemma 13,* AIC$(l)$ *and* AIC$(l')$ *remain true forever after some point. Then, some correct processes $p_i$ and $p_j$ confirmed $l$ and $l'$, respectively. Suppose (by contradiction) that the lemma is false, and let us consider the longest common prefix $log$ of $l$ and $l'$, as well as the first value $v_i \neq v_j$ by which $l$ and $l'$ differ. As correct processes append values one by one on Line 17, $p_i$ (resp. $p_j$) confirmed $log \oplus v_i$ (resp. $log \oplus v_j$). By the condition of Line 17, some correct process approved $v_i$ (resp. $v_j$), which contradicts Lemma 7.*

**Lemma 17 (Log and aic)** *If $p_i$ is a correct process,* AIC$(log_i)$ *holds at all times.*

**Proof 18** *We prove the lemma by induction on $|log_i|$. Initially,* AIC$(\varepsilon)$ *holds. Suppose* AIC$(log_i)$ *holds for some $log_i$, and let us suppose $p_i$ logs $v$ on Line 18. By the condition of that line, $log_i \oplus v$ was confirmed by at least $2t + 1$ processes, hence by at least $t + 1$ correct processes. Therefore,* AIC$(log_i \oplus v)$ *holds at that time, and by Lemma 13,* AIC$(log_i \oplus v)$ *remains true forever afterward.*

**Lemma 19 (Reads and aic)** *Let $l$ be a sequence such that* AIC$(l)$ *is true when a correct process $p_i$ invokes a* `read`() *operation. Then $l$ is a prefix of the sequence returned to $p_i$.*

**Proof 20** *By Line 5, $|l| \leq count_i$, so by Line 6, $|l| \leq |log|_i$. Moreover,* AIC$(log_i)$ *holds by Lemma 17. Since* AIC$(l)$ *and* AIC$(log_i)$ *are satisfied, Lemma 15 implies that $l$ is a prefix of $log_i$.*

**Lemma 21 (Linearizability)** *Let $H$ be a distributed history admitted by Algorithm 1. Then $H$ is Byzantine linearizable with respect to the Read/Append register.*

**Proof 22** *Following the characterization of Proposition 2, we prove the four properties that imply Byzantine linearizability.*

Proof of the Validity property. *Correct processes $p_i$ return $log_i$ on Line 7, which is composed of logged values that have been written by $p_w$ according to Lemma 11.*

Proof of the Read after Write property. *Let us suppose $p_w$ is correct and completed its $s^{th}$ append operation (of some value $v$) before a correct process $p_i$ starts reading. By Lemma 17,* AIC$(log_w)$ *holds on Line 3, with $|log_w| \geq s$. By Lemma 13,* AIC$(log_w)$ *is still true when $p_i$ starts its read, so by Lemma 19, $log_w$ is a prefix of the sequence returned by $p_i$, of size at least $s$.*

Proof of the Inclusion property. *Let us consider two reads $r_i$ and $r_j$, done by two processes $p_i$ and $p_j$, that return respectively $l_i$ and $l_j$. By Lemma 17, we have* AIC$(l_i)$ *(resp.* AIC$(l_j)$*) when $p_i$ (resp. $p_j$) executes Line 7, which implies the inclusion property by Lemma 15.*

Proof of the Read after read property. *Let $r_i$ and $r_j$ be two* `read` *operations done by correct processes $p_i$ and $p_j$, that return respectively $l_i$ and $l_j$, such that $r_i$ completes before $r_j$ starts. By Lemmas 13, 17, and 19,* AIC$(l_i)$ *is verified when $p_i$ returns, then when $p_j$ starts its read, so $l_i$ is a prefix of $l_j$.*

**Lemma 23 (Liveness)** *If some correct process $p_i$ confirms a sequence of size $s$, all correct processes eventually log a $s^{th}$ value.*

**Proof 24** *Suppose some correct process $p_i$ confirms a sequence of size $s$, and some process $p_j$ never writes a sequence of size $s$ in $log_j$. Since $p_j$ appends values one by one in $log_j$, this means that $|log_j|$ plateaus at a size $s' < s$ after some point in time. Without loss of generality, let us assume that $s'$ is minimal, i.e. $|log_k|$ reaches at least $s'$ for all correct process $p_k$. By Lemma 9 and the condition on Line 17, when $|log_i|$ reaches $s'+1$, at least $t+1$ approved $s'+1$ values.*

