Reliable Shared Memory Abstractions on Top of Asynchronous t-Resilient Byzantine Message-passing Systems

Damien Imbs, Sergio Rajsbaum, Michel Raynal, Julien Stainer

Publications Internes de l'IRISA
ISSN : 2102-6327
PI 2018 – January 2014

Reliable Shared Memory Abstractions on Top of Asynchronous t-Resilient Byzantine Message-passing Systems

Damien Imbs[*] Sergio Rajsbaum[*] Michel Raynal[**] [***] Julien Stainer[***]

Abstract: This paper is on the construction and the use of a shared memory abstraction on top of an asynchronous message-passing system in which up to t processes may commit Byzantine failures. This abstraction consists of arrays of n single-writer/multi-reader atomic registers, where n is the number of processes. A distributed algorithm building such a shared memory abstraction it first presented. This algorithm assumes $t < n/3$, which is shown to be a necessary and sufficient condition for such a construction. Hence, the algorithm is resilient-optimal. Then the paper presents distributed algorithms built on top of this shared memory abstraction, which cope with up to t Byzantine processes. The simplicity of these algorithms constitutes a strong motivation for such a shared memory abstraction in the presence of Byzantine processes.

For a lot of problems, algorithms are more difficult to design and prove correct in a message-passing system than in a shared memory system. Using a protocol stacking methodology, the aim of the proposed abstraction is to allow an easier design (and proof) of distributed algorithm, when one has the underlying system is an asynchronous message-passing system prone to Byzantine failures.

Key-words: Approximate agreement, Asynchronous message-passing system, Atomic read/write register, Broadcast abstraction, Byzantine process, Distributed computing, Message-passing system, Quorum, Reliable broadcast, Reliable shared memory, Single-writer/multi-reader register, t-Resilience.

Abstraction d'une mémoire partagée fiable dans les systèmes asynchones à passage de messages où jusqu'à t fautes Byzantines peuvent survenir

Résumé : *Cet article propose une abstraction de mémoire partagée adaptée aux systèmes asynchones à passage de messages où jusqu'à t fautes Byzantines peuvent survenir. Il présente également une implémentation de cette abstraction ainsi que sa preuve.*

Mots clés : *Accord approché, systèmes asynchrones à passage de messages, registres atomiques read/write, diffusion fiable, processus Byzantins, calcul distribué, quorums , mémoire partagée fiable, tolérance à t fautes.*

---

[*] Department of Mathematics, UNAM, Mexico
[**] Institut Universitaire de France
[***] ASAP : équipe commune avec l'Université de Rennes 1 et Inria

1 Introduction

Distributed computing Distributed computing occurs when one has to solve a problem in terms of physically distinct entities (usually called nodes, processors, processes, agents, sensors, etc.) such that each entity has only a partial knowledge of the many parameters involved in the problem. In the following, we use the term *process* to denote a computing entity. From an operational point of view this means that the processes of a distributed system need to exchange information, and agree in some way or another, in order to cooperate to a common goal. If processes do not cooperate, the system is no longer a distributed system. Hence, a distributed system has to provide the processes with communication and agreement abstractions.

Understanding and designing distributed applications is not an easy task [3, 14, 22, 23, 24]. This is due to the fact that, due its very nature, no process can capture instantaneously the global state of the application it is part of. More precisely, as processes are geographically localized at distinct places, distributed applications have to cope with the uncertainty created by asynchrony and failures. As a simple example, it is impossible to distinguish a crashed process from a very slow process in an asynchronous system prone to process crashes.

As in sequential computing, a simple approach to facilitate the design of distributed applications consists in designing appropriate abstractions. With such abstractions, the application designer can think about solutions to solve problems at a higher conceptual level than the one offered by the basic send/receive communication layer.

Byzantine behavior This failure type has first been introduced in the context of synchronous distributed systems (e.g., [12, 21, 23]), and then investigated in the context of asynchronous ones (e.g., [3, 14, 22]). A process has a *Byzantine* behavior when it arbitrarily deviates from its intended behavior; it then commits a Byzantine failure. Otherwise it is non-faulty (or non-Byzantine). This bad behavior can be intentional (malicious) or simply the result of a transient fault that altered the local state of a process, thereby modifying its behavior in an unpredictable way. Let us notice that process crashes (unexpected halting) and communication omissions, define a strict subset of Byzantine failures.

The major part of the papers on Byzantine failures considers (synchronous or asynchronous) message-passing systems, and mainly addresses agreement problems, such as consensus and total order broadcast, or the construction of a Byzantine-tolerant disk storage (e.g., [7, 11, 15, 16, 17]). Many of these papers consider registers built on top of duplicated disks (servers), which are accessed by clients, and where disks and clients may exhibit different type of failures (e.g., [2]). Moreover, in these client/server models, clients communicate only with servers and vice versa (the communication graph is bipartite).

The reason why different types of failures are addressed in [2, 11] comes from the fact that these papers consider "classical" read/write registers, namely, a register may be modified any number of times, and each read must return the last value that was written. In this context, a Byzantine client can overwrite a value it has previously written, and trick correct clients so that they believe that it wrote only once. To prevent this bad scenario, [11] considers clients that can fail only by crashing, while [2] restricts clients to be "semi-Byzantine": they can issue a bounded number of faulty writes, but otherwise have to respect the code of their algorithm.

Content of the paper: Construction of an atomic read/write memory Differently, this paper is on the construction of a shared memory (atomic registers) on top of an asynchronous message-passing system where processes may exhibit a Byzantine behavior. Its first contribution is the definition of a shared memory (atomic registers) in the context of Byzantine processes, and the design of an algorithm that builds such a shared memory on top of an asynchronous message-passing systems where up to t processes may be Byzantine. These registers differ from classical registers in that each read returns the complete history of writes to the registers. This prevents the problem mentioned previously where a Byzantine process can make different correct processes read different values from the same register. This t-resilient shared memory is made up of n single-writer/multi-reader (SWMR) atomic registers (one per process). The paper also shows that $t < n/3$ is a necessary and sufficient requirement for such a construction.

This construction and the associated upper bound $t < n/3$ complement the previous result known on the construction of an atomic shared memory in asynchronous crash-prone message-passing systems, where it has been shown that $t < n/2$ is an upper bound on the number of faulty processes [1]. Interestingly, the upper bound $t < n/3$ is the same as the one for solving consensus in both Byzantine synchronous systems [12] and Byzantine asynchronous systems (enriched with an appropriate oracle such as a common coin) e.,g. [6, 19].

Content of the paper: From read/write registers to higher level abstractions The second contribution of the paper is a set of algorithms that solve distributed computing problems on top of the previous t-resilient shared memory abstraction. These algorithms illustrate the versatility of these Byzantine-tolerant atomic registers. The two first algorithms, which are pretty simple, solve the "one-shot write-snapshot" problem, and the "correct-only" agreement problem (a weakened version of the consensus problem), respectively. Then the paper describes an algorithm that solves the multidimensional approximate agreement on top of the t-resilient shared memory abstraction. This algorithm can be seen as an adaptation to a Byzantine read/write shared memory system of

Mendes-Herlihy's algorithm [18], which solves the same problem "directly" on top of an asynchronous Byzantine message-passing system.

As shown by these examples, the important feature of the proposed shared memory abstraction lies in the fact that it prevents Byzantine processes from corrupting synchronization among the correct processes. A Byzantine process can create inconsistency only on the values it writes, but any two correct processes see the same sequence of written values.

Roadmap The paper is composed of 6 sections. Section 2 introduces the underlying Byzantine asynchronous message-passing model. Section 3 defines the notion of Byzantine-tolerant atomic read/write registers, and presents an algorithm that builds such registers on top of the basic Byzantine asynchronous message-passing model. This section shows also that $t < n/3$ is a necessary and sufficient requirement for such a construction. Then, Section 4 and Section 5 present algorithms that solve distributed computing problems on top of Byzantine-tolerant atomic registers. Finally, Section 6 concludes the paper.

2 Computation Model, Reliable Broadcast and Two Lemmas

2.1 Computation Model

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

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

Byzantine failures The model parameter t is an upper bound on the number of processes that can exhibit a Byzantine behavior [12, 21]. A Byzantine process is a process that behaves arbitrarily: it may crash, fail to send or receive messages, send arbitrary messages, start in an arbitrary state, perform arbitrary state transitions, etc. Hence, a Byzantine process, which is assumed to send the same message m to all the processes, can send a message m_1 to some processes, a different message m_2 to another subset of processes, and no message at all to the other processes. Moreover, Byzantine processes can collude to "pollute" the computation.

