# Atomic Read/Write Memory in Signature-free Byzantine Asynchronous Message-passing Systems

Achour Mostefaoui, Matoula Petrolia, Michel Raynal, Claude Jard

**Publications Internes de l'IRISA**
**ISSN : 2102-6327**
**PI 2028 – December 2015**

UMR IRISA

# Atomic Read/Write Memory in Signature-free Byzantine Asynchronous Message-passing Systems

Achour Mostéfaoui[*] Matoula Petrolia[**] Michel Raynal[***] Claude Jard[****]

**Abstract:** This article presents a signature-free distributed algorithm which builds an atomic read/write shared memory on top of an $n$-process asynchronous message-passing system in which up to $t < n/3$ processes may commit Byzantine failures. From a conceptual point of view, this algorithm is designed to be as close as possible to the algorithm proposed by Attiya, Bar-Noy and Dolev (JACM 1995), which builds an atomic register in an $n$-process asynchronous message-passing system where up to $t < n/2$ processes may crash. The proposed algorithm is particularly simple. It does not use cryptography to cope with Byzantine processes, and is optimal from a $t$-resilience point of view ($t < n/3$). A read operation requires $O(n)$ messages, and a write operation requires $O(n^2)$ messages.

**Key-words:** Asynchronous message-passing system, Atomic read/write register, Byzantine process, Linearizability, Reliable broadcast abstraction.

---

*Mémoire partagée fiable dans les systèmes asynchones avec processus Byzantins*

**Résumé :** *Cet article présente une construction de mémoire partagée au dessus d'un système asynchone à passage de messages dans lequel jusqu'à $t < n/3$ processus peuvent avoir des comportements arbitraires (fautes Byzantines), $n$ étant le nombre total de processsus.*

**Mots clés :** *systèmes asynchrones à passage de messages, registres atomiques read/write, diffusion fiable, processus Byzantins.*

---

[*] LINA, Université de Nantes, 44322 Nantes, France
[**] LINA, Université de Nantes, 44322 Nantes, France
[***] Institut Universitaire de France and ASAP (équipe commune avec l'Université de Rennes 1 et Inria)
[****] LINA, Université de Nantes, 44322 Nantes, France

# 1 Introduction

**Shared memory abstraction on top of a message-passing system** *Informatics* is a science of abstractions, and accordingly (as in sequential computing) the writing of distributed applications can be greatly facilitated by the design and the use of underlying appropriate abstractions.

This paper considers the design of such an abstraction, namely an atomic read/write memory, on top of an asynchronous message-passing distributed system made up of $n$ processes, and where up to $t$ processes may commit failures. The case of crash failures was solved by Attiya, Bar-Noy and Dolev in [3] (a) where it is shown that $t < n/2$ is an upper bound for the model parameter $t$, and (b) where a simple, elegant, and $t$-resilient optimal algorithm is proposed. This algorithm is called ABD in the following.

This paper focuses on the case where processes may commit Byzantine failures, i.e., may behave in a way that does not respect their intended behavior (as defined by their specification).

**Related work** Considering the *clients/servers* distributed model, several articles have addressed the design of servers implementing a shared memory accessible by clients. The servers are usually managing a set of disks (e.g., [8, 13, 18]). Moreover, while they consider that some servers can be Byzantine, some articles restrict the failure type allowed to clients. As an example, [9, 10] explore the efficiency issues (relation between resilience and fast reads) in the context where only servers can be Byzantine, while clients (the single writer and the readers) can fail by crashing. As other examples, [13] considers that clients can only commit crash failures, while [4] considers that clients can only be "semi-Byzantine" (i.e., they can issue a bounded number of faulty writes, but otherwise respect their code). The algorithm presented in [17] allows clients and some number of servers to be Byzantine, but requires clients to sign their messages. As far as we know, [1] was the first paper considering Byzantine readers while still offering maximal resilience (with respect to the number of Byzantine servers) without using cryptography. However, the writer can fail only by crashing, and the fact that a –possibly Byzantine– reader does not write a fake value in a register (to ensure the "reader have to write" rule required to implement atomicity) is insured only with some probability.

In the *peer-to-peer* model (defined here as a model in which all processes are "equal"), the construction of an atomic register requires that each process manages a copy the register that is built. The first algorithm building a read/write shared memory in a message-passing system where processes may commit Byzantine failures is (to our knowledge) the one presented in [12]. This paper considers the implementation of an SWMR (single-writer/multi-reader) atomic register. It also shows that $t < n/3$ is an upper bound the resilience parameter $t$ for such a construction. In this algorithm, each SWMR atomic read/write register is represented, at each process, by the full history of all its modifications.

The fact that an SWMR register is considered is due to the following observation: as a Byzantine process can corrupt any register it can write, the design of a multi-writer/multi-reader register with non-trivial correctness guarantees is impossible in the presence of Byzantine processes.

**Content of the paper** This paper presents a new algorithm implementing an array of $n$ SWMR atomic read/write registers (one per process) in an asynchronous message-passing system where up to $t < n/3$ processes may commit Byzantine failures. This algorithm does not require to enrich the underlying system with cryptography-based techniques.

When designing this algorithm, an aim was to obtain an algorithm whose "spirit" is "as close as possible" to ABD. We think that this is important from both understanding and pedagogical point of views. It helps better understand the "gap" between crash failures and Byzantine failures. From an algorithmic point of view, we have the following:

- With respect to the algorithm described in [12], the proposed algorithm requires a process to store only a single pair (value, sequence number) per atomic register.
- With respect to ABD, there are two main differences:
  - One is the way processes implement the "reads have to write" requirement needed to obtain the atomicity property of a register [16].
  - The other one lies in the broadcast operation used to disseminate new values. While a simple unreliable broadcast[1] is sufficient in the presence of process crash failures, a stronger broadcast needs to be used to cope with Byzantine processes in a signature-free system.

The resulting algorithm is particularly simple. Moreover, when considering the non-faulty processes, a read costs $O(n)$ messages and a write costs $O(n^2)$ messages.

[1] This broadcast is a simple send of the same message to all processes. If a process crashes during its execution, it is possible that only a subset of the processes receive the message.

