Two-Bit Messages are Sufficient to Implement Atomic Read/Write Registers in Crash-prone Systems

Achour Mostefaoui, Michel Raynal

Two-Bit Messages are Sufficient to Implement Atomic Read/Write Registers in Crash-prone Systems

Achour Mostéfaoui[†], Michel Raynal[⋆,‡]

[†]LINA, Université de Nantes, 44322 Nantes, France
[⋆]Institut Universitaire de France
[‡]IRISA, Université de Rennes, 35042 Rennes, France
achour.mostefaoui@univ-nantes.fr raynal@irisa.fr

Abstract

Atomic registers are certainly the most basic objects of computing science. Their implementation on top of an n-process asynchronous message-passing system has received a lot of attention. It has been shown that $t < n/2$ (where t is the maximal number of processes that may crash) is a necessary and sufficient requirement to build an atomic register on top of a crash-prone asynchronous message-passing system. Considering such a context, this paper presents an algorithm which implements a single-writer multi-reader atomic register with four message types only, and where no message needs to carry control information in addition to its type. Hence, two bits are sufficient to capture all the control information carried by all the implementation messages. Moreover, the messages of two types need to carry a data value while the messages of the two other types carry no value at all. As far as we know, this algorithm is the first with such an optimality property on the size of control information carried by messages. It is also particularly efficient from a time complexity point of view.

Keywords: Asynchronous message-passing system, Atomic read-write register, Message type, Process crash failure, Sequence number, Upper bound.

1 Introduction

Since Sumer time [9], and –much later– Turing's machine tape [20], read/write objects are certainly the most basic communication objects. Such an object, usually called a *register*, provides its users (processes) with a write operation which defines the new value of the register, and a read operation which returns the value of the register. When considering sequential computing, registers are universal in the sense that they allow to solve any problem that can be solved [20].

Register in message-passing systems In a message-passing system, the computing entities communicate only by sending and receiving messages transmitted through a communication network. Hence, in such a system, a register is not a communication object given for free, but constitutes a communication abstraction which must be built with the help of the underlying communication network and the local memories of the processes.

Several types of registers can be defined according to which processes are allowed to read or write the register, and the quality (semantics) of the value returned by each read operation. We consider here registers which are single-writer multi-reader (SWMR), and atomic. Atomicity means that (a) each read or write operation appears as if it had been executed instantaneously at a single point of the time line, between its start event and its end event, (b) no two operations appear at the same point of the time line, 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) [10]. Algorithms building multi-writer multi-reader (MWMR) atomic registers from single-writer single-reader (SWSR) registers with a weaker semantics (safe or regular registers) have been introduced by L. Lamport in [10, 11] (such algorithms are described in several papers and textbooks, e.g., [4, 12, 18, 21]).

Many distributed algorithms have been proposed, which build a register on top of a message-passing system, be it failure-free or failure-prone. In the failure-prone case, the addressed failure models are the process crash failure model, or the Byzantine process failure model (see, the textbooks [4, 12, 16, 17]). The most famous of these algorithms was proposed by H. Attiya, A. Bar-Noy, and D. Dolev in [3]. This algorithm, which is usually called ABD according to the names of its authors, considers an n-process asynchronous system in which up to $t < n/2$ processes may crash (it is also shown in [3] that $t < n/2$ is an upper bound of the number of process crashes which can be tolerated). This simple and elegant algorithm, relies on (a) quorums [22], and (b) a simple broadcast/reply communication pattern. ABD uses this pattern once in a write operation, and twice in a read operation implementing an SWMR register (informal presentations of ABD can be found in [2, 19]).

Content of the paper ABD and its successors (e.g., [1, 15, 22]) associate an increasing sequence number with each value that is written. This allows to easily identify each written value. Combined with the use of majority quorums, this value identification allows each read invocation to return a value that satisfies the atomicity property (intuitively, a read always returns the "last" written value).

Hence, from a communication point of view, in addition to the number of messages needed to implement a read or a write operation, important issues are the number of different message types, and the size of the control information that each of them has to carry. As sequence numbers increase according to the number of write invocations, this number is not bounded, and the size of a message that carries a sequence number can become arbitrarily large.

A way to overcome this drawback consists in finding a modulo-based implementation of sequence numbers [8], which can be used to implement read/write registers. Considering this approach, one of the algorithms presented in [3] uses messages that carry control information whose size is upper bounded by $O(n^5)$ bits (where n is the total number of processes). The algorithm presented in [1] reduced this size to $O(n^3)$ bits. Hence the natural question: "*How many bits of control information, a message has to carry, when one wants to implement an atomic read/write register?*".

This is the question that gave rise to this paper, which shows that it is possible to implement an SWMR atomic register with four types of message carrying no control information in addition to their type. Hence, the result: *messages carrying only two bits of control information are sufficient to implement an SWMR atomic register in the presence of asynchrony and up to $t < n/2$ unexpected process crashes*. Another important property of the proposed algorithm lies in its time complexity, namely, in a failure-free context and assuming a bound Δ on message transfer delays, a write operation requires at most 2Δ time units, and a read operation requires at most 4Δ time units.

Roadmap The paper is made up of 5 sections. The computing model and the notion of an atomic register are presented in Section 2. The algorithm building an SWMR atomic register, where messages carry only two bits of control information (their type), in an asynchronous message-passing system prone to any minority of process crashes is presented in Section 3. Its proof appears in Section 4. Finally, Section 5 concludes the paper.

2 Computation Model and Atomic Read/Write Register

2.1 Computation model

Processes The computing model is composed of a set of n sequential processes denoted p_1, ..., p_n. Each process is asynchronous which means that it proceeds at its own speed, which can be arbitrary and remains always unknown to the other processes.

A process may halt prematurely (crash failure), but executes correctly its local algorithm until it possibly crashes. The model parameter t denotes the maximal number of processes that may crash in a run. A process that crashes in a run is said to be *faulty*. Otherwise, it is *correct* or *non-faulty*. Given a run, $\mathcal{C}$ denotes the set of correct processes.

Communication Each pair of processes communicate by sending and receiving messages through two uni-directional channels, one in each direction. Hence, the communication network is a complete network: any process p_i can directly send a message to any process p_j.A process p_i invokes the operation "send $\text{TYPE}(m)$ to p_j" to send to p_j the message m, whose type is TYPE. The operation "receive TYPE() from p_j" allows p_i to receive from p_j a message whose type is TYPE.