Terminology and notation A Byzantine process is also called a *faulty* process. A process that never commits a failure is called a *correct* (or *non-faulty*) process. Given an execution, let $\mathcal{C}$ and $\mathcal{F}$ denote the sets of correct and faulty processes, respectively.

This process model is denoted $\mathcal{BAMP}_{n,t}[c(n,t)]$, where $c(n,t)$ is a constraint imposed on the model parameter t. As an example $c(n,t) \equiv (n > 3t)$ means that less than one third of processes may be faulty.

2.2 Reliable Broadcast Abstraction

This section presents a reliable broadcast abstraction (denoted r-broadcast), that will be used to build atomic read/write registers. (Section 3). This abstraction is a simple generalization of a reliable broadcast due to Bracha [5]. While Bracha's abstraction is for a single broadcast, the proposed abstraction considers that each process can issue a sequence of broadcasts. It is shown in [5] that $t < n/3$ is a necessary requirement when one has to build such an abstraction in the presence of asynchrony and Byzantine failures.

Specification The reliable broadcast abstraction provides each process with the operations R_broadcast() and R_deliver(). When a process p_i invokes R_broadcast() we say that "p_i r-broadcasts a value". Similarly, when p_i returns from an invocation of R_deliver() and obtains a value, we say "p_i r-delivers a value".

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

- RB-Validity. If a correct process r-delivers a pair (v, sn) from a correct process p_x, then p_x invoked the operation R_broadcast(v, sn).

- RB-Integrity. Given any process p_i and any sequence number sn, a correct process r-delivers at most once a pair $(-, sn)$ from p_i.
- RB-Uniformity. If a correct process r-delivers a pair (v, sn) from p_i (possibly faulty), then all the correct processes eventually r-deliver the same pair (v, sn) from p_i.
- RB-Termination. If the process that invokes R_broadcast(v, sn) is correct, all the correct processes eventually r-deliver the pair (v, sn).

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

An algorithm implementing the reliable broadcast abstraction is presented in Appendix A. This algorithm is a simple variant of Bracha's algorithm [5].

2.3 Two Preliminary Quorum-related Lemmas

This section states and proves two lemmas that will be central in the understanding and the proof of the construction of atomic SWMR registers described in the next section.

Lemma 1 *Let m, n, and t be positive integers.* $\big(m > \frac{n+t}{2}\big) \Leftrightarrow \big(m \geq \lfloor \frac{n+t}{2} \rfloor + 1\big)$.

Proof

- Direction $\Leftarrow$: As $x - 1 < \lfloor x \rfloor$, it follows that $\big(m \geq \lfloor \frac{n+t}{2} \rfloor + 1\big) \implies \big(m > \frac{n+t}{2}\big)$.
- Direction $\Rightarrow$: $(m > \frac{n+t}{2}) \Rightarrow (m \geq \lfloor \frac{n+t}{2} \rfloor + 1)$.
  - Case $(n+t) \bmod 2 = 0$. We have then $m > \frac{n+t}{2} \Rightarrow m \geq \frac{n+t}{2} + 1 = \lfloor \frac{n+t}{2} \rfloor + 1$.
  - Case $(n+t) \bmod 2 = 1$. We have then $m > \frac{n+t}{2} \Rightarrow m \geq \frac{n+t}{2} + \frac{1}{2} = \lfloor \frac{n+t}{2} \rfloor + 1$.

$\square_{Lemma\ 1}$

Lemma 2 *Any two sets of processes Q_1 and Q_2 of size at least $\lfloor \frac{n+t}{2} \rfloor + 1$ have at least one correct process in their intersection.*

Proof When considering integers, it follows from Lemma 1, that "strictly more than $\frac{n+t}{2}$" is equivalent to "at least $\lfloor \frac{n+t}{2} \rfloor + 1$".

- $Q_1 \cup Q_2 \subseteq \{p_1, \ldots, p_n\}$. Hence, $|Q_1 \cup Q_2| \leq n$.
- $|Q_1 \cap Q_2| = |Q_1| + |Q_2| - |Q_1 \cup Q_2| \geq |Q_1| + |Q_2| - n \geq 2(\lfloor \frac{n+t}{2} \rfloor + 1) - n > 2(\frac{n+t}{2}) - n = t$. Hence, $|Q_1 \cap Q_2| \geq t+1$, from which it follows that $Q_1 \cap Q_2$ contains at least one correct process.

$\square_{Lemma\ 2}$

3 Construction of Single-Writer/Multi-Reader Atomic Registers

3.1 Atomic Read/Write Registers in the Presence of Byzantine Processes

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

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

The value returned by a read operation A register $REG[i]$ contains the sequence of values (also called a *history*) that have been written into it, and such a sequence is returned by the invocations of the operation $REG[i]$.read$()$. Each register $REG[i]$ is initialized to the empty sequence (denoted ϵ), which corresponds to the default value $\perp$. It is assumed that no process can write $\perp$ into its register. Let us remark that returning a sequence of values is not a restriction, as, when a process obtains such a sequence h, it can always consider only its last value (or $\perp$ if $h = \epsilon$).

Notations Let p_i and p_j be correct processes. We use the following notations.

- $\mathsf{op\_read}[j, i, h]$: execution by a correct process p_j of $REG[i].\mathsf{read}()$, returning the history h.
- $\mathsf{op\_write}[i, wsn]$: wsn^{th} execution by a correct process p_i of $REG[i].\mathsf{write}()$.
- $|h|$: length of the history h ($|\epsilon| = 0$).

Specification The correct behavior of an SWMR register is defined by the following properties.

- R-Termination (liveness). Let p_i be a correct process.
  - Each invocation of $REG[i].\mathsf{write}()$ issued by p_i terminates.
  - $\forall\, j$, each invocation of $REG[j].\mathsf{read}()$ issued by p_i terminates.
- R-Consistency (safety).
  - Single history. Let p_i be a (correct or faulty) process. There exists exactly one history H_i such that, for any correct process p_j, any $\mathsf{op\_read}[j, i, h]$ is such that h is a prefix of H_i.
  - Read followed by write. Let p_i and p_j be two correct processes. If $\mathsf{op\_read}[j, i, h]$ terminates before $\mathsf{op\_write}[i, wsn]$ starts, then $|h| < wsn$.
  - Write followed by read. Let p_i and p_j be two correct processes. If $\mathsf{op\_write}[i, wsn]$ terminates before $\mathsf{op\_read}[j, i, h]$ starts, then $wsn \leq |h|$.
  - No read inversion. Let p_i and p_j be two correct processes. If $\mathsf{op\_read}[j, i, h]$ terminates before $\mathsf{op\_read}[k, i, h']$ starts, then $|h'| \geq |h|$.

As, whatever i, the invocations of $REG[i].\mathsf{read}()$ by a faulty process p_j can return any value, the previous specification do not need to take them into account. Moreover, it is possible that, while executing a code different from the code of the write operation, a faulty process modifies the content of its register $REG[j]$ (at the operational level, this happens when the messages it generates could have been sent by a correct implementation of the operation $\mathsf{write}()$). The specification of the consistency of such a register $REG[j]$ takes this into account in the "no read inversion" property.

The previous properties state that each register is linearizable [10]. This means that it is possible to totally order the executions of the operations in such a way that (a) the execution of each operation appears as if it has been executed at a single point of the time line between its start event and its end event, (b) no two operations have been executed at the same time, and (c) each read by a correct process returns the sequence of values written before it in the sequence (when considering read/write registers, linearizability is equivalent to atomicity [13]).

An important theorem associated with linearizability is the following [10]: If each object (register) is linearizable, then the set of all the objects, considered as a single object, is linearizable. This means that linearizable objects compose for free.

3.2 The Construction

An algorithm constructing an SWMR atomic (linearizable) register in the presence of up to t Byzantine processes, is described in Figure 1. This algorithm requires $t < n/3$, hence it is suited to the computing model $\mathcal{BAMP}_{n,t}[t < n/3]$. This algorithm uses a **wait**$(condition)$ statement. The corresponding process is blocked until $condition$ becomes satisfied. While a process is blocked, it can process the messages it receives.

Local variables Each process p_i manages four local variables whose scope is the full computation (local variables are denoted with lower case letters, and subscripted by the process index i).