*Since all correct processes $p_j$ repeatedly execute* `synch()` *thanks to Line 8, all processes eventually read $v$ at least $t+1$ times in* APPROVE$[s']$ *(and only $v$, by Lemma 7) and approve it (Line 15). This eventually satisfies the condition on Line 17, and then the condition on Line 18. Hence, $|log_j|$ reaches $s'+1$. A contradiction.*

**Lemma 25 ($t$-resilience)** *Algorithm 1 is $t$-resilient.*

**Proof 26** Termination of the `append` operation. *Suppose a correct process $p_w$ invokes* `append` $(v)$ *to write its $s^{th}$ value, and let $l_w$ be $log_w$ on Line 1. By Lemma 17* AIC$(l_w)$ *holds at the beginning of the execution. Hence, some correct process $p_i$ writes $l_w$ in* CONFIRM$[i]$*, and by Lemma 23, each correct process $p_i$ eventually reach a state where $|log_i| = |l_w|$.*

*Since all correct processes repeatedly execute* `synch()` *thanks to Line 8, all processes eventually read $v$ in* ENDORSE$[|l_w|][w]$ *(Line 9) and write it in* ENDORSE$[|l_w|][i]$ *(Line 12). Since there are at least $n-t > \frac{2n}{3} > \frac{n+t}{2}$ correct processes, eventually, all of them write $v$ in* APPROVE$[|l_w|]$ *(Line 15). Since there are at least $n-t > \frac{2n}{3} > 2t$ correct processes, eventually, each of them appends $v$ at the end of* CONFIRM *(Line 17). Then, $p_w$ can read $v$ in the end of* CONFIRM$[j]$ *(Line 11) for more than $2t$ different $j$, and appends it to $log_w$ (Line 18) and terminates its while loop (Line 3).*

Termination of the `read` operation. *Suppose a correct process $p_i$ invokes* `read()`*, and let us study its loop on Line 6. By Line 5, at least one correct $p_j$ process wrote a log containing at least $count_i$ values in* CONFIRM$[j]$*. By Lemma 23, $p_i$ eventually writes at least $count_i$ values in $log_i$ and complete its* `read` *operation.*

**Theorem 27 (Correctness of Algorithm 1)** *Algorithm 1 implements a $t$-resilient linearizable SWRM Read/Append register in the model $\mathcal{BASM}_{n,t}[R/W, t < \frac{n}{3}]$.*

**Proof 28** *By Lemma 21, Algorithm 1 is linearizable. By Lemma 25, Algorithm 1 is $t$-resilient.*

# 4 From R/WI registers to R/A registers

This section shows that the hypothesis $t < \frac{n}{2}$ is necessary, and sufficient, to implement a R/A register on top of R/WI registers. More precisely, Section 4.1 proves that any such reduction algorithm has a resilience $t < \frac{n}{2}$ as an upper bound, and then Section 4.2 presents an algorithm with an optimal resilience.

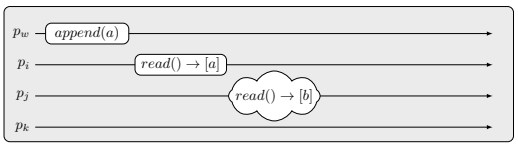
## 4.1 An upper bound on resilience

This section proves that for any implementation of a linearizable Read/Append register in a system where processes communicating only through Read/Write-Increment registers, the maximal resilience is at most $t < \frac{n}{2}$.

