Christian Cachin • Rachid Guerraoui
Luís Rodrigues

Introduction to

Reliable and Secure Distributed Programming

Second Edition

Springer

4. Shared Memory

I always tell the truth, even when I lie.
(Tony Montana – Scarface)

This chapter presents abstractions of shared memory. They represent distributed programming abstractions, which are shared among processes and encapsulate data storage functionality accessible by read and write operations. The memory abstractions are called *registers* because they resemble those provided by multiprocessor machines at the hardware level, though in many cases, including in this chapter, they are implemented over processes that communicate by exchanging messages over a network and do not share any physical storage device. A register abstraction also resembles a disk device accessed over a storage-area network, a file in a distributed file system, or a shared working space in a collaborative editing environment. Therefore, understanding how to implement register abstractions helps us understand how to implement such distributed storage systems.

We study here different variants of register abstractions. These differ in the number of processes that are allowed to read from and write to them, as well as in the semantics of their read operations in the face of concurrency and failures. We distinguish three kinds of semantics: *safe*, *regular*, and *atomic*.

4.2 We first consider the $(1, N)$ *regular* register abstraction. The notation $(1, N)$
means that one specific process can write and all N processes in the system can
4.3 read. Then we consider the $(1, N)$ *atomic* register and the (N, N) atomic register 4.4
abstractions. We specify and implement regular and atomic register abstractions in
four of the distributed system models identified in Chap. 2: the fail-stop, fail-silent,
fail-recovery, and fail-arbitrary models.

The $(1, N)$ *safe* register abstraction is the simplest one among the three; we skip it first and treat it only in the fail-arbitrary model toward the end of the chapter.

C. Cachin et al., *Introduction to Reliable and Secure Distributed Programming*,
DOI: 10.1007/978-3-642-15260-3_4,
© Springer-Verlag Berlin Heidelberg 2011

4.1 Introduction

4.1.1 Shared Storage in a Distributed System

In a multiprocessor machine, processes typically communicate through shared memory provided at the hardware level. The shared memory can be viewed as an array of shared registers. It is a convenient abstraction to use for programmers. One may also build a register abstraction from a set of processes that communicate by sending messages to each other over a network; this results in an *emulation of shared-memory*. The programmer using this abstraction can develop algorithms using shared memory, without being aware that, behind the scenes, the processes actually communicate by exchanging messages and that there is no physical shared memory. Such an emulation is very appealing because programming with a shared memory is usually considered significantly easier than working with message exchanges, precisely because the programmer can ignore the consistency problems introduced by the distribution of data. Of course, the programmer has to respect the complexity of the emulation.

As we pointed out, studying register specifications and algorithms is also useful when implementing networked storage systems, distributed file systems, and shared working spaces for collaborative work. For example, the abstraction of a distributed storage device that can be accessed through read and write operations is similar to the notion of a register. Not surprisingly, the algorithms that one needs to devise to build a distributed storage system are directly inspired by those used to implement register abstractions.

In this section, we introduce *safe*, *regular*, and *atomic* semantics for registers. To describe them, we consider the behavior of a register when it is accessed concurrently by multiple processes.

4.1.2 Register Overview

Assumptions. Registers store values and can be accessed through two operations, *read* and *write*. A process starts a *read* operation by triggering a ⟨ *Read* ⟩ event and starts a *write* operation by triggering a ⟨ *Write* | v ⟩ event with a value v. We say that a process *invokes* an operation on a register when it triggers the event. The processes in the system use registers for communicating with each other and for storing information.

After a process has invoked an operation like this, the register abstraction may trigger an event that carries the reply from the operation. We say that the process *completes* the operation when this event occurs. Each correct process accesses the registers in a *sequential* manner, which means that after a process has invoked an operation on a register, the process does not invoke any further operation on that register until the previous operation completes. (There were no such restrictions for the broadcast abstractions in Chap. 3.)

A register may store values from an arbitrary domain and is initialized to a special value $\perp$. In other words, we assume that some write operation was initially invoked on the register with parameter $\bot$ and completed before any other operation was invoked. (The value $\bot$ cannot be written otherwise.) For presentation simplicity, but without loss of generality, we also assume that the values written to a particular register are unique. This can be implemented by adding unique timestamps provided by the processes to the written values and is similar to the assumption from the previous chapters that messages sent or broadcast are unique.

Some of the presented register abstractions and algorithms restrict the set of processes that may write to and read from a register. The simplest case is a register with one writer and one reader, which is called a $(1, 1)$ register; the writer is a specific process known in advance, and so is the reader. We will also consider registers with one specific writer and N readers, which means that any process can read from the register. It is called a $(1, N)$ register. Finally, a register to which every process may write to and read from is called an (N, N) register. Sometimes a $(1, 1)$ register is also called a *single-writer*, *single-reader* register, a $(1, N)$ register is called a *single-writer*, *multi-reader* register, and an (N, N) register is called a *multi-writer*, *multi-reader* register.

Signature and Semantics. A process interacts with a register abstraction through events. Basically, the register abstraction stores a value, a read operation returns the stored value, and a write operation updates the stored value. More precisely:

1. A process invokes a *read operation* on a register *r* by triggering a request event $\langle$ *r*, *Read* $\rangle$ with no input parameters. The register signals that it has terminated a read operation by triggering an indication event $\langle$ *r*, *ReadReturn* $\mid v \rangle$, containing a *return value* v as an output parameter. The return value presumably contains the current value of the register.
2. A process invokes a *write operation* on a register *r* by triggering a request event $\langle$ *r*, *Write* $\mid v \rangle$ with one input parameter v, called the *written value*. The register signals that it has terminated a write operation by triggering an indication event $\langle$ *r*, *WriteReturn* $\rangle$ with no parameters. The write operation serves to update the value in the register.