**Roadmap** The paper is composed of 6 sections. Section 2 presents the computation model, and the underlying reliable broadcast abstraction. Section 3 presents a specification of an SWMR read/write atomic register in the presence of Byzantine processes. Then, Section 4 presents the algorithm, and Section 5 proves its correctness. Finally, Section 6 concludes the paper.

# 2 Computation model

## 2.1 Process model, communication model, and failure model

**Computing entities** The system is made up of a set $\Pi$ of $n$ sequential processes, denoted $p_1$, $p_2$, ..., $p_n$. These processes are asynchronous in the sense that each process progresses at its own speed, which can be arbitrary and remains always unknown to the other processes.

**Communication model** The processes cooperate by sending and receiving messages through bi-directional channels. The communication network is a complete network, which means that each process $p_i$ can directly send a message to any process $p_j$ (including itself). It is assumed that the Byzantine processes cannot control the network, hence when a process receives a message, it can unambiguously identify its sender. Each channel is reliable (no loss, corruption, or creation of messages), not necessarily first-in/first-out, and asynchronous (while the transit time of each message is finite, there is no upper bound bound on message transit times).

A process $p_i$ invokes the operation "send TAG($m$) to $p_j$" to send the message tagged TAG and carrying the value $m$. It receives a message tagged TAG by invoking the operation "receive TAG()". "broadcast TAG($m$)" is a macro-operation that expands as "**for each** $j \in \{1, \cdots, n\}$ send TAG($m$) to $p_j$ **end for**". (The sending order is arbitrary, which means that, if the sender crashes while executing this statement, an arbitrary subset of processes of processes will receive the message.)

**Byzantine failures** The model parameter $t$ is an upper bound on the number of processes that can exhibit a Byzantine behavior [14, 21]. A Byzantine process is a process that behaves arbitrarily: it can crash, fail to send or receive messages, send arbitrary messages, start in an arbitrary state, perform arbitrary state transitions, etc. Hence, a Byzantine process, which is assumed to send a message $m$ to all the processes, can send a message $m_1$ to some processes, a different message $m_2$ to another subset of processes, and no message at all to the other processes. Moreover, while they cannot modify the content of the messages sent by non-Byzantine processes, they can read their content and reorder their deliveries. More generally, Byzantine processes can collude to "pollute" the computation.

A Byzantine process is also called a *faulty* process. A process that commits no failure (i.e., a non-Byzantine process) is also called a *correct* process.

**Notation** In the following, the previous computation model, restricted to the case where $t < n/3$, is denoted $\mathcal{BAMP}_{n,t}[t < n/3]$.

## 2.2 Reliable broadcast abstraction

This section presents a reliable broadcast abstraction (denoted r-broadcast) that will be used to build a read/write register (Section 4). This abstraction is a simple generalization of a reliable broadcast due to Bracha [6]. While Bracha's abstraction is for a single broadcast, the proposed abstraction considers that each process can issue a sequence of broadcasts. It is shown in [6] that $t < n/3$ is a necessary requirement to cope with the net effect of asynchrony and Byzantine failures.

**Specification** The reliable broadcast abstraction is defined by two operations denoted R_broadcast() and R_deliver(). When a process $p_i$ invokes R_broadcast() we say that "$p_i$ r-broadcasts a value". Similarly, when $p_i$ returns from an invocation of R_deliver() and obtains a value, we say "$p_i$ r-delivers a value".

The operation R_broadcast() has two input parameters: a broadcast value $v$, and an integer $sn$, which is a local sequence number used to identify the successive r-broadcasts issued by each process $p_i$. The sequence of numbers used by each (correct) process is the increasing sequence of consecutive integers.

- RB-Validity. If a correct process r-delivers a pair $(v, sn)$ from a correct process $p_i$, then $p_i$ invoked the operation R_broadcast$(v, sn)$.
- RB-Integrity. Given any process $p_i$, a correct process r-delivers at most once a pair $(-, sn)$ from $p_i$.
- RB-Uniformity. If a correct process r-delivers a pair $(v, sn)$ from $p_i$ (possibly faulty), then all the correct processes eventually r-deliver the same pair $(v, sn)$ from $p_i$.

- RB-Termination. If the process that invokes R_broadcast$(v, sn)$ is correct, all the correct processes eventually r-deliver the pair $(v, sn)$.

RB-Validity is on correct processes and relates their outputs to their inputs, namely no correct process r-delivers spurious messages from correct processes. RB-Integrity states that there is no r-broadcast duplication. RB-Uniformity is an "all or none" property (it is not possible for a pair to be delivered by a correct process and to be never delivered by the other correct processes). RB-Termination is a liveness property: at least all the pairs r-broadcast by correct processes are r-delivered by them.

For completeness, an algorithm (due to Bracha [6]), which implements the r-broadcast abstraction in the model $\mathcal{BAMP}_{n,t}[t < n/3]$, is described in Appendix A.

# 3 Atomic Read/Write Registers in the Presence of Byzantine Processes

## 3.1 Definitions and specification

**Single-writer/multi-reader (SWMR) registers** The fault-tolerant shared memory supplied to the upper abstraction layer is an array denoted $REG[1..n]$. For each $i$, $REG[i]$ is a single-writer/multi-reader (SWMR) register. This means that $REG[i]$ can be written only by $p_i$. To that end, $p_i$ invokes the operation $REG[i]$.write$(v)$ where $v$ is the value it wants to write into $REG[i]$. Differently, any process $p_j$ can read $REG[i]$. It invokes then the operation $REG[i]$.read$()$.

As already noticed in the Introduction, the "single-writer" requirement is natural in the presence of Byzantine processes. If registers could be written by any process, it would be possible for the Byzantine processes to pollute the whole memory, and no non-trivial computation could be possible.

**On write operations by Byzantine processes** A Byzantine process $p_k$ may invoke the write operation $REG[k]$.write$()$ to assign a new value to $REG[k]$, but it may also try to modify $REG[k]$ without using this operation. In such a case, its fraudulent attempt to modify $REG[k]$ may succeed or not. If it succeeds, the corresponding modification of $REG[k]$ is considered (from an external observer's point of view) as if it has been produced by an invocation of $REG[k]$.write$()$[2]. This is because no correct process can distinguish such a modification of $REG[k]$ from a call to the write operation by $p_k$. Let us nevertheless notice that this does not prevent the fact that the value assigned to $REG[k]$ can be a fake value. Moreover, at the abstraction level defined by $REG[1..n]$, as $p_k$ is sequential, its modifications of $REG[k]$ appear as if they have been executed sequentially.