Each channel is reliable (no loss, corruption, nor 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 on message transit times).

Let us notice that, due to process and message asynchrony, no process can know if an other process crashed or is only very slow.

Notation In the following, the previous computation model is denoted $\mathcal{CAMP}_{n,t}[\emptyset]$ (unconstrained $\mathcal{C}$rash $\mathcal{A}$synchronous $\mathcal{M}$essage-$\mathcal{P}$assing).

2.2 Atomic read/write register

Definition A *concurrent object* is an object that can be accessed by several processes (possibly simultaneously). An SWMR *atomic* register (say REG) is a concurrent object which provides exactly one process (called the writer) with an operation denoted $REG.\mathsf{write}()$, and all processes with an operation denoted $REG.\mathsf{read}()$. When the writer invokes $REG.\mathsf{write}(v)$ it defines v as being the new value of REG. An SWMR atomic register is defined by the following set of properties [10].

- Liveness. An invocation of an operation by a correct process terminates.
- Consistency (safety). All the operations invoked by the processes, except possibly –for each faulty process– the last operation it invoked, appear as if they have been executed sequentially and this sequence of operations is such that:

– each read returns the value written by the closest write that precedes it (or the initial value of REG if there is no preceding write),
– if an operation $op1$ terminates before an operation $op2$ starts, then $op1$ appears before $op2$ in the sequence.

This set of properties states that, from an external observer point of view, the read/write register appears as if it is accessed sequentially by the processes, and this sequence (a) respects the real time access order, and (ii) belongs to the sequential specification of a register. More formal definitions can be found in [10, 14]. (When considering any object defined by a sequential specification, atomicity is also called linearizability [7], and it is then said that the object is *linearizable*.)

Necessary and sufficient condition The constraint $(t < n/2)$ is a necessary and sufficient condition to implement an atomic read/write register in $\mathcal{CAMP}_{n,t}[\emptyset]$ [3]. Hence, the corresponding constrained model is denoted $\mathcal{CAMP}_{n,t}[t < n/2]$.

3 An Algorithm with Two-Bit Messages

A distributed algorithm implementing an SWMR atomic register in $\mathcal{CAMP}_{n,t}[t < n/2]$ is described in Figure 1. As already indicated, this algorithm uses only four types of messages, denoted WRITE0$()$, WRITE1$()$, READ$()$, and PROCEED$()$. The messages WRITE0$()$ and WRITE1$()$ carry a data value, while the messages READ$()$ and PROCEED$()$ carry only their type.

3.1 Notation and underlying principles

Notation p_w denotes the writer process, v_x denotes the x^{th} value written by p_w, and v_0 is the initial value of the register REG that is built.

Underlying principles The principle that underlies the algorithm is the following. First, each process (a) manages a local copy of the sequential history made up of the values written by the writer, and (b) forwards, once to each process, each new value it learns. Then, in order that all processes obtain the same sequential history, and be able to read up to date values, each process p_i follows rules to forward a value to another process p_j, and manages accordingly appropriate local variables, which store sequence numbers.

- Rule R1. When, while it knows the first $(x - 1)$ written values, and only them, p_i receives the x^{th} written value, it forwards it to all the processes that, from its point of view, know the first $(x - 1)$ written values and no more. In this way, these processes will learn the x^{th} written value (if not yet done when they receive the corresponding message forwarded by p_i).
- Rule R2. The second forwarding rule is when p_i receives the x^{th} written value from a process p_j, while it knows the first y written values, where $y > x$. In this case, p_i sends the $(x + 1)^{th}$ written value to p_j, and only this value, in order p_j increases its local sequential history with its next value (if not yet done when it receives the message from p_i).
- Rule R3. To ensure a correct management of the local histories, and allow a process to help other processes in the construction of their local histories (Rules R1 and R2), each process manages a sequence number-based local view of the progress of each other process (as far as the construction of their local history is concerned).

As we are about to see, translating these rules into an algorithm, provides us with a distributed algorithm where, while each process locally manages sequence numbers, the only control information carried by each message is its type, the number of different message types being very small (namely 4, as already indicated)[1].

[1]Such a constant number of message types is not possible from a "modulo $f(n)$" implementation of sequence numbers carried by messages. This is because, from a control information point of view, each of the values in $\{0, 1, \ldots, f(n) - 1\}$ defines a distinct message type.

3.2 Local data structures

Each process p_i manages the following local data structures.

- $history_i$ is the prefix sequence of the values already written, as known by p_i; $history_i$ is accessed with an array like-notation, and we have $history_i[0] = v_0$. As there is a single writer p_w, $history_w$ represents the history of the values written so far.
- $w\_sync_i[1..n]$ is an array of sequence numbers; $w\_sync_i[j] = \alpha$ means that, to p_i's knowledge, p_j knows the prefix of $history_w$ until $history_w[\alpha]$. Hence, $w\_sync_i[i]$ is the sequence number of the most recent value known by p_i, and $w\_sync_w[w]$ is the sequence number of the last value written (by p_w).
- $r\_sync_i[1..n]$ is an array of sequence numbers; $r\_sync_i[j] = \alpha$ means that, to p_i's knowledge, p_j answered α of its read requests.
- wsn, rsn and sn are auxiliary local variables, the scope of each being restricted to the algorithm implementing an operation, or the processing of a message, in which it occurs.

3.3 Channel behavior with respect to the message types WRITE0() and WRITE1()

As far as the messages WRITE0() and WRITE1() are concerned, the notation WRITE$(0, v)$ is used for WRITE0(v), and similarly, WRITE$(1, v)$ is used for WRITE1(v).

When considering the two uni-directional channels connecting p_i and p_j, the algorithm, as we will see, requires (a) p_i to send to p_j the sequence of messages WRITE$(1, v_1)$, WRITE0$(0, v_2)$, WRITE$(1, v_3)$, ..., WRITE$(x \bmod 2, v_x)$, etc., and (b) p_j to send to p_i the very same sequence of messages WRITE$(1, v_1)$, WRITE0$(0, v_2)$, WRITE$(1, v_3)$, ..., WRITE$(x \bmod 2, v_x)$, etc.