If a register is accessed by read and write operations of a single process, and we assume there is no failure, we define the specification of a register through the following simple properties:

- *Liveness:* Every operation eventually completes.
- *Safety:* Every read operation returns the value written by the *last* write operation.

In fact, even if a register is accessed by a set of processes one at a time, in a *serial* manner, and if no process crashes, we could still specify a register using those simple properties. By a serial execution we mean that a process does not invoke an operation on a register if some other process has invoked an operation and has not received any reply for the operation. (Note that this notion is stronger than the notion of sequential access introduced earlier.)

Failures. If we assume that processes might fail, say, by crashing, we can no longer require that any process who invokes an operation eventually completes that operation. Indeed, a process might crash right after invoking an operation and may not have the time to complete this operation. We say that the operation has failed. Because crashes are unpredictable, precisely this situation makes distributed computing challenging. We assume that a process who invokes an operation on a register can only fail by crashing (i.e., we exclude other faults for processes that invoke read/write operation, such as arbitrary faults). This restriction is important for implementing registers in the fail-arbitrary model.

Nevertheless, it makes sense to require that if a process p invokes some operation and does not subsequently crash then p eventually gets back a reply to its invocation, i.e., completes the operation. In other words, any process that invokes a read or write operation and does not crash should eventually return from that invocation. In this sense, its operation should not fail. This requirement makes the register *fault-tolerant*. Algorithms with this property are sometimes also called *robust* or *wait-free*.

If we assume that processes access a register in a serial manner, we may at first glance still want to require from a read operation that it returns the value written by the last write operation. However, we need to care about failures when defining the very notion of *last*. To illustrate the underlying issue, consider the following example execution:

> A process p invokes a write operation on a register with a value v and completes this write. Later on, some other process q invokes a write operation on the register with a new value w, and then q crashes before the operation completes. Hence, q does not get any indication that the operation has indeed taken place before it crashes, and the operation has failed. Now, if a process r subsequently invokes a read operation on the register, what is the value that r is supposed to return? Should it be v or w?

In fact, both values may be valid replies depending on what happened. Intuitively, process q may or may not have the time to complete the write operation. In other words, when we require that a read operation returns the last value written, we consider the following two cases as possible:

1. The value returned has indeed been written by the last process that completed its write, even if some other process invoked a write later but crashed. In this case, no future read should return the value written by the failed write; everything happens as if the failed operation was never invoked.
2. The value returned was the input parameter of the last write operation that was invoked, even if the writer process crashed before it completed the operation. Everything happens as if the operation that failed actually completed.

The underlying difficulty is that the failed write operation (by the crashed process q in the example) did not complete and is, therefore, “concurrent” to the last read operation (by process r) that happened after the crash. The same problem occurs even if process q does not fail and is simply delayed. This is a particular problem resulting from the concurrency of two operations, which we discuss now.

Concurrency. When multiple processes access a register in practice, executions are most often not serial (and clearly not sequential). What should we expect a read operation to return when it is concurrent with some write operation? What is the meaning of the "last" write in this context? Similarly, if two write operations were invoked concurrently, what is the "last" value written? Can a subsequent read return one of the values, and then a read that comes even later return the other value?

In this chapter, we specify three register abstractions, called *safe*, *regular*, and *atomic*, which differ mainly in the way we answer these questions. Roughly speaking, a safe register may return an arbitrary value when a write is concurrently ongoing. A regular register, in contrast, ensures a minimal guarantee in the face of concurrent or failed operations, and may only return the previous value or the newly written value. An atomic register is even stronger and provides a strict form of consistency even in the face of concurrency and failures. We also present algorithms that implement these specifications; we will see that algorithms implementing atomic registers are more complex than those implementing regular or safe registers.

To make the specifications more precise, we first introduce some definitions that aim to capture this intuition. For the moment, we assume fail-stop process abstractions, which may only fail by crashing and do not recover after a crash; later in the chapter, we consider algorithms in the fail-recovery model and in the fail-arbitrary model.

4.1.3 Completeness and Precedence

We first define more precise notions for the *completeness* of the execution of an operation and for the *precedence* between different operation executions. When there is no possible ambiguity, we simply take *operations* to mean *operation executions*.

These notions are defined in terms of the events that occur in the interface of a register abstraction, that is, using ⟨ *Read* ⟩, ⟨ *Write* ⟩, ⟨ *ReadReturn* ⟩, and ⟨ *WriteReturn* ⟩ events; the first two represent the *invocation* of an operation, and the latter two indicate the *completion* of an operation. Remember that these events occur at a single indivisible point in time, using a fictional notion of global time that only serves to reason about specifications and algorithms. This global time is not directly accessible to the processes.

We say that an operation is *complete* if its invocation and completion events have *both* occurred. In particular, this means that the process which invokes an operation o does not crash before operation o terminates and the completion event occurs at the invoking algorithm of the process. An operation is said to *fail* when the process that invoked it crashes *before* the corresponding completion event occurs. (We only consider implementations with crash-stop process abstractions here; the corresponding concepts in the fail-recovery and fail-arbitrary models are introduced later.)

The temporal relation between operations is given by the following notions:

- An operation o is said to *precede* an operation o' if the completion event of o occurs before the invocation event of o'. As an immediate consequence of this

definition, note that if an operation o invoked by a process p precedes some other operation (possibly invoked by a different process) then o must be complete and its completion event occurred at p.
- If two operations are such that one precedes the other then we say that the operations are *sequential*. If neither one of two operations precedes the other then we say that they are *concurrent*.

Basically, the execution of operations on a register defines a partial order on its read and write operations. If only one process invokes operations then the order is total, according to our assumption that every process operates sequentially on one register. When no two operations are concurrent and all operations are complete, as in a serial execution, the order is also total.

- When a read operation o_r returns a value v, and v was the input parameter of some write operation o_w, we say that operation o_r *reads from* o_w or that value v is *read from* o_w.
- When a write operation (o_w) with input parameter v completes, we say that value v is *written (by* o_w*)*.

