Time-Efficient Read/Write Register in Crash-prone Asynchronous Message-Passing Systems

Achour Mostefaoui, Michel Raynal, Matthieu Roy

Time-Efficient Read/Write Register in Crash-prone Asynchronous Message-Passing Systems

Achour Mostéfaoui · Michel Raynal · Matthieu Roy

May 3, 2018

Abstract The atomic register is one of the most basic and useful object of computing science, and its simple read-write semantics is appealing when programming distributed systems. Hence, its implementation on top of crash-prone asynchronous message-passing systems has received a lot of attention. It was shown that having a strict minority 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.

This paper visits the notion of a fast implementation of an atomic register, and presents a new time-efficient asynchronous algorithm that reduces latency in many cases: a write operation always costs a round-trip delay, while a read operation costs a round-trip delay in favorable circumstances (intuitively, when it is not concurrent with a write). When designing this algorithm, the design spirit was to be as close as possible to the original algorithm proposed by Attiya, Bar-Noy, and Dolev.

Keywords: Asynchronous message-passing system, Atomic read/write register, Concurrency, Fast operation, Process crash failure, Synchronous behavior, Time-efficient operation.

1 Introduction

Since Sumer time [8], and –much later– Turing's machine tape [15], read/write objects are certainly the most basic memory-based communication objects. Such an

A. Mostéfaoui
LINA, Université de Nantes, Nantes, France E-mail: achour.mostefaoui@univ-nantes.fr

M. Raynal
IRISA, Université de Rennes 1, France & Department of Computing, Hong Kong Polytechnic University
E-mail: michel.raynal@irisa.fr

M. Roy
CNRS, LAAS-CNRS, Université de Toulouse, Toulouse, France E-mail: roy@laas.fr

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 us to solve any problem that can be solved [15]. Although named differently in recent architectures such as multi-core systems, the read/write semantics of a register is the most clean and easy to understand abstraction of shared memory and is extensively used in multi-threaded programs.

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 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 it, 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 is 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) [9]. Algorithms building multi-writer multi-reader (MWMR) atomic registers from single-writer single-reader (SWSR) registers with a weaker semantics (safe or regular registers) are described in several textbooks (e.g., [3, 10, 13]).

Many distributed algorithms have been proposed to 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 [3, 10, 12, 14]). The most famous of these algorithms was proposed by H. Attiya, A. Bar-Noy, and D. Dolev in [2]. This algorithm, which is usually called ABD according to the names its authors, considers an n-process asynchronous system in which up to $t < n/2$ processes may crash (it is also shown in [2] 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 [16], 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.

Fast operation. To our knowledge, the notion of a *fast implementation* of an atomic register operation, in failure-prone asynchronous message-passing systems, was introduced in [5] for process crash failures, and in [6] for Byzantine process failures. These papers consider a three-component model, namely there are three different types of processes: a set of writers W, a set of readers R, and a set of servers S which implements the register. Moreover, a client (a writer or a reader) can communicate only with the servers, and the servers do not communicate among themselves.

In these papers, *fast* means that a read or write operation must entail exactly one communication round-trip delay between a client (the writer or a reader) and the

servers. When considering the process crash failure model (the one we are interested in in this paper), it is shown in [5] that, when $(|W| = 1) \wedge (t \geq 1) \wedge (|R| \geq 2)$, the condition $(|R| < \frac{|S|}{t} - 2)$ is necessary and sufficient to have fast read and write operations (as defined above), which implement an atomic register. It is also shown in [5] that there is no fast implementation of an MWMR atomic register if $\big((|W| \geq 2) \wedge (|R| \geq 2) \wedge (t \geq 1)\big)$.

Algorithms suited to this three-tier model (writers, readers, and servers) providing very fast read operations are described in [7].

Another notion of efficiency is related to the size of the control information carried by messages. This issue is mainly addressed in [11], where it is shown that two bits are sufficient.

Content of the paper: bounded delay assumption and time-efficiency. The work described in [5,6] is mainly on the limits of the three-component model (writers, readers, and servers constitute three independent sets of processes) in the presence of process crash failures, or Byzantine process failures. These limits are captured by predicates involving the set of writers (W), the set of readers (R), the set of servers (S), and the maximal number of servers that can be faulty (t). Both the underlying model used in this paper and its aim are different from this previous work.