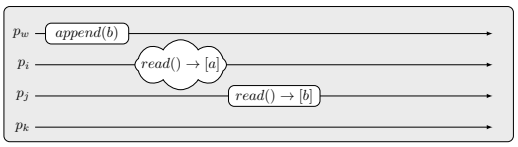
**Theorem 29** *It is impossible to implement a linearizable Read/Append register in the model $\mathcal{BASM}_{n,t}[R/WI]$, when $n \geq 4$ and $t \geq \frac{n}{2}$.*

**Proof 30** *Similarly to Section 3.1, suppose there is an Algorithm $A$ implementing a R/A register from a collection of R/WI registers in a system made up of $n \geq 4$ processes, from which $t \geq \frac{n}{2}$ maybe Byzantine. We will prove, by contradiction, that $A$ allows a run that is not linearizable. Since $n \geq 4$, processes can be divided into four non-empty subsets $W$, $I$, $J$ and $K$, each having a size not exceeding $\frac{t}{2}$ processes. Let us pick four processes $p_w \in W$ (the writing process), $p_i \in I$, $p_j \in J$ and $p_k \in K$ (three reading processes). We will consider the four situations represented in Figure 4 and described thereafter.*

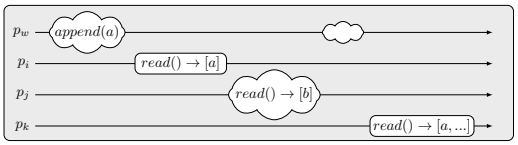
**S1:** *In this first scenario, all processes of the sets $J$ are Byzantine whereas $W$, $I$ and $K$ consist of correct processes. Processes of $K$ are too slow to participate. Process $p_w$ appends $a$ to $x$. The write terminates because $|J| \leq t$ processes are Byzantine. Then $p_i$ reads $x$ and obtains $[a]$ because $x$ is linearizable. After that, the Byzantine processes in $J$ will simulate the steps they would have taken in a read of $x$ by $p_j$, if $p_w$ had appended $b$ instead of $a$.*



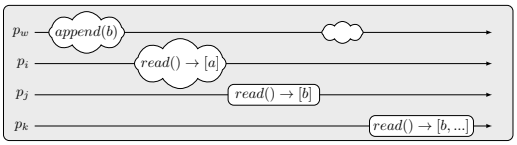

(a) Scenario $S_1$: $p_j$ and $p_k$ are Byzantine, but cannot prevent $p_i$ from reading $[a]$




(b) Scenario $S_2$: $p_i$ and $p_k$ are Byzantine, but cannot prevent $p_j$ from reading $[b]$




(c) Scenario $S_3$: $p_w$ and $p_j$ are Byzantine, but cannot prevent $p_i$ and $p_j$ from reading $[a, ...]$




(d) Scenario $S_4$: $p_w$ and $p_i$ are Byzantine, but cannot prevent $p_j$ and $p_k$ from reading $[b, ...]$

Figure 4: Illustration of the scenarios from the proof of Theorem 29, with $n = 4$ and $t = 2$

**S2:** *In the second scenario, the processes of $I$ are byzantine while the others are correct. Processes of $K$ are still slow. Process $p_w$ appends $b$ to $x$, then the Byzantine processes of $I$ simulate the steps they would take in a read of $x$ by $p_i$ if the written value were $a$. Finally, the correct process $p_j$ reads the contents of $x$ and gets $[b]$ as a result because $x$ is linearizable.*

*For all R/WI registers $x$, let $x.\mathsf{count}_1$ (resp $x.\mathsf{count}_2$) the value of $x.\mathsf{count}_1$ at the end of $S_1$ (resp. $S_2$). The two remaining scenarios are built as extensions of $S_1$ and $S_2$, except that processes of $W$ are Byzantine as well, although $p_w$ appends its value ($a$ in $S_3$ and $b$ in $S_4$) by following $A$ properly.*

**S3:** *After $p_i$ and $p_j$ finish their read in S1, all processes $p'_w \in W$ follow the following strategy to confuse correct processes. For all R/WI registers $x$ on which $p'_w$ can write, and such that $x.\mathsf{count}_1 + x.\mathsf{count}_2 \neq 0$, $p'_w$ calls* `write-incr`$(\perp)$ *on $x$ until the value of $x$ is $\langle \perp, \max(x.\mathsf{count}_1, x.\mathsf{count}_2) + 1 \rangle$.*

*Then, process $p_k$ reads the contents of $x$. Since the correct process $p_i$ already read $[a]$, $p_k$ is forced to return a sequence whose first value is $a$, because $x$ is linearizable.*