Recall that every value is written only once and, hence, the write operations in the definition are unique.

In the following, we give specifications of various forms of register abstractions and algorithms to implement them. Some algorithms use multiple instances of simpler register abstractions.

4.2 $(1, N)$ Regular Register

We start the description of shared-memory abstractions with the specification and two algorithms for a $(1, N)$ *regular* register. This means that one specific process p can invoke a write operation on the register, and any process can invoke a read operation on the register. The notion of regularity, which we explain later, is not considered for multiple writers. (There is no consensus in the distributed computing literature on how to generalize the notion of regularity to multiple writers.)

4.2.1 Specification

The interface and properties of a $(1, N)$ *regular register* abstraction (ONRR) are given in Module 4.1. In short, every read operation that is not concurrent with any write operation returns the last value written. If there is a concurrent write, the read is allowed to return the last value written or the value concurrently being written. Note that if a process invokes a write and crashes (without recovering), the write is considered to be concurrent with any read that did not precede it. Hence, such a read can return the value that was supposed to be written by the failed write or the last value written before the failed write was invoked. In any case, the returned value must be read from some write operation invoked on the register. That is, the value returned by any read operation must be a value that some process has tried to write

Module 4.1: Interface and properties of a $(1, N)$ regular register

Module:

Name: $(1, N)$-RegularRegister, **instance** *onrr*.

Events:

Request: ⟨ *onrr*, *Read* ⟩: Invokes a read operation on the register.

Request: ⟨ *onrr*, *Write* | v ⟩: Invokes a write operation with value v on the register.

Indication: ⟨ *onrr*, *ReadReturn* | v ⟩: Completes a read operation on the register with return value v.

Indication: ⟨ *onrr*, *WriteReturn* ⟩: Completes a write operation on the register.

Properties:

ONRR1: *Termination:* If a correct process invokes an operation, then the operation eventually completes.

ONRR2: *Validity:* A read that is not concurrent with a write returns the last value written; a read that is concurrent with a write returns the last value written or the value concurrently written.

(even if the write was not complete), and it cannot be invented out of thin air. The value may be the initial value $\bot$ of the register.

To illustrate the specification of a regular register, we depict two executions of operations on one register in Figs. 4.1 and 4.2; the operations are executed by two processes. The notation in the figure uses two dots and a thick line to denote the execution of an operation, where the dots represent the invocation and the completion event. The type of the operation and the parameters are described with text. The execution of Fig. 4.1 is not regular because the first read does not return the last written value. In contrast, the execution in Fig. 4.2 is regular.

As an outlook to the specification of a *safe* register (in Sect. 4.6), which is a weaker abstraction than a regular register, we note that one obtains the *validity* property a safe register by dropping the second part of the *validity* property in Module 4.1, namely, the condition on reads that are not concurrent with any write. When a read is concurrent with a write in a safe register, it may return an arbitrary value.

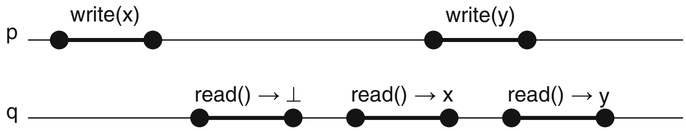

Figure 4.1: A register execution that is not regular because of the first read by process q

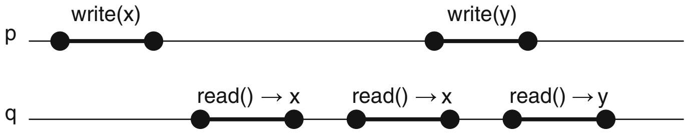

Figure 4.2: A regular register execution

4.2.2 Fail-Stop Algorithm: Read-One Write-All Regular Register

Algorithm 4.1 implements a $(1, N)$ regular register. The algorithm is particularly simple because it uses the fail-stop model and relies on a perfect failure detector. When a process crashes, the failure detector ensures that eventually all correct processes detect the crash (*strong completeness*), and no process is detected to have crashed until it has really crashed (*strong accuracy*).

Algorithm 4.1: Read-One Write-All

```
Implements:
    $(1, N)$-RegularRegister, instance onrr.

Uses:
    BestEffortBroadcast, instance beb;
    PerfectPointToPointLinks, instance pl;
    PerfectFailureDetector, instance $\mathcal{P}$.

upon event ⟨ onrr, Init ⟩ do
    $val := \perp$;
    $correct := \Pi$;
    $writeset := \emptyset$;

upon event ⟨ $\mathcal{P}$, Crash | $p$ ⟩ do
    $correct := correct \setminus \{p\}$;

upon event ⟨ onrr, Read ⟩ do
    trigger ⟨ onrr, ReadReturn | $val$ ⟩;

upon event ⟨ onrr, Write | $v$ ⟩ do
    trigger ⟨ beb, Broadcast | [WRITE, $v$] ⟩;

upon event ⟨ beb, Deliver | $q$, [WRITE, $v$] ⟩ do
    $val := v$;
    trigger ⟨ pl, Send | $q$, ACK ⟩;

upon event ⟨ pl, Deliver | $p$, ACK ⟩ do
    $writeset := writeset \cup \{p\}$;

upon $correct \subseteq writeset$ do
    $writeset := \emptyset$;
    trigger ⟨ onrr, WriteReturn ⟩;
```

The algorithm has each process store a copy of the *current register value* in a local variable val. In other words, the value of the register is replicated at all processes. The writer updates the value of all presumably correct processes (i.e., those that it does not detect to have crashed) by broadcasting a WRITE message with the new value. All processes acknowledge the receipt of the new value with an ACK message. The write operation returns when the writer has received an acknowledgment from every process that it considers to be correct. When the write of a new value is complete, all processes that did not crash have stored the new value. The reader simply returns the value that it has stored locally. In other words, the reader *reads one value* and the writer *writes all values*. Hence, Algorithm 4.1 is called "Read-One Write-All."