**Definitions**

- A sequence of values, denoted $H_i$, is associated with each register $REG[i]$. $H_i$ is the sequence of values written in $REG[i]$. Let $H_i[x]$ denote the $x^{\text{th}}$ element of $H_i$.
- The following notations are used.
  - Let $p_i$ be a correct process. $read[i, j, x]$: execution of $REG[j]$.read$()$ returning $H_i[x]$.
  - $write[i, x]$: $x^{\text{th}}$ update of $REG[i]$ by $p_i$. Hence, $write[i, x]$ defines the value of $H_i[x]$.

    If $p_i$ is a correct process, $write[i, x]$ corresponds to an execution of $REG[i]$.write$()$. If $p_i$ is Byzantine, according to the previous discussion, these "$write[i, x]$" capture all the modifications of $REG[i]$ by $p_i$, be them associated with a call to the write operation or not. (Let us remember that, at this abstraction level, any process is sequential.)

**Specification**

The correct behavior of the array of registers $REG[1..n]$ is defined by the following set of properties.

- Termination (liveness). Let $p_i$ be a correct process.
  - Each invocation of $REG[i]$.write$()$ terminates.
  - For any $j$, any invocation of $REG[j]$.read$()$ by $p_i$ terminates.
- Consistency (safety)[3]. Let $p_i$ and $p_j$ be correct processes and $p_k$ any process.
  - Read followed by write: ($read[i, k, x]$ terminates before $write[k, y]$ starts) $\Rightarrow (x < y)$.

[2] As we will see, at the operational level, when a modification of $REG[i]$ by a Byzantine process $p_i$ succeeds, the underlying messages generated by $p_i$ could have been sent by a correct implementation of the operation write$()$.

[3] It would be possible to associate a start event and an end event with each $read[i, j, x]$ and each $write[i, y]$ issued by a correct process, and a start event with each $write[j, y]$ issued by a Byzantine process, so that all these events define a total order from which the notion of "terminates before" could be formally defined (as in [11, 15, 23]). To not overload the presentation, we do not use this formalization here.

– Write followed by read: ($write[j, x]$ terminates before $read[i, j, y]$ starts) $\Rightarrow (x \leq y)$.[4]
– No read inversion: ($read[i, k, x]$ terminates before $read[j, k, y]$ starts) $\Rightarrow (x \leq y)$.

As there is no way to constrain the behavior of a Byzantine process, the termination property is only on correct processes, and there is no requirement on the value returned by a read issued by a Byzantine process. The safety property concerns only the values read by correct processes. The first property states that there is no read from the future, while the second property states that no read can obtain an overwritten value. Due to the possible concurrent accesses to a same register, these two properties actually defines a regular register [15]. Hence the "no read inversion" property, which allows to obtain an atomic register from a regular register [7, 15, 23].

## 3.2 Linearizability

Atomic registers were formally defined in [15, 19]. Then, the atomicity concept was extended to any concurrent object defined by a sequential specification under the name linearizability [11]. Hence in our context, the terms "atomic register" and "linearizable register" are synonymous. The properties provided by linearizability are investigated in [11].

**Definition** Given a register $REG[i]$, *linearizability* [11] means that it is possible to totally order the executions of its read and write operations in such a way that (a) each operation appears as if it has been executed at a single point of the time line between its start event and its end event, (b) no two operation executions appear at the same point, and (c) each read operation returns the value written by the closest write operation that precedes it in the sequence (or the initial value if there is no such write operation).

A register is *linearizable* if its operations satisfy the previous items (a), (b), and (c). The *linearization point* of an operation is the point of the timeline at which this operation appears to have been instantaneously executed.

**An important property of Linearizability** An important theorem associated with linearizability is the following [11]: If each object (here a register) is linearizable, then the set of all the objects, considered as a single object, is linearizable. This means that linearizable objects compose for free.

**Theorem 1** *The register specification defined in Section* 3 *defines atomic (linearizable) registers.*

**Proof** As linearizable (atomic) objects compose for free [11], it is sufficient to consider a single register and, starting from its specification defined in Section 3.1, show that it is linearizable.

Let $REG[i]$ be a register. Let $H_i$ be the sequence of values written by $p_i$ in $REG[i]$.[5] The proof consists in building a sequence $S_i$ which (a) includes all the read operations of $REG[i]$ issued by the correct processes plus the writes of $REG[i]$ issued by $p_i$, and (b) satisfies the definition of linearizability.

To simplify and without loss of generality, let us assume that there is an initial write that gives $REG[i]$ its initial value. Let us start with $S_i$ being the sequence of write operations that produced the sequence $H_i$.

Let $read[j, i, x]$ be a read operation issued by a correct process $p_j$. Let $write[i, a]$ be the last write of $REG[i]$ that terminates before $read[j, i, x]$ starts. Let $write[i, a+1]$, ..., $write[i, a+c]$ be (if any) the writes of $REG[i]$ that are concurrent with $read[j, i, x]$. If there is no such writes $c = 0$. Let $b = c + 1$. Hence, if any, $write[i, b]$ is the first write of $REG[i]$ starting after $read[j, i, x]$ has terminated. We have the following.

- It follows that from the properties "read followed by write" and "write followed by read" that $x \in \{a, a + 1, ..., a + c\}$.
- It follows from the "no read inversion" property that if $read[\ell, i, x']$ (issued a correct process $p_\ell$) starts after $read[j, i, x]$, we have $x \leq x'$.

The operation $read[j, i, x]$ is added to $S_i$ just after $write[i, x]$. It there are two (or more) operations $read[j_1, i, x]$ and $read[j_2, i, x]$ issued by correct processes, they are placed one after the other in the sequence $S_i$. All the read operations issued by the correct processes are added to $S_i$ as described.

It is easy to see that the execution associated with $S_i$ is linearizable (i.e., satisfies the items (a), (b), and (c) stated above). $\square_{Theorem\ 1}$

[4] Let us notice that this property considers that the write of $REG[j]$ is issued by a correct process. This is because it is not always possible to define when the modification of $REG[j]$ has terminated when $p_j$ is Byzantine.