While keeping the spirit (basic principles and simplicity) of ABD, our aim is to design a *time-efficient* implementation of an atomic register in the classical model used in many articles and textbooks (see, e.g., [2,3,10,13]). This model, where any process can communicate with any process, can be seen as a peer-to-peer model in which each process is both a client (it can invoke operations) and a server (it manages a local copy of the register that is built).[1]

Adopting the usual distributed computing assumption that (a) local processing times are negligible and assumed consequently to have zero duration, and (b) only communication takes time, this paper focuses on the communication time needed to complete a read or write operation. For this reason the term *time-efficiency* is defined here in terms on message transfer delays, namely, the cost of a read or write operation is measured by the number of "consecutive" message transfer delays they require to terminate. Let us notice that this includes transfer delays due to causally related messages (for example round trip delays generated by request/acknowledgment messages), but also (as we will see in the proposed algorithm) message transfer delays which occur sequentially without being necessarily causally related. Let us notice that this notion of a time-efficient operation does not involve the model parameter t.

In order to give a precise meaning to the notion of a "time-efficient implementation" of a register operation, this paper considers the duration of read and write operations based on a specific additional synchrony assumption, usually named "bounded delay" assumption. This assumptions and the associated time-efficiency of the proposed algorithm are the following.

[1] Considering the three-component model where each reader is also a server (i.e., $R = S$), we obtain a two-component model with one writer and reader-server processes. In this model, the necessary and sufficient condition $(|R| < \frac{|S|}{t} - 2)$ can never be satisfied, which means that, it is impossible to design a fast implementation of a SWMR atomic register in such a two-component model.

Let us consider the case where every message takes at most Δ time units to be transmitted from its sender to any of its receivers. In such a context, the algorithm presented in the paper has the following time-efficiency properties:

- A write operation takes at most 2Δ time units.
- A read operation which is write-latency-free takes at most 2Δ time units. (The notion of write-latency-freedom is defined in Section 3. Intuitively, it captures the fact that the behavior of the read does not depend on a concurrent or faulty write operation, which is the usual case in read-dominated applications.) Otherwise, it takes at most 3Δ time units, except in the case where the read operation is concurrent with a write operation and the writer crashes during this write, where it can take up to 4Δ time units. (Let us remark that a process can experience at most once the 4Δ read operation scenario.)

Hence, while it remains correct in the presence of any asynchronous message pattern (e.g., when each message takes one more time unit than any previous message), the proposed algorithm is particularly time-efficient when "good" scenarios occur. Those are the ones defined by the previous synchrony patterns where the duration of a read or a write operation is a single round-trip delay. Moreover, in the other synchrony scenarios, where a read operation is concurrent with a write, the maximal duration of the read operation is precisely quantified. A concurrent write adds uncertainty whose resolution by a read operation requires one more message transfer delay (and two if the concurrent write crashes).

Roadmap. The paper consists of 7 sections. Section 2 presents the system model. Section 3 defines the atomic register abstraction, and the notion of a time-efficient implementation. Then, Section 4 presents an asynchronous algorithm providing an implementation of an atomic register with time-efficient operations, as previously defined. Section 5 proves its properties. Section 6 provides insights on the practical impact of the latency-efficiency in real systems. Finally, Section 7 concludes the paper.

2 System 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*.

Communication. The processes cooperate by sending and receiving messages through bi-directional channels. The communication network is a complete network, which means that any process p_i can directly send a message to any process p_j (including itself). 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).

A process p_i invokes the operation "send TAG(m) to p_j" to send p_j the message tagged TAG and carrying the value m. It receives a message tagged TAG by invoking the operation "receive TAG()". The macro-operation "broadcast TAG(m)" is a shortcut for "**for each** $j \in \{1, \ldots, n\}$ 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 – possibly empty– subset of processes will receive the message.)

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

Notations. In the following, the previous computation model, restricted to the case where $t < n/2$, is denoted $\mathcal{CAMP}_{n,t}[t < n/2]$ (Crash Asynchronous Message-Passing).

It is important to notice that, in this model, all processes are a priori "equal". As we will see, this allows each process to be at the same time a "client" and a "server". In this sense, and as noticed in the Introduction, this model is the "fully connected peer-to-peer" model (whose structure is different from other computing models such as the client/server model, where processes are partitioned into clients and servers, playing different roles).

3 Atomic Register and Time-efficient Implementation

3.1 Atomic read/write register

Read/write register. A *concurrent object* is an object that can be accessed by several processes (possibly simultaneously). An SWMR register (say REG) is a concurrent object which provides exactly one process (called the writer) with an operation denoted REG.write(), and all processes with an operation denoted REG.read(). When the writer invokes REG.write(v) it defines v as being the new value of REG.

Effective operation. The notion of an *effective* read or write operation is from [14]. When a process crashes while executing an operation, this operation may take effect or not. The notion of an *effective* operation captures the fact that, due to process crashes, some operations "participate" in the computation, while others do not. More precisely, we have the following [14].

- A read operation is *effective* if the invoking process does not crash during its execution.
- A write operation is *effective* if the invoking process does not crash during its execution, or, despite the fact that it crashes during its execution, the value it writes is returned by an effective read operation.