- $reg_i[1..n]$ is the local representation of the array $REG[1..n]$ of atomic SWMR registers. Each local register $reg_i[j]$ is initialized to the empty sequence ϵ whose size is 0 (the corresponding value being the default value $\perp$). The content of $reg_i[j]$ is called the local *history* of $REG[j]$, as known by p_i.
- wsn_i is a sequence number generator (initialized to 0) for the writes of $REG[i]$ (issued by p_i).
- $rsn_i[1..n]$ is a local array such that $rsn_i[j]$ is used to associate sequence numbers to the invocations of $REG[j].\mathsf{read}()$ issued by p_i. Each $rsn_i[j]$ is initialized to 0.
- $approx\_rsn_i[1..n, 1..n]$ is a matrix of sequence numbers, each initialized to 0; $approx\_rsn_i[k, j]$ is the last read sequence number ($rsn_k[j]$) used by p_k to to read $REG[j]$, as known by p_i.

local variables initialization:
$reg_i[1..n] \leftarrow [\epsilon, \ldots, \epsilon]$; $wsn_i \leftarrow 0$; $rsn_i[1..n] \leftarrow [0, \ldots, 0]$, $approx\_rsn_i[1..n, 1..n] \leftarrow [0, \ldots, 0]$.
%———————————————————————————————————————

operation $REG[i]$.write(v) **is**
(01) $wsn_i \leftarrow wsn_i + 1$;
(02) R_broadcast WRITE(v, wsn_i);
(03) **wait** (WRITE_DONE(wsn_i) received from strictly more than $\frac{n+t}{2}$ different processes);
(04) return ().

operation $REG[j]$.read() **is**
(05) $rsn_i[j] \leftarrow rsn_i[j] + 1$;
(06) broadcast READ($j, rsn_i[j]$);
(07) **wait** ($\exists h$: READ_VALUE($j, rsn_i[j], h$) received from strictly more than $\frac{n+t}{2}$ different processes);
(08) return (h). % the last value in h can be returned if not interested in the history of $REG[j]$ %
%———————————————————————————————————————

when a message WRITE(v, wsn) **from** p_j **is** R_delivered:
(09) **wait** ($|reg_i[j]| + 1 = wsn$);
(10) $reg_i[j] \leftarrow reg_i[j] \cdot v$;
(11) send WRITE_DONE(wsn) to p_j;
(12) **for** $k \in [1..n]$ **do** send READ_VALUE($j, approx\_rsn_i[k, j], reg_i[j]$) to p_k **end for**.

when a message READ(j, rsn) from p_k **is** received:
(13) **if** ($approx\_rsn_i[k, j] < rsn$) **then**
(14) $approx\_rsn_i[k, j] \leftarrow rsn$;
(15) send READ_VALUE($j, rsn, reg_i[j]$) to p_k
(16) **end if**.

Figure 1: Array of SWMR atomic registers in $\mathcal{BAMP}_{n,t}[t < n/3]$ (code for p_i)

The operation $REG[i]$.write(v) This operation is implemented by the client lines 01-04 and the server lines 09-12. Process p_i first increases wsn_i and r-broadcasts the message WRITE(v, wsn_i). Let us remark that this is the only place where the algorithm uses the underlying reliable broadcast abstraction. The process p_i then waits for acknowledgments (message WRITE_DONE(wsn_i)) from a quorum including strictly more than $\frac{n+t}{2}$ processes, and finally terminates the write operation. As we have seen (Lemmas 1 and 2), the intersection of any two quorums of such a size contains at least one correct process. This property will be used to prove the consistency of the register $REG[i]$.

When p_i is r-delivered a message WRITE(v, wsn) from a process p_j, it first waits until the previous write of $REG[j]$ by p_j has locally terminated (line 09). Hence, all the writes of $REG[j]$ are locally processed by p_i in the order they have been issued by p_j. When $|reg_i[j]| + 1 = wsn$, p_i adds v at the tail of $reg_i[j]$ (line 10), and sends back an acknowledgment to p_j (line 11).

Finally, p_i sends to each process p_k the customized message READ_VALUE($j, approx\_rsn_i[k, j], reg_i[j]$) to inform p_k that this write from p_j has locally been taken into account at p_i (line 12). As we will see, this is to help terminate the invocations of $REG[j]$.read() issued by the correct processes.

The operation $REG[j]$.read() This operation is implemented by the client lines 05-08 and the server lines 12-16. When p_i wants to read $REG[j]$, it first broadcasts a read request appropriately identified (message READ($j, rsn_i[j]$), lines 05-06), and waits for acknowledgment messages carrying the same history h (message READ_VALUE($j, rsn_i[j], h$)), from a quorum of strictly more than $\frac{n+t}{2}$ distinct processes (lines 07). When p_i stops waiting, it knows that, when they sent their acknowledgments, the history of $REG[j]$ was equal to h for strictly more than $\frac{n+t}{2}$ processes. When this occurs, p_i returns this history h as the result of the read operation (line 08).

On its server side, when a process p_i receives a read request message READ(j, rsn) from a process p_k, it first checks if its view of the read of $REG[j]$ by p_k is "late", i.e., if $approx\_rsn_i[k, j] < rsn$ (line 13). If it is the case, p_i updates $approx\_rsn_i[k, j]$ (line 14), and sends by return to p_k the message READ_VALUE($j, rsn, reg_i[j]$), to inform it of its current value of $REG[j]$ (line 15). If $approx\_rsn_i[k, j] \geq rsn$, the read request is an old message and p_i ignores it.

Comparing with the crash failure model It is known that the algorithms implementing an atomic register on top of an asynchronous message-passing system prone to process crashes, require that "read have to write" [1]. More precisely, before returning a value, a process must write this value to ensure atomicity. Doing so, it is not possible that two sequential read invocations, both

concurrent with a write invocation, are such that the first read obtains the new value while the second read obtains the old value (demanding the reader to write the value it is about to return guarantees that there is no new/old inversion [22]).

As Byzantine failures are more severe than crash failures, the algorithm of Figure 1 needs to use a mechanism analogous to the "read have to write" to prevent new/old inversion from occurring. This is done by sending to each process p_k the customized message READ_VALUE$(j, approx_rsn_i[j,k], reg_i[j])$ issued at line 12. These messages play the role of writes, that allow the wait statement of line 07 to always terminate with a correct history value for $REG[j]$.

Message complexity It follows from the previous discussion that, in addition to a reliable broadcast, each write generates $O(n)$ messages WRITE_DONE() and $O(n^2)$ messages READ_VALUE()), and each read generates at most $2n$ messages. Hence, while the algorithm presented in [1] requires the assumption $t < n/2$ (which is a necessary and sufficient requirement on the model parameter t) and $O(n)$ messages to implement an SWMR atomic register in the crash failure model, the proposed algorithm requires the assumption $t < n/3$ (which is shown to be necessary and sufficient in Section 3.4), and $O(n^2)$ messages plus a reliable broadcast, to implement SWMR atomic registers in the Byzantine asynchronous message-passing model.

3.3 Proof of the Construction

Lemma 3 *Let $n \geq 3t+1$. If p_i is correct and invokes $REG[i]$.write(), its invocation terminates.*

Proof Let us first consider the first invocation of $REG[i]$.write() by p_i. Due to the RB-termination property of the underlying reliable broadcast abstraction invoked by p_i at line 02, each correct process p_j r-delivers the message WRITE$(-,1)$. When this occurs, the predicate of line 09 is satisfied, and p_j sends the message WRITE_DONE(1) to p_i. As there are strictly more than $\frac{n+t}{2}$ correct processes (since $n > 3t$, $\frac{n+t}{2} = n - t - \frac{n-3t}{2} < n - t$), it follows that p_i cannot remain blocked forever at line 03, and the write invocation terminates.

Let us now observe that each correct process p_j processes the messages WRITE$(-, wsn)$ r-broadcast by p_i in their sequence number order (line 09). As p_i uses the sequence of the consecutive integers to define its next sequence number wsn, it follows that the same reasoning as for $wsn = 1$ applies iteratively to the subsequent invocations of $REG[i]$.write() issued by p_i, which concludes the proof. $\square_{Lemma\ 3}$

Lemma 4 (Single history). *Let p_i be a (correct or faulty) process. It exists a history H_i such that any invocation of the operation $REG[i]$.read() by a correct process returns a prefix of H_i.*

Proof Given a register $REG[i]$, let H_i be the the sequence of values defined by the sequence of messages WRITE(v, wsn) r-delivered from p_i to the correct processes, where the order of the values in H_i is defined by the sequence numbers $wsn = 1, 2$, etc. It follows from the RB-integrity and RB-uniformity properties of the reliable broadcast, and the lines 09-10 of the algorithm, that the history $reg_j[i]$ of any correct process p_j is a prefix of H_i. Consequently, the messages READ_VALUE$(-,-,reg_k[i])$ sent by any correct process p_k at line 12 or 15 are such that $reg_k[i]$ is a prefix of H_i. It then follows that, when a correct process p_j computes h at line 07, this history comes from strictly more than $\frac{n+t}{2} \geq t$ processes, thus at least from one correct process, and is consequently a prefix of H_i. Hence, any invocation of the operation $REG[i]$.read() by a correct process returns a prefix of H_i. $\square_{Lemma\ 4}$

Lemma 5 *Let $n \geq 3t+1$. If p_j is correct and invokes $REG[i]$.read(), its invocation terminates.*

Proof The proof is by contradiction. let us assume that a correct process p_j invokes $REG[i]$.read() and this invocation never terminates. This means that the predicate associated with the wait statement of line 07 remains false forever, namely, $\nexists h$ such that the message READ_VALUE$(i, rsn_j[i], h)$ is received from strictly more than $\frac{n+t}{2}$ different processes.

