Chapter 9

Atomic Read/Write Registers in the Presence of Byzantine Processes

Theorem 18 (stated and proved in Section 5.4) has shown that $t < n/2$ is an upper bound on the resilience parameter t to build atomic read/write registers in the asynchronous crash process model $CAMP_{n,t}[\emptyset]$. Section 6.3 and Section 6.4 then presented an incremental construction of Single-Writer Multi-Reader (SWMR) and Multi-Writer Multi-Reader (MW-MR) atomic registers.

This chapter addresses the construction of SWMR atomic read/write registers (one per process) in the failure context where up to t processes may exhibit a Byzantine behavior. It first shows that $t < n/3$ is a necessary condition for such a construction. Then, it presents an algorithm building an array $REG[1..n]$ of SWMR atomic registers (only p_i can write $REG[i]$) in the system model $BAMP_{n,t}[t < n/3]$. This algorithm is consequently t-resilient optimal.

Keywords Asynchronous system, Atomicity, Byzantine process, Byzantine reliable broadcast, Impossibility, Linearization point, Upper bound, Read/write register.

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

9.1.1 Why SWMR (and Not MWMR) Atomic 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 this end, p_i invokes the operation $REG[i].\mathsf{write}(v)$ where v is the value it wants to write into $REG[i]$. However, any process p_j can read $REG[i]$ by invoking the operation $REG[i].\mathsf{read}()$.

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 flood the whole memory with fake values, so that no non-trivial computation could be possible.

9.1.2 Reminder on Possible Behaviors of a Byzantine Process

Reminder on Byzantine behavior A Byzantine process is a process that behaves arbitrarily. As seen in Section 4.1, this means that, when looking at the implementation level of the array $REG[1..n]$, 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 a message m to all the processes, can send a message m_1 to some processes, a different message m_2 to another subset of processes, and no message at all to the other processes. Moreover, while they cannot modify the

M. Raynal, *Fault-Tolerant Message-Passing Distributed Systems*,
https://doi.org/10.1007/978-3-319-94141-7_9

content of the messages sent by non-Byzantine processes, they can read their content and reorder their deliveries. More generally, Byzantine processes can collude to "pollute" the computation.

Notation As already indicated, the asynchronous message-passing system made up of n processes, among which up to t may be Byzantine, is denoted $BAMP_{n,t}[\emptyset]$.

On the modifications of $REG[k]$ by a Byzantine process p_k Let p_k be a Byzantine process. Like a correct process, p_k may invoke the write operation $REG[k].\mathsf{write}(v)$ to assign a value v to $REG[k]$ (where v can be a correct or a fake value).

Such a process p_k can also try to modify $REG[k]$ without using this operation, e.g., by sending "protocol messages" which, from the point of view of correct processes, simulate an invocation of $REG[k].\mathsf{write}(v)$. Such an attempt to modify $REG[k]$, without invoking the operation $REG[k].\mathsf{write}()$, may or not succeed. "Succeed" means that, from the point of view of all the correct processes, v was assigned to $REG[k]$, namely, this modification of $REG[k]$ appears as if it had been produced by an invocation of $REG[k].\mathsf{write}()$ by p_k.

The problem in the implementation of $REG[k]$ is then to ensure that $REG[k]$ does not appear as having been modified to some correct processes, and not modified to other correct processes. Moreover, the implementation of $REG[k]$ must also ensure that none of the modifications by the Byzantine process p_k are seen by some correct processes as if a was written, and seen by other correct processes as if $b \neq a$ was written. Hence, $REG[k]$ must appear as having been modified to the same value to all correct processes or none of them.

9.1.3 SWMR Atomic Registers Despite Byzantine Processes: Definition

Notations Let p_i and p_j be two correct processes.

- Let $read[i, j, x]$ denote the execution of the operation $REG[j].\mathsf{read}()$ issued by p_i which returns the x^{th} value written by p_j.
- Let $write[i, y]$ denote the y^{th} execution of the operation $REG[i].\mathsf{write}()$ by p_i.
- H being a sequence of values, let $H[x]$ denote the value at position x in H.

As seen in Section 5.2, it would be possible to associate a start event and an end event with each $read[i, j, x]$ and each $write[i, y]$ issued by a correct process p_i, so that all the events produced by the correct processes define a total order from which the notion of "terminates before" (used below) can be formally defined. To not overload the presentation, we do not use this formalization here.

Atomic SWMR registers in the presence of Byzantine processes The atomicity of a set of n SWMR registers $REG[1]$, ..., $REG[n]$ (some of them possibly associated with Byzantine processes) is defined by the following set of properties:

- R-termination (liveness). Let p_i be a correct process.
  - Each invocation of $REG[i].\mathsf{write}()$ terminates.
  - For any j, any invocation of $REG[j].\mathsf{read}()$ by p_i terminates.
- R-consistency (safety). Let p_i and p_j be two correct processes, and p_k a faulty or correct process.
  - Single history per process. There is exactly one sequence of values H_k associated with each process p_k. More, if p_k is correct, $H_k[x]$ contains the value written by $write[k, x]$.
  - Read followed by write. ($read[j, i, x]$ terminates before $write[i, y]$ starts) $\Rightarrow$ $(x < y)$.
  - Write followed by read. ($write[j, x]$ terminates before $read[i, j, y]$ starts) $\Rightarrow$ $(x \leq y)$.
  - No new/old read inversion. ($read[i, k, x]$ terminates before $read[j, k, y]$ starts) $\Rightarrow$ $(x \leq y)$.