[5] As we have seen, if $p_i$ is Byzantine, this sequence contains all the modifications of $REG[i]$ which cannot be distinguished by the correct processes from invocations of $REG[i]$.write() by $p_i$.

# 4 Construction of Single-Writer/Multi-Reader Atomic Registers

An algorithm constructing an SWMR atomic (linearizable) register in the presence of up to $t$ Byzantine processes, is described in Figure 1. As it assumes $t < n/3$, this algorithm is suited for the computing model $\mathcal{BAMP}_{n,t}[t < n/3]$. The algorithm presents the code associated with a correct process $p_i$.

The design of the algorithm strives to be as close as possible to the ABD algorithm [3], which implements an atomic register in an asynchronous system where at most $t < n/2$ may crash.[6] It uses a **wait**$(condition)$ statement. The corresponding process is blocked until the predicate $condition$ becomes satisfied. While a process is blocked, it can process the messages it receives.

**Local variables** Each process $p_i$ manages the following local variables whose scope is the full computation (local variables are denoted with lower case letters, and sub-scripted by the process index $i$).

- $reg_i[1..n]$ is the local representation of the array $REG[1..n]$ of atomic SWMR registers. Each local register $reg_i[j]$ contains two fields, a sequence number $reg_i[j].sn$, and the corresponding value $reg_i[j].val$. It is initialized to the pair $\langle init_j, 0\rangle$, where $init_j$ is the initial value of $REG[j]$.
- $wsn_i$ is an integer, initialized to 0, used by $p_i$ to associate sequence numbers with its successive write invocations.
- $rsn_i[1..n]$ is an array of sequence numbers (initialized to $[0, \cdots, 0]$) such that $sn_i[j]$ is used by $p_i$ to identify its successive read invocations of $REG[j]$.[7]

**The operation** $REG[i]$.write$(v)$ This operation is implemented by the client lines 1-4 and the server lines 11-12 (which are similar to the algorithm implementing a write operation in a crash-prone system [3]).

Process $p_i$ first increases $wsn_i$ and r-broadcasts the message WRITE$(v, wsn_i)$. Let us remark that this is the only use of the reliable broadcast abstraction by the algorithm. The process $p_i$ then waits for acknowledgments (message WRITE_DONE$(v, wsn_i)$) from $(n - t)$ distinct processes, and finally terminates the write operation. As we will see (Lemma 2), the intersection of any two quorums of $(n - t)$ processes contains at least $(t + 1)$ correct processes. This intersection property will be used to prove the consistency of the register $REG[i]$.

When $p_i$ r-delivers a message WRITE$(v, wsn)$ from a process $p_j$, it waits until $wsn = reg_i[j] + 1$ (line 12). Hence, whatever the sender $p_j$, its messages WRITE() are processed in their sending order. When this predicate becomes true, $p_i$ updates accordingly its local with respect to $REG[j]$ (line 13), and sends back to $p_j$ an acknowledgment to inform it that its new write has locally been taken into account (line 14).

**Write of** $REG[j]$ **by a Byzantine process** $p_j$ Let us observe that the only way for a process $p_i$ to modify $reg_i[j]$ is to r-deliver a message WRITE$(v, wsn)$ from a (correct or faulty) process $p_j$. Due to the RB-Uniformity of the r-broadcast abstraction, it follows that, if a correct process $p_i$ r-delivers such a message, all correct processes will r-deliver the same message, be its sender correct or faulty. Consequently each of them will eventually execute the statements of lines 12-14.

Hence, when a faulty process invokes R_broadcast WRITE$(v, wsn)$ (be the r-broadcast invocation involved in an invocation of $REG[j]$.write$(v)$ or not), its faulty behavior is restricted to broadcast fake values for $v$ and $wsn$.

**The operation** $REG[j]$.read() This operation is implemented by the client lines 5-11 and the server line 15. The corresponding algorithm is the core of the implementation of an SWMR atomic register.

When $p_i$ wants to read $REG[j]$, it first broadcasts a read request (message READ$(j, rsn_i[j])$), and waits for corresponding acknowledgments (message STATE$(rsn_i[j], -)$). Each of these acknowledgment carries the sequence number associated with the current value of $REG[j]$, as known by the sender of the message (line 15). For $p_i$ to progress, the wait predicate (line 7) states that its local representation of $REG[j]$, namely $reg_i[j]$, must be fresh enough (let us remember that the only line where $reg_i[j]$ can be modified is line 13, i.e., when $p_i$ r-delivers a message WRITE$(-, -)$ from $p_j$). This *freshness* predicate states that $p_i$'s current value of $reg_i[j]$ is as fresh as the current value of at least $(n - t)$ processes (i.e., at least $(n - 2t)$ correct processes). If the freshness predicate is false, it will become true when $p_i$ will have r-delivered WRITE$(-, -)$ messages, which have been r-delivered by other correct processes, but not yet by it.

When this waiting period terminates, $p_i$ considers the current value $\langle w, wsn\rangle$ of $reg_i[j]$ (line 8). It then broadcasts the message CATCH_UP$(j, wsn)$, and returns the value $w$ as soon as its message CATCH_UP() is acknowledged by $(n - t)$ processes (lines 9-10).

The aim of the CATCH_UP$(j, wsn)$ message is to allow each destination process $p_k$ to have a value in its local representation of $REG[j]$ (namely $reg_k[j].val$) at least as recent as the one whose sequence number is $wsn$ (line 15). The aim of this *value*

[6] In addition to the stronger necessary and sufficient condition $t < n/3$, this presentation style allows people aware of the ABD algorithm to see the additional statements needed to go from crash failures to Byzantine behavior.

[7] If we assume that no correct process $p_i$ reads its own register $REG[i]$, $rsn_i[i]$ can be used to store $wsn_i$.

**local variables initialization:**
$reg_i[1..n] \leftarrow [\langle init_0, 0\rangle, \ldots, \langle init_n, 0\rangle]$; $wsn_i \leftarrow 0$; $rsn_i[1..n] \leftarrow [0, \cdots, 0]$.
%———————————————————————————————————————

**operation** $REG[i]$.write($v$) **is**
(1) $wsn_i \leftarrow wsn_i + 1$;
(2) R_broadcast WRITE($v, wsn_i$);
(3) **wait** WRITE_DONE($wsn_i$) received from $(n - t)$ different processes;
(4) return()
**end operation**.