As p_j is correct, it broadcasts the request message READ(i, sn) where $sn = rsn_j[i]$ (line 06), and this message is received by all correct processes. Moreover, sn is the greatest sequence number ever used by p_j to read $REG[i]$, and, due to the contradiction assumption, $rsn_j[i]$ keeps forever the value sn.

Let p_k be any correct process. When p_k receives READ(i, sn) from p_j, the predicate $approx\_rsn_k[j,i] < sn$ is satisfied (line 13). This is because sn is greater than all previous sequence numbers used by p_j to read $REG[i]$. It follows that p_k updates $approx\_rsn_k[j,i]$ to $sn = rsn_j[i]$, and sends to p_j the message READ_VALUE$(i, sn, reg_k[i])$ (lines 14-15). Moreover, as the read of $REG[i]$ by p_j never terminates, $approx\_rsn_k[j,i]$ remains forever equal to $sn = rsn_j[i]$.

As the predicate of line 07 remains forever false at p_j, and p_j receives at least $(n-t)$ messages READ_VALUE$(i, sn, -)$ (one from each correct process), it follows that p_j receives at least two messages READ_VALUE(i, sn, h) and READ_VALUE(i, sn, h') such that h is a strict prefix of h or h' is a strict prefix of h (this is because, due to Lemma 4, both h and h' are prefixes of H_i). Without loss of generality, let h' be the shortest history received, and h be the longest.

Due to the RB-uniformity property of the underlying broadcast abstraction, it follows that all the correct processes r-delivers the same messages WRITE() from p_i, and process them in the same order (line 09). Let p_k be a correct process. It follows directly from the code of the algorithm that, each time p_k adds a value to $reg_k[i]$ (line 10), it sends a message READ_VALUE$(i, sn, reg_k[i])$ to p_j (line 12). It follows that there is a finite time after which p_j has received the very same message READ_VALUE(j, sn, h) from strictly more than $\frac{n+t}{2}$ different processes. The predicate of line 07 becomes then satisfied. This contradicts the initial assumption, and the lemma follows. $\square_{Lemma\ 5}$

Lemma 6 (Read followed by write). *Let p_i and p_j be two correct processes. If* op_read$[j, i, h]$ *terminates before* op_write$[i, wsn]$ *starts, then the returned history h is such that $|h| < wsn$.*

Proof As p_i is correct and has not yet invoked op_write$[i, wsn]$ when p_j terminates op_read$[j, i, h]$, it follows that no correct process r-delivers a message WRITE(v, wsn) from p_i before op_read$[j, i, h]$ terminates. Hence, when they received from p_j the message READ(i, sn) generated by op_read$[j, i, -]$, strictly more than $\frac{n+t}{2}$ different processes p_x (i.e., strictly more than $\frac{n-t}{2}$ correct processes) returned the same message READ_VALUE(i, sn, h) where $h = reg_x[i]$ and $|h| < wsn$. Consequently, the history h returned by p_j is smaller than wsn. $\square_{Lemma\ 6}$

Lemma 7 (Write followed by read). *Let $n > 3t$. Let p_i and p_j be two correct processes. If* op_write$[i, wsn]$ *terminates before* op_read$[j, i, h]$ *starts, then the returned history h is such that $|h| \geq wsn$.*

Proof It follows from line 03 and lines 10-11 that, when op_write$[i, wsn]$ terminates (which implies before op_read$[j, i, h]$ starts), there is a quorum Q_W of strictly more than $\frac{n+t}{2}$ processes p_x such that $|reg_x[i]| \geq wsn$. Moreover, the invocation op_read$[j, i, h]$ obtains the same message READ_VALUES(i, sn, h) from a quorum Q_R including strictly more than $\frac{n+t}{2}$ correct processes. According to Lemmas 1 and 2, it follows that $Q_W \cap Q_R$ contains at least one correct process p_y. As this process is such that $|reg_y[i]| \geq wsn$, it follows that $|h| \geq wsn$. $\square_{Lemma\ 7}$

Lemma 8 (No read inversion). *Let $n > 3t$. Let p_j and p_k be two correct processes. If* op_read$[j, i, h]$ *terminates before* op_read$[k, i, h']$ *starts, we have then $|h| \leq |h'|$.*

Proof To terminate, op_read$[j, i, h]$ received the same message READ_VALUE(j, rsn, h) from a quorum Q_{R1} of strictly more than $\frac{n+t}{2}$ different processes. Similarly, let Q_{R2} be the quorum of strictly more than $\frac{n+t}{2}$ processes that allowed op_read$[k, i, h']$ to terminate.

According to Lemmas 1 and 2, there is a correct process p_x in $Q_{R1} \cap Q_{R2}$. As, (a) p_x is correct and sent the message READ_VALUE(j, rsn, h) to p_j, and later sent the message READ_VALUE(j, rsn, h') to p_k, and (b) $reg_x[i]$ can only increase, we necessarily have $|h| \leq |h'|$, which concludes the proof of the lemma. $\square_{Lemma\ 8}$

The following theorem follows from the previous lemmas 3-8.

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

To better understand and capture the behavior of the algorithm, the linearization points associated with the executions of (a) read and write operations issued by the correct processes, and (b) the writes by faulty processes, whose values have been read by correct processes, are described in Appendix B.

3.4 The Condition $t < n/3$ is Necessary and Sufficient

Theorem 2 *The condition $t < n/3$ is necessary and sufficient to built an SWMR atomic register in $\mathcal{BAMP}_{n,t}[\emptyset]$.*

Proof The algorithm presented in the previous section has shown that the condition $t < n/3$ is sufficient to built an SWMR atomic register. So, the rest of the proof addresses the necessity part of the condition.

The proof is by contradiction. Let us assume that there is an algorithm A that builds an atomic register in $\mathcal{BAMP}_{n,t}[n \leq 3t]$, which means that it satisfies the R-consistency and R-termination properties stated in Section 3.1. For simplicity, and without loss of generality, we consider the largest possible value of t, i.e., $n = 3t$. Let us first observe that to guarantee R-termination, a process cannot wait for messages from more than $n - t = 2t$ processes.

Let us partition the set of processes into three sets Q_1, Q_2 and Q_3, each of size (at most) t. Moreover, let us consider two processes $p_1 \in Q_1$ and $p_2 \in Q_2$.

Let us consider a first execution E_1 defined as follows.

- The set of Byzantine processes is Q_1; these processes do not send messages and appear as crashed.
- The process $p_2 \in Q_2$ writes a value v in $REG[2]$. Due to the R-termination property of the algorithm A, the invocation of $REG[2].\mathsf{write}(v)$ by p_2 terminates. Let τ_w be the time instant at which this write terminates.

Let E_2 be a second execution defined as follows.

- All processes are correct, but the processes of Q_2 execute no step before τ_r (defined below).
- After τ_w, the process $p_1 \in Q_1$ reads the register $REG[2]$. Due to the R-termination property of the algorithm A, and because the processes of Q_2 could be Byzantine, the invocation of $REG[2].\mathsf{read}()$ by p_1 terminates. Let τ_r be the time instant at which this read terminates. As, no process of Q_2 executes a step before the read terminates, p_2 does not write $REG[2]$ before τ_r. Consequently, according to the R-consistency property *read followed by write*, $REG[2]$ has still its initial value ϵ. It follows that the read operation by p_1 returns this initial value.

Let us finally consider E_3, a third execution defined as follows.