As the behavior of a Byzantine process escapes the control of a correct algorithm, both the termination property and the constraint on the values returned by read invocations can only be on correct processes.

The "single history per process" property states that the write operations on any register are totally ordered. Hence, if p_k is correct, H_k is the sequence of values it wrote in $REG[k]$.

The three other safety properties concern only the values read by correct processes. The "read followed by write" property states that there is no read from the future, while the "write followed by read"' property states that no read can obtain an overwritten value. Due to the possibiliry of concurrent access to the same register, these two properties actually defines a regular register. Hence the "no new/old read inversion" property, which allows us to obtain an atomic register from a regular register.

9.2 An Impossibility Result

This section shows that $t < n/3$ is a necessary condition to implement an SWMR atomic register $BAMP_{n,t}[\emptyset]$. This theorem is due to D. Imbs, S. Rajsbaum, M. Raynal, and J. Stainer (2017).

Theorem 37. *It is impossible to implement an atomic* SWMR *register in* $BAMP_{n,t}[t \geq n/3]$.

Proof The proof is by contradiction. It is based on classic partitioning and indistinguishability arguments. Let us assume that there is an algorithm A that builds an atomic read/write register in $BAMP_{n,t}[t \geq n/3]$, which means that it satisfies the R-consistency and R-termination properties stated in the previous section. Let us notice that to guarantee the R-termination property, a correct process cannot wait for messages from more than $n - t = 2t$ processes.

Let us partition the processes into three sets Q_1, Q_2 and Q_3, each of size $\lfloor \frac{n}{3} \rfloor$ or $\lceil \frac{n}{3} \rceil$. As $\lfloor \frac{n}{3} \rfloor \leq \lceil \frac{n}{3} \rceil \leq t$, it follows that, in any execution, all processes of Q_1 (or Q_2, or Q_3) can be Byzantine. In the following p_1 is a process of Q_1, while p_2 is a process of Q_2. Let us assume that all SWMR atomic registers are initialized to $\perp$.

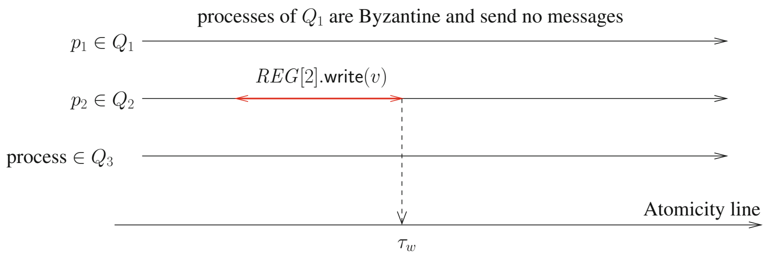

Figure 9.1: Execution $E1$ (impossibility of an SWMR register in $BAMP_{n,t}[t \geq n/3]$)

Let us consider a first execution E_1, depicted in Fig. 9.1 and defined as follows. (In this figure and the two following figures, a single process of each set is represented.)

- The set of Byzantine processes is Q_1. They do not send messages and appear as crashed to the processes of Q_2 and Q_3.
- 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 $REG[2]$.write(v) by p_2 terminates. Let τ_w be the time instant at which this write terminates.

Let E_2 (Fig. 9.2) 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).

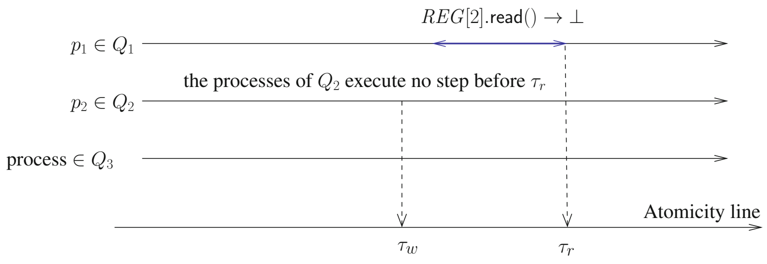
$REG[2].\mathsf{read}() \to \perp$
$p_1 \in Q_1$
the processes of Q_2 execute no step before τ_r
$p_2 \in Q_2$
process $\in Q_3$
Atomicity line
τ_w
τ_r

Figure 9.2: Execution $E2$ (impossibility of an SWMR register in $BAMP_{n,t}[t \geq n/3]$)

- 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 it follows that the invocation of $REG[2].\mathsf{read}()$ by p_1 terminates (let us notice that, as $|Q_2| \leq t$, and $n - 2t \leq t$, the processes of Q_2 appear as crashed to the invocation of $REG[2].\mathsf{read}()$, and they cannot prevent it from terminating). Let τ_r be the time instant at which this read terminates. According to the R-consistency property *read followed by write*, $REG[2]$ still has its initial value $\perp$. It follows that the read operation by p_1 returns this value.