Moreover, the algorithm forces process p_i to send to p_j the message WRITE$(x \bmod 2, v_x)$, only when it has received from p_j the message WRITE$((x-1) \bmod 2, v_{x-1})$. From the point of view of the write messages, these communication rules actually implement the *alternating bit* protocol [6, 13], which ensures the following properties:

- Property P1: each of the two uni-directional channels connecting p_i and p_j allows at most one message WRITE$(-,-)$ to bypass another message WRITE$(-,-)$, which, thanks to the single control bit carried by these messages allows the destination process (e.g., p_i) to process the messages WRITE$(-,-)$ it receives from (e.g., p_j) in their sending order.
- Property P2: p_i and p_j are synchronized in such a way that $0 \leq |w\_sync_i[j] - w\_sync_j[i]| \leq 1$. This is the translation of Property P1 in terms of the pair of local synchronization-related variables $\langle w\_sync_i[j], w\_sync_j[i] \rangle$.

Let us insist on the fact that this "alternating bit" message exchange pattern is only on the write messages. It imposes no constraint on the messages of the types READ() and PROCEED() exchanged between p_i and p_j, which can come in between, at any place in the sequence of the write messages sent by a process p_i to a process p_j.

3.4 The algorithm implementing the write() operation

This algorithm is described at lines 1-4, executed by the writer p_w, and line 11-18, executed by any process.

Invocation of the operation write() When p_w invokes write(v_x) (we have then $w\_sync_w[w] = x - 1$), it increases $w\_sync_w[w]$ and writes v_x at the tail of its local history variable (line 1). This value is locally identified by its sequence number $x = wsn$.

Then p_w sends the message WRITE(b, v_x), where $b = (wsn \bmod 2)$, to each process p_j that (from its point of view) knows all the previous write invocations, and only to these processes. According to the definition of $w\_sync_w[1..n]$, those are the processes p_j such that $w\_sync_w[j] = wsn - 1 = w\_sync_w[w] - 1$

(line 2). Let us notice that this ensures the requirement p_i needs to satisfy when it sends a message in order to benefit from the properties provided by the alternating bit communication pattern.

Finally, p_w waits until it knows that a quorum of at least $(n-t)$ processes knows the value v_x is it writing. The fact that a process p_j knows this x^{th} value is captured by the predicate $w\_sync_w[j] = wsn(= x)$ (line 3).

local variables initialization:
$history_i[0] \leftarrow v_0$; $w\_sync_i[1..n] \leftarrow [0, \dots, 0]$; $r\_sync_i[1..n] \leftarrow [0, \dots, 0]$.

operation write(v) **is** % invoked by $p_i = p_w$ (the writer) %
(1) $wsn \leftarrow w\_sync_w[w] + 1$; $w\_sync_w[w] \leftarrow wsn$; $history_w[wsn] \leftarrow v$; $b \leftarrow wsn$ mod 2;
(2) **for each** j such that $w\_sync_w[j] = wsn - 1$ **do** send WRITE(b, v) to p_j **end for**;
(3) wait $\big(z \geq (n-t)$ where z is the number of processes p_j such that $w\_sync_w[j] = wsn\big)$;
(4) return()
end operation.

operation read() **is** % the writer can directly returns $history_i[w\_synch_i[i]]$ %
(5) $rsn \leftarrow r\_sync_i[i] + 1$; $r\_sync_i[i] \leftarrow rsn$;
(6) **for each** $j \in \{1, ...n\} \setminus \{i\}$ **do** send READ() to p_j **end for**;
(7) wait $\big(z \geq (n-t)$ where z is the number of processes p_j such that $r\_sync_i[j] = rsn\big)$;
(8) let $sn = w\_sync_i[i]$;
(9) wait $\big(z \geq (n-t)$ where z is the number of processes p_j such that $w\_sync_i[j] \geq sn\big)$;
(10) return($history_i[sn]$)
end operation.
%---

when WRITE(b, v) **is received from** p_j **do**
(11) wait $\big(b = (w\_sync_i[j] + 1)$ mod $2\big)$;
(12) $wsn \leftarrow w\_sync_i[j] + 1$;
(13) **if** $(wsn = w\_sync_i[i] + 1)$
(14) **then** $w\_sync_i[i] \leftarrow wsn$; $history_i[wsn] \leftarrow v$; $b \leftarrow wsn$ mod 2;
(15) **for each** ℓ such that $w\_sync_i[\ell] = wsn - 1$ **do** send WRITE(b, v) to p_ℓ **end for**
(16) **else** **if** $(wsn < w\_sync_i[i])$ **then** $b \leftarrow (wsn + 1)$ mod 2; send WRITE($b, history_i[wsn + 1]$) to p_j **end if**
(17) **end if**;
(18) $w\_sync_i[j] \leftarrow wsn$.

when READ() **is received from** p_j **do**
(19) $sn \leftarrow w\_sync_i[i]$;
(20) wait $(w\_sync_i[j] \geq sn)$;
(21) send PROCEED() to p_j.

when PROCEED() **is received from** p_j **do**
(22) $r\_sync_i[j] \leftarrow r\_sync_i[j] + 1$.

Figure 1: Single-writer multi-reader atomic register in $\mathcal{CAMP}_{n,t}[t < n/2]$ with counter-free messages

Reception of a message WRITE(b, v) **from a process** p_j When p_i receives a message WRITE(b, v) from a process p_j, it first waits until the waiting predicate of line 11 is satisfied. This waiting statement is nothing else than the the reception part of the alternating bit algorithm, which guarantees that the messages WRITE$()$ from p_j are processed in their sending order. When, this waiting predicate is satisfied, all messages sent by p_j before WRITE(b, v) have been received and processed by p_i, and consequently the message WRITE(b, v) is the swn^{th} message sent by p_j (FIFO order), where $wsn = w\_sync_i[j]+1$, which means that $history_j[wsn] = v$ (line 12).

When this occurs, p_i learns that v is the next value to be added to its local history if additionally we have $w\_sync_i[i] = wsn - 1$. In this case (predicate of line 13), p_i (a) adds v at the tail of its history (line 14), and (b) forwards the message WRITE(b, v) to the processes that, from its local point of view, know the first $(wsn - 1)$ written values and no more (line 15, forwarding Rule R1).

If $wsn < w\_sync_i[i]$, from p_i's local point of view, the history known by p_j is a strict prefix of its own history. Consequently, p_i sends to p_j the message $\text{WRITE}(b', v')$, where $b' = ((wsn + 1) \bmod 2)$ and $v' = history_i[wsn + 1]$ (line 16 applies the forwarding Rule R2 in order to allow p_j to catch up its lag, if not yet done when it will receive the message $\text{WRITE}(b', v')$ sent by p_i). Finally, as p_j sends to p_i a single message per write operation, whatever the value of wsn, p_i updates $w\_sync_i[j]$ (line 18).