- The set of Byzantine processes is Q_3, and these processes behave exactly as in E_1 with respect to the processes of Q_2, and exactly as in E_2 with those of Q_1.
- The messages sent by the processes of Q_1 to the processes of Q_2 and by the processes of Q_2 to the processes of Q_1 are delayed until after τ_r.
- The messages exchanged between themselves by the processes of $Q_2 \cup Q_3$ are received at exactly the same time instants as in E_1. Similarly, the messages exchanged between themselves by the processes of $Q_1 \cup Q_3$ are received at exactly the same time instants as in E_2.
- As the very same time instants as in E_1, process $p_2 \in Q_2$ writes a value v in $REG[2]$. Since, from the point of view of the processes of Q_2, the executions E_1 and E_3 are indistinguishable, the invocation of $REG[2].\mathsf{write}(v)$ by p_2 terminates at τ_w too.
- As in execution E_2, after τ_w the process $p_1 \in Q_1$ reads the register $REG[2]$. Since, from the point of view of the processes of Q_1, the executions E_2 and E_3 are indistinguishable, the invocation of $REG[2].\mathsf{read}()$ by p_1 terminates at τ_r and returns ϵ. But this violates the R-Consistency property *write followed by read*, which contradicts the existence of Algorithm A.

$\square_{Theorem\ 2}$

4 A Few Simple Abstractions on Top of SWMR Atomic Registers

This section presents two simple uses of the previous construction of an array of n SWMR atomic registers. Another example is given in Appendix C [4]. In these algorithms, the simplicity comes from the SWMR atomic registers built on top of a message-passing system. This SWMR atomic register abstractions provides us with an abstraction level that allows for simple solutions.

The classical notations for atomic registers are used in the following, namely, given an atomic register XX, $XX \leftarrow v$ stands for $XX.\mathsf{write}(v)$, and $x \leftarrow XX$ stands for $x \leftarrow XX.\mathsf{read}(v)$. Moreover, only the last value of the sequence returned by $XX.\mathsf{read}(v)$ is considered.

4.1 One-Shot Write-Snapshot

This section describes a very simple one-shot write-snapshot object. This object provides the processes with a single operation denoted write_snapshot(). This operation has a single parameter, namely the value that the invoking process wants to write in the snapshot object. A process p_i can invoke write_snapshot() at most once. This operation returns to the invoking process p_i a set $output_i$ made up of pairs $\langle j, w\rangle$, where w is the value written by the process p_j.

Specification A one-shot write-snapshot object is defined by the following properties.

- Termination. The invocation of write_snapshot(v) by a correct process p_i terminates.
- Self-inclusion. If p_i is correct and invokes write_snapshot(v), then $\langle i, v\rangle \in output_i$.
- Containment. If both p_i and p_j are correct and invoke write_snapshot(), then $output_i \subseteq output_j$ or $output_j \subseteq output_i$.
- Validity. If $\langle j, w\rangle \in output_i$, then p_j wrote w in the write_snapshot object.

```
operation write_snapshot(v_i):
(01) IN[i] ← v_i;
(02) for x ∈ {1, ..., n} do aux1[x] ← IN[x] end for;
(03) for x ∈ {1, ..., n} do aux2[x] ← IN[x] end for;
(04) while (aux1 ≠ aux2) do
(05)    aux1 ← aux2;
(06)    for x ∈ {1, ..., n} do aux2[x] ← IN[x] end for
(07) end while;
(08) output_i ← { ⟨j, aux1[j]⟩ | aux1[j] ≠ ⊥ };
(09) return(output_i).
```

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

The algorithm The internal representation of the write-snapshot object is an array of SWMR atomic registers. This array, denoted $IN[1..n]$, is obtained from the construction of Figure 1 (slightly and easily modified at line 09 to ensure that a process writes at most once its register). It is assumed that $IN[1..n]$ is initialized to $[\bot, \ldots, \bot]$.

The algorithm implementing the operation write_snapshot() is very simple (Figure 2). The invoking process p_i first deposits its value in $IN[i]$ (line 01), and issues an asynchronous "sequential double scan" (lines 02-03). If the sequential double scan is not successful (line 04), it executes other double scans (lines 02-03) until a pair of them is successful, i.e., $aux1[1..n] = aux2[1..n]$. After the successful double scan, p_i computes its output $output_i$, namely, a set containing the pairs $\langle j, w\rangle$ such that w is the value written by p_j (as known by the last successful double scan).

The termination of the algorithm is an immediate consequence of the bounded number of processes, and the fact that each register $IN[i]$ is a one-write register. The validity and self-inclusion are trivial. The containment property follows from the fact that the number of non-$\bot$ entries can only increase.

4.2 Correct-Only Agreement

A correct-only agreement object is a one-shot object that provides the processes with a single operation denoted correct_only_agreement(). This operation is used by each process to propose a value and decide a set of values. A decided set contains only values proposed by correct processes and the decided sets satisfy the containment property. According to the topological bounds stated in [9], the problem captured by the correct-only agreement object can be solved if and only if $n > (\dim(I) + 2)t$, where $\dim(I)$ is the dimension of the colorless input complex I with which the problem is instantiated. In our context, this necessary and sufficient condition boils down to $n > (w + 1)t$, where $w > 1$ is the maximal number of distinct values that can be proposed by the correct processes in any execution (in the topology parlance, $w - 1$ is the greatest dimension of a simplex of the colorless input complex I).

Specification A correct-only agreement object is defined by the following properties. As in the previous section, $output_i$ denotes the set of values output by a correct process p_i.

- Termination. The invocation of correct_only_agreement() by a correct process p_i terminates.
- Containment. If both p_i and p_j are correct and invoke correct_only_agreement(), then $output_i \subseteq output_j$ or $output_j \subseteq output_i$.
- Validity. The set $output_i$ of a correct process p_i is not empty and contains only values proposed by at least one correct process.

The algorithm The algorithm implementing the operation correct_only_agreement(), is described in Figure 3. As the reader can see, this algorithm is obtained from a simple adaptation of the algorithm implementing the operation write_snapshot(). The modified parts are the predicate used at line 04, and the computation of the output at line 08.

More precisely, a successful double scan is now necessary to exit the while loop, but is no longer sufficient. In addition, a process p_i must observe there is at least one value that has been proposed by $(t + 1)$ processes (i.e., by at least one correct process). Finally, the output $output_i$ contains all the values that, from p_i's point of view, have been proposed by at least $(t + 1)$ processes.

As in the previous section, the containment property is a consequence of the fact that the writes in the array $IN[1..n]$ are atomic, and the number of non-$\bot$ entries can only increase. The termination property is a consequence of the following observations: (a) there is a bounded number of processes, (b) the atomic registers are one-write registers, and (c) the condition $n > (w + 1)t$. The validity follows from the condition $n > (w + 1)t$ (hence there is at least one value that appears $(t + 1)$ times), and the predicate of line 04.

```
operation correct_only_agreement(v_i):
(01) IN[i] ← v_i;
(02) for x ∈ {1, ..., n} do aux1[x] ← IN[x] end for;
(03) for x ∈ {1, ..., n} do aux2[x] ← IN[x] end for;
(04) while [(aux1 ≠ aux2) ∨ (∄v : |{j : aux1[j] = v}| > t)] do
(05)    aux1 ← aux2;
(06)    for x ∈ {1, ..., n} do aux2[x] ← IN[x] end for
(07) end while;
(08) output_i ← { v : |{j : aux1[j] = v}| > t};
(09) return(output_i).
```

Figure 3: Correct-only agreement in $\mathcal{BAMP}_{n,t}[t < n/(w+1)]$ (code for p_i)

Remark Both the one-write write-snapshot object and the correct-only agreement object share the same termination and containment properties. They can be seen as dual the one from the other in the following sense. One-write write-snapshot satisfies self-inclusion and a weak validity property, while correct-only agreement is not required to satisfy self-inclusion, but is constrained by a stronger validity property. As we have seen, both objects can be implemented by the same generic algorithm whose instances differ essentially in the predicate used to exit the while loop (line 04).

5 Solving Multidimensional Approximate Agreement

This section shows how an algorithm designed for the Byzantine asynchronous message-passing system model can be easily adapted to the Byzantine asynchronous shared memory model introduced in Section 3. The shared memory abstraction being at a higher abstraction level than message-passing, the shared memory version of the algorithm seems much easier to understand.

5.1 The Multidimensional Approximate Agreement Problem

Approximate agreement The ϵ-approximate agreement problem has been introduced in the context of synchronous and asynchronous message-passing systems where processes can commit Byzantine failures [8]. Each process proposes a value in $\mathbb{R}$, and each correct process has to decide a value such that: (a) any decided value is in the range of the values proposed by the correct processes (validity), and (b) the distance between any two values decided by correct processes is at most ϵ (agreement). It is shown in [8] that $t < n/3$ is a necessary and sufficient condition on the number of Byzantine processes when one has to solve ϵ-approximate agreement in both synchronous and asynchronous systems.

Multidimensional approximate agreement The ϵ-approximate agreement problem has been generalized in [18] to the case where each input value is a point in $\mathbb{R}^d$ (space of dimension d). Such a point is defined by a size d vector (one entry per coordinate). The problem is then defined by the following properties. Let multi_approx_agreement() be the associated operation invoked by processes.