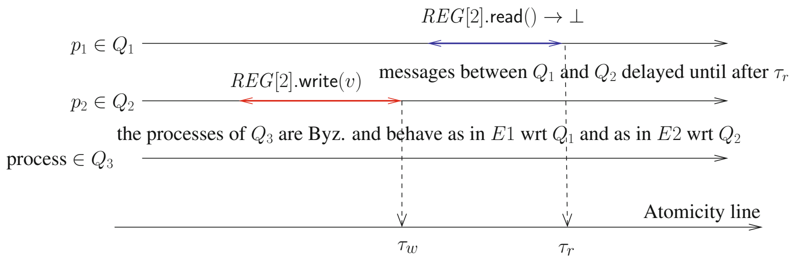
$REG[2].\mathsf{read}() \to \perp$
$p_1 \in Q_1$
$REG[2].\mathsf{write}(v)$
messages between Q_1 and Q_2 delayed until after τ_r
$p_2 \in Q_2$
the processes of Q_3 are Byz. and behave as in $E1$ wrt Q_1 and as in $E2$ wrt Q_2
process $\in Q_3$
Atomicity line
τ_w
τ_r

Figure 9.3: Execution $E3$ (impossibility of an SWMR register in $BAMP_{n,t}[t \geq n/3]$)

Let us finally consider E_3, a third execution depicted in Fig. 9.3 and defined as follows.

- The set of Byzantine processes is Q_3, and the processes of Q_3 behave exactly as in E_1 with respect to the processes of Q_2, and exactly as in E_2 with respect to 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.
- At the very same time instant as in E_1, process $p_2 \in Q_2$ writes 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.
- 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 $\perp$. But this violates the R-consistency property *write followed by read*, which contradicts the existence of Algorithm A.

$\square_{Theorem\ 37}$

9.3 Reminder on Byzantine Reliable Broadcast

This section is a reminder of Section 4.4 where a reliable broadcast algorithm suited to the system model $BAMP_{n,t}[t < n/3]$ was presented. This algorithm is extended here to include sequence numbers, which allows a process to send a sequence of messages instead of a single message. This extension constitutes a basic building block on which the algorithm implementing SWMR atomic registers in $BAMP|n, t[t < n/3]$ presented in Section 9.4 relies.

9.3.1 Specification of Multi-shot Reliable Broadcast

Including sequence numbers The multi-shot BRB-broadcast communication abstraction provides the processes with the operations $\mathsf{BRB\_broadcast}()$ and $\mathsf{BRB\_deliver}()$. $\mathsf{BRB\_broadcast}()$ has now two input parameters: a broadcast value v and an integer sn, which is a local sequence number used to identify the successive brb-broadcasts issued by the sender process. The sequence of numbers used by each (correct) process is the increasing sequence of consecutive integers. This BRB-broadcast communication abstraction is defined by the following properties:

- BRB-validity. If a non-faulty process BRB-delivers a pair (v, sn) from a correct process p_i, then p_i invoked $\mathsf{BRB\_broadcast}(v, sn)$.
- BRB-integrity. No correct process BRB-delivers a pair (v, sn) more than once.
- BRB-no-duplicity. If a non-faulty process brb-delivers a pair (v, sn) from a process p_i, no non-faulty process brb-delivers a pair $(v', sn,)$ such that $v \neq v'$ from p_i.
- BRB-termination-1. If a non-faulty process p_i invokes $\mathsf{BRB\_broadcast}(v, sn)$, all the non-faulty processes eventually brb-deliver the pair (v, sn).
- BRB-termination-2. If a non-faulty process brb-delivers a pair (v, sn) from p_i (possibly faulty) then all the non-faulty processes eventually brb-deliver a pair from p_i.

Let us notice that it follows from the BRB-no-duplicity property and the BRB-termination-2 properties that, if a correct process brb-delivers a pair (v, sn) from a process p_i (possibly faulty), then all the correct processes eventually brb-deliver the same pair (v, sn) from p_i (this property is called BRB-uniformity).

BRB-validity is on correct processes and relates their outputs to their inputs, namely no correct process brb-delivers spurious messages from correct processes. BRB-integrity states that there is no brb-broadcast duplication. BRB-uniformity is an "all or none" property (it is not possible for a pair to be delivered by a correct process and to never be delivered by the other correct processes). BRB-termination-1 is a liveness property: at least all the pairs brb-broadcast by correct processes are brb-delivered by them.

Adding FIFO delivery As a process p_i may execute several write operation on $REG[i]$, it is possible to associate a sequence number with each of them. So, we require that these messages be processed in their sequence number order.