Remark As far as the written values are concerned, the algorithm implementing the operation write() can be seen as a fault-tolerant "synchronizer" (in the spirit of [5]), which ensures the mutual consistency of the local histories between any two neighbors with the help of an alternating bit algorithm executed by each pair of neighbors [6, 13].

3.5 The algorithm implementing the read() operation

This algorithm is described at lines 5-10 executed by a reader p_i, and lines 19-22 executed by any process.

Invocation of the operation read() The invoking process p_i first increments its local read request sequence number $r\_sync_i[i]$ and broadcasts its read request in a message $\text{READ}()$, which carries neither additional control information, nor a data value (lines 5-6). If p_i crashes during this broadcast, the message $\text{READ}()$ is received by an arbitrary subset of processes (possibly empty). Otherwise, p_i waits until it knows that at least $(n - t)$ processes received its current request (line 7).

When this occurs, p_i considers the sequence number of the last value in its history, namely $sn = w\_sync_i[i]$ (line 8). This is the value it will return, namely $history_i[sn]$ (line 10). But in order to ensure atomicity, before returning $history_i[sn]$, p_i waits until at least $(n - t)$ processes know this value (and may be more). From p_i's point of view, the corresponding waiting predicate translates in "at least $(n - t)$ processes p_j are such that $w\_sync_i[j] \geq sn$".

Reception of a message $\text{READ}()$ **sent by a process** p_j When a process p_i receives a message $\text{READ}()$ from a process p_j (hence, p_j issued a read operation), it considers the most recent written value it knows (the sequence number of this value is $sn = w\_sync_i[i]$, line 19), and waits until it knows that p_j knows this value, which is locally captured by the sequence number-based predicate $w\_sync_i[j] \geq sn$ (line 20). When this occurs, p_i sends the message $\text{PROCEED}()$ to p_j which is allowed to progress as far as p_i is concerned.

The control messages $\text{READ}()$ and $\text{PROCEED}()$ (whose sending is controlled by a predicate) implement a synchronization which –as far as p_i is concerned– forces the reader process p_j to wait until it knows a "fresh" enough value, where "freshness" is locally defined by p_i as the last value it was knowing when it received the message $\text{READ}()$ from p_j (predicate of line 20).

Reception of a message $\text{PROCEED}()$ **sent by a process** p_j When p_i receives a message $\text{PROCEED}()$ from a process p_j, it learns that its local history is as fresh as p_j's history when p_j received its message $\text{READ}()$. Locally, this is captured by the incrementation of $r\_sync_i[j]$, namely p_j answered all the read requests of p_i until the $(r\_sync_i[j])^{th}$ one.

4 Proof of the Algorithm

Let us remind that $\mathcal{C}$ is the set of correct processes, p_w the writer, and v_x the x^{th} value written by p_w. Due to page limitation, the missing proofs are given in an Appendix.

Lemma 1 $\forall i, j$: $w\_sync_i[j]$ *increases by steps equal to* 1.

Lemma 2 $\forall i, j$: $w\_sync_i[i] \geq w\_sync_j[i]$.

Lemma 3 $\forall i$: $w\_sync_i[i] = \max\{w\_sync_i[j]\}_{1 \leq j \leq n}$.

Lemma 4 $\forall i$: $history[0..w\_sync_i[i]]$ *is a prefix of* $history[0..w\_sync_w[w]]$.

Lemma 5 $\forall i \in \mathcal{C}, \forall j$: *we have:*
R1: $(w\_sync_i[i] = w\_sync_i[j] = x) \Rightarrow p_i$ *sent* x *messages* WRITE$(-,-)$ *to* p_j,
R2: $(w\_sync_i[i] > w\_sync_i[j] = x) \Rightarrow p_i$ *sent* $x+1$ *messages* WRITE$(-,-)$ *to* p_j.

Lemma 6 $\forall i, j \in \mathcal{C}$, *if* $w\_sync_i[i] = x$, *there is a finite time after which* $w\_sync_i[j] \geq x$.

Proof Let us first notice that, due to Lemma 7, all WRITE$(-,-)$ messages received by correct processes will eventually satisfy the predicate line 11 and will be processed.

The proof is by contradiction. Let us assume that there exists some correct process p_j such that $w\_sync_i[j]$ stops increasing forever at some value $y < x$. Let us first notice that there is no message WRITE$(-,-)$ in transit from p_j to p_i otherwise its reception by p_i will entail the incrementation of $w\_sync_i[j]$ from y to $y+1$, contradicting the assumption. So, let us consider the last message WRITE$(-,-)$ sent by p_j to p_i and processed by p_i. There are three cases to consider when this message is received by p_i at line 11. (Let us remind that, due to to Lemma 3, $w\_sync_i[i] \geq w\_sync_i[j]$.)

- Case 1. $w\_sync_i[i] = w\_sync_i[j] = y-1 < x-1$. The variables $w\_sync_i[i]$ and $w\_sync_i[j]$ are both incremented at lines 14 and 18 respectively to the value $y < x$. As by assumption, $w\_sync_i[i]$ will attain the value x, it will be necessarily incremented in the future to reach x. The next time $w\_sync_i[i]$ is incremented, a message WRITE$(-,-)$ is sent by p_i to p_j (at line 15). Due to Lemma 5, p_i sent $y+1$ messages WRITE$(-,-)$ to p_j and eventually $w\_sync_j[i]$ will be equal to $y+1$. When the last of these messages arrives and is processed by p_j, there are two cases.
  - Case $w\_sync_j[j] = y$ (as p_i sent $y+1$ messages WRITE$(-,-)$ to p_j, $w\_sync_j[j]$ cannot be smaller than y). In this case, $w\_sync_j[j] = y$ is increased, and a message WRITE$(-,-)$ is necessarily sent by p_j to p_i (line 15). This contradicts the assumption that the message we considered was the last message sent by p_j to p_i.
  - Case $w\_sync_j[j] \geq y+1$. In this case, as p_i sent previously y messages to p_j, we necessarily have $w\_sync_j[i] = y$. In this case, the predicate of line 13 is false, while the one of line 16 is satisfied. Hence, p_j sends a message WRITE$(-,-)$ to p_i. A contradiction.