**operation** $REG[j]$.read() **is**
(5) $rsn_i[j] \leftarrow rsn_i[j] + 1$;
(6) broadcast READ($j, rsn_i[j]$);
(7) **wait** $\big(reg_i[j].sn \geq \max(wsn_1, ..., wsn_{n-t})$ where $wsn_1, ..., wsn_{n-t}$ are from messages STATE($rsn_i[j], -$) received from $n - t$ different processes$\big)$;
(8) **let** $\langle w, wsn\rangle$ the value of $reg_i[j]$ which allows the previous wait to terminate;
(9) broadcast CATCH_UP($j, wsn$);
(10) **wait** $\big($CATCH_UP_DONE($j, wsn$) received from $(n - t)$ different processes$\big)$;
(11) return($w$)
**end operation**.
%———————————————————————————————————————

**when a message** WRITE($v, wsn$) **is** R_delivered **from** $p_j$ **do**
(12) wait($wsn = reg_i[j].sn + 1$);
(13) $reg_i[j] \leftarrow \langle v, wsn\rangle$;
(14) send WRITE_DONE($wsn$) to $p_j$.

**when a message** READ($j, rsn$) **is** received **from** $p_k$ **do**
(15) send STATE($rsn, reg_i[j].sn$) to $p_k$.

**when a message** CATCH_UP($j, wsn$) **is** received **from** $p_k$ **do**
(16) wait $(reg_i[j].sn \geq wsn)$;
(17) send CATCH_UP_DONE($j, wsn$) to $p_k$.

Figure 1: Atomic SWMR Registers in $\mathcal{BAMP}_{n,t}[t < n/3]$ (code for process $p_i$)

*resynchronization* is to prevent read inversions. When $p_i$ has received the $(n - t)$ acknowledgments it was waiting for (line 10), it knows that no other correct process can obtain a value older than the value $w$ it returns.

**Message cost of the algorithm** In addition to a reliable broadcast (whose message cost is $O(n^2)$), a write operation generates $n$ messages WRITE_DONE. Hence the cost of a write is $O(n^2)$ message. A read operation cost $4n$ messages, $n$ messages for each of the four kinds of messages READ, STATE, CATCH_UP and $catch\_up\_done$.

**Comparing with the crash failure model** It is known that the algorithms implementing an atomic register on top of an asynchronous message-passing system prone to process crashes, require that "reads have to write" [2, 3, 5, 16, 22]. More precisely, before returning a value, in one way or another, a reader must write this value to ensure atomicity (otherwise, we have only a "regular" register [15]). Doing so, it is not possible that two sequential read invocations, concurrent with one or more write invocations, be such that the first read obtains a value while the second read obtains older value (this prevents *read inversion*).

As Byzantine failures are more severe than crash failures, the algorithm of Figure 1 needs to use a mechanism analogous to the "reads have to write" to prevent read inversions from occurring. As previously indicated, This is done by the messages CATCH_UP() broadcast at line 9 and the associated acknowledgments messages CATCH_UP_DONE() received at line 10. As previously indicated, these messages realize a synchronization during which $(n-t)$ processes (i.e., at least $(n-2t)$ correct processes) have resynchronized their value, if needed (line 15).

A comparison of two instances of the ABD algorithm [3] and the algorithm of Figure 1 is presented in Table 1. The first instance is the version of the ABD algorithm which builds an array of $n$ SWMR (single-writer/multi-reader) atomic registers (one register per process). The second instance is the version of the ABD algorithm which builds a single MWMR (multi-writer/multi-reader) atomic register.

As they depend on the application and not on the algorithm, the size of the values which are written is considered as constant. The parameters $n$ and $t$ have the same meaning as before; $m$ denotes an upper bound on the number of read and write operations on each register. The value $\log n$ is due to the fact that a message carries a constant number of process identities. Similarly, $\log m$

is due to the fact that (a) a message carries a constant number of sequence numbers, and (b) there is a constant number of message tags (including the underlying reliable broadcast).

| algorithm | failure type | requirement | msgs/write | msgs/read | msg size | local mem./proc. |
|---|---|---|---|---|---|---|
| ABD: $n$ SWMR | crash | $t < n/2$ | $O(n)$ | $O(n)$ | $O(\log n + \log m)$ | $O(n \log m)$ |
| ABD: 1 MWMR | crash | $t < n/2$ | $O(n)$ | $O(n)$ | $O(\log n + \log m)$ | $O(n \log m)$ |
| Fig. 1: $n$ SWMR | Byzantine | $t < n/3$ | $O(n^2)$ | $O(n)$ | $O(\log n + \log m)$ | $O(n \log m)$ |

Table 1: Crash vs Byzantine failures: cost comparisons

# 5 Proof of the construction

The model assumption $n > 3t$ is implicit in all the statements and proofs that follow.

## 5.1 Preliminary lemmas

**Lemma 1** *If a correct process $p_i$ r-delivers a message* WRITE$(w, sn)$ *(from a correct or faulty process), any correct process r-delivers it.*

**Proof** This is an immediate consequence of the RB-Uniformity property of the the r-broadcast abstraction. $\square_{Lemma\ 1}$

**Lemma 2** *Any two sets (quorums) of $(n - t)$ processes have at least on correct process in their intersection.*

**Proof** Let $Q_1$ and $Q_2$ be two sets of processes such that $|Q_1| = |Q_2| = n - t$. In the worst case, the $t$ processes that are not in $Q_1$ belong to $Q_2$, and the $t$ processes that are not in $Q_2$ belong to $Q_1$. It follows that $|Q_1 \cap Q_2| \geq n - 2t$. As $n > 3t$, it follows that $|Q_1 \cap Q_2| \geq n - 2t \geq t + 1$, which concludes the proof of the lemma. $\square_{Lemma\ 2}$

## 5.2 Proof of the termination properties

**Lemma 3** *Let $p_i$ be a correct process. Any invocation of $REG[i]$.*write() *terminates.*

**Proof** Let us consider the first invocation of $REG[i]$.write() by a correct process $p_i$. This write operation generates the r-broadcast of message WRITE$(-, 1)$ (lines 1-2). Due to Lemma 1, all correct processes r-deliver this message, and the waiting predicate of line 13 is eventually satisfied. Consequently, each correct process $p_k$ eventually sets $reg_k[i].sn$ to 1, and sends back to $p_i$ an acknowledgment message WRITE_DONE$(1)$. As there are least $(n - t)$ correct processes, $p_i$ receives such acknowledgments from at least $(n - t)$ different processes, and terminates its first invocation (lines 3-4).