**S4:** *Processes $p'_w \in W$ continue S2 with the same strategy. For all R/WI registers $x$ $p'_w$ can write such that $x.\mathsf{count}_1 + x.\mathsf{count}_2 \neq 0$, $p'_w$ calls* `write-incr`$(\perp)$ *on $x$ until the value of $x$ is $\langle \perp, \max(x.\mathsf{count}_1, x.\mathsf{count}_2) + 1 \rangle$.*

*Then, process $p_k$ reads a sequence starting with $b$ in $x$, because $x$ is linearizable and the correct process $p_j$ already read $[b]$.*

*The contradiction comes from the fact that the scenarios $S_3$ and $S_4$ are indistinguishable to process $p_k$ during its last read: all the registers that can be written by processes of $W$ are in the same state that exposes no relevant information, all processes of $I$ pretend they have read $a$, and all processes of $J$ pretend they have read $b$. Therefore, it is impossible for $p_k$ to return a different value in $S_3$ and $S_4$, which means $A$ cannot exist.*

## 4.2 A resilience-optimal algorithm

Algorithm 2 presents an implementation of a R/A register in the model $\mathcal{BASM}_{n,t}[\text{R/WI}, t < \frac{n}{2}]$. The writing process is denoted by $p_w$.

**Shared memory and local variables** The $n$ processes share two variables called ENDORSE and COUNTER, defined as follows.

- ENDORSE$[0...][1..n]$ plays the same role as in Algorithm 1: it is an infinite array of arrays of $n$ SWMR atomic R/WI registers such that, for any $s \in \mathbb{N}$ and $i \in \{1, ..., n\}$, ENDORSE$[s][i]$ is initialized to $\langle \perp, 0 \rangle$, can only be written by $p_i$, and eventually contains $p_i$'s opinion on what the $s^{\text{th}}$ appended value is.

  Each process $p_i$ is only supposed to write once in ENDORSE$[s][i]$, for each $s$. Hence, a Byzantine process can erase a value that it has already written in ENDORSE$[s][i]$, but doing so passes the $\mathsf{count}$ field to 2, which informs the other processes that it is faulty and its values cannot be trusted.

**operation** append($v$) *invoked by* $p_w$ **is**

1 $count_w \leftarrow |log_w|$;
2 ENDORSE$[count_w][w]$.write-incr($v$);
3 COUNTER.write-incr($\perp$);
4 **while** $|log_w| \leq count_w$ **do** synch();

**operation** read() *invoked by any* $p_i$ **is**

5 $count_i \leftarrow$ COUNTER.read().count;
6 **do** $old_i \leftarrow log_i$; synch(); **while** $|log_i| < count_i \wedge old_i \neq log_i$;
7 **return** $log_i$;

**background task** $T()$ *repeatedly executed by all* $p_i$ **is**

8 synch()

**procedure** synch() *invoked by any* $p_i$ **is**

9 **if** COUNTER.read().count $\leq |log_i|$ **then return**;
10 **for** $j$ **from** 1 **to** $n$ **do** $endorse_i[j] \leftarrow$ ENDORSE$[|log_i|][j]$.read() ;
11 **if** $endorse_i[i]$.count $= 0 \wedge endorse_i[w]$.count $= 1$ **then**
12 ENDORSE$[|log_i|][i]$.write-incr($endorse_i[w]$.value)
13 **if** $\exists v : |\{j : endorse_i[j] = \langle v, 1\rangle \vee endorse_i[j]$.count $> 1\}| > t$ **then**
14 $log_i \leftarrow log_i \oplus v$;

**Algorithm 2:** Implementation of a R/A register in the model $\mathcal{BASM}_{n,t}[\text{R/WI}, t < \frac{n}{2}]$

- COUNTER is an SWMR atomic R/WI register, initialized to $\langle \perp, 0\rangle$ as well, and can only be written by the writing process $p_w$. Only the field count of COUNTER is used to represent the number of appended values, so $p_w$ only writes a dummy value $\perp$ in it.