The algorithm uses a perfect failure-detector abstraction and two underlying communication abstractions: perfect point-to-point links and best-effort broadcast.

We will be using multiple instances of regular registers to build stronger abstractions later in this chapter. As mentioned before, the instances are differentiated by their identifiers, and all messages exchanged using the underlying communication primitives implicitly carry an instance identifier to match the same instance at all processes.

Correctness. The *termination* property is straightforward for any read invocation, because a process simply returns its local value. For a write invocation, *termination* follows from the properties of the underlying communication abstractions (*reliable delivery* of perfect point-to-point links and *validity* of best-effort broadcast) and the *completeness* property of the perfect failure detector (every crashed process is eventually detected by every correct process). Any process that crashes is detected and any process that does not crash sends back an acknowledgment, which is eventually delivered by the writer.

Consider *validity*. Assume that there is no concurrency and all operations are complete. Consider a read invoked by some process p and assume, furthermore, that v is the last value written. Because of the *accuracy* property of the perfect failure detector, at the time when the read is invoked, all processes that did not crash store value v. In particular, also p stores v and returns v, and this is the last value written.

Assume now that the read is concurrent with some write of a value v and the value written prior to v was v' (it may be that v' is the initial value $\perp$). According to the above argument, every process stores v' before the write operation of v was invoked. Because of the properties of the communication abstractions (*no creation* properties), no message is altered and no value is stored by a process unless the writer has invoked a write operation with this value as a parameter. At the time of the read, every process therefore stores either still v' or has *beb*-delivered the WRITE message with v and stores v. The return value of the read is either v or v', as required from a regular register.

Performance. Every write operation requires two communication steps corresponding to the WRITE and ACK exchange between the writer and all processes and $O(N)$ messages. A read operation does not require any communication, it is purely local.

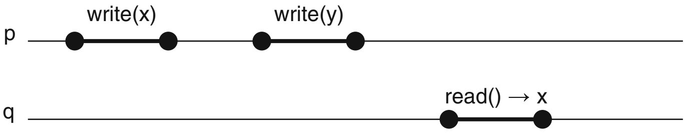

Figure 4.3: A non-regular register execution

4.2.3 Fail-Silent Algorithm: Majority Voting Regular Register

It is easy to see that if the failure detector is not perfect, the “Read-One Write-All” algorithm (Algorithm 4.1) may violate the *validity* property of the register. The execution illustrated in Fig. 4.3 shows how this could happen, even without concurrency and without any failure. When the writer process p falsely suspects process q to have crashed, the write operation may return before receiving the acknowledgment from q and, thus, before q has locally stored the new value y. Hence, the read by q may return x and not the last written value.

In the following, we present Algorithm 4.2 that implements a regular register in the fail-silent model. This algorithm does not rely on any failure detection abstraction. Instead, the algorithm assumes that a majority of the processes is correct. We leave it as an exercise (at the end of the chapter) to show that this majority assumption is actually needed, even when an eventually perfect failure detector can be used.

The general principle of the algorithm requires for the writer and the readers to use a set of *witness* processes that keep track of the most recent value of the register. The witnesses must be chosen in such a way that at least one witness participates in any pair of such operations, and does not crash in the meantime. Every two such sets of witnesses must, therefore, overlap. In other words, they form *quorums*, a collection of sets such that the intersection of every two sets is not empty (Sect. 2.7.3). Majorities are one of the simplest kinds of quorums, which is the reason for calling Algorithm 4.2 “Majority Voting.” The algorithm implements a $(1, N)$ regular register, where one specific process is the writer and any process can be a reader.

Similar to the “Read-One Write-All” algorithm presented before, the “Majority Voting” algorithm also has each process store a copy of the *current register value* in a local variable val. In addition, the “Majority Voting” algorithm also stores a *timestamp* ts together with the stored value at every process. This timestamp is defined by the writer and represents the number of times the write operation has been invoked.

The algorithm uses a best-effort broadcast instance *beb* and a perfect links instance *pl*. When the unique writer p invokes a write operation with a new value, the process increments its *write-timestamp* and associates it with the value to be written. Then p *beb*-broadcasts a WRITE message to all processes, and has a majority “adopt” this value and the associated timestamp. To *adopt* a value in this context means to store it locally as the current register value. The writer completes the write (and hence returns from the operation) when it has received an acknowledgment from a majority of the processes, indicating that they have indeed adopted the new

Algorithm 4.2: Majority Voting Regular Register

```
Implements:
    (1, $N$)-RegularRegister, instance onrr.
Uses:
    BestEffortBroadcast, instance beb;
    PerfectPointToPointLinks, instance pl.

upon event ⟨ onrr, Init ⟩ do
    $(ts, val) := (0, \bot)$;
    $wts := 0$;
    $acks := 0$;
    $rid := 0$;
    $readlist := [\bot]^N$;

upon event ⟨ onrr, Write | $v$ ⟩ do
    $wts := wts + 1$;
    $acks := 0$;
    trigger ⟨ beb, Broadcast | [WRITE, $wts, v$] ⟩;

upon event ⟨ beb, Deliver | $p$, [WRITE, $ts', v'$] ⟩ do
    if $ts' > ts$ then
        $(ts, val) := (ts', v')$;
    trigger ⟨ pl, Send | $p$, [ACK, $ts'$] ⟩;

upon event ⟨ pl, Deliver | $q$, [ACK, $ts'$] ⟩ such that $ts' = wts$ do
    $acks := acks + 1$;
    if $acks > N/2$ then
        $acks := 0$;
        trigger ⟨ onrr, WriteReturn ⟩;

upon event ⟨ onrr, Read ⟩ do
    $rid := rid + 1$;
    $readlist := [\bot]^N$;
    trigger ⟨ beb, Broadcast | [READ, $rid$] ⟩;

upon event ⟨ beb, Deliver | $p$, [READ, $r$] ⟩ do
    trigger ⟨ pl, Send | $p$, [VALUE, $r, ts, val$] ⟩;

upon event ⟨ pl, Deliver | $q$, [VALUE, $r, ts', v'$] ⟩ such that $r = rid$ do
    $readlist[q] := (ts', v')$;
    if $\#(readlist) > N/2$ then
        $v := highestval(readlist)$;
        $readlist := [\bot]^N$;
        trigger ⟨ onrr, ReadReturn | $v$ ⟩;
```