9.3.2 An Algorithm for Multi-shot Byzantine Reliable Broadcast

The BRB-broadcast algorithm presented in Fig. 9.4 is the one of Section 4.4 enriched with sequence numbers. The lines with the same meaning in both algorithms have the same line numbers. Line (2) is split into two lines denoted (2)-1 and (2)-2. There are also two new lines related to the management of sequence numbers, denoted (N1) and (N2). Instead of INIT, the tag of an application message is denoted APPL, and each message carries the sequence number of the application message it is associated with.

```
operation BRB_broadcast APPL($v, sn$) is
(1)    broadcast APPL($v, sn$).

when a message APPL($v, sn$) is received from $p_j$ do
(2)-1 discard the message if it is not the first message from $p_j$ with sequence number $sn$;
(N1)  wait ($next_i[j] = sn$);
(2)-2 broadcast ECHO($j, v, sn$).

when a message ECHO($j, v, sn$) is received do
(3)    if  (ECHO($j, v, sn$) received from strictly more than $\frac{n+t}{2}$ different processes)
          $\wedge$(READY($j, v, sn$) never broadcast)
(4)       then broadcast READY($j, v, sn$)
(5)    end if.

when a message READY($j, v, sn$) is received do
(6)    if  (READY($j, v, sn$) received from at least $(t + 1)$ different processes)
          $\wedge$(READY($j, v, sn$) never sent)
(7)       then broadcast READY($j, v, sn$)
(8)    end if;
(9)    if  (READY($j, v, sn$) received from at least $(2t + 1)$ different processes)
          $\wedge$ ($\langle j, v, sn\rangle$ brb-delivered from $p_j$)
(10)      then BRB_deliver $\langle j, v, sn\rangle$;
(N2)            $next_i[j] \leftarrow next_i[j] + 1$
(11)   end if.
```

Figure 9.4: Reliable broadcast with sequence numbers in $BAMP_{n,t}[t < n/3]$ (code for p_i)

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, APPL$(-, sn)$) from p_j, which p_i will process (line N1). Initially, for all i, j, $next_i[j] = 1$. Then, $next_i[j]$ increases at line (N2).

Let us remember that broadcast TAG(m) is a simple macro-operation standing for "**for all** $j \in \{1, ...n\}$ **do** send TAG(m) to p_j **end for**".

When, on its "client" side, a process p_i invokes BRB_broadcast APPL(v, sn), it broadcasts the message APPL(v, sn), where sn is the value of its next sequence number (line 1).

On its "server" side, the behavior of a process p_i is as follows:

- When it receives a message APPL(v, sn) from a process p_j, p_i discards it if it has already received a message APPL$(-, sn')$ from p_j such that $sn' = sn$ (line (2)-1). This is because in this case p_j is Byzantine (a correct process issues a single BRB-broadcast per sequence number). Otherwise, p_i waits until it can process this message according to its sequence number (line N1). When this occurs, p_i broadcasts the message ECHO(j, v, sn) to inform the other processes it has received the application message APPL(v, sn) (line (2)-2).
- Then, when p_i has received the same message ECHO(j, v, sn) from "enough" processes (where "enough" means here "more than $(n + t)/2$ different processes"), and has not yet broadcast a message READY(j, v, sn), it does so (lines 3-5).

  The aim of (a) the messages ECHO(j, v, sn), and (b) the cardinality "greater than $(n + t)/2$ processes", is to ensure that no two correct processes brb-deliver distinct messages from p_j (even if p_j is Byzantine). The aim of the messages READY(j, v, sn) is related to the liveness of the algorithm. More precisely, their aim is to allow the brb-delivery, by the correct processes, of the very same triple $\langle j, v, sn\rangle$ from p_j, and this must always occur if p_j is correct. It is nevertheless possible that a message brb-broadcast by a Byzantine process p_j is never brb-delivered by the correct processes.

- Finally, when p_i has received the message READY(j, v, sn) from $(t + 1)$ different processes, it broadcasts the same message READY(j, v, sn), if not yet done. This is required to ensure the BRB-termination properties. If p_i has received "enough" messages READY(j, v, sn) ("enough" means here "from at least $(2t + 1)$ different processes"), it brb-delivers the triple $\langle j, v, sn\rangle$ generated by the message APPL(v, sn) brb-broadcast by p_j.

9.4 Construction of SWMR Atomic Registers in $BAMP_{n,t}[t < n/3]$

An algorithm constructing an array $REG[1..n]$ of SWMR atomic registers, where each p_i can write only $REG[i]$, in the presence of up to t Byzantine processes is described in Fig. 9.5. As it assumes $t < n/3$, this algorithm is t-resilience optimal.

This algorithm is due to A. Mostéfaoui, M. Petrolia, M. Raynal, and Cl. Jard (2017). Its design strives to be as close as possible to the ABD algorithms presented in Section 6.3.2 (SWMR atomic register) and Section 6.4.2 (MWMR atomic register). In addition to the necessary and sufficient condition $t < n/3$, this presentation allows the reader to better see, and understand, the additional statements needed to go from crash failures to Byzantine process failures.

The algorithm uses a **wait**$(condition)$ statement. The corresponding process is blocked until the predicate $condition$ is satisfied. While a process is blocked, it can process the messages it receives.

9.4.1 Description of the Algorithm

Local variables Each process p_i manages the following local variables whose scope is the full computation:

- $reg_i[1..n]$ is the local representation of the array $REG[1..n]$ of SWMR registers. Each local register $reg_i[j]$ contains two fields, a sequence number $reg_i[j].sn$, and the corresponding value $reg_i[j].val$. It is initialized to the pair $\langle \perp_j, 0\rangle$, where $\perp_j$ is the initial value of $REG[j]$.
- wsn_i is an integer, initialized to 0, used by p_i to associate sequence numbers with its successive write invocations.
- $rsn_i[1..n]$ is an array of sequence numbers (initialized to $[0, \cdots, 0]$) such that $sn_i[j]$ is used by p_i to identify its successive read invocations of $REG[j]$. (If we assume that no correct process p_i reads its own register $REG[i]$, $rsn_i[i]$ can be used to store wsn_i.)