Besides these two shared variables, each process $p_i$ maintains four local variables:

- $log_i$ is a sequence that represents the current state of the shared R/A register seen by $p_i$.
- $old_i$ is a local copy of $log_i$, used to detect when the value of $log_i$ changes locally.
- $count_i$ is an integer that represents the number of appended values.
- $endorse_i[1..n]$ is an array of size $n$, that stores a local copy of ENDORSE$[|log_i|]$ by $p_i$.

**Notations** Let $v$ be a value and $s \in \mathbb{N}$. We say that a process $p_i$ *endorses* $v$ as the $(s+1)^{\text{th}}$ value (or simply *endorses* $v$ when $s$ is immaterial) if the first value $p_i$ writes in ENDORSE$[s][i]$ is $v$. Similarly, we say that $p_i$ *logs* $v$ as the $(s+1)^{\text{th}}$ value (or simply *logs* $v$) when $p_i$ executes $log_i \leftarrow log_i \oplus v$ in Line 18, with $|log_i| = s$.

**Description of the algorithm** Similarly to Algorithm 1, Algorithm 2 implements a confirmation mechanism to ensure that a read value can never be lost. In order to write its $(s+1)^{\text{th}}$ value $v$, the writing process $p_w$ writes $v$ in ENDORSE$[s][w]$ (Line 2), then increments the count field of COUNTER (Line 3) by calling write-incr($\perp$), and then waits until it has logged at $s$ values. This happens eventually after enough correct processes have endorsed $v$ as their $(s+1)^{\text{th}}$ value by calling the procedure synch() (which they do regularly on Line 8).

When any process $p_i$ invokes synch(), it first checks whether a new value was appended by comparing the count field of COUNTER, to the length of its $log_i$ variable (Line 9), and it updates its local copy of the shared variable ENDORSE (Line 10). Then on Lines 11-12, $p_i$ endorses the value $endorse_i[w]$.value as its $(|log_i|+1)^{\text{th}}$ value if 1) it has not done so yet (i.e. $endorse_i[i]$.count $= 1$), and 2) $p_w$ has endorsed this value and has not overwritten it (i.e. $endorse_i[w]$.count $= 1$).

Finally, $p_i$ logs $v$ if enough correct processes have endorsed the same value $v$ to ensure its persistence (Lines 13-14). The condition for persistence, captured by the predicate AIH$(|log_i|, v)$ in Definition 31, is similar to the condition of Line 13, and has two important properties:

- It can only be true if $v$ was endorsed by some correct process, hence if $v$ was endorsed by the writer itself. Therefore, only one value $v$ can ever satisfy this property
- Once this property is true at the same process, it cannot be falsified at another process later.

When any process $p_i$ reads the Read/Append register, it first sets its local variable $count_i$ to the count field of COUNTER, which indicates the number of written values if the writer is correct, and a bound on the complexity of the operation otherwise. Then, it invokes synch() repeatedly until either 1) it has read $count_i$ values, or 2) its value for $log_i$ does not change during one iteration (which indicates that a Byzantine writer incremented COUNTER without updating ENDORSE properly). Then, $p_i$ returns its local value for $log_i$.

**Definition 31 (Add In History)** *For all $s \in \mathbb{N}$ and all values $v$, let $\text{AIH}(s, v)$, be the predicate $\text{AIH}_1(s, v) \wedge \text{AIH}_2(s, v)$ defined as follows:*

- $\text{AIH}_1(s, v) \triangleq \text{COUNTER.count} > s$,
- $\text{AIH}_2(s, v) \triangleq |\{i : \text{ENDORSE}[s][i] = \langle v, 1\rangle \vee \text{ENDORSE}[s][i].\text{count} > 1\}| > t$.

## 4.3 Correctness of the Algorithm

We now prove the correctness of Algorithm 2.

**Lemma 32 (Stability of aih)** *If the condition $\text{AIH}(s, v)$ is true at a given time for some pair $(s, v)$, it can never become false afterward.*

**Proof 33** Proof for $\text{AIH}_1$. COUNTER.count *cannot decrease by the definition of write-increment.*

Proof for $\text{AIH}_2$. *Let $p_i$ be a process such that* $\text{ENDORSE}[s][i] = \langle v, 1\rangle \vee$ $\text{ENDORSE}[s][i].\text{count} > 1$. *If $p_i$ does not overwrite* $\text{ENDORSE}[s][i]$, $\text{ENDORSE}[s][i]$ *keeps its value because it is a SWMR register, and if $p_i$ writes in* $\text{ENDORSE}[s][i]$ *afterwards, it remains true that* $\text{ENDORSE}[s][i].\text{count} > 1$.