Atomic read/write register. An SWMR atomic register (we also say the register is *linearizable* [4]) is defined by the following set of properties [9].

- Liveness. The invocation of an effective operation by a correct process terminates.
- Consistency (safety). All the effective operations appear as if they have been executed sequentially and this sequence S of operations is such that:
  - each effective read returns the value written by the closest write that precedes it in S (or the initial value of REG if there is no preceding write),
  - if an effective operation $op1$ terminated before an effective operation $op2$ started, then $op1$ appears before $op2$ in the sequence S.

This set of properties states that, from an external observer point of view, the object appears as if it was accessed sequentially by the processes, this sequence $i)$ respecting the real time access order, and $ii)$ belonging to the sequential specification of a read/write register.

3.2 *Bounded delay*-based time-efficient implementation

The notion of a time-efficient operation is not related to its correctness, but is a property of its implementation. It is sometimes called *non-functional* property. In the present case, it captures the time efficiency of operations[2]. Time-efficiency is a property that is closely related to the notion of fast [5] implementation. While a fast implementation focus on the minimum number of round-trip transfers that are necessary, a time-efficient operation, as we consider here, provides us with time guarantees when favorable synchrony conditions hold on the system execution.

Let us remember that it is assumed that the local processing times needed to implement these high level read and write operations are negligible, and consider a scenario where all message transfer delays are upper bounded by Δ. Notice that we consider the synchrony assumption applies to parts of the execution only.

Write-latency-free read operation and interfering write. Intuitively, a read operation is *write-latency-free* if its execution does "not interleave" with the execution of a write operation. More precisely, let τ_r be the starting time of a read operation. This read operation is *write-latency-free* if (a) it is not concurrent with a write operation, and (b) the closest preceding write did not crash and started at a time $\tau_w < \tau_r - \Delta$.

Let $\mathsf{op_r}$ be a read operation, which started at time τ_r. Let $\mathsf{op_w}$ be the closest write preceding $\mathsf{op_r}$. If $\mathsf{op_w}$ started at time $\tau_w \geq \tau_r - \Delta$, it is said to be *interfering* with $\mathsf{op_r}$. A graphical representation of write-latency-freedom and interfering operations is depicted in Fig. 1.

[2] Another example of a non-functional property is *quiescence*. This property is on algorithms implementing reliable communication on top of unreliable networks [1]. It states that the number of underlying implementation messages generated by an application message must be finite. Hence, if there is a time after which no application process sends messages, there is a time after which the system is quiescent.

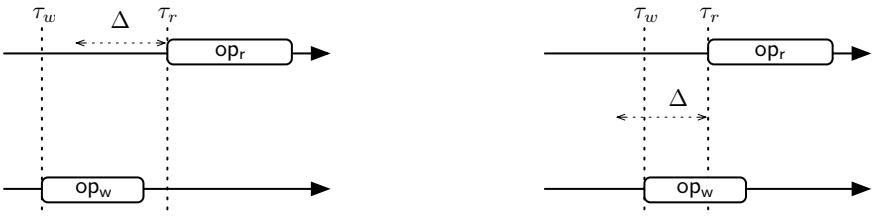

Fig. 1 Left: A write-latency-free read operation $\mathsf{op_r}$. Right: a write operation $\mathsf{op_w}$ interfering with a read.

Time-efficiency: Bounded delay-based definition. An implementation of a read/write register is *time-efficient* (from a bounded delay point of view) if it satisfies the following properties.

- A write operation takes at most 2Δ time units.
- A read operation which is write-latency-free takes at most 2Δ time units.
- A read operation which is not write-latency-free takes at most
  - 3Δ time units if the writer does not crash while executing the interfering write,
  - 4Δ time units if the writer crashes while executing the interfering write (this scenario can appear at most once for each process).

4 An Algorithm with Time-efficient Operations

The algorithm described in Algorithm 1 implements an SWMR atomic register in the asynchronous system model $\mathcal{CAMP}_{n,t}[t < n/2]$, and is time-efficient with respect to the "bounded delay"-based definition. This algorithm is voluntarily formulated to be as close as possible to ABD, for facilitating its understanding by ABD-aware readers.

Local variables. Each process p_i manages the following local variables.

- reg_i contains the value of the constructed register REG, as currently known by p_i. It is initialized to the initial value of REG (e.g., the default value $\perp$).
- wsn_i is the sequence number associated with the value in reg_i.
- rsn_i is the sequence number of the last read operation invoked by p_i.
- $swsn_i$ is a synchronization local variable. It contains the sequence number of the most recent value of REG that, to p_i's knowledge, is known by at least $(n - t)$ processes. This variable (which is new with respect to other algorithms) is at the heart of the time-efficient implementation of the read operation.
- res_i is the value of REG whose sequence number is $swsn_i$.