As, for any given any process $p_j$, all correct processes process the messages WRITE() from $p_j$ in their sequence order, the lemma follows from a simple induction (whose previous paragraph is the proof of the base case).

$\square_{Lemma\ 3}$

**Lemma 4** *Let $p_i$ be a correct process. For any $j$, any invocation of $REG[j]$.*read() *terminates.*

**Proof** When a correct process $p_i$ invokes $REG[j]$.read(), it broadcasts a message READ$(j, rsn)$ where $rsn$ is a new sequence number (lines 5-6). Then, it waits until the freshness predicate of line 7 becomes satisfied. As $p_i$ is correct, each correct process $p_k$ receives READ$(j, rsn)$, and sends back to $p_i$ a message STATE$(rsn, wsn)$, where $wsn$ is the sequence number of the last value of $REG[j]$ it knows (line 15). It follows that $p_i$ receives a message STATE$(j, -)$ from at least $(n - t)$ correct processes. Let STATE$(j, wsn_1)$, $\cdots$, STATE$(j, wsn_{n-t})$ be these messages.

To show that the wait of line 7 terminates we have to show that the freshness predicate $reg_i[j].sn \geq \max(wsn_1, \cdots, wsn_{n-t})$ is eventually satisfied. Let $wsn$ be one of the previous sequence number, and $p_k$ the correct process that send it. This means that $reg_k[j].sn = wsn$ (line 15), from which we conclude (as $p_k$ is correct) that $p_k$ has previously r-delivered a message WRITE$(-, wsn)$ and updated accordingly $reg_k[j]$ at line 13 (let us remember that this is the only line at which the local register $reg_k[j]$ is updated). It follows from Lemma 1 that eventually $p_i$ r-delivers the message WRITE$(-, sn)$. It follows then from line 13 that eventually we

have $reg_i[j].sn \geq sn$. As this is true for any sequence number in $\{wsn_1, ..., wsn_{n-t}\}$, it follows that the freshness predicate is eventually satisfied, and consequently the wait statement of line 7 is satisfied.

Let us now consider the wait statement of line 10, which appears after $p_i$ has broadcast the message CATCH_UP$(j, wsn)$, where $wsn = reg_i[j].sn$ (sequence number in $reg_i[j]$ just after $p_i$ stopped waiting at line 7). We show that any correct process sends back to $p_i$ an acknowledgment CATCH_UP_DONE$(j, wsn)$ at line 17. Process $p_i$ updated $reg_i[j].sn$ to $wsn$ at line 13, and this occurred when it r-delivered a message WRITE$(-, wsn)$. The reasoning is the same as in the previous paragraph, namely, it follows from Lemma 1 that all correct processes r-deliver this message and consequently we have $reg_k[j].sn \geq wsn$ at every correct process $p_k$. Hence, the value resynchronization predicate of line 16 is eventually satisfied at all correct processes, that consequently sends back a message CATCH_UP_DONE$(j, wsn)$ at line 17, which concludes the proof of the lemma. $\square_{Lemma\ 4}$

## 5.3 Proofs of the consistency (atomicity) properties

The next lemma shows that a sequence $H_i$, as defined in Section 3, can be associated with each register $REG[i]$.

**Lemma 5** *Given any register $REG[i]$, there is a sequence of values $H_i$ such that, if $p_i$ is correct, $H_i$ is the sequence of values written by $p_i$.*

**Proof** Let us define $H_i$ as follows. Let us consider all the messages WRITE$(-, sn)$ r-delivered from a (correct or faulty) process $p_i$ by the correct processes (due to Lemma 1, these messages are r-delivered to all correct processes). Let us order these messages according to their processing order as defined by the predicate of line 12. $H_i$ is the corresponding sequence of values. (Let us notice that, if $p_i$ is Byzantine, it is possible that some of its messages WRITE() are r-delivered but never processed at line 14; if any, such messages are never added to $H_i$).

Let us now consider the case where $p_i$ is correct. It follows from the RB-Validity property of the r-broadcast abstraction that any message r-delivered from $p_i$, was r-broadcast by $p_i$. It then follows from lines 1-2 that $H_i$ is the sequence of values written by $p_i$. $\square_{Lemma\ 5}$

**Lemma 6** *Let $p_i$ be a correct process. If $read[i, j, x]$ terminates before $write[j, y]$ starts, we have $x < y$.*

**Proof** Let $p_i$ a correct process that returns value $v$ from the invocation of $REG[j]$.read(). Let $reg_i[j] = \langle v, x\rangle$ the pair obtained by $p_i$ at line 8, i.e., $v = H_j[x]$ and $reg_i[j].sn \geq x$ when $read[i, j, x]$ terminates.

As $write[j, y]$ defines $H_j[y]$, it follows that a message WRITE$(-, y)$ is r-delivered from $p_j$ at each correct process $p_k$ which executes $reg_k[j] \leftarrow \langle -, y\rangle$ at line 13. As this occurs after $read[i, j, x]$ has terminated, we necessarily have $x < y$. $\square_{Lemma\ 6}$

**Lemma 7** *Let $p_i$ and $p_j$ be correct processes. If $write[i, x]$ terminates before $read[j, i, y]$ starts, we have $x \leq y$.*

**Proof** Let $p_i$ a correct process that returns from its $x^{\text{th}}$ invocation of $REG[i]$.write(). It follows from line 1 that the sequence number $x$ is associated with the written value. It follows from the r-broadcast of the message WRITE$(v, x)$ issued by $p_i$ (line 2), and its r-delivery (line 12) at each correct process (RB-uniformity of the r-broadcast), that $p_i$ receives $(n-t)$ messages WRITE_DONE$(x)$ (line 3). Let $Q_1$ be this set of $(n-t)$ processes that sent these messages (line 14). Let us notice that there are at least $(n-2t)$ correct processes in $Q_1$ and, due to line 13, any of them, say $p_k$, is such that $reg_k[i].sn \geq x$.