- Case 2. $w\_sync_i[i] = w\_sync_i[j] + 1 = y < x$. In this case, when p_i receives the last message WRITE$(-,-)$ from p_j, the variable $w\_sync_i[j]$ is incremented at line 18 to the value $y < x$. Moreover, by the contradiction assumption, no more message WRITE$(-,-)$ is sent by p_j to p_i.

  Hence, we have now $w\_sync_i[i] = w\_sync_i[j] = y < x$, and the variable $w\_sync_i[i]$ will be incremented in the future to reach x. A reasoning similar to the previous one shows that p_j will send a message WRITE$(-,-)$ to p_i in the future, which contradicts the initial assumption.

- Case 3. $w\_sync_i[i] > w\_sync_i[j] + 1$. The reception by p_i of the last message WRITE$(-,-)$ from p_j entails the incrementation of $w\_sync_i[j]$ to its next value. However as $w\_sync_i[i] > w\_sync_i[j]$ remains true, a message WRITE$(-,-)$ is sent by p_i to p_j at line 16. Similarly to the previous cases, the reception of this message by p_j will direct it to send another message WRITE$(-,-)$ to p_i, contradicting the initial assumption.

Hence, $w\_sync_i[j]$ cannot stop increasing before reaching x, which proves the lemma. $\square_{Lemma\ 6}$

Lemma 7 *No correct process blocks forever at line* 11.

Lemma 8 *If the writer does not crash during a write operation, it terminates it.*

Proof Let us first notice that, due to Lemma 7, the writer cannot block forever at line 11.

When it invokes a new write operation, the writer p_w first increases the write sequence number $w\_sync_w[w]$ to its next value wsn (line 1). If p_w does not crash, it follows from Lemma 6 that we eventually have

$w\_sync_i[i] \geq w\_sync_w[i] = wsn$ at each correct process p_i. Consequently, the writer cannot block forever at line 3 and the lemma follows. $\Box_{Lemma\ 8}$

Lemma 9 *If a process does not crash during a read operation, it terminates it.*

Proof Let us first notice that, due to Lemma 7, the reader cannot block forever at line 11.

Each time a process p_i executes a read operation it broadcasts a message READ() to all the other processes (line 6). Let us remind that its local variable $r\_sync_i[i]$ counts the number of messages READ() it has broadcast, while $r\_sync_i[j]$ counts the number of messages PROCEED() it has received from p_j (line 22) in response to its READ messages READ().

When the predicate of line 7 becomes true at the reader p_i, there are at least $(n-t)$ processes that answered the $r\_sync_i[i]$ messages READ() it sent (note that $r\_sync_i[i]$ is incremented line 5 and p_i does not send messages READ() to itself). We claim that each message READ() sent by p_i to a correct process p_j is eventually acknowledged by a a message PROCEED() send by p_j to p_i. It follows from this claim and line 22 executed by p_i when it receives a message PROCEED(), that the predicate of line 7 is eventually satisfied, and consequently, p_i cannot block forever at line 7.

Proof of the claim. Let us consider a correct process p_j when it receives a message READ() from p_i. It saves $w\_sync_i[i]$ in sn and waits until $w\_sync_j[i] \geq sn$ (lines 19-20). Due to Lemma 6, the predicate $w\_sync_j[i] \geq sn$ eventually becomes true at p_j. When this occurs, p_j sends the message PROCEED() to p_i (line 21), which proves the claim.

Let us now consider the wait statement at line 9, where sn is the value of $w\_sync_i[i]$ when the wait statement of line 7 terminates. Let p_j be a correct process. Due to Lemma 6 the predicate $w\_sync_i[j] \geq sn$ eventually holds. As this is true for any correct process p_j, p_i eventually exits the wait statement, which concludes the proof of the lemma. $\Box_{Lemma\ 9}$

Lemma 10 *The register that is built is atomic.*

Proof Let $read[i,x]$ be a read operation issued by a process p_i which returns the value with sequence number x (i.e., $history_i[x]$), and $write[y]$ be the write operation which writes the value with sequence number y (i.e., $history_w[y]$). The proof of the lemma is the consequence of the three following claims.

- Claim 1. If $read[i,x]$ terminates before $write[y]$ starts, then $x < y$.
- Claim 2. If $write[x]$ terminates before $read[i,y]$ starts, then $x \leq y$.
- Claim 3. If $read[i,x]$ terminates before $read[j,y]$ starts, then $x \leq y$.

Claim 1 states that no process can read from the future. Claim 2 states that no process can read overwritten values. Claim 3 states that there is no new/old read inversion [4, 18].

Proof of Claim 1.
Due to Lemma 4, the value returned by $read[i,x]$ is $history_i[x] = history_w[x] = v_x$. As each write generate a greater sequence number, and p_w has not yet invoked write(v_y), we necessarily have $y > x$.

Proof of Claim 2.
It follows from lines 1-3 that when $write[x]$ terminates, there is a quorum Q_w of at least $(n-t)$ processes p_i such that $w\_sync_w[j] = x$. On another side, $read[i,y]$ obtains messages PROCEED() from a quorum Q_r at least $(n-t)$ processes (lines 22 and 7). As $|Q_w| \geq n-t$, $|Q_r| \geq n-t$, and $n-t > n/2$, we have $Q_w \cap Q_r \neq \emptyset$. Let p_k be a process of $Q_w \cap Q_r$. As $w\_sync_w[k] = x$, and $w\_sync_k[k] \geq w\_sync_w[k]$ (Lemma 2), and $write[x]$ is the last write before $read[i,y]$, we have $w\_sync_k[k] = x$ when $read[i,y]$ starts.

When p_k received the message READ() from p_i, we had $w\_sync_k[k] = x$, and p_k waited until $w\_sync_k[i] \geq x$ (line 20) before sending the message PROCEED() that allowed p_i to progress in its waiting at line 7. As $w\_sync_i[i] \geq w\_sync_k[i]$ (Lemma 2), it follows that we have $w\_sync_i[i] \geq x$, when p_i computes at line 8 the sequence number sn of the value it will return at line 10). Hence, the index $y = sn$ computed by p_i at line 8 is such that $y = sn = w\_sync_i[i] \geq x$.

Proof of Claim 3.
On one side, when $read[i, x]$ stops waiting at line 9, there is a quorum Q_{ri} of at least $(n - t)$ processes p_k such that $w\_sync_i[k] \geq x$ (predicate of line 9 at p_i). Due to Lemma 2, we have then $w\_sync_k[k] \geq x$ for any process p_k of Q_{ri}, when $read[i, x]$ terminates.