Client side: operation $\mathsf{write}()$ *invoked by the writer.* Let p_i be the writer. When it invokes $REG.\mathsf{write}(v)$, it increases wsn_i, updates reg_i, and broadcasts the message

Algorithm 1 Time-efficient SWMR atomic register in $\mathcal{CAMP}_{n,t}[t < n/2]$

local variables initialization: $reg_i \leftarrow \perp$; $wsn_i \leftarrow 0$; $swsn_i \leftarrow 0$; $rsn_i \leftarrow 0$.

operation write(v) **is**
(1) $wsn_i \leftarrow wsn_i + 1$; $reg_i \leftarrow v$; broadcast WRITE(wsn_i, v);
(2) **wait** (WRITE($wsn_i, -$) received from $(n - t)$ different processes);
(3) return()
end operation.

operation read() **is** % the writer may directly return reg_i %
(4) $rsn_i \leftarrow rsn_i + 1$; broadcast READ($rsn_i$);
(5) **wait** ((msgs STATE($rsn, -$) rec. from $(n - t)$ different proc.) $\wedge$ $(swsn_i \geq maxwsn)$
where $maxwsn$ is the greatest seq. nb in the previous STATE($rsn, -$) msgs);
(6) return(res_i)
end operation.

when WRITE(wsn, v) **is received do**
(7) **if** $(wsn > wsn_i)$ **then** $reg_i \leftarrow v$; $wsn_i \leftarrow wsn$ **end if**;
(8) **if** (not yet done) **then** broadcast WRITE(wsn, v) **end if**;
(9) **if** (WRITE($wsn, -$) received from $(n - t)$ different processes)
(10) **then if** $(wsn > swsn_i) \wedge$ (not already done) **then** $swsn_i \leftarrow wsn$; $res_i \leftarrow v$ **end if**
(11) **end if**.

when READ(rsn) **is received from** p_j **do**
(12) send STATE(rsn, wsn_i) to p_j.

WRITE(wsn_i, v) (line 1). Then, it waits until it has received an acknowledgment message from $(n - t)$ processes (line 2). When this occurs, the operation terminates (line 3). Let us notice that the acknowledgment message is a copy of the very same message as the one it broadcast.

Server side: reception of a message WRITE(wsn, v). When a process p_i receives a WRITE(wsn, v) message, if this message carries a more recent value than the one currently stored in reg_i, then p_i updates accordingly wsn_i and reg_i (line 7). Moreover, if this message is the first message carrying the sequence number wsn, p_i forwards to all the processes the message WRITE(wsn, v) it has received (line 8). This broadcast has two aims: to be an acknowledgment for the writer, and to inform the other processes that p_i "knows" this value.[3]

Moreover, when p_i has received the message WRITE(wsn, v) from $(n - t)$ different processes, and $swsn_i$ is smaller than wsn, it updates its local synchronization variable $swsn_i$ and accordingly assigns v to res_i (lines 9-11).

Server side: reception of a message READ(rsn). When a process p_i receives such a message from a process p_j, it sends by return to p_j the message STATE(rsn, wsn_i), thereby informing it on the freshness of the last value of REG it knows (line 12). The parameter rsn allows the sender p_j to associate the messages STATE($rsn, -$) it will receive with the corresponding request identified by rsn.

[3] Let us observe that, due to asynchrony, it is possible that $wsn_i > wsn$ when p_i receives a message WRITE(wsn, v) for the first time.

Client side: operation read(). When a process p_i invokes REG.read(), it first broadcasts the message READ(rsn_i) with a new sequence number. Then, it waits until "some" predicate is satisfied (line 5), and finally returns the current value of res_i. Let us notice that the value res_i that is returned is the one whose sequence number is $swsn_i$.

The waiting predicate is the heart of the algorithm. Its first part states that p_i must have received a message STATE$(rsn, -)$ from $(n - t)$ processes. Its second part, namely $(swsn_i \geq maxwsn)$, states that the value in p_i's local variable res_i is as recent or more recent than the value associated with the greatest write sequence number wsn received by p_i in a message STATE$(rsn, -)$. Combined with the broadcast of messages WRITE$(wsn, -)$ issued by each process at line 8, this waiting predicate ensures both the correctness of the returned value (atomicity), and the fact that the read implementation is time-efficient.

5 Proof of the Algorithm

5.1 Termination and atomicity

The properties proved in this section are independent of the message transfer delays (provided they are finite).