Let $p_j$ be a correct process that invokes $REG[i]$.read(). The freshness predicate of line 7 blocks $p_j$ until $reg_j[i].sn \geq \max(wsn_1, ..., wsn_{n-t})$. Let $Q_2$ be the set of the $(n-t)$ processes that sent the messages STATE() (line 15) which allowed $p_j$ to exit the wait statement of line 7.

It follows from Lemma 2 that at least one correct process $p_k$ belongs to $Q_1 \cap Q_2$. Hence, when $p_i$ returns from $REG[i]$.write() it received the message WRITE_DONE$(x)$ from $p_k$, and we have then $reg_k[i].sn \geq x$. As $REG[i]$.read() by $p_j$ started after $REG[i]$.write() by $p_i$ terminated, when $p_k$ sends to $p_j$ the message STATE$(-, reg_k[i].sn)$, we have $reg_k[i].sn \geq x$. It follows that, when $p_j$ exits the wait statement of line 8 we have $reg_j[i].sn \geq x$, which concludes the proof of the lemma. $\square_{Lemma\ 7}$

**Lemma 8** *Let $p_i$ and $p_j$ be two correct processes. If $read[i, k, x]$ terminates before $read[j, k, y]$ starts, we have $x \leq y$.*

**Proof** Let us consider process $p_i$. When it terminates $read[i, k, x]$, if follows from the messages CATCH_UP() and CATCH_UP_DONE() (lines 9-10 and lines 16-17) that $p_i$ received the acknowledgment message CATCH_UP_DONE$(k, x)$ from $(n-t)$ different processes. Let $Q_1$ be this set of $(n-t)$ processes. Let us notice that there are at least $(n-2t)$ correct processes in $Q_1$, and for each of them, say $p_\ell$, we have $reg_\ell[k].sn \geq x$.

When $p_j$ invokes $REG[k]$.read() it broadcasts the message READ() and waits until the freshness predicate is satisfied (lines 7). The messages STATE$(-,-)$ it receives are from $(n-t)$ different processes. Let $Q_2$ be this set of $(n-t)$ processes.

It follows from Lemma 2 that at least one correct process $p_\ell$ belongs to $Q_1 \cap Q_2$. According to the fact that $read[i,k,x]$ terminates before $read[j,k,y]$ starts, it follows that $p_\ell$ sent CATCH_UP_DONE$(k,x)$ to $p_i$ before sending the message STATE$(-,s)$ to $p_j$. As $reg_\ell[k].sn$ never decreases, it follows that $x \leq s$. It finally follows that, when the freshness predicate is satisfied at $p_j$, we have $reg_j[k].sn \geq s$. As $y = reg_j[k].sn$ (lines 8-11), it follows that $x \leq y$, which concludes the proof. $\square_{Lemma\ 8}$

### 5.4 Piecing together the lemmas

**Theorem 2** *The algorithm described in Figure* 1 *implements an array of* $n$ SWMR *atomic (linearizable) registers (one register per process) in the system model* $\mathcal{BAMP}_{n,t}[t < n/3]$.

**Proof** The proof follows from Lemmas 3-8 and Theorem 1. $\square_{Theorem\ 2}$

## 6 Conclusion

This paper presented a signature-free algorithm building an array of $n$ single-writer/multi-reader atomic registers (with a register per process) in an $n$-process asynchronous message-passing system where up to $t < n/3$ processes may commit Byzantine failures.

This algorithm relies on an underlying reliable broadcast [6], an appropriate freshness predicate and a value resynchronization mechanism which ensure that a correct process always reads up-to-date values. A noteworthy property of this algorithm lies in its conceptual simplicity.

According to the result of [12] this algorithm is optimal from a $t$-resilience point of view. While the cost of a read operation is linear with respect to $n$, a problem which remains open lies in its $O(n^2)$ message complexity for write operations. This cost is due to the use of a Byzantine-tolerant reliable broadcast. Hence the question: Is it possible to reduce it, or is $O(n^2)$ a lower bound when one has to implement an atomic register in a signature-free message-passing distributed system prone to Byzantine failures? We conjecture it is a lower bound.

## Acknowledgments

This work has been partially supported by the French ANR project DISPLEXITY devoted to computability and complexity in distributed computing, the Franco-German ANR-DFG project DISCMAT (devoted to connections between mathematics and distributed computing), and the French ANR project $CO_2$Dim.

## References

[1] Aiyer A.S., Alvisi L., and Bazzi R.A., Bounded wait-free implementation of optimally resilient Byzantine storage without (unproven) cryptographic assumptions. *Proc. 21st Int'l Symposium on Distributed Computing (DISC'07)*, Springer LNCS 4731, pp. 7-19, 2007.

[2] Attiya H., Efficient and robust sharing of memory in message-passing systems. *Journal of Algorithms*, 34(1):109-127, 2000.

[3] Attiya H., Bar-Noy A. and Dolev D., Sharing memory robustly in message passing systems. *Journal of the ACM*, 42(1):121-132, 1995.

[4] Attiya H. and Bar-Or A., Sharing memory with semi-Byzantine clients and faulty storage servers. *Parallel Processing Letters*, 16(4):419-428, 2006.

[5] Attiya H. and Welch J.L., *Distributed computing: fundamentals, simulations and advanced topics, (2nd Edition)*, Wiley-Interscience, 414 pages, 2004 (ISBN 0-471-45324-2).

[6] Bracha G., Asynchronous Byzantine agreement protocols. *Information & Computation*, 75(2):130-143, 1987.

[7] Chaudhuri S., Kosa M.J., and Welch J.L., One-write algorithms for multivalued regular and atomic registers. *Acta Informatica*, 37(3):161-192, 2000.

[8] Chockler G. and Malkhi D., Active disk Paxos with infinitely many processes. *Distributed Computing*, 18(1):73-84, 2005.

[9] Dobre D., Guerraoui R., Majuntke M., Suri N., and Vukolic M., The complexity of robust atomic storage. *Proc. 30th ACM Symposium on Principles of Distributed Computing (PODC'11)*, ACM Press, pp. 59-68, 2011.