**Lemma 34 (Safety of aih)** *Suppose the condition $\text{AIH}(s, v)$ is true at a given time for some pair $(s, v)$. Then at some point in the execution,* $\text{ENDORSE}[s][w] = \langle v, 1\rangle$.

**Proof 35** *For $\text{AIH}_2(s)$ to be true, at least $t + 1$ processes $p_i$ must have written in* $\text{ENDORSE}[s][i]$. *Among them, some process $p_j$ must be correct. If $p_j$ is the writer $p_w$, then the write happened on Line 2, and after that* $\text{ENDORSE}[s][w] = \langle v, 1\rangle$. *Otherwise, the write happened on Line 12. By the condition on Line 11, $p_j$ read $\langle v, 1\rangle$ in* $\text{ENDORSE}[s][w]$ *on Line 10, which concludes the proof.*

**Lemma 36 (Update of $log_i$)** *Let us consider a call of* synch() *by any correct process $p_i$, let $s$ be $|log_i|$ at the moment of the invocation, and let $v$ be any value. Then:*

- *if $\text{AIH}(s, v)$ when $p_i$ invokes* synch(), *then $p_i$ appends $v$ on Line 14;*
- *if $p_i$ appends $v$ on Line 14, then $\text{AIH}(s, v)$ is true when $p_i$ returns from* synch().

**Proof 37** *Suppose $\text{AIH}(s, v)$ when $p_i$ invokes* synch(). *Then $p_i$ does not return on Line 9 by $\text{AIH}_1(s, v)$, and $\text{AIH}_2(s, v)$ is true when $p_i$ executed Line 10, so the condition on Line 13 is true.*

*Suppose $p_i$ appends $v$ on Line 14. Then $\text{AIH}_1(s, v)$ was true when $p_i$ executed Line 9, and $\text{AIH}_2(s, v)$ was true after Line 10. Hence, Lemma 32 concludes the proof.*

**Lemma 38 (Linearizability)** *Let $H$ be a distributed history admitted by Algorithm 2. Then $H$ is Byzantine linearizable with respect to the Read/Append register.*

**Proof 39** *Following the characterization of Proposition 2, we prove the four properties that imply Byzantine linearizability.*

Proof of the Validity property. *Suppose the writing process $p_w$ is correct, let us consider the sequence $log_i$ returned by the read of a correct process $p_i$ on Line 7, let $s \in \{1, ..., |log|\}$, and let $v = log[s - 1]$. By Lemma 36, $\text{AIH}(s - 1, v)$ is true after $p_i$ appended $v$ to $log$, so by Lemma 34,* $\text{ENDORSE}[s - 1][w] = \langle v, 1\rangle$. *Since $p_w$ is correct, $p_w$ executed Line 2 when it wrote $v$ as its $s^{th}$ value.*

Proof of the Read after Write property. *Let us suppose $p_w$ is correct and completed its $s^{th}$ write (of some value $v$) before a correct process $p_i$ starts reading. By Line 4, the write can only stop when $|log_w| = s$, so by Lemma 36, $\text{AIH}(s - 1, v)$ is true at the end of the write. By Lemma 32, $\text{AIH}(s - 1, v)$ is still true at the beginning of the read. By the same reasoning for previous writes (that have been completed as well), $\text{AIH}(s', \_)$ is true for all $s' < s$. By $\text{AIH}_1(s-1, v)$, $count_i \geq s-1$ after Line 5, and by Lemma 36, some value is appended to $log_i$ each time* synch() *is called on Line 6 when $|log_i| < s$. Hence, $|log_i| \geq s$ when Line 6 completes.*

Proof of the Inclusion property. *The sequence returned by a correct process is the content of its variable $log_i$. By Lemmas 36 and 34, all processes update their $log_i$ value by appending the same values in the same order.*