The operation $REG[i]$.write(v) This operation is implemented by the client lines 1-4 and the server lines 12-14.

When a process p_i invokes $REG[i]$.write(v), it first increases wsn_i and brb-broadcasts the message WRITE(v, wsn_i). Let us notice that this is the only use of the reliable broadcast abstraction by the algorithm. The process p_i then waits for acknowledgments (message WRITE_DONE(v, wsn_i)) from $(n - t)$ distinct processes, and finally terminates the write operation.

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

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

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

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

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

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

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

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

Figure 9.5: Atomic SWMR Registers in $BAMP_{n,t}[t < n/3]$ (code for p_i)

Hence, if a correct process brb-delivers a message WRITE(v, wsn) from a Byzantine process p_j, be this message due to an invocation of BRB_broadcast WRITE() by p_j or a spurious message it sent, its faulty behavior is restricted to the broadcast of fake values for v and wsn.

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

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

When this waiting period terminates, p_i considers the current value $\langle w, wsn\rangle$ of $reg_i[j]$ (line 8). It then broadcasts the message CATCH_UP(j, wsn), and returns the value w as soon as its message

CATCH_UP() is acknowledged by $(n-t)$ processes (lines 9-10).

The aim of the CATCH_UP(j, wsn) message is to allow each destination process p_k to have a value in its local representation of $REG[j]$ (namely $reg_k[j].val$) at least as recent as the one whose sequence number is wsn (line 15). The aim of this *value resynchronization* is to prevent read inversions. When p_i has received the $(n-t)$ acknowledgments it was waiting for (line 10), it knows that no other correct process can obtain a value older than the value w it is about to return.

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

9.4.2 Comparison with the Crash Failure Model

As we have seen in Chapter 6 and Chapter 8, the algorithms implementing an atomic register on top of an asynchronous message-passing system prone to process crashes, require that “reads have to write”. More precisely, before returning a value, in one way or another, a reader must write this value to ensure atomicity (otherwise, we obtain only a “regular” register). In doing so, it is not possible that two sequential read invocations, concurrent with one or more write invocations, are such that the first read obtains one value while the second read obtains an older value (this prevents *read inversion*).

As Byzantine failures are more severe than crash failures, the algorithm of Figure 9.5 needs to use a mechanism analogous to the “reads have to write” to prevent read inversions from occurring. As previously indicated, this is done by the messages CATCH_UP() broadcast at line 9 and the associated acknowledgments messages CATCH_UP_DONE() received at line 10. These messages realize a synchronization during which $(n-t)$ processes (i.e., at least $(n-2t)$ correct processes) have resynchronized their value, if needed (line 15).

A comparison of two instances of the ABD algorithm and the algorithm of Fig. 9.5 is presented in Table 9.1. The first instance is the version of the ABD algorithm presented in Fig. 6.4, which builds an array of n SWMR (single-writer/multi-reader) atomic registers (one register per process). The second instance is the version of the ABD algorithm, presented in Fig. 6.5, which builds a single MWMR (multi-writer/multi-reader) atomic register.

As they depend on the application and not on the algorithm that implements registers, the size of the values which are written is considered to be constant. The parameter m denotes an upper bound on the number of read and write operations on each register. The value $\log n$ is due to the fact that a message carries a constant number of process identities. Similarly, $\log m$ is due to the fact that (a) a message carries a constant number of sequence numbers, and (b) there is a constant number of message tags (including the tags used by the underlying reliable broadcast).

Algorithm	Fig. 6.4: n SWMR	Fig. 6.5: 1 MWMR	Fig. 9.5: n SWMR
Failure type	crash	crash	Byzantine
Requirement	$t < n/2$	$t < n/2$	$t < n/3$
Msgs/write	$O(n)$	$O(n)$	$O(n^2)$
Msgs/read	$O(n)$	$O(n)$	$O(n)$
Msg size	$O(\log n + \log m)$	$O(\log n + \log m)$	$O(\log n + \log m)$
Local mem./proc.	$O(n \log m)$	$O(n \log m)$	$O(n \log m)$

Table 9.1: Crash vs Byzantine failures: cost comparisons

9.5 Proof of the Algorithm

9.5.1 Preliminary Lemmas

Lemma 30. *If a correct process p_i brb-delivers a message* $\text{WRITE}(w, sn)$ *(from a correct or faulty process), any correct process brb-delivers it.*

Proof This is an immediate consequence of the BRB-uniformity property of the BRB-broadcast abstraction. $\Box_{Lemma\ 30}$

Lemma 31. *Any two sets of $(n-t)$ processes have at least one correct process in their intersection.*

Proof Let Q_1 and Q_2 be two sets of processes such that $|Q_1| = |Q_2| = n - t$. In the worst case, the t processes that are not in Q_1 belong to Q_2, and the t processes that are not in Q_2 belong to Q_1. It follows that $|Q_1 \cap Q_2| \geq n - 2t$. As $n > 3t$, it follows that $|Q_1 \cap Q_2| \geq n - 2t \geq t + 1$, which concludes the proof of the lemma. $\Box_{Lemma\ 31}$