Lemma 1 *If the writer is correct, all its write invocations terminate. If a reader is correct, all its read invocations terminate.*

Proof Let us first consider the writer process. As by assumption it is correct, it broadcasts the message WRITE$(sn, -)$ (line 1). Each correct process broadcasts WRITE$(sn, -)$ when it receives it for the first time (line 8). As there are at least $(n - t)$ correct processes, the writer eventually receives WRITE$(sn, -)$ from these processes, and stops waiting at line 2.

Let us now consider a correct reader process p_i. It follows from the same reasoning as before that the reader receives the message STATE$(rsn, -)$ from at least $(n - t)$ processes (lines 5 and 12). Hence, it remains to prove that the second part of the waiting predicate, namely $swsn_i \geq maxwsn$ (line 5) becomes eventually true, where $maxwsn$ is the greatest write sequence number received by p_i in a message STATE$(rsn, -)$. Let p_j be the sender of this message. The following list of items is such that item $x \implies$ item $(x + 1)$, from which follows that $swsn_i \geq maxwsn$ (line 5) is eventually satisfied.

1. p_j updated wsn_j to $maxwsn$ (line 7) before sending STATE$(rsn, maxwsn)$ (line 12).
2. Hence, p_j received previously the message WRITE$(maxwsn, -)$, and broadcast it the first time it received it (line 8).
3. It follows that any correct process receives the message WRITE$(maxwsn, -)$ (at least from p_j), and broadcasts it the first time it receives it (line 8).
4. Consequently, p_i eventually receives the message WRITE$(maxwsn, -)$ from $(n-t)$ processes. When this occurs, it updates $swsn_i$ (line 10), which is then $\geq maxwsn$, which concludes the proof of the termination of a read operation.

$\square_{Lemma\ 1}$

Lemma 2 *The register REG 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, and $write[y]$ be the write operation which writes the value with sequence number 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 inversions [3, 12].

Proof of Claim 1.
This claim follows from the following simple observation. When the writer executes $write[y]$, it first increases its local variable wsn which becomes greater than any sequence number associated with its previous write operations (line 1). Hence if $read[i,x]$ terminates before $write[y]$ starts, we necessarily have $x < y$.

Proof of Claim 2.
It follows from line 2 and lines 7-8 that, when $write[x]$ terminates, there is a set Q_w of at least $(n-t)$ processes p_k such that $wsn_k \geq x$. On another side, due to lines 4-5 and line 12, $read[i,y]$ obtains a message STATE() from a set Q_r of at least $(n-t)$ processes.

As $|Q_w| \geq n-t$, $|Q_r| \geq n-t$, and $n > 2t$, it follows that $Q_w \cap Q_r$ is not empty. There is consequently a process $p_k \in Q_w \cap Q_r$, such that that $wsn_k \geq x$. Hence, p_k sent to p_i the message STATE$(-,z)$, where $z \geq x$.

Due to (a) the definition of $maxwsn \geq z$, (b) the predicate $swsn_i \geq maxwsn \geq z$ (line 5), and (c) the value of $swsn_i = y$, it follows that $y = swsn_i \geq z$ when $read[i,y]$ stops waiting at line 5. As, $z \geq x$, it follows $y \geq x$, which proves the claim.

Proof of Claim 3.
When $read[i,x]$ stops waiting at line 5, it returns the value res_i associated with the sequence number $swsn_i = x$. Process p_i previously received the message WRITE$(x,-)$ from a set Q_{r1} of at least $(n-t)$ processes. The same occurs for p_j, which, before returning, received the message WRITE$(y,-)$ from a set Q_{r2} of at least $(n-t)$ processes.

As $|Q_{r1}| \geq n-t$, $|Q_{r2}| \geq n-t$, and $n > 2t$, it follows that $Q_{r1} \cap Q_{r2}$ is not empty. Hence, there is a process p_k which sent STATE$(,x)$ to p_i, and later sent STATE$(-,y)$ to p_j. As $swsn_k$ never decreases, it follows that $x \leq y$, which completes the proof of the lemma. $\square_{Lemma\ 2}$

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

Proof The proof follows from Lemma 1 (termination) and Lemma 2 (atomicity). $\square_{Theorem\ 1}$

5.2 Time-efficiency with respect to the *bounded delay* assumption

As already indicated, this underlying synchrony assumption considers that every message takes at most Δ time units. Moreover, let us remind that a read (which started at time τ_r) is write-latency-free if it is not concurrent with a write, and the last preceding write did not crash and started at time $\tau_w < \tau_r - \Delta$.

Lemma 3 *A write operation takes at most* 2Δ *time units.*

Proof The case of the writer is trivial. The message WRITE() broadcast by the writer takes at most Δ time units, as do the acknowledgment messagesWRITE() sent by each process to the writer. In this case 2Δ correspond to a causality-related maximal round-trip delay (the reception of a message triggers the sending of an associated acknowledgment). $\square_{Lemma\ 3}$

When the writer does not crash while executing a write operation The cases where the writer does not crash while executing a write operation are captured by the next two lemmas.