Proof of the Read after read property. *Let $r_i$ and $r_j$ be two* `read` *operations done by correct processes $p_i$ and $p_j$, that return respectively $log_i$ and $log_j$, such that $r_i$ completes before $r_j$ starts. By Lemma 36,* $\text{AIH}(s', \_)$ *is true for all $s' < |log_i|$ at the end of $r_i$. Hence, applying the same reasoning as for the Read after write property, $|log_j| \geq |log_i|$. Finally, the Inclusion property implies that $log_i$ is a prefix of $log_j$.*

**Lemma 40 ($t$-resilience)** *Algorithm 1 is $t$-resilient.*

**Proof 41** Termination of the `append` operation. *Suppose a correct process $p_w$ invokes* `append`$(v)$ *to write its $(s+1)^{th}$ value. In particular, $p_w$ completed all previous writes, by Lemma 36,* $\text{AIH}(s', \_)$ *is true for all $s' < s$, and* $\text{AIH}_1(s, v)$ *is true by Line 3.*

*All correct processes repeatedly execute* `synch`() *thanks to Line 8, hence by Lemma 36 again, for all correct processes $p_i$, eventually $|log_i| = s - 1$. The next time $p_i$ executes* `synch`(), *it reads $v$ in* $\text{ENDORSE}[s-1][w]$ *(Line 10) and writes it in* $\text{ENDORSE}[s-1][i]$ *(Line 12), which is enough to satisfy* $\text{AIH}_2(s, v)$ *and allow $p_w$ to complete its write.*

Termination of the `read` operation. *Suppose a correct process $p_i$ invokes* `read`(). *Since $log_i$ can only be updated by appending values at its end (Line 14), at each iteration of the loop Line 6, either $log_i$ remains unchanged which stops the loop, or the length of $log_i$ growths, until it reaches $count_i$ which stops the loop as well.*

**Theorem 42 (Correctness of Algorithm 2)** *Algorithm 2 implements a $t$-resilient linearizable SWRM R/A register in the model $\mathcal{BASM}_{n,t}[R/WI, t < \frac{n}{2}]$.*

**Proof 43** *This is a direct consequence of lemmas 38 and 40.*

# 5 Conclusion

The goal of this paper is to investigate the relationships between three register specifications: the Read/Write register, whose `read` operation returns the last written value, the Read/Write-Increment register, whose `read` operation returns a pair composed of the last written value and the total number of values written, and the Read/Append register, whose `read` operation returns the sequence of all written values. We identified necessary and sufficient bounds on the number of Byzantine failures that can be tolerated in algorithms that build one from another. More precisely, a Read/Write-Increment register can be implemented on top of Read/Write registers if, and only if, $t < \frac{n}{3}$. Differently, a Read/Append register can be implemented on top of Read/Write-increment registers at the condition that $t < \frac{n}{2}$.

In order to prove that Read/Write-Increment registers can be implemented on top of Read/Write registers when $t < \frac{n}{3}$, Algorithm 1 actually provides an implementation of a Read/Append register. This is correct for computability reasons because Read/Write-Increment registers can be trivially obtained from Read/Append registers. However, this poses the question of the memory complexity of such algorithms: it is expected that a Read/Append register needs to keep the entire sequence of written values because of its specification, but is it possible to only keep track of the current value and a sequence number in a Read/Write-Increment register implementation?

# References


[1] Ittai Abraham, Gregory V. Chockler, Idit Keidar, and Dahlia Malkhi. Byzantine disk paxos: optimal resilience with byzantine shared memory. *Distributed Comput.*, 18(5):387–408, 2006.

[2] Ittai Abraham and Dahlia Malkhi. The blockchain consensus layer and BFT. *Bull. EATCS*, 123, 2017. URL: http://eatcs.org/beatcs/index.php/beatcs/article/view/506.

[3] Amitanand Aiyer, Lorenzo Alvisi, and Rida A. Bazzi. Bounded wait-free implementation of optimally resilient byzantine storage without (unproven) cryptographic assumptions. In *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '07, page 310–311, New York, NY, USA, 2007. Association for Computing Machinery. https://doi.org/10.1145/1281100.1281147 doi:10.1145/1281100.1281147.