- Termination. The invocation of multi_approx_agreement() by a correct process p_i terminates.
- Validity. The value decided by any correct process is a point of $\mathbb{R}^d$ within the convex hull of the points proposed by the correct processes.
- Agreement. The Euclidean distance between any two points decided by correct processes is at most ϵ.

It is shown in [18] that $n > (d+2)t$ is a sufficient and necessary requirement for the problem to be solved.

5.2 Solving Multidimensional Approximate Agreement

An algorithm solving the multidimensional approximate agreement problem on top of the shared memory abstraction is presented in Figure 4. This algorithm is an adaptation of Mendes-Herlihy's algorithm suited to asynchronous message-passing with up to $t < n/3$ Byzantine processes [18].

Shared memory: arrays of n SWMR atomic registers The processes cooperate through the following arrays of SWMR atomic registers. Each register is written at most once by its writer. An input value is a d-dimensional vector (coordinates of the input of p_i in $\mathbb{R}^d$).

- $VAL[1..n]$: array such that $VAL[i]$ is written by p_i to publish its input.

- $VIEW[1..n]$: array such that $VIEW[i]$ contains p_i's view of the inputs.
- $EST[1..][1..n]$: array such that $EST[r][i]$ contains p_i's estimate of its decision value at round r.
- $MAX[1..n]$: array such that $MAX[i] = r$ means that p_i has estimated that r rounds are enough to reach approximate agreement. Initially, $MAX[i] = +\infty$.

The value of any of these arrays is obtained with a collect() operation, which (differently from a snapshot) is an asynchronous and unordered read of the corresponding entries of the array.

Procedures used in the algorithm Let X denote a multiset of values of $\mathbb{R}^d$ and $\mathcal{CH}(X)$ the convex hull of such a multiset. Algorithm 4 uses the notion of *safe area* introduced in [18]. The safe area of X is defined by

$$\mathsf{Safe}_t(X) = \bigcap_{X' \subseteq X, |X'|=|X|-t} \mathcal{CH}(X').$$

Informally, this captures the region of $\mathbb{R}^d$ that is contained in all the convex hulls of the subsets of cardinal $n-t$ of $|X|$. If the values of $|X|$ are proposed by distinct processes, then this region is consequently contained in the convex hull of the subset of the values of X proposed by correct processes. It is shown in [18] that for any X such that $|X| > t(d+1)$, the safe area $\mathsf{Safe}_t(X)$ is non-empty [18, Lemma 3.6].

The procedures bary(S), MultiDimMidpoint(S), and SingleDimMaxDist(S), where S is a convex polytope of $\mathbb{R}^d$, are called locally by the processes; bary(S) denotes the barycenter of S, while the two later are defined as follows, by (where $s[x]$ designate the x^{th} coordinate of $s \in \mathbb{R}^d$:

$$\forall x \in \{1, \ldots, d\} \ : \ \mathsf{MultiDimMidpoint}(S)[x] = (\max\{s[x], s \in S\} + \min\{s[x], s \in S\})/2,$$

$$\text{and} \qquad \mathsf{SingleDimMaxDist}(S) = \max_{x \in \{1,\ldots,d\}} (\max\{s[x], s \in S\} - \min\{s[x], s \in S\}).$$

Local variables at each process p_i Each process p_i manages the following local data structures: two arrays $my\_view_i[1..n]$ and $views_i[1..n]$, both initialized to $[\perp, ..., \perp]$; a set of points $safe\_init_i$; a local estimate est_i of the point that will be locally decided; an array $vals_i[1..n]$ of estimates values; an array $max\_r_i[1..n]$ containing round number upper bounds; and r_i which contains p_i's current round number.

Behavior of a process p_i A process p_i first writes its input value v_i (point in $\in \mathbb{R}^d$) in $VAL[i]$, and collects a local view ($my\_view_i[1..n]$) including at least $(n-t)$ inputs (lines 01-02). Then, to make it public, p_i writes its view in $VIEW[i]$, and collects in $views_i$ the views of at least $(n-t)$ processes (lines 03-04). The process p_i then calculates in $safe\_init_i$ the barycenter of the safe area of these views (line 05). The parameters d and ϵ of the problem, and the set of barycenters $safe\_inits_i$ of its current instance, allow p_i to locally compute an upper bound on the number of rounds to be executed, which is written into the atomic register $MAX[i]$ (line 06). The set of barycenters $safe\_inits_i$ is also used to compute p_i's first estimate (est_i) of its decision value (line 07).

Then, process p_i starts a sequence of asynchronous rounds whose aim is to refine its current estimate est_i (lines 09-13) until it returns its last estimate value. This occurs when p_i attains a round r_i during which it sees a set of more than t processes –i.e., at least one correct process– such that each process p_j of this set posted in its atomic register $MAX[j]$ a last round upper bound smaller than r_i. This is captured by the outer termination predicate ($|\{x : \ max\_r_i[x] < r_i\}| > t$) used at lines 12 and 13.

During a loop iteration r_i, p_i first writes its current decision value estimate in $EST[r_i][i]$ (line 09), and repeatedly collects both current estimates of decision values written in $EST[r_i][1..n]$, and upper bounds of the last round number from $MAX[1..n]$, until either the outer termination predicate is satisfied, or it sees at least $(n-t)$ estimates computed at round r_i (first sub-predicate of line 11). The use of the outer termination predicate in the inner repeat loop (line 11) allows p_i not to remain blocked forever waiting for a correct process that has already terminated. Then, if it knows enough values deposited in $EST[r_i]$, p_i collects again current estimates of decision values written in $EST[r_i]$ (line 12), and computes a new estimate est_i (line 13). Finally, once a large enough number of rounds is reached (line 14), p_i returns (decides) its current estimate value (line 15).

5.3 Proof of the Algorithm

The proof of the correctness of the algorithm is similar to, and follows the structure of, the one of Mendes-Herlihy's algorithm [18].

The following lemmas prove that, at lines 03, 05, and 12, all the correct processes receive at least $n-t$ values in common. This replaces the guarantee given by the module denoted RBReceiveWitness() in [18], which is in charge on the reception of messages sent with an underlying reliable broadcast on FIFO channels.

```
operation multi_approx_agreement(v_i):
(01) VAL[i] ← v_i; r_i ← 0;
(02) repeat my_view_i[1..n] ← VAL.collect() until (|{x | my_view_i[x] ≠ ⊥}| ≥ n − t) end repeat;
(03) my_view_i[1..n] ← VAL.collect(); VIEW[i] ← my_view_i[1..n];
(04) repeat views_i[1..n] ← VIEW.collect() until (|{x | views_i[x] ≠ ⊥}| ≥ n − t) end repeat;
(05) views_i[1..n] ← VIEW.collect(); safe_inits_i ← {bary(Safe_t(X)) : X ∈ views_i};
(06) MAX[i] ← ⌈log_2 (√d/ε · SingleDimMaxDist(safe_inits_i))⌉;
(07) est_i ← bary(Safe_t(safe_inits_i));
(08) repeat r_i ← r_i + 1;
(09)       EST[r_i][i] ← est_i;
(10)       repeat vals_i[1..n] ← EST[r_i].collect(); max_r_i[1..n] ← MAX.collect()
(11)       until (|{x : vals_i[x] ≠ ⊥}| ≥ n − t) ∨ (|{x : max_r_i[x] < r_i}| > t) end repeat;
(12)       if (|{x : vals_i[x] ≠ ⊥}| ≥ n − t) then vals_i[1..n] ← EST[r_i].collect();
(13)                                              est_i ← MultiDimMidpoint(Safe_t(vals_i)) end if
(14) until (|{x : max_r_i[x] < r_i}| > t) end repeat;
(15) return(est_i).
```

Figure 4: Multidimensional approximate agreement in $\mathcal{BAMP}_{n,t}[t < n/(d+2)]$ (code for p_i)

Lemma 9 *For any pair correct processes p_i and p_j, $|my\_view_i \cap my\_view_j| \geq n - t$.*

Proof A correct process performs a collect at line 03 only if it observes at least $n-t$ values during its last collect at line 02. Without loss of generality, let p_i be the first process that starts its collect at line 03.

Let τ be the time at which p_i starts its collect at line 03. At least $(n-t)$ values have already been written in VAL by time τ (otherwise, p_i would not perform this collect). Because of the linearizability (atomicity) property of the registers, any collect that starts after τ will read the values already written in the array VAL, which concludes the proof of the lemma. $\square_{Lemma\ 9}$

The reasoning is the same for the next two lemmas. Lemma 10 is on the sets $view_i$ computed at line 05, while Lemma 11 is on the sets $vals_i$ computed at line 12.