[10] Guerraoui R. and Vukolic M., How fast can a very robust read be? *Proc. 25th ACM Symposium on Principles of Distributed Computing (PODC'06)*, ACM Press, pp. 248-257, 2006.

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

[12] Imbs D., Rajsbaum S., Raynal M., and Stainer J., Reliable shared memory abstraction on top of asynchronous byzantine message-passing systems. *Proc. 21st Int. Colloquium on Structural Information and Communication Complexity (SIROCCO'14)*, Springer LNCS 8576, pp. 37-53, 2014.

[13] Ittai A., Chockler G., Keidar I., and Malkhi D., Byzantine disk paxos: optimal resilience with byzantine shared memory. *Distributed Computing*, 18(5):387-408, 2006.

[14] Lamport L., Shostack R., and Pease M., The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3)-382-401, 1982.

[15] Lamport L., On interprocess communication, Part I: basic formalism. *Distributed Computing*, 1(2):77-85, 1986.

[16] Lamport L., On interprocess communication, Part II: algorithms. *Distributed Computing*, 1(2):77-101, 1986.

[17] Malkhi D. and Reiter M., Secure and scalable replication in Phalanx. *Proc. 17th IEEE Symposium on Reliable Distributed Systems (SRDS'98)*, IEEE Press, pp. 51-58, 1998.

[18] Martin J.-Ph. and Alvisi L., A framework for dynamic Byzantine storage. *Proc. Int'l Conference on Dependable Systems and Networks (DSN'04)*, IEEE Press, pp. 325-334, 2004.

[19] Misra J., Axioms for memory access in asynchronous hardware systems. *ACM Transactions on Programming Languages and Systems*, 8(1):142-153, 1986.

[20] Mostéfaoui A. and Raynal M., Communication and agreement abstractions in the presence of Byzantine processes. To appear in *IEEE Transactions on Parallel and Distributed Systems*, 2016.

[21] Pease M., R. Shostak R., and Lamport L., Reaching agreement in the presence of faults. *Journal of the ACM*, 27:228-234, 1980.

[22] Raynal M., *Communication and agreement abstractions for fault-tolerant asynchronous distributed systems*. Morgan & Claypool Publishers, 251 pages, 2010 (ISBN 978-1-60845-293-4).

[23] Raynal M., *Concurrent programming: algorithms, principles and foundations*. Springer, 515 pages, 2013 (ISBN 978-3-642-32026-2).

# A A Reliable Broadcast Algorithm

The r-broadcast algorithm presented in Figure 2 is Bracha's algorithm [6] enriched with sequence numbers. Each process $p_i$ manages a local array $next_i[1..n]$, where $next_i[j]$ is the sequence number $sn$ of the next application message (namely, $\text{APP}(-, sn)$) from $p_j$, that $p_i$ will process (line 3). Initially, for all $i, j$, $next_i[j] = 1$.

When a process $p_i$ invokes R_broadcast $\text{APP}(v, sn)$, it broadcasts the message $\text{APP}(v, sn)$ (line 1) where $sn$ is its next sequence number. On its "server" role, the behavior of a process $p_i$ is as follows.

- When a process $p_i$ receives a message $\text{APP}(v, sn)$ from a process $p_j$ for the first time, it first waits until it can process this message (line 3). Process $p_i$ then broadcasts a message $\text{ECHO}(j, v, sn)$ (line 4). If the message just received is not the first message $\text{APP}(-, sn)$, $p_j$ is Byzantine and the message is discarded.
- Then, when $p_i$ has received the same message $\text{ECHO}(j, v, sn)$ from "enough" processes (where "enough" means "more than $(n + t)/2$ different processes"), and has not yet broadcast a message $\text{READY}(j, v, sn)$, it does it (lines 6-9).

  The aim of (a) the messages $\text{ECHO}(j, v, sn)$, and (b) the cardinality "greater than $(n + t)/2$ processes", is to ensure that no two correct processes can r-deliver distinct messages from $p_j$ (in the case where $p_j$ is Byzantine). The aim of the messages $\text{READY}(j, v, sn)$ is related to the liveness of the algorithm. Namely, its aim is to allow (at least when $p_j$ is correct) the r-delivery by the correct processes of the very same message $\text{APP}(v, sn)$ from $p_j$, and this must always occur if $p_j$ is correct. It is nevertheless possible that a message r-broadcast by a Byzantine process $p_j$ be never r-delivered by the correct processes.

```
operation R_broadcast APP(v, sn):
(1)  broadcast APP(v, sn).

when a message APP(v, sn) from p_j is received:
(2)  if no message APP(−, sn)) received from p_j
(3)     then wait (next_i[j] = sn);
(4)          broadcast ECHO(j, v, sn)
(5)  end if.

when a message ECHO(j, v, sn) is received:
(6)  if   ECHO(j, v, sn) received from strictly more than (n+t)/2 different processes
(7)    ∧ READY(j, v, sn) never sent
(8)    then broadcast READY(j, v, sn)
(9)  end if.

when a message READY(j, v, sn) is received:
(10) if   READY(j, v, sn) received from at least t + 1 different processes
(11)   ∧ READY(j, v, sn) never sent
(12)   then broadcast READY(j, v, sn)
(13) end if;
(14) if   READY(j, v, sn) received from at least 2t + 1 different processes
(15)   ∧ APP(v, sn) by p_j never R_delivered
(16)   then R_deliver APP(v, sn) from p_j;
(17)        next_i[j] ← next_i[j] + 1
(18) end if.
```

Figure 2: Reliable Broadcast in $\mathcal{BAMP}_{n,t}[t < n/3]$, (code for process $p_i$)

- Finally, when $p_i$ has received the message $\text{READY}(j, v, sn)$ from $(t + 1)$ different processes, it broadcasts the same message $\text{READY}(j, v, sn)$, it not yet done. This is required to ensure the RB-termination property. If $p_i$ has received "enough" messages $\text{READY}(j, v, sn)$ (as before "enough" means "from more than $(n + t)/2$ different processes"), it r-delivers the message $\text{APP}(v, sn)$ r-broadcast by $p_j$.

Proofs that this algorithm satisfies the properties defining the reliable broadcast abstraction can be found in [6, 20].