Lemma 4 *A write-latency-free read operation takes at most* 2Δ *time units.*

Proof Let p_i be a process that issues a write-latency-free read operation, and τ_r be its starting time. Moreover, Let τ_w the starting time of the last preceding write. As the read is write latency-free, we have $\tau_w + \Delta < \tau_r$. Moreover, as messages take at most Δ time units, and the writer did not crash when executing the write, each non-crashed process p_k received the message WRITE$(x, -)$ (sent by the preceding write at time $\tau_w + \Delta < \tau_r$), broadcast it (line 8), and updated its local variables such that we have $wsn_k = x$ (lines 7-11) at time $\tau_w + \Delta < \tau_r$. Hence, all the messages STATE() received by the reader p_i carry the write sequence number x. Moreover, due to the broadcast of line 8 executed by each correct process, we have $swsn_i = x$ at some time $\tau_w + 2\Delta < \tau_r + \Delta$. It follows that the predicate of line 5 is satisfied at p_i within 2Δ time units after it invoked the read operation. $\square_{Lemma\ 4}$

Lemma 5 *A read operation which is not write-latency-free, and during which the writer does not crash during the interfering write operation, takes at most* 3Δ.

Proof Let us consider a read operation that starts at time τ_r, concurrent with a write operation that starts at time τ_w and during which the writer does not crash. From the read operation point of view, the worst case occurs when the read operation is

invoked just after time $\tau_w - \Delta$, let us say at time $\tau_r = \tau_w - \Delta + \epsilon$. As a message STATE$(rsn, -)$ is sent by return when a message READ(rsn) is received, the messages STATE$(rsn, -)$ received by p_i by time $\tau_r + 2\Delta$ can be such that some carry the sequence number x (due to last previous write) while others carry the sequence number $x + 1$ (due to the concurrent write)[4]. Hence, $maxwsn = x$ or $maxwsn = x + 1$ (predicate of line 5). If $maxwsn = x$, we also have $swsn_i = x$ and p_i terminates its read. If $maxwsn = x + 1$, p_i must wait until $swsn_i = x + 1$, which occurs at the latest at $\tau_w + 2\Delta$ (when p_i receives the last message of the $(n - t)$ messages WRITE$(y, -)$ which makes true the predicates of lines 9-10, thereby allowing the predicate of line 5 to be satisfied). When this occurs, p_i terminates its read operation. As $\tau_w = \tau_r + \Delta - \epsilon$, p_i returns at the latest $\tau_r + 3\Delta - \epsilon$ time units after it invoked the read operation. $\square_{Lemma\ 4}$

When the writer crashes while executing a write operation. The problem raised by the crash of the writer while executing the write operation is when it crashes while broadcasting the message WRITE$(x, -)$ (line 1): some processes receive this message by Δ time units, while other processes do not. This issue is solved by the propagation of the message WRITE$(x, -)$ by the non-crashed processes that receive it (line 8). This means that, in the worst case (as in synchronous systems), the message WRITE$(x, -)$ must be forwarded by $(t + 1)$ processes before being received by all correct processes. This worst scenario may entail a cost of $(t + 1)\Delta$ time units.

Algorithm 2 Time-efficient read in case of concurrent writer crash

when WRITE(wsn, v) or STATE(rsn, wsn, v) **is received do**
(7) **if** $(wsn > wsn_i)$ **then** $reg_i \leftarrow v$; $wsn_i \leftarrow wsn$; broadcast WRITE(wsn, v) **end if**;
(8) **if** (not yet done) **then** broadcast WRITE(wsn, v) **end if**;
(9) **if** (WRITE$(wsn, -)$ received from $(n - t)$ different processes)
(10) **then if** $(wsn > swsn_i) \wedge$ (not already done) **then** $swsn_i \leftarrow wsn$; $res_i \leftarrow v$ **end if**
(11) **end if**.

when READ(rsn) **is received from** p_j **do**
(12) send STATE(rsn, wsn_i, reg_i) to p_j.

Algorithm 2 presents a simple modification of Algorithm 1, which allows a fast implementation of read operations whose executions are concurrent with a write operation during which the writer crashes. The modifications are underlined.

When a process p_i receives a message READ(), it now returns a message STATE() containing an additional field, namely the current value of reg_i, its local copy of REG (line 12).