value and the associated timestamp. It is important to note that a process q will adopt a value v' sent by the writer only if q has not already adopted a value v with a larger timestamp. This might happen if the WRITE message containing v was *beb*-delivered by q *before* the WRITE message containing v'. In this case, process q was also not in the majority that made it possible for p to complete the write of v' before proceeding to writing v.

To read a value, a reader process *beb*-broadcasts a READ message to all processes, every process replies with the stored value and its timestamp, and the reader selects the value with the largest timestamp from a majority of the replies. The processes in this majority act as witnesses of what was written before. This majority does not have to be the same as the one used by the writer. Choosing the largest timestamp ensures that the value written last is returned, provided there is no concurrency. To simplify the presentation of Algorithm 4.2, the reader uses a function *highestval*(S) that takes a list S of timestamp/value pairs as input and returns the value of the pair with the largest timestamp, that is, the value v of a pair $(ts, v) \in S$ such that

$$\text{forall } (ts', v') \in S \, : \, ts' < ts \lor (ts', v') = (ts, v).$$

The function is applied to the received pairs as soon as timestamp/value pairs have been received from a majority of the processes.

Note that every WRITE and READ message is tagged with a unique identifier, and the corresponding reply carries this tag. For a write operation, the tag is simply the write-timestamp wts associated with the value written. In the case of a read operation, the tag is a read-request identifier *rid*, solely used for identifying the messages belonging to different reads. In this way, the reader can figure out whether a given reply message matches a request (and is not a reply in response to an earlier READ message). This mechanism is important to prevent the reader from confusing two replies from different operations and counting them toward the wrong operation. Likewise, the *ack* counter and the list of values in *readlist* must be initialized freshly whenever a new write or read operation starts, respectively. Without these mechanisms, the algorithm may violate the *validity* property of the register.

Correctness. The *termination* property follows from the properties of the underlying communication abstractions and from the assumption that a majority of processes in the system are correct.

For the *validity* property, consider a read operation that is not concurrent with any write. Assume, furthermore, that the read is invoked by process q and the last value written by the writer p is v with associated timestamp wts. This means that, at the time when the read is invoked, a majority of the processes store wts in their timestamp variable ts, and that there is no larger timestamp in the system. This is because the writer uses increasing timestamps. Before returning from the read operation, process q consults a majority of the processes and, hence, receives at least one reply containing timestamp wts. This follows from the use of majority quorums that always intersect. Process q hence returns value v, which is indeed the last value written, because wts is the largest timestamp.

Consider now the case where the read is concurrent with some write of value v with associated timestamp wts, and the previous write was for value v' and timestamp $wts-1$. If any process returns the pair (wts, v) to the reader q then q returns v, which is a valid reply. Otherwise, all replies from more than $N/2$ processes contain v' and associated timestamp $wts - 1$, and q returns v', which is also a valid reply.

Performance. Every write operation requires one communication roundtrip between the writer and a majority of the processes, and every read requires one communication roundtrip between the reader and a majority of the processes. In both operations, $O(N)$ messages are exchanged.

4.3 $(1, N)$ Atomic Register

We give here the specification and two algorithms for a $(1, N)$ *atomic register*. The generalization to multiple writers will be discussed in the next section.

4.3.1 Specification

Consider a $(1, N)$ regular register with initial value $\perp$, and suppose the writer p invokes an operation to write a value v. Because of the regular register specification, nothing prevents a process that reads the register multiple times from returning first v, subsequently $\perp$, then again v, and so on, as long as the reads and the write of p are concurrent. Furthermore, if the writer crashes before completing the write, the operation is not complete, and one subsequent reader might read v, whereas another reader, coming even later, might still return $\perp$. An atomic register is a regular register that prevents such behavior.

The interface and properties of a $(1, N)$ *atomic register* abstraction (ONAR) are given in Module 4.2. A $(1, N)$ atomic register is a regular register that, in addition to the properties of a regular register (Module 4.1) ensures a specific *ordering* property

Module 4.2: Interface and properties of a $(1, N)$ atomic register

Module:

Name: $(1, N)$-AtomicRegister, **instance** *onar*.

Events:

Request: $\langle$ *onar*, *Read* $\rangle$: Invokes a read operation on the register.

Request: $\langle$ *onar*, *Write* | v $\rangle$: Invokes a write operation with value v on the register.

Indication: $\langle$ *onar*, *ReadReturn* | v $\rangle$: Completes a read operation on the register with return value v.

Indication: $\langle$ *onar*, *WriteReturn* $\rangle$: Completes a write operation on the register.

Properties:

ONAR1–ONAR2: Same as properties ONRR1–ONRR2 of a $(1, N)$ regular register (Module 4.1).

ONAR3: *Ordering:* If a read returns a value v and a subsequent read returns a value w, then the write of w does not precede the write of v.