9.5.2 Proof of the Termination Properties

Lemma 32. *Let p_i be a correct process. Any invocation of $REG[i]$.write() terminates.*

Proof Let us consider the first invocation of $REG[i]$.write() by a correct process p_i. This write operation generates the brb-broadcast of the message $\text{WRITE}(-, 1)$ (lines 1-2). Due to Lemma 30, all correct processes brb-deliver this message, and the waiting predicate of line 13 is eventually satisfied. Consequently, each correct process p_k eventually sets $reg_k[i].sn$ to 1, and sends back to p_i an acknowledgment message $\text{WRITE\_DONE}(1)$. As there are least $(n-t)$ correct processes, p_i receives such acknowledgments from at least $(n-t)$ different processes, and terminates its first invocation (lines 3-4).

As, for any given process p_j, all correct processes will process the messages WRITE() from p_j in their sequence order, the lemma follows from a simple induction (whose previous paragraph is the proof of the base case). $\Box_{Lemma\ 32}$

Lemma 33. *Let p_i be a correct process. For any j, any invocation of $REG[j]$.read() terminates.*

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

To show that the wait of line 7 terminates we have to show that the freshness predicate $reg_i[j].sn \geq \max(wsn_1, \cdots, wsn_{n-t})$ is eventually satisfied. Let wsn be one of the previous sequence numbers, and p_k the correct process that send it. This means that $reg_k[j].sn = wsn$ (line 15), from which we conclude (as p_k is correct) that p_k has previously brb-delivered a message $\text{WRITE}(-, wsn)$ and updated accordingly $reg_k[j]$ at line 13 (let us remember that this is the only line at which the local register $reg_k[j]$ is updated). It follows from Lemma 30 that eventually p_i brb-delivers the message $\text{WRITE}(-, sn)$. It follows then from line 13 that eventually we have $reg_i[j].sn \geq sn$. As this is true for any sequence number in $\{wsn_1, ..., wsn_{n-t}\}$, it follows that the freshness predicate is eventually satisfied, and consequently the wait statement of line 7 is satisfied.

Let us now consider the wait statement of line 10, which appears after p_i has broadcast the message $\text{CATCH\_UP}(j, wsn)$, where $wsn = reg_i[j].sn$ (the sequence number in $reg_i[j]$ just after p_i

stopped waiting at line 7). We show that any correct process sends an acknowledgment message CATCH_UP_DONE(j, wsn) back to p_i at line 17. Process p_i updated $reg_i[j].sn$ to wsn at line 13, and this occurred when it brb-delivered a message WRITE$(-, wsn)$. The reasoning is the same as in the previous paragraph, namely, it follows from Lemma 30 that all correct processes brb-deliver this message and consequently we have $reg_k[j].sn \geq wsn$ at every correct process p_k. Hence, the value resynchronization predicate of line 16 is eventually satisfied at all correct processes, which consequently send back a message CATCH_UP_DONE(j, wsn) at line 17, which concludes the proof of the lemma. $\Box_{Lemma\ 33}$

9.5.3 Proof of the Consistency (Atomicity) Properties

Lemma 34. *It is possible to associate a single sequence of values H_i with each register $REG[i]$. Moreover, if p_i is correct, H_i is the sequence of values written by p_i in $REG[i]$.*

Proof To define H_i let us consider all the messages WRITE$(-, sn)$ brb-delivered from a (correct or faulty) process p_i by the correct processes (due to Lemma 30, these messages are brb-delivered to all correct processes). Let us order these messages according to their processing order as defined by the predicate of line 12. H_i is the corresponding sequence of values. (Let us notice that, if p_i is Byzantine, it is possible that some of its messages WRITE() are brb-delivered but never processed at lines 12-14; such messages if any are never added to H_i).

Let us now consider the case where p_i is correct. It follows from the BRB-validity property of the brb-broadcast abstraction that any message brb-delivered from p_i, was brb-broadcast by p_i. It then follows from lines 1-2 that H_i is the sequence of values written by p_i. $\Box_{Lemma\ 34}$

Lemma 35. *Let p_i and p_j be two correct processes. If $read[i, j, x]$ terminates before $write[j, y]$ starts, we have $x < y$.*

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

As $write[j, y]$ defines $H_j[y]$, it follows that a message WRITE$(-, y)$ is brb-delivered from p_j at each correct process p_k which executes $reg_k[j] \leftarrow \langle -, y\rangle$ at line 13. As this occurs after $read[i, j, x]$ has terminated, we necessarily have $x < y$. $\Box_{Lemma\ 35}$

Lemma 36. *Let p_i and p_j be two correct processes. If $write[i, x]$ terminates before $read[j, i, y]$ starts, we have $x \leq y$.*

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

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

It follows from Lemma 31 that at least one correct process p_k belongs to $Q_1 \cap Q_2$. Hence, when p_i returns from $REG[i]$.write() it received the message WRITE_DONE(x) from p_k, and we then have $reg_k[i].sn \geq x$. As $REG[i]$.read() by p_j started after $REG[i]$.write() by p_i terminated, when p_k sends the message STATE$(-, reg_k[i].sn)$ to p_j, we have $reg_k[i].sn \geq x$. It follows that, when p_j exits the wait statement at line 8 we have $reg_j[i].sn \geq x$, which concludes the proof of the lemma. $\square_{Lemma\ 36}$