[4] Hagit Attiya. Efficient and robust sharing of memory in message-passing systems. *J. Algorithms*, 34(1):109–127, 2000.

[5] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *J. ACM*, 42(1):124–142, 1995.

[6] Hagit Attiya and Amir Bar-Or. Sharing memory with semi-byzantine clients and faulty storage servers. *Parallel Process. Lett.*, 16(4):419–428, 2006.

[7] E. Borowsky and E. Gafni. Generalized FLP impossibility result for t-resilient asynchronous computations. In *Proc. of the 25th Annual ACM Symposium on Theory of Computing STOC*, pages 91–100, 1993.

[8] Gabriel Bracha and Sam Toueg. Asynchronous consensus and broadcast protocols. *Journal of the ACM (JACM)*, 32(4):824–840, 1985.

[9] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In Margo I. Seltzer and Paul J. Leach, editors, *Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation (OSDI), New Orleans, Louisiana, USA, February 22-25, 1999*, pages 173–186. USENIX Association, 1999. URL: https://dl.acm.org/citation.cfm?id=296824.

[10] Gregory V. Chockler and Dahlia Malkhi. Active disk paxos with infinitely many processes. *Distributed Comput.*, 18(1):73–84, 2005.

[11] V. Cholvi, A. Fernández Anta, C. Georgiou, N. Nicolaou, and M. Raynal. Atomic appends in asynchronous byzantine distributed ledgers. In *16th European Dependable Computing Conference EDCC*, pages 77–84. IEEE, 2020.

[12] Shir Cohen and Idit Keidar. Tame the Wild with Byzantine Linearizability: Reliable Broadcast, Snapshots, and Asset Transfer. In Seth Gilbert, editor, *35th International Symposium on Distributed Computing (DISC 2021)*, volume 209 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 18:1–18:18, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. URL: https://drops.dagstuhl.de/opus/volltexte/2021/14820, https://doi.org/10.4230/LIPIcs.DISC.2021.18 doi:10.4230/LIPIcs.DISC.2021.18.

[13] Dan Dobre, Rachid Guerraoui, Matthias Majuntke, Neeraj Suri, and Marko Vukolić. The complexity of robust atomic storage. In *Proceedings of the 30th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, PODC '11, page 59–68, New York, NY, USA, 2011. Association for Computing Machinery. https://doi.org/10.1145/1993806.1993816 doi:10.1145/1993806.1993816.

[14] Rachid Guerraoui and Marko Vukolić. How fast can a very robust read be? In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Principles of Distributed Computing*, PODC '06, page 248–257, New York, NY, USA, 2006. Association for Computing Machinery. https://doi.org/10.1145/1146381.1146419 doi:10.1145/1146381.1146419.

[15] Maurice Herlihy and Jeannette Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[16] Damien Imbs, Sergio Rajsbaum, Michel Raynal, and Julien Stainer. Read/write shared memory abstraction on top of asynchronous byzantine message-passing systems. *J. Parallel Distributed Comput.*, 93-94:1–9, 2016.

[17] Leslie Lamport. On interprocess communication. part I: basic formalism. *Distributed Comput.*, 1(2):77–85, 1986.

[18] Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

[19] Dahlia Malkhi and Michael K. Reiter. Secure and scalable replication in phalanx. In *Proceedings of the The 17th IEEE Symposium on Reliable Distributed Systems*, SRDS '98, page 51, USA, 1998. IEEE Computer Society.

[20] Jean-Philippe Martin and Lorenzo Alvisi. A framework for dynamic byzantine storage. In *Proceedings of the 2004 International Conference on Dependable Systems and Networks*, DSN '04, page 325, USA, 2004. IEEE Computer Society.

[21] Achour Mostéfaoui, Matoula Petrolia, Michel Raynal, and Claude Jard. Atomic read/write memory in signature-free byzantine asynchronous message-passing systems. *Theory Comput. Syst.*, 60(4):677–694, 2017.

[22] Marshall C. Pease, Robert E. Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2):228–234, 1980.

[23] Fred B. Schneider. The state machine approach: A tutorial. In *Proc. of Asilomar Workshop on Fault-Tolerant Distributed Computing*, volume 448 of *LNCS*, pages 18–41. Springer, 1986.