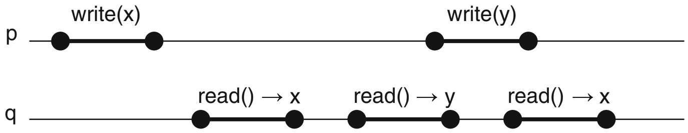

Figure 4.4: A register execution that is not atomic because of the third read by process q

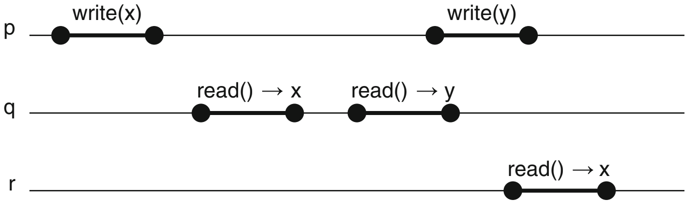

Figure 4.5: Violation of atomicity in the "Read-One Write-All" regular register algorithm

which, roughly speaking, prevents an "old" value from being read by process p, once a "newer" value has been read by process q (even if $p \neq q$). More precisely, this property implies that every operation of an atomic register can be thought to occur at a single indivisible point in time, which lies between the invocation and the completion of the operation.

A $(1, N)$ atomic register prevents that a reader process reads a value w after the completion of a read operation that returned a value v (possibly by another process), when w was written before v. In addition, if the single writer process started to write some value v and crashed before completing this operation, the atomic register ensures that once any reader completes a read operation and returns v, then no subsequent read operation returns a different value.

The execution depicted in Fig. 4.4 is not atomic because the *ordering* property of an atomic register should prevent the last read of process q from returning x after the previous read returned y, given that x was written before y. If the execution is changed so that the last read of q also returns y, the execution becomes atomic. Another atomic execution is the regular execution shown in Fig. 4.2.

It is important to note that none of our previous algorithms implements a $(1, N)$ atomic register, even if no failures occur. We illustrate this through the execution depicted in Fig. 4.5 as a counterexample for Algorithm 4.1 ("Read-One Write-All"), and the execution depicted in Fig. 4.6 as a counterexample for Algorithm 4.2 ("Majority Voting").

The scenario of Fig. 4.5 can occur with Algorithm 4.1 if during the second write operation of process p, the new value y is received and read by process q before it is received by process r. Before receiving the new value, r will continue to read the previous value x, even if its read operation occurs after the read by q.

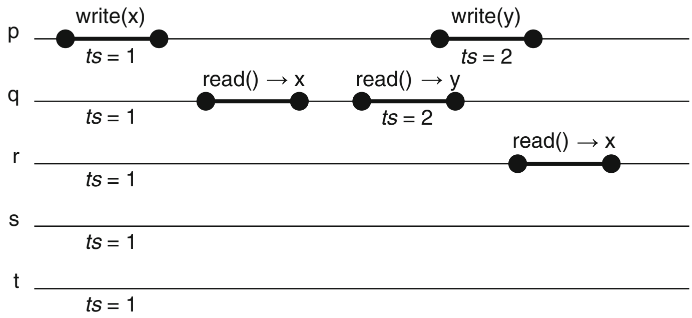

Figure 4.6: Violation of atomicity in the "Majority Voting" regular register algorithm

The scenario of Fig. 4.6 can occur with Algorithm 4.2 if process q has received replies from processes p, q, and s in its second read, and the replies already included timestamp 2 from the second write of p. On the other hand, process r has accessed processes r, s, and t, which have not yet received the WRITE message with timestamp 2 from p.

In the following, we give several algorithms that implement the $(1, N)$ atomic register abstraction. We first describe how to transform an abstract $(1, N)$ regular register into a $(1, N)$ atomic register algorithm; the regular register might be implemented by a fail-stop or fail-silent algorithm, which will determine the system model of the resulting implementation. Such a transformation is modular and helps to understand the fundamental difference between atomic and regular registers. It does not lead to very efficient algorithms, however. We subsequently describe how to directly extend our two regular register algorithms to obtain efficient $(1, N)$ atomic register algorithms.

4.3.2 Transformation: From $(1, N)$ Regular to $(1, N)$ Atomic Registers

This section describes how to transform any $(1, N)$ regular register abstraction into a $(1, N)$ atomic register abstraction. For pedagogical reasons, we divide the transformation in two parts. We first explain how to transform any $(1, N)$ regular register abstraction into a $(1, 1)$ atomic register abstraction and then how to transform any $(1, 1)$ atomic register abstraction into a $(1, N)$ atomic register abstraction. These transformations do not use any other means of communication between processes than the underlying registers.

From $(1, N)$ Regular to $(1, 1)$ Atomic Registers. The first transformation is given in Algorithm 4.3 and realizes the following simple idea. To build a $(1, 1)$ atomic register with process p as writer and process q as reader, we use one $(1, N)$ regular register, also with writer p and reader q. Furthermore, the writer maintains a timestamp that it increments and associates with every new value to be written. The reader also maintains a timestamp, together with the value associated to the highest timestamp that it has read from the regular register so far. Intuitively, the reader

Algorithm 4.3: From $(1, N)$ Regular to $(1, 1)$ Atomic Registers

```
Implements:
    $(1,1)$-AtomicRegister, instance $ooar$.

Uses:
    $(1,N)$-RegularRegister, instance $onrr$.

upon event ⟨ $ooar$, Init ⟩ do
    $(ts, val) := (0, \perp)$;
    $wts := 0$;

upon event ⟨ $ooar$, Write | $v$ ⟩ do
    $wts := wts + 1$;
    trigger ⟨ $onrr$, Write | $(wts, v)$ ⟩;

upon event ⟨ $onrr$, WriteReturn ⟩ do
    trigger ⟨ $ooar$, WriteReturn ⟩;

upon event ⟨ $ooar$, Read ⟩ do
    trigger ⟨ $onrr$, Read ⟩;

upon event ⟨ $onrr$, ReadReturn | $(ts', v')$ ⟩ do
    if $ts' > ts$ then
        $(ts, val) := (ts', v')$;
    trigger ⟨ $ooar$, ReadReturn | $val$ ⟩;
```

stores these items in order to always return the value with the highest timestamp and to avoid returning an old value once it has read a newer value from the regular register.