When a process p_i receives from a process p_j a message STATE$(-, wsn, v)$, it uses it in the waiting predicate of line 5, but executes before the lines 7-11, as if this message was WRITE(wsn, v). According to the values of the predicates of lines 7, 9,

[4] Messages STATE(rsn, x) are sent by the processes that received READ(rsn) before τ_w, while the messages STATE$(rsn, x+1)$ are sent by the processes that received READ(rsn) between τ_w and $\tau_r + \Delta = \tau_w + \epsilon$.

and 10, this allows p_i to expedite the update of its local variables wsn_i, reg_i, $swsn_i$, and res_i, thereby favoring fast termination.

The reader can check that these modifications do not alter the proofs of Lemma 1 (termination) and Lemma 2 (atomicity). Hence, the proof of Theorem 1 is still correct.

Lemma 6 *A read operation which is not write-latency-free, and during which the writer crashes during the interfering write operation, takes at most 4Δ time units.*

Proof Let τ_r be the time at which the read operation starts. As in the proof of Lemma 4, the messages STATE$(rsn, -, -)$ received p_i by time $\tau_r + 2\Delta$ can be such that some carry the sequence number $wsn = x$ (due to last previous write) while some others carry the sequence number $wsn = x + 1$ (due to the concurrent write during which the writer crashes). If all these messages carry $wsn = x$, the read terminates by time $\tau_r + 2\Delta$. If at least one of these messages is STATE$(rsn, x + 1, -)$, we have $maxwsn = x+1$, and p_i waits until the predicate $swsn_i \geq maxwsn\ (= x+1)$ becomes true (line 5).

When it received STATE$(rsn, x+1, -)$, if not yet done, p_i broadcast the message WRITE$(rsn, x + 1, -)$, (line 8 of Algorithm 2), which is received by the other processes within Δ time units. If not yet done, this entails the broadcast by each correct process of the same message WRITE$(rsn, x + 1, -)$. Hence, at most Δ time units later, p_i has received the message WRITE$(rsn, x + 1)$ from $(n - t)$ processes, which entails the update of $swsn_i$ to $(x + 1)$. Consequently the predicate of line 5 becomes satisfied, and p_i terminates its read operation.

When counting the number of consecutive communication steps, we have: The message READ(rsn) by p_i, followed by a message STATE$(rsn, x+1, -)$ sent by some process and received by p_i, followed by the message WRITE$(rsn, x + 1)$ broadcast by p_i, followed by the message WRITE$(rsn, x + 1)$ broadcast by each non-crashed process (if not yet done). Hence, when the writer crashes during a concurrent read, the read returns within at most $\tau_r + 4\Delta$ time units. $\square_{Lemma\ 6}$

Theorem 2 *The algorithm described in Algorithm* 1*, modified as indicated in Algorithm* 2*, implements a time-efficient* SWMR *atomic register in the distributed system model $\mathcal{CAMP}_{n,t}[t < n/2]$.*

Proof The proof follows from Theorem 1 (termination and atomicity), Lemma 3, Lemma 4, Lemma 5, and Lemma 6 (time-efficiency). $\square_{Theorem\ 2}$

6 Practical Implications: a System's Viewpoint

The algorithm presented in this paper shares many similarities with the well-known ABD algorithm. ABD is at the basis of real-life implementations of scalable replicated storage services used in today's large data centers and clouds, and distributed shared memory systems. Keeping as close as possible to the design ideas of the original algorithm eases the integration of the proposed ideas in implemented systems.

While the notion of a time-efficient algorithm has a theoretical interest —the lower bounds and the worst cases are not the "end of the road"— its practical implication is much more important.

In all possible scenarios that may happen in an actual execution, the worst case scenarios in terms of message transfer delays and asynchrony patterns are usually rare. From a practical viewpoint, it is more important to optimize cases that occur regularly than just focus on those worst case scenarios. What we show with this algorithm is that it possible to match the lower bound, whatever asynchrony pattern happens in the system, and still provide a fast and time-efficient solution when the system is more synchronous. The price to pay for optimizing those good cases is an increase of the message complexity of a write operation, making the algorithm well suited for read-dominated applications that run on almost always synchronous systems.

Indeed, many real-life systems experience lots of "stability" periods during which the results described in this paper apply —cloud systems deployed by Google, Apple, Facebook, Amazon and Microsoft run on dedicated in-house hardware solutions, and are not purely asynchronous. With regards to IoT applications, sensor networks usually have a synchronous mode of operation in order to reduce energy consumption: a Time-Division Multiple Access (TDMA) communication scheme allows sensors to sleep most of the time and wake up at predefined time slots to communicate.

7 Conclusion

This work presented a new distributed algorithm that implements an atomic read/write register on top of an asynchronous n-process message-passing system in which up to $t < n/2$ processes may crash. When designing it, the constraints we imposed on this algorithm were (a) from an efficiency point of view: provide time-efficient implementations for read and write operations, (b) and from a design principle point of view: remain "as close as possible" to the flagship ABD algorithm introduced by Attiya, Bar-Noy and Dolev [2].