Lemma 37. *Let p_i and p_j be two correct processes. If $read[i, k, x]$ terminates before $read[j, k, y]$ starts, we have $x \leq y$.*

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

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

It follows from Lemma 31 that at least one correct process p_ℓ belongs to $Q_1 \cap Q_2$. According to the fact that $read[i, k, x]$ terminates before $read[j, k, y]$ starts, it follows that p_ℓ sent CATCH_UP_DONE(k, x) to p_i before sending the message STATE$(-, s)$ to p_j. As $reg_\ell[k].sn$ never decreases, it follows that $x \leq s$. It finally follows that, when the freshness predicate is satisfied at p_j, we have $reg_j[k].sn \geq s$. As $y = reg_j[k].sn$ (lines 8-11), it follows that $x \leq y$, which concludes the proof. $\square_{Lemma\ 37}$

9.5.4 Piecing Together the Lemmas

Theorem 38. *The algorithm described in Fig. 9.5 implements an* array of n SWMR atomic registers *(one per process) in the system model $BAMP_{n,t}[t < n/3]$.*

Proof The proof follows from Lemmas 32-37. $\square_{Theorem\ 38}$

9.6 Building Objects on Top of SWMR Byzantine Registers

This section presents two objects illustrating the use of an SWMR shared memory build on top of $BAMP_{n,t}[t < n/3]$. Both these objects assume that, not only can each register $REG[i]$ be written by p_i, but p_i can write it only once. Hence, the underlying shared memory $REG[1..n]$ is made up of n write-once SWMR atomic registers. It is easy to modify (simplify) the algorithm presented in Fig. 9.5 to obtain write-once registers. This is left to the reader, and constitutes Exercise 1 of Section 9.9.

9.6.1 One-shot Write-snapshot Object

Definition A *one-shot write-snapshot* 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 object. A process p_i can invoke write_snapshot() at most once (whereas, there is no control on the number of times a Byzantine process invokes write_snapshot()). 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. 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 both p_i and p_j are correct and $\langle j, w \rangle \in output_i$, then p_j invoked write_snapshot(w).

The algorithm The internal representation of the write-snapshot object is an array ($REG[1..n]$) of write-once SWMR atomic registers. It is assumed that $REG[1..n]$ is initialized to $[\perp, \ldots, \perp]$, and all correct processes invoke write_snapshot(). Each process manages two auxiliary variables $aux1$ and $aux2$.

```
operation write_snapshot($v_i$) is
(1) $REG[i]$.write($v_i$);
(2) for $x \in \{1, ..., n\}$ do $aux1[x] \leftarrow REG[x]$.read() end for;
(3) for $x \in \{1, ..., n\}$ do $aux2[x] \leftarrow REG[x]$.read() end for;
(4) while ($aux1 \neq aux2$) do
(5)     $aux1[1..n] \leftarrow aux2[1..n]$;
(6)     for $x \in \{1, ..., n\}$ do $aux2[x] \leftarrow REG[x]$.read() end for
(7) end while;
(8) $output_i \leftarrow \{ \langle j, aux1[j] \rangle \mid aux1[j] \neq \perp \}$;
(9) return($output_i$).
```

Figure 9.6: One-shot write-snapshot in $BAMP_{n,t}[t < n/3]$ (code for p_i)

The algorithm implementing the operation write_snapshot() is very simple (Fig. 9.6). The invoking process p_i first deposits its value in $REG[i]$ (line 1), and issues an asynchronous "sequential double scan" (lines 2-3). If the sequential double scan is not successful (line 4), it executes other double scans (lines 2-3) 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).

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

9.6.2 Correct-only Agreement Object

Definition and assumptions A *correct-only agreement* object is a one-shot object that provides processes with a single operation denoted correct_only_agreement(). This operation is used by each process to propose a value and decide (return) a set of values. A decided set contains only values proposed by correct processes and the decided sets satisfy the containment property. It is assumed that $n > (w + 1)t$, where $w > 1$ is the maximal number of distinct values that can be proposed by the correct processes in an execution.

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 $output_i$ returned by a correct process p_i is not empty and does not contain values proposed only by Byzantine processes.

The algorithm The algorithm implementing the operation correct_only_agreement(), is described in Fig. 9.7. This algorithm is almost the same as the algorithm implementing the previous operation write_snapshot(). The modified lines are prefixed by "M", and concern the predicate used at line M4, and the computation of the output at line M8.