To implement a $(1, 1)$ atomic register instance *ooar*, Algorithm 4.3 maintains one instance *onrr* of a $(1, N)$ regular register. The writer maintains a writer-timestamp wts, and the reader maintains a timestamp ts, both initialized to 0. In addition, the reader stores the most recently read value in a variable val. The algorithm proceeds as follows:

- To *ooar*-write a value v to the atomic register, the writer p increments its timestamp wts and *onrr*-writes the pair (wts, v) into the underlying regular register.
- To *ooar*-read a value from the atomic register, the reader q first *onrr*-reads a timestamp/value pair from the underlying regular register. If the returned timestamp ts' is larger than the local timestamp ts then q stores ts' together with the returned value v in the local variables, and returns v. Otherwise, the reader simply returns the value from val, which it has already stored locally.

Correctness. The *termination* property of the atomic register follows from the same property of the underlying regular register.

Consider *validity* and assume first that a read is not concurrent with any write, and the last value written by p is v and associated with timestamp ts'. The reader-timestamp stored by the reader q is either ts', if q has already read v in some previous read, or a strictly smaller value. In both cases, because of the *validity* property of the regular register, a read by q will return v. Consider now a read that is concurrent with some write of value v and timestamp ts', and the previous write was for value v' and timestamp $ts' - 1$. The reader-timestamp stored by q cannot be larger than ts'. Hence, because of the *validity* property of the underlying regular register, q will return either v or v'; both are valid replies.

Consider now *ordering* and assume that p writes v and subsequently writes w. Suppose that q returns w for some read and consider any subsequent read of q. The reader-timestamp stored by q is either the one associated with w or a larger one. Hence, the last check in the algorithm when returning from a read prevents that the return value was written before w and there is no way for the algorithm to return v.

Performance. The transformation requires only local computation, such as maintaining timestamps and performing some checks, in addition to writing to and reading from the regular register.

From $(1,1)$ Atomic to $(1,N)$ Atomic Registers. We describe here an algorithm that implements the abstraction of a $(1, N)$ atomic register out of $(1, 1)$ atomic registers. To get an intuition of the transformation, think of a teacher (the writer), who needs to communicate some information to a set of students (the readers), through the abstraction of a traditional blackboard. The board is a good match for the abstraction of a $(1, N)$ register, as long as only the teacher writes on it. Furthermore, it is made of a single physical entity and atomic.

Assume now that the teacher cannot physically gather all students within the same classroom, and hence cannot use one physical board for all. Instead, this global board needs to be emulated with one or several individual boards (i-boards) that can also be written by one person but may only be read by one person. For example, every student can have one or several such electronic i-boards at home, which only he or she can read.

It makes sense to have the teacher write each new piece of information to at least one i-board per student. This is intuitively necessary for the students to eventually read the information provided by the teacher, i.e., to ensure the *validity* property of the register. However, this is not enough to guarantee the *ordering* property of an atomic register. Indeed, assume that the teacher writes two pieces of information consecutively, first x and then y. It might happen that a student reads y and later on, some other student still reads x, say, because the information flow from the teacher to the first student is faster than the flow to the second student. This *ordering* violation is similar to the situation of Fig. 4.5.

One way to cope with this issue is for every student, before terminating the reading of some information, to transmit this information to all other students, through other i-boards. That is, every student would use, besides the i-board devoted to the teacher to provide new information, another one for writing new information to the other students. Whenever a student reads some information from the teacher, the student first writes this information to the i-board that is read by the other students, before returning the information. Of course, the student must in addition also read the i-boards on which the other students might have written newer information. The

Algorithm 4.4: From $(1, 1)$ Atomic to $(1, N)$ Atomic Registers

```
Implements:
    $(1, N)$-AtomicRegister, instance $onar$.

Uses:
    $(1, 1)$-AtomicRegister (multiple instances).

upon event ⟨ $onar$, $Init$ ⟩ do
    $ts := 0$;
    $acks := 0$;
    $writing :=$ FALSE;
    $readval := \bot$;
    $readlist := [\bot]^N$;
    forall $q \in \Pi, r \in \Pi$ do
        Initialize a new instance $ooar.q.r$ of $(1, 1)$-AtomicRegister
            with writer $r$ and reader $q$;

upon event ⟨ $onar$, $Write$ | $v$ ⟩ do
    $ts := ts + 1$;
    $writing :=$ TRUE;
    forall $q \in \Pi$ do
        trigger ⟨ $ooar.q.self$, $Write$ | $(ts, v)$ ⟩;

upon event ⟨ $ooar.q.self$, $WriteReturn$ ⟩ do
    $acks := acks + 1$;
    if $acks = N$ then
        $acks := 0$;
        if $writing =$ TRUE then
            trigger ⟨ $onar$, $WriteReturn$ ⟩;
            $writing :=$ FALSE;
        else
            trigger ⟨ $onar$, $ReadReturn$ | $readval$ ⟩;

upon event ⟨ $onar$, $Read$ ⟩ do
    forall $r \in \Pi$ do
        trigger ⟨ $ooar.self.r$, $Read$ ⟩;

upon event ⟨ $ooar.self.r$, $ReadReturn$ | $(ts', v')$ ⟩ do
    $readlist[r] := (ts', v')$;
    if $\#(readlist) = N$ then
        $(maxts, readval) := highest(readlist)$;
        $readlist := [\bot]^N$;
        forall $q \in \Pi$ do
            trigger ⟨ $ooar.q.self$, $Write$ | $(maxts, readval)$ ⟩;
```

teacher adds a timestamp to the written information to distinguish new information from old one.