On the other side, when $read[j, y]$ stops waiting at line 7 (which defines the value it returns, namely, $history_j[y]$), there is a quorum Q_{rj} of at least $(n - t)$ processes p_ℓ such that (due to the waiting predicate of line 20) $w\_sync_\ell[j] \geq sn(\ell)$, where $sn(\ell)$ is the value of $w\_sync_\ell[\ell]$ when p_ℓ receives the message READ() from p_j.

As each of Q_{ri} and Q_{rj} contains at least $(n - t)$ processes, and there is a majority of correct processes, there is at least one correct process in their intersection, say p_m. It follows that we have $w\_sync_m[m] \geq x$ when $read[i, x]$ terminates, and $w\_sync_m[j] \geq sn(m)$, where $sn(m)$ is the value of $w\_sync_m[m]$, when p_m received the message READ() from p_j. As $w\_sync_m[m]$ never decreases, and p_m receives the message READ() from p_j after $read[i, x]$ terminated, we necessarily have $sn(m) \geq x$. Hence, $w\_sync_m[j] \geq x$, when p_m sends PROCEED() to p_j. As (Lemma 2) $w\_sync_j[j] \geq w\_sync_m[j]$, it follows that the index sn computed by p_i at line 8 is such that $sn = y \geq x$. $\Box_{Lemma\ 10}$

Theorem 1 *The algorithm described in Figure* 1 *implements an* SWMR *atomic register in the system model* $\mathcal{CAMP}_{n,t}[t < n/2]$.

Proof The theorem follows from Lemma 8 and Lemma 9 (Termination properties), and Lemma 10 (Atomicity property). $\Box_{Theorem\ 1}$

Theorem 2 *The algorithm described in Figure* 1 *uses only four types of messages, and those carry no additional control information. Moreover, a read operation requires* $O(n)$ *messages, and a write operation requires* $O(n^2)$ *messages.*

Proof The message content part of the theorem is trivial. A read generates n messages READ(), and each of generates a message PROCEED(). A write operation generates $(n - 1)$ messages WRITE$(b, -)$ from the writer to the other processes, and then each process forward once this message to each process. $\Box_{Theorem\ 2}$

5 Concluding Remarks

The aim and the paper

As indicated in the introduction, our aim was to investigate the following question: "*How many bits of control information messages have to carry to implement an atomic register in* $\mathcal{CAMP}_{n,t}[t < n/2]$?".

As far as we know, all the previous works addressing this issue have reduced the size of control information with the use of a "modulo n" implementation technique. Table 1 presents three algorithms plus ours. These three algorithms are the unbounded version of the ABD algorithm [3], its bounded version, and the bounded algorithm due to H. Attiya [1]. They all associate a sequence number with each written value, but differently from ours, the last two require each message to carry a "modulo representative" of a sequence number.

For each algorithm, the table considers the number of messages it uses to implement the write operation (line 1), the read operation (line 2), the number of control bits carried by messages (line 3), the size of local memory used by each process (line 4), the time complexity of the write operation (line 5), and the time complexity of the read operation (line 6), both in a failure-free context. For time complexity it is assumed that message transfer delays are bounded by Δ, and local computations are instantaneous. The values appearing in the table for the bounded version of ABD and Attiya's algorithm are from [1, 19]. The reader can see that the proposed algorithm is particularly efficient from a time complexity point of view, namely, it is as good as the unbounded version of ABD.

line number	What is measured	ABD95 [3] unbounded seq. nb	ABD95 [3] bounded seq. nb	H. Attiya's algorithm [1]	Proposed algorithm
1	#msgs: write	$O(n)$	$O(n^2)$	$O(n)$	$O(n^2)$
2	#msgs: read	$O(n)$	$O(n^2)$	$O(n)$	$O(n)$
3	msg size (bits)	unbounded	$O(n^5)$	$O(n^3)$	2
4	local memory	unbounded	$O(n^6)$	$O(n^5)$	unbounded
5	Time: write	2Δ	12Δ	14Δ	2Δ
6	Time: read	4Δ	12Δ	18Δ	4Δ

Table 1: A few algorithms implementing an SWMR atomic register in $\mathcal{CAMP}_{n,t}[t < n/2]$

The result presented in the paper As we have seen, our algorithm also uses sequence numbers, but those remain local. Only four types of messages are used, which means that each implementation message carries only two bits of control information. Moreover, only two message types carry a data value, the other two carry no data at all. Hence, this paper answers a long lasting question: "*it is possible to implement an atomic register, despite asynchrony and crashes of a minority of processes, with messages whose control part is constant?*".

The unbounded feature of the proposed algorithm (when looking at the local memory size) is due to the fact that the algorithm introduces a fault-tolerant version of a "synchronizer"[2] suited to the implementation of an atomic register, which disseminates new values, each traveling between each pair of processes in both directions, in such a way that a strong synchronization is ensured between any pair of processes, independently from the other processes, (namely, $\forall i, j : \ 0 \leq |w\_sync_i[j] - w\_sync_j[i]| \leq 1$). This fault-tolerant synchronization is strong enough to allow sequence numbers to be eliminated from messages. Unfortunately, it does not seem appropriate to allow a local modulo-based representation of sequence numbers at each process.

In addition to its theoretical interest, and thanks to its time complexity, the proposed algorithm is also interesting from a practical point of view. Due to the $O(n)$ message cost of its read operation, it can benefit to read-dominated applications and, more generally, to any setting where the communication cost (time and message size) is the critical parameter[3].

A problem that remains open According to the previous discussion, a problem that still remains open is the following. Is it possible to design an implementation where (a) a constant number of bits is sufficient to encode the control information carried by messages, and (b) the sequence numbers have a local modulo-based implementation? We are inclined to think that this is not possible.

[2] As introduced in [5], and presented in textbooks such as [4, 12, 17].

[3] In addition to the way they use sequence numbers, an interesting design difference between our algorithm and ABD-like algorithms is the following. When a process receives a message READ(), it has two possibilities. Either send by return the last written value it knows, as done in ABD-like algorithms. Or wait until it knows that the sender has a value as up to date as it own value, and only then send it a signal, as done in our algorithm with the message PROCEED().

Acknowledgments

This work has been partially supported by the French ANR project DISPLEXITY, which is devoted to computability and complexity in distributed computing, and the Franco-German ANR project DISCMAT devoted to connections between mathematics and distributed computing.