More precisely, a successful double scan is still 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.

```
operation correct_only_agreement($v_i$) is
(1)    $REG[i]$.write($v_i$);
(2)    for $x \in \{1,...,n\}$ do $aux1[x] \leftarrow REG[x]$ end for;
(3)    for $x \in \{1,...,n\}$ do $aux2[x] \leftarrow REG[x]$ end for;
(M4)   while $[(aux1 \neq aux2) \vee (\nexists v : |\{j : aux1[j] = v\}| > t)]$ do
(5)        $aux1 \leftarrow aux2$;
(6)        for $x \in \{1,...,n\}$ do $aux2[x] \leftarrow REG[x]$ end for
(7)    end while;
(M8)   $output_i \leftarrow \{ v : |\{j : aux1[j] = v\}| > t\}$;
(9)    return($output_i$).
```

Figure 9.7: Correct-only agreement in $BAMP_{n,t}[t < n/(w+1)]$

Proof of the algorithm As previously, the containment property is a consequence of the fact that the writes in the array $REG[1..n]$ are atomic, and the number of non-$\perp$ entries can only increase. The termination property is a consequence of the following observations: (a) there is a bounded number of processes, (b) the registers are write-once atomic 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 M4.

Remark Both the previous objects share the same termination and containment properties. They can be seen as dual in the following sense. One-shot 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 4).

9.7 Summary

This chapter addressed the implementation of single-writer/multi-reader registers in asynchronous message-passing systems where processes may commit Byzantine failures. It has first shown that $(t < n/3)$ is a necessary condition for such a construction. It has then presented an t-resilient algorithm which builds an array of n SWMR atomic registers (one per process) in such a context (system model $BAMP_{n,t}[t < n/3]$). This algorithm relies on an underlying reliable broadcast, an appropriate freshness predicate and a value resynchronization mechanism which ensure that a correct process always reads up-to-date values. A read operation costs $O(n)$ protocol messages, while a write operation costs $O(n^2)$ messages. It is important to notice that SWMR atomic registers can be implemented without using cryptography notions.

The fact that SWMR registers are considered is due to the following observation: as a Byzantine process can corrupt any register it can write, the design of multi-writer/multi-reader registers with non-trivial correctness guarantees is impossible in the presence of Byzantine processes. Whereas the values written in the SWMR register associated with a non-Byzantine process cannot be corrupted by a Byzantine process.

9.8 Bibliographic Notes

- Byzantine process failures were introduced in [263, 342] in the context of synchronous distributed systems.
- The impossibility proof stated in Theorem 37 is from [230]. The algorithm presented in Section 9.4 is due to A. Mostéfaoui, M. Petrolia, M. Raynal, and Cl. Jard [311].
- As far as we know, the first algorithm building SWMR atomic read/write registers in the system model $BAMP_{n,t}[t < n/3]$ is the one presented in [230]. In this algorithm, each register $REG[j]$ is locally represented at each process p_i by the sequence of all the values written by p_j in $REG[j]$. This article also presents implementations of high level objects on top of SWMR atomic registers that cope with Byzantine processes.
- Byzantine-tolerant broadcast was investigated in [81, 235, 325] (see also Chapter 4 and [88, 89]).
- The construction of Byzantine-tolerant objects was investigated in [241, 275].
- The topological structure of executions with Byzantine processes was investigated in [214, 286, 287].
- The ABD algorithms were introduced in [36] (see Chapter 6).
- The one-shot write-snapshot object and the correct-only agreement objects, and the associated algorithms, presented in Section 9.6 are due to D. Imbs, S. Rajsbaum, M. Raynal, and J. Stainer [230]. The one-shot write-snapshot object is a variant of an object called immediate snapshot object, defined by E. Borowsky and E. Gafni in [76].
- This chapter has considered the *peer-to-peer* model in which each process has both the role of a client (when it invokes an operation) and the role of a server (where it manages a local representation of the state of the implemented registers).

  In the *clients/servers* distributed model, some processes are clients while other are servers. Several articles have addressed the design of servers implementing a shared memory accessible by clients. The servers are usually managing a set of disks (e.g., [111, 1, 280]). Moreover, while they consider that some servers can be Byzantine, some articles restrict the failure type allowed to clients. As an example, [131, 203] explore efficiency issues (relation between resilience and fast reads) in the context where only servers can be Byzantine, while clients (the single writer and the readers) can fail by crashing.

  As other examples, [1] considers that clients can only commit crash failures, while [38] considers that clients can only be "semi-Byzantine" (i.e., they can issue a bounded number of faulty writes, but otherwise respect their code). The algorithm presented in [278] allows clients and some number of servers to be Byzantine, but requires clients to sign their messages. As far as we know, [25] was the first paper considering Byzantine readers while still offering maximal resilience (with respect to the number of Byzantine servers) without using cryptography. However, the writer can fail only by crashing, and the fact that a – possibly Byzantine – reader does not write a fake value in a register (to ensure the "reads have to write" rule required to implement atomicity) is ensured only with some probability.

9.9 Exercises and Problems

1. A *one-write* SWMR atomic register is a register that can written only once. Modify the algorithm described in Fig. 9.5 so that it implements an array $REG[1..n]$ of one-write SWMR atomic registers.
2. Is the one-shot write-snapshot object presented in Section 9.6 an atomic object?

   If it is atomic, you have to associate a linearization point with each operation invocation, such that no two invocations have the same linerarization point, and, for any two operations op1 and op2, if op1 terminates before op2 starts, the linearization point of op1 appears before the one of op2. If it is not atomic, you have to show that there are executions of the one-shot write-snapshot object for which it is impossible to build a linearization (atomicity line) as just described.

Solution in [230].

3. Same question with the correct-only agreement object.

   Solution in [230].