Lemma 10 *For any pair correct processes p_i and p_j, $|views_i \cap views_j| \geq n - t$.*

Let $vals_{i,r}$ be the set of values observed at line 12 by process p_i at round r.

Lemma 11 *For any pair of correct processes p_i and p_j that perform a collect at line 12 during round r, $|vals_{i,r} \cap vals_{j,r}| \geq n - t$.*

The following lemma replaces Lemma 4.16 in [18]. It proves that all correct processes execute enough rounds. Let r_{min} be the minimum round estimate calculated by a correct process at line 06.

Lemma 12 *All correct processes run for at least r_{min} rounds.*

Proof A correct process p_i may exit the loop at lines 08-14 only if it observes at least $t+1$ estimates of the number of rounds (values written in MAX) lower than its round number. These $t+1$ values must include at least one value written by a correct process. Process p_i will thus execute at least r_{min} rounds. $\square_{Lemma\ 12}$

The following lemma guarantees that no correct process can run for an infinite number of steps.

Lemma 13 *All correct processes terminate their invocation of* multi_approx_agreement().

Proof The algorithm does not contain any wait statement and contains four loops.

- The loop at line 02. All the correct processes write in $VAL[1..n]$, thus all correct processes will eventually observe at least $(n-t)$ values during their collect and exit the loop.
- The loop at line 04. The reasoning is the same as previously. All the correct processes write in $VIEW$, thus all correct processes will eventually observe at least $(n-t)$ values during their collect and exit the loop.
- The loop at lines 10-11. Let r_{min_exit} be the smallest round after which a correct process p_i exits the loop. Thus, at any round $r \leq r_{min\_exit}$, each correct process writes a value in $EST[r]$. Because p_i exited the loop at round r_{min_exit}, it observed strictly more than t finite values smaller than r_{min_exit} in $MAX[1..n]$. Hence, any other correct process p_j, at round $r \geq r_{min\_exit}$, will then observe strictly more than t finite values smaller than r in $MAX[1..n]$ and exit the loop.
- The loop at lines 08-14. As all the correct processes write a finite value in $MAX[1..n]$, it follows that each of them will eventually observe at least than $(n-t)t$ finite values during a collect at line 10. It will consequently execute the loop at lines 08-14 a finite number of times, which concludes the proof of the lemma.

$\square_{Lemma\ 13}$

As noticed in [18], for the decided values to be within ϵ of each other, it is sufficient that they be within $\epsilon/\sqrt{d}$ of each other in every coordinate. The reader can find in [18] the proof of Lemma 14 and Lemma 15 stated below. In [18], their proofs depend only on the guarantees given by the module RBReceiveWitness() devoted to the management of message reception. These guarantees are replaced here by the previous Lemmas 9, 10, and 11.

Lemma 14 *For any correct process p_i, after $R \geq \log_2\left(\sqrt{d}/\epsilon \cdot \mathsf{SingleDimMaxDist}(safe\_inits_i)\right)$ rounds, the values of the correct processes are within distance $\epsilon/\sqrt{d}$ of each other at any dimension x* [18, Lemma 4.10].

Lemma 15 *At any round, for any correct process p_i, est_i is within the convex hull of the inputs of the correct processes* [18, Lemma 4.17].

Using the previous lemmas, we obtain the following theorem.

Theorem 3 *The algorithm presented in Figure* 4 *is a correct implementation of multidimensional approximate agreement in the $\mathcal{BAMP}_{n,t}[t < n/(d+2)]$ model.*

Proof The proof follows from Lemma 13 (termination), Lemmas 12 and 14 (agreement) and Lemma 15 (validity). $\square_{Theorem\ 3}$

6 Conclusion

This paper has first proposed a clean notion of atomic registers in the presence of Byzantine failures, and has then shown how to build it on top of a Byzantine asynchronous message-passing system. More precisely, an algorithm building a shared memory abstraction, made up of n single-writer/multi-reader atomic registers, has been presented and proved correct. The paper has also shown that $t < n/3$ is a necessary and sufficient condition for such an algorithm.

The paper has then presented distributed algorithms suited to such a shared memory abstraction, which can cope with up to t Byzantine processes. The simplicity of these algorithms constitutes a strong motivation for the use of such a shared memory abstraction in the presence of Byzantine processes. As, for a lot of problems, algorithms are more difficult to design and prove correct in a message-passing system than in a shared memory system, the proposed abstraction should allow easier designs and proofs for other algorithms that have to cope with Byzantine failures.

Acknowledgments

This work has been partially supported by the French ANR project DISPLEXITY devoted to computability and complexity in distributed computing.

References

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

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

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

[4] Borowsky E. and Gafni E., Immediate atomic snapshots and fast renaming. *Proc. 12th ACM Symposium on Principles of Distributed Computing (PODC'93)*, pp. 41-51, 1993.

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

[6] Cachin Ch., Kursawe K., and Shoup V., Random oracles in Constantinople: practical asynchronous Byzantine agreement using cryptography. *Proc. 19th Annual ACM Symposium on Principles of Distributed Computing (PODC'00)*, ACM Press, pp. 123-132, 2000.

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

[8] Dolev D., Lynch N.A., Pinter S.S., Stark E.W., and Weihl W.E., Reaching approximate agreement in the presence of faults. *journal of the ACM* 33(3):499-516, 1986.

[9] Herlihy M.P., Kozlov D., and Rajsbaum S., *Distributed computing through combinatorial topology*, Morgan Kaufmann/Elsevier, 336 pages, 2014 (ISBN 9780124045781).

[10] 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.

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

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

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

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

[15] Malkhi D., Merritt M., Reiter M.K., and Taubenfeld G., Objects shared by Byzantine processes. *Distributed Computing*, 16(1):37-48, 2003.

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

[17] Martin J.-Ph. and Alvisi L., Fast Byzantine consensus. *IEEE Transactions on Dependable and Secure Computing*, 3(3):202-215, 2006.

[18] Mendes H. and Herlihy M., Multidimensional approximate agreement in Byzantine asynchronous systems. *Proc. 46th ACM Symposium on Theory of Computing (STOCS'13)*, ACM Press, pp. 391-400, 2013.

[19] Mostéfaoui A., Moumem H., and Raynal M., Signature-free asynchronous Byzantine consensus with $t < n/3$ and $O(n^2)$ messages. *Proc. 33rd ACM Symposium on Principles of Distributed Computing (PODC'14)*, ACM Press, 2014.

[20] Mostéfaoui A. and Raynal M., Communication and agreement abstractions in the presence of Byzantine processes. *Tech Report 2015*, 24 pages, IRISA, Université de Rennes (France), 2014.

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

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

[23] Raynal M., *Fault-tolerant agreement in synchronous message-passing systems*. Morgan & Claypool Publishers, 165 pages, 2010 (ISBN 978-1-60845-525-6).

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

A An Algorithm Implementing the Reliable Broadcast Abstraction

The r-broadcast algorithm presented in Figure 5 assumes $t < n/3$. It is a simple variant of Bracha's algorithm [5]. The operation broadcast XXX(), where XXX is a message tag, is a simple macro-operation standing for "**for** $x \in \{1, \ldots, n\}$ **do** send XXX() to p_x **end for**" (the sending order is arbitrary, which means that, if the sender crashes while executing this statement, an arbitrary subset of processes will receive the message).

Each process p_i manages a local array $next_i[1..n]$, where $next_i[j]$ is the sequence number sn of the next application message (namely, APP$(-, sn)$) from p_j, that p_i will process (line 03). Initially, for all i, j, $next_i[j] = 1$.

When a process p_i invokes R_broadcast APP(v, sn), it broadcasts the message APP(v, sn) (line 01) where sn is its next sequence number. in its "server" role, the behavior of a process p_i is as follows.

- When a process p_i receives a message APP(v, sn) from a process p_j for the first time, it first waits until it can process this message (line 03). Process p_i then broadcasts a message ECHO(j, v, sn) (line 04). If the message just received is not the first message APP$(-, sn)$, p_j is Byzantine and the message is discarded.

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

when a message APP(v, sn) from p_j is received:
(02) if APP(−, sn)) never received from p_j
(03)    then wait (next_i[j] = sn);
(04)         broadcast ECHO(j, v, sn)
(05) end if.

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

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

Figure 5: Generalized Reliable Broadcast in $\mathcal{BAMP}_{n,t}[t < n/3]$, (code for process p_i)

- Then, when p_i has received the same message $\text{ECHO}(j, v, sn)$ from "enough" processes (where "enough" means "strictly more than $(n + t)/2$ different processes"), and has not yet broadcast a message $\text{READY}(j, v, sn)$, it does so (lines 06-09).