References

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

[2] Attiya H., Robust simulation of shared memory: 20 years after. *Bulletin of the EATCS*, 100:99-113 (2010)

[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 Welch J., *Distributed computing: fundamentals, simulations and advanced topics*, (2d Edition), Wiley-Interscience, 414 pages (2004)

[5] Awerbuch B., Complexity of network synchronization. *Journal of the ACM*, 4:804-823 (1985)

[6] Bartlett K. A., Scantlebury S. A., and Wilkinson P. T., A note on reliable full-duplex transmission over half-duplex links. *Communications of the ACM*, 12(5):260-261 (1969)

[7] 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)

[8] Israeli A. and Li M., Bounded time-stamps, *Distributed Computing*, 6(4):205-209 (1993)

[9] Kramer S. N., *History Begins at Sumer: Thirty-Nine Firsts in Man's Recorded History*. University of Pennsylvania Press, 416 pages, ISBN 978-0-8122-1276-1 (1956)

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

[11] Lamport L., On interprocess communication, Part II: Algorithms. *Distributed Computing*, 1(2):86-101 (1986)

[12] Lynch N. A., *Distributed algorithms*. Morgan Kaufmann Pub., San Francisco (CA), 872 pages, ISBN 1-55860-384-4 (1996)

[13] Lynch W. C., Reliable full-duplex file transmission over half-duplex telephone lines. *Communications of the ACM*, 11(6):407-410 (1968)

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

[15] Mostéfaoui A. and Raynal M., Time-efficient read/write register in crash-prone asynchronous message-passing systems. *Tech Report 2031*, IRISA, Université de Rennes (F), (2016) https://hal.inria.fr/hal-01256067

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

[17] Raynal M., *Distributed algorithms for message-passing systems*. Springer, 510 pages, ISBN 978-3-642-38122-5 (2013)

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

[19] Ruppert E., Implementing shared registers in asynchronous message-passing systems. *Springer Encyclopedia of Algorithms*, pp. 400-403 (2008)

[20] Turing A. M., On computable numbers with an application to the Entscheidungsproblem. *Proc. of the London Mathematical Society*, 42:230-265, 1936.

[21] Vitányi P. M. B. and Awerbuch B., Atomic shared register access by asynchronous hardware (Detailed abstract). *Proc. 27th Annual Symposium on Foundations of Computer Science (FOCS'86)*, IEEE Press, pp. 233-243 (1986)

[22] Vukolic M., *Quorum systems, with applications to storage and consensus*. Morgan & Claypool Publishers, 132 pages, 2012 (ISBN 978-1-60845-683-3).

A Proof of the Lemmas 1-5 and Lemma 7

Lemma 1 $\forall i, j$: $w\_sync_i[j]$ increases by steps equal to 1.

As this lemma is used in all other lemmas, it will not be explicitly referenced.
Proof Let us first observe that, due to the sending predicates of line 2 (for the writer), and lines 15 and 16 for any process p_i, no process sends a message WRITE$(-,-)$ to itself.

As far as $w\_sync_i[i]$ is concerned, and according to the previous observation,we have the following. The writer increases $w\_sync_w[w]$ only at line 1. Any reader process p_i increases $w\_sync_i[i]$ at line 14, and due to line 12 and the predicate of line 13, the increment is 1. Let us now consider the case of $w\_sync_i[j]$ when $i \neq j$. An incrementation of such a local variable occurs only at line 18, where (due to line 12) we have $wsn = w\_sync_i[j] + 1$, and the lemma follows. $\Box_{Lemma\ 1}$

Lemma 2 $\forall i, j : w\_sync_i[i] \geq w\_sync_j[i]$.

Proof Let us first observe, that the predicate is initially true. Then, a local variable $w\_sync_j[i]$ is increased by 1, when p_j receives a message WRITE$(-,-)$ from p_i (lines 12 and 18). Process p_i sent this message at line 2 or 16 if $i = w$, and at lines 15 or 16 for any $i \neq w$. If the sending of the message WRITE$(b,-)$ by p_i occurs at line 2 or 15, p_i increased $w\_sync_i[i]$ at the previous line. If the sending occurs at line 16, $w\_sync_i[i]$ was increased during a previous message reception. $\Box_{Lemma\ 2}$

Lemma 3 $\forall i$: $w\_sync_i[i] = \max\{w\_sync_i[j]\}_{1 \leq j \leq n}$.

Proof The lemma is trivially true for the writer process p_w. Let us consider any other process p_i, different from p_w. The proof is by induction on the number of messages WRITE$(-,-)$ received by p_i. Let $P(i,m)$ be the predicate $w\_sync_i[i] = \max\{w\_sync_i[j]\}_{1 \leq j \leq n}$, where m is the number of messages WRITE$(-,-)$ processed by p_i. The predicate $P(i,0)$ is true. Let us assume $P(i,m')$ is true for any m' such that $0 \leq m' \leq m$. Let p_j be the process that sends to p_i the $(m+1)^{th}$ message WRITE$(b,-)$, and let $w\_sync_i[i] = x$ when p_i starts processing this message. There are four cases to consider.

- Case 1. When the message WRITE$(-,-)$ from p_j is processed by p_i, we have $w\_sync_i[i] + 1 = w\_sync_i[j] + 1$. As the predicate of line 13 is satisfied when this message is processed, p_i updates $w\_sync_i[i]$ to the value $(x + 1)$ at line 14. Moreover, it also updates $w\_sync_i[j]$ to the same value $(x+1)$ at line 18. As $P(i,m)$ is true, it follows that $P(i,m+1)$ is true after p_i processed the message.
- Case 2. When the message WRITE$(-,-)$ from p_j is processed by p_i, we have $w\_sync_i[j] + 1 < w\_sync_i[i] = x$. In this case, p_i does not modify $w\_sync_i[i]$. It only updates $w\_sync_i[j]$ to its next value (line 18), which is smaller than x. As $P(i,m)$ is true, it follows that $P(i,m+1)$ is true after p_i processed the message.
- Case 3. When the message WRITE$(-,-)$ from p_j is processed by p_i, we have $w\_sync_i[j] + 1 = w\_sync_i[i] = x$. In this case, both the predicates of lines 13 and 16 are false. It follows that p_i executes only the update of line 18, and we have then $w\_sync_i[j] = w\_sync_i[i] = x$. As $P(i,m)$ is true, $P(i,m+1)$ is true after p_i processed the message.
- Case 4. When the message WRITE$(-,-)$ from p_j is processed by p_i, we have $w\_sync_i[j] + 1 > w\_sync_i[i] + 1 = x + 1$. In this case, due to (a) $w\_sync_i[j] \leq w\_sync_i[i]$ (induction assumption satisfied when the message WRITE$(-,-)$ arrives at p_i from p_j), and (b) the fact that $w\_sync_i[j]$ increases by step 1 (Lemma 1), we necessarily have $w\_sync_i[i] + 1 \geq w\_sync_i[j] + 1$, when the message is received. Hence, we obtain $w\_sync_i[j] + 1 > w\_sync_i[i] + 1 \geq w\_sync_i[j] + 1$, a contradiction. It follows that this case cannot occur.