Table 1 recapitulates the "time-efficiency" property of the proposed algorithm. Under the "upper bound Δ on message transfer delays", any write operation takes then at most 2Δ time units, and a read operation takes at most 2Δ time units when executed in good circumstances (i.e., when there is no write operation concurrent with the read operation). Hence, the inherent cost of an operation is a round-trip delay, always for a write and in favorable circumstances for a read. A read operation concurrent with a write operation during which the writer does not crash, may require an additional cost of Δ, which means that it takes at most 3Δ time units. Finally, if the writer crashes during a write concurrent with a read, the read may take at most 4Δ time units. This shows clearly the incremental cost imposed by the adversaries (concurrency of write operations, and failure of the writer).

It is important to remind that the proposed algorithm remains correct in the presence of any asynchrony pattern. Its time-efficiency features are particularly interesting when the system has long synchrony periods.

Cost of the operations according to the underlying synchrony assumption	upper bound Δ on transfer delays
write operation	2Δ
read operation (no concurrent write)	2Δ
read operation (concurrent write)	3Δ
read operation (conc. crashing write)	4Δ

Table 1 Summary for the upper bound delay assumption

Differently from the proposed algorithm, the ABD algorithm does not display different behaviors in different concurrency and failure patterns. In ABD, the duration of all write operations is upper bounded by 2Δ time units, and the duration of all read operations is upper bounded by 4Δ time units. The trade-off between ABD and our algorithm lies the message complexity, which is $O(n)$ in ABD for both read and write operations, while it is $O(n^2)$ for a write operation and $O(n)$ for a read operation in the proposed algorithm. Hence our algorithm is particularly interesting for registers used in read-dominated applications run on systems that are often synchronous.

Acknowledgments

The authors want to thank the referees for their constructive comments. This work has been partially supported by the Franco-German DFG-ANR Project DISCMAT (DFG-ANR 40300781) devoted to connections between mathematics and distributed computing, and the French ANR project DESCARTES (ANR-16-CE40-0023-03) devoted to distributed software engineering.

References

1. Aguilera M.K., Chen W. and Toueg S., On quiescent reliable communication. *SIAM Journal of Computing*, 29(6):2040-2073 (2000)
2. Attiya H., Bar-Noy A. and Dolev D., Sharing memory robustly in message passing systems. *Journal of the ACM*, 42(1):121-132 (1995)
3. Attiya H. and Welch J., *Distributed computing: fundamentals, simulations and advanced topics*, (2d Edition), Wiley-Interscience, 414 pages (2004)
4. 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)
5. Dutta P., Guerraoui R., Levy R., and Chakraborty A., How fast can a distributed atomic read be? *Proc. 23rd ACM Symposium on Principles of distributed computing (PODC'04)*, ACM Press, pp. 236-245 (2004)
6. Dutta P., Guerraoui R., Levy R., and Vukolic M., Fast access to distributed atomic memory. *SIAM Journal of Computing*, 39(8):3752-3783 (2010)
7. Hadjistasi T., Nicolaou N., and Schwarzmann A.A., Oh-RAM! One and a half round atomic memory. *Proc. 5th Int'l Conference on Networked Systems (NETYS'17)*, Springer LNCS 10299, pp. 117-132 (2017)
8. 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).
9. Lamport L., On interprocess communication, Part I: basic formalism. *Distributed Computing*, 1(2):77-85,((1986)

10. Lynch N.A., *Distributed algorithms*. Morgan Kaufmann Pub., San Francisco (CA), 872 pages, ISBN 1-55860-384-4) (1996)
11. Mostéfaoui A. and Raynal M., Two-bit messages are sufficient to implement atomic read/write registers in crash-prone systems. *Proc. 35th ACM Symposium on Principles of Distributed Computing (PODC'16)*, ACM Press, pp. 381-390 (2026)
12. Raynal M., *Distributed algorithms for message-passing systems*. Springer, 510 pages, ISBN: 978-3-642-38122-5 (2013)
13. Raynal M., *Concurrent programming: algorithms, principles and foundations*. Springer, 515 pages, ISBN 978-3-642-32026-2 (2013)
14. Raynal M., *Fault-tolerant message-passing distributed systems: an algorithmic approach*. To appear, Springer, 550 pages (2018)
15. Turing A.M., On computable numbers with an application to the Entscheidungsproblem. *Proc. of the London Mathematical Society*, 42:230-265 (1936)
16. Vukolic M., *Quorum systems, with applications to storage and consensus*. Morgan & Claypool Publishers, 132 pages, ISBN 978-1-60845-683-3 (2012)