  The aim of (a) the messages $\text{ECHO}(j, v, sn)$, and (b) the cardinality "strictly greater than $(n+t)/2$ processes", is to ensure that no two correct processes can r-deliver distinct messages from p_j (in the case where p_j is Byzantine). The aim of the messages $\text{READY}(j, v, sn)$ is related to the liveness of the algorithm. Namely, its aim is to allow the r-delivery by the correct processes of the very same message $\text{APP}(v, sn)$ from p_j, and this must always occur if p_j is correct. It is nevertheless possible that a message r-broadcast by a Byzantine process p_j be never r-delivered by the correct processes.
- Finally, when p_i has received the message $\text{READY}(j, v, sn)$ from $(t + 1)$ different processes, it broadcasts the same message $\text{READY}(j, v, sn)$, it not yet done. This is required to ensure the RB-termination property. If p_i has received "enough" messages $\text{READY}(j, v, sn)$ (as before "enough" means "from strictly more than $(n+t)/2$ different processes"), it r-delivers the message $\text{APP}(v, sn)$ r-broadcast by p_j.

Proofs that this algorithm satisfies the properties defining the reliable broadcast abstraction can be found in [5, 20].

B Linearization Points Associated with SWMR Registers

To better understand and capture the behavior of the algorithm, we exhibit below a *linearization* order [10]. A linearization point is a point of the timeline (defined from an omniscient external observer point of view) at which a read or write operation issued by a correct process (or a write by a faulty process whose value has been read by a correct process) appears to have been instantaneously executed. For an operation issued by a correct process, this point has to lie between the beginning and the end of the corresponding execution.

Let us notice that a read by a Byzantine process may return any value. Hence, whatever the value it returns such a read invocation does not need to be linearized. We have the same for a write by a Byzantine process, whose value is never returned by a read from a correct process.

According to the Lemmas 4, 6, 7, and 8, we can define the following linearization points.

- An invocation of the operation $REG[i].\mathsf{write}(v)$ issued by a correct process p_i is linearized at the time $\min(\tau_w, \tau_r)$ where
  - τ_w is the end of the wait statement executed by p_i at line 03, and
  - τ_r is the end of the wait statement of line 07 executed by the first correct process that returns a history including the value v.

- A write of a value v into $REG[i]$ by a Byzantine process is linearized at the end of the wait statement of line 07 executed by the first correct process that returns the value v.
- Let op_read$[i, j, h]$ such that the last value of h is v. Let τ_b and τ_e be the times at which op_read$[i, j, h]$ starts and terminates, respectively. Let τ_w be the linearization point at which v has been added to $REG[j]$.

  Let us observe that $\tau_w < \tau_e$ (otherwise, the history h returned by op_read$[i, j, h]$ could not end with v). There are two cases.
  - If $\tau_w < \tau_b$: op_read$[i, j, h]$ is linearized at time τ_b.
  - If $\tau_b < \tau_w$: op_read$[i, j, h]$ is linearized just after τ_w. If two or more read invocations must be linearized just after the same time τ, their linearization times are ordered according to their starting times.

C Immediate Snapshot

The immediate snapshot object has been introduced in [4]. Such an object is a one-shot object that provides the processes with a single operation, denoted immediate_snapshot(). When a process invokes this operation, it proposes a value, and obtains an output which is a set of pairs similar to the one obtained in the one-shot write-snapshot.

Specification in the crash failure model When considering the asynchronous crash failure model, a one-shot immediate snapshot object is defined by the following properties. As in the previous section, $output_i$ denotes the set of values output by a correct process p_i.

- Termination. The invocation of immediate_snapshot() by a correct process p_i terminates.
- Validity. If $\langle i, v_i\rangle \in output_j$, then p_i invoked immediate_snapshot(v_i).
- Self-inclusion. If p_i returns from immediate_snapshot(), then $\langle i, v_i\rangle \in output_i$.
- Containment. If p_i and p_j return from immediate_snapshot(), then $output_i \subseteq output_j$ or $output_j \subseteq output_i$.
- Immediacy. $[(\langle i, v_i\rangle \in output_j) \wedge (\langle j, v_j\rangle \in output_i)] \Rightarrow (output_i = output_j)$.

Algorithm The implementation of immediate_snapshot() in the crash failure model given in [4] is described in Figure 6 (the reader can consult Chapter 8 of [24], for a pedagogical presentation and a proof of this algorithm).

```
operation immediate_snapshot(v_i) is
(01) REG[j] ← v_i;
(02) repeat LEVEL[i] ← LEVEL[i] − 1;
(03)       for each j ∈ {1, ..., n} do level_i[j] ← LEVEL[j] end for;
(04)       view_i ← {x | level_i[x] ≤ level_i[i]}
(05) until (|view_i| ≥ level_i[i]) end repeat;
(06) let output_i = { ⟨x, REG[x]⟩ : x ∈ view_i };
(07) return(output_i).
```

Figure 6: Immediate snapshot algorithm (code for p_i) [4]

The algorithm uses two arrays of SWMR atomic registers denoted $REG[1..n]$ and $LEVEL[1..n]$, initialized to $[\perp, ..., \perp]$ and $[n + 1, ..., n + 1]$, respectively. Only p_i can write $REG[i]$ and $LEVEL[i]$).

A process p_i first writes its value in $REG[i]$ (line 01). The array $LEVEL[1..n]$ can be thought of as a ladder, where initially each process p_i is at the top of the ladder, namely, at level $(n + 1)$. Then p_i descends the ladder (line 02), one step after the other, according to predefined rules, until it stops at some level (or crashes). While descending the ladder, p_i registers its current position in the ladder in the atomic register $LEVEL[i]$. After it has stepped down from one ladder level to the next one, p_i computes a local view (denoted $view_i$) of the progress of the other processes in their descent of the ladder. That view contains the processes p_x seen by p_i at the same or a lower ladder level (i.e., such that $level_i[x] \leq LEVEL[i]$, line 04). Then, if the current level ℓ of p_i is such that p_i sees at least ℓ processes in its view (i.e., processes that are at its level or a lower level), p_i stops at the level ℓ of the ladder. Finally, p_i computes $output_i$, which is a set of pairs $\langle x, v_x\rangle$ determined from the values of its last view $view_i$. This concise algorithm is such that the sets $view_i$ of the processes that terminate the algorithm, satisfy the following main property: if $|view_i| = \ell$, p_i stopped at the level ℓ, and there are ℓ processes whose current level is $\leq \ell$. From this property, follow the validity, self-inclusion, containment and immediacy properties that define the one-shot immediate snapshot object.

From the crash failure model to the Byzantine failure model Interestingly, the previous algorithm still works when considering that the arrays of SWMR atomic registers $REG[1..n]$ and $LEVEL[1..n]$ are implemented in $\mathcal{BAMP}_{n,t}[t < n/3]$ with the construction of Figure 1.

In this case, the previous specification and line 03 have to be slightly modified as follows. Remembering that $\mathcal{C}$ denote the set of correct processes, let $c(output_i) = output_i \cap \mathcal{C}$ (projection of $output_i$ on the correct processes). The specification of immediate snaspshot in the Byzantine failure model is defined by the following properties.

- Termination. The invocation of immediate_snapshot() by a correct process p_i terminates.
- Validity. If p_i and p_j are correct and $\langle i, v_i\rangle \in output_j$, then p_i invoked immediate_snapshot(v_i).
- Self-inclusion. If p_i is correct and invokes immediate_snapshot(), then $\langle i, v_i\rangle \in output_i$.
- Containment. If p_i and p_j are correct and invoke immediate_snapshot(), then $c(output_i) \subseteq c(output_j)$ or $c(output_j) \subseteq c(output_i)$.
- Immediacy. If p_i and p_j are correct, $[(\langle i, v_i\rangle \in output_j) \wedge (\langle j, v_j\rangle \in output_i)] \Rightarrow (c(output_i) = c(output_j))$.

As far as line 03 is concerned, we have the following. If p_j is Byzantine, $LEVEL[j]$ can contain an arbitrary value. Hence, instead of considering the content of $LEVEL[j]$, p_i considers the number of times that p_j has written this register. Due to the algorithm of Figure 1, this number is equal to $|LEVEL[j].\mathsf{read}()|$ (the length of the sequence returned by $LEVEL[j].\mathsf{read}()$). It follows that p_i has to assign the value $(n+1) - |LEVEL[j].\mathsf{read}()|$ to $level_i[j]$. It is easy to see that this quantity is the value of $LEVEL[j]$ when only crash failures are committed.