$\Box_{Lemma\ 3}$

Lemma 4 $\forall i$: $history[0..w\_sync_i[i]]$ is a prefix of $history[0..w\_sync_w[w]]$.

Proof The proof of this lemma rests on the properties P1 and P2 provided by the underlying "alternating bit" communication pattern imposed on the messages WRITE$(-,-)$ exchanged by any pair of processes p_i and p_j. If follows from these properties (obtained from the use of parity bits carried by every message WRITE$(-,-)$, and the associated wait statement of line 11) that, p_i sends to p_j the message WRITE$(-,v_x)$, only after it knows that p_j received WRITE$(-,v_{x-1})$. Moreover, it follows from the management of the local sequence numbers $w\_sync_i[1..n]$, that no process sends twice the same message WRITE$(-,v_x)$. Finally, due to the predicate of line 11, two consecutive messages WRITE$(0,-)$ and WRITE$(1,-)$ sent by a process p_i to a process p_j are processed in their sending order.

The lemma then follows from these properties, and the fact that, when at lines 13-14 a process p_i assigns a value v to $history_i[x]$, this value was carried by x^{th} message WRITE$(-,v)$ sent by some process p_j, and is the value of $history_j[x]$. It follows that no two processes have different histories, from which we conclude that $history_i[x] = history_w[x]$. $\Box_{Lemma\ 4}$

Lemma 5 $\forall i \in \mathcal{C}, \forall j$: we have:
R1: $(w\_sync_i[i] = w\_sync_i[j] = x) \Rightarrow p_i$ sent x messages WRITE$(-,-)$ to p_j,
R2: $(w\_sync_i[i] > w\_sync_i[j] = x) \Rightarrow p_i$ sent $x+1$ messages WRITE$(-,-)$ to p_j.

Proof Both predicates are initially true ($w\_sync_i[i] = w\_sync_i[j] = 0$ and no message was previously sent by p_i to p_j). The variables involved in the premises of the predicates R1 and R2 can be modified in the execution of a write operation (if p_i is the writer), or when a message WRITE$(-,-)$ arrives at process p_i from process p_j. Let us suppose that R1 and R2 are true until the value x, and let us show that they remain true for the value $(x+1)$.

During the execution of a write operation, if $w\_sync_w[w] = w\_sync_w[j] = x$, the local variable $w\_sync_w[w]$ is incremented to $(x+1)$, and the $(x+1)^{th}$ message WRITE$(-,-)$ is sent by p_w to p_j (lines 1-2). R1 and R2 remain true. If $w\_sync_w[w] > w\_sync_w[j] = x$, the local variable $w\_sync_w[w]$ is incremented at line 1, but no message is sent to p_j at line 2, which falsifies neither R1 nor R2.

When a process p_i receives a message WRITE$(-,-)$ from a process p_j, there are also two cases, according to the values of $w\_sync_i[i]$ and $w\_sync_i[j]$ when p_i starts processing the message at line 12.

- Case 1. $w\_sync_i[i] = w\_sync_i[j] = x$. In this case, the predicate of line 13 is satisfied. It follows that both $w\_sync_i[i]$ and $w\_sync_i[j]$ are incremented to $(x+1)$ (at line 14 for $w\_sync_i[i]$ and line 18 for $w\_sync_i[j]$). Moreover, when p_i executes line 15 we have $w\_sync_i[i] = w\_sync_i[j] - 1$, and consequently p_i sends a message WRITE$(-,-)$ to p_j (the fact this message is the $(x+1)^{th}$ follows from the induction assumption). Hence, R1 and R2 are true when p_i terminates the processing of the message WRITE$(-,-)$ received from p_j.
- Case $w\_sync_i[i] > w\_sync_i[j] = x$. In this case, $w\_sync_i[j]$ is incremented to $x+1$ at line 18, while $w\_sync_i[i]$ is not (because the predicate of line 13 is false). Two sub-cases are considered according to the values of $w\_sync_i[i]$ and $w\_sync_i[j]$.
  - If $w\_sync_i[i] = x+1$ (this is the value $w\_sync_i[j]$ will obtain at line 18), the predicate of line 16 is false, and no message is sent to p_j. R1 and R2 remains true, as, by the induction assumption, p_i already sent $(x+1)$ messages WRITE$(-,-)$.
  - If $w\_sync_i[i] > x+1$, the predicate of line 16 is satisfied, and the $(x+2)^{th}$ message WRITE$(-,-)$ is sent to p_j at this line, maintaining satisfied the predicates R1 and R2.

$\square_{Lemma\ 5}$

Lemma 7 No correct process blocks forever at line 11.

Proof The fact that the waiting predicate of line 11 is eventually satisfied follows from the following observations.

- As the network is reliable, all the messages that are sent are received. Due to lines 2 and 15-16, this means that, for any x, if $\text{WRITE}(-, v_x)$ is received while $m = \text{WRITE}(-, v_{x-1})$ has not, then m will be eventually received.
- The message exchange pattern involving any two messages $\text{WRITE}(0, -)$ and $\text{WRITE}(1, -)$ (sent consecutively) exchanged between each pair of processes is the "alternating bit pattern", from which it follows that no two messages $\text{WRITE}(b, -)$ (with the same b) can be received consecutively.
- It follows that the predicate of line 11 is a simple re-ordering predicate for any pair of messages such that $\text{WRITE}(-, v_x)$ was received before $\text{WRITE}(-, v_{x-1})$. When this predicate is not satisfied for a message $m = \text{WRITE}(b, -)$, this is because a message $m' = \text{WRITE}(1 - b, -)$, will necessarily arrive and be processed before m. After that, the predicate of line 11 becomes true for m.

$\square_{Lemma\ 7}$