The transformation in Algorithm 4.4 implements one $(1, N)$ atomic register instance *onar* from N^2 underlying $(1, 1)$ atomic register instances. Suppose the writer of the $(1, N)$ atomic register *onar* is process p (note that the writer is also a reader here, in contrast to the teacher in the story). The $(1, 1)$ registers are organized in a $N \times N$ matrix, with register instances called *ooar.q.r* for $q \in \Pi$ and $r \in \Pi$. They are used to communicate among all processes, from the writer p to all N readers and among the readers. In particular, register instance *ooar.q.r* is used to inform process q about the last value read by reader r; that is, process r writes to this register and process q reads from it. The register instances *ooar.q.p*, which are written by the writer p, are also used to store the written value in the first place; as process p may also operate as a reader, these instances have dual roles.

Note that both write and read operations require N registers to be updated; the *acks* counter keeps track of the number of updated registers in the write and read operation, respectively. As this is a local variable of the process that executes the operation, and as a process executes only one operation at a time, using the same variable in both operations does not create any interference between reading and writing. A variable *writing* keeps track of whether the process is writing on behalf of a write operation, or whether the process is engaged in a read operation and writing the value to be returned.

Algorithm 4.4 also relies on a timestamp ts maintained by the writer, which indicates the version of the current value of the register. For presentation simplicity, we use a function *highest*$(\cdot)$ that returns the timestamp/value pair with the largest timestamp from a list or a set of such pairs (this is similar to the *highestval* function introduced before, except that the timestamp/value pair is returned whereas *highestval* only returns the value). More formally, *highest*(S) with a set or a list of timestamp/value pairs S is defined as the pair $(ts, v) \in S$ such that

$$\text{forall } (ts', v') \in S \;:\; ts' < ts \lor (ts', v') = (ts, v).$$

The variable *readlist* is a length-N list of timestamp/value pairs; in the algorithm for reading, we convert it implicitly to the set of its entries. Recall that the function $\#(S)$ denotes the cardinality of a set S or the number of non-$\perp$ entries in a list S.

Correctness. Because of the *termination* property of the underlying $(1, 1)$ atomic registers, it is easy to see that every operation in the transformation algorithm eventually returns.

Similarly, because of the *validity* property of the underlying $(1, 1)$ atomic registers, and due to the choice of the value with the largest timestamp as the return value, we also derive the *validity* of the $(1, N)$ atomic register.

For the *ordering* property, consider an *onar*-write operation of a value v with associated timestamp ts_v that precedes an *onar*-write of value w with timestamp ts_w; this means that $ts_v < ts_w$. Assume that a process r *onar*-reads w. According to the algorithm, process r has written (ts_w, w) to N underlying registers, with identifiers *ooar.q.r* for $q \in \Pi$. Because of the *ordering* property of the $(1, 1)$ atomic registers, every subsequent read operation from instance *onar* reads at least one of the underlying registers that contains (ts_w, w), or a pair containing a higher timestamp. Hence, the read operation returns a value associated with a timestamp that is at least ts_w, and there is no way for the algorithm to return v.

Performance. Every write operation into the $(1, N)$ register requires N writes into $(1, 1)$ registers. Every read from the $(1, N)$ register requires one read from N $(1, 1)$ registers and one write into N $(1, 1)$ registers.

We give, in the following, two direct implementations of $(1, N)$ atomic register abstractions from distributed communication abstractions. The first algorithm is in the fail-stop system model and the second one uses the fail-silent model. These are adaptations of the "Read-One Write-All" and "Majority Voting" $(1, N)$ regular register algorithms, respectively. Both algorithms use the same approach as presented transformation, but require fewer messages than if the transformation would be applied automatically.

4.3.3 Fail-Stop Algorithm: Read-Impose Write-All $(1, N)$ Atomic Register

If the goal is to implement a $(1, N)$ register with one writer and multiple readers, the "Read-One Write-All" regular register algorithm (Algorithm 4.1) clearly does not work: the scenario depicted in Fig. 4.5 illustrates how it fails.

To cope with this case, we define an extension to the "Read-One Write-All" regular register algorithm that circumvents the problem by having the reader also *impose* the value it is about to return on all other processes. In other words, the read operation also *writes back* the value that it is about to return. This modification is described as Algorithm 4.5, called "Read-Impose Write-All." The writer uses a timestamp to distinguish the values it is writing, which ensures the *ordering* property of every execution. A process that is asked by another process to store an older value than the currently stored value does not modify its memory. We discuss the need for this test, as well as the need for the timestamp, through an exercise (at the end of this chapter).

The algorithm uses a request identifier *rid* in the same way as in Algorithm 4.2. Here, the request identifier field distinguishes among WRITE messages that belong to different reads or writes. A flag *reading* used during the writing part distinguishes between the write operations and the write-back part of the read operations.

Correctness. The *termination* and *validity* properties are ensured in the same way as in the "Read-One Write-All" algorithm (Algorithm 4.1). Consider now *ordering* and assume process p writes a value v, which is associated to some timestamp ts_v, and subsequently writes a value w, associated to some timestamp $ts_w > ts_v$. Assume, furthermore, that some process q reads w and, later on, some other process r invokes another read operation. At the time when q completes its read, all processes that did not crash have a timestamp variable ts that is at least ts_w. According to the algorithm, there is no way for r to change its value to v after this time because $ts_v < ts_w$.

Performance. Every write or read operation requires two communication steps, corresponding to the roundtrip communication between the writer or the reader and all processes. At most $O(N)$ messages are needed in both cases.

Algorithm 4.5: Read-Impose Write-All

```
Implements:
    (1, N)-AtomicRegister, instance onar.

Uses:
    BestEffortBroadcast, instance beb;
    PerfectPointToPointLinks, instance pl;
    PerfectFailureDetector, instance $\mathcal{P}$.

upon event ⟨ onar, Init ⟩ do
    $(ts, val) := (0, \bot)$;
    correct := $\Pi$;
    writeset := $\emptyset$;
    readval := $\bot$;
    reading := FALSE;

upon event ⟨ $\mathcal{P}$, Crash | p ⟩ do
    correct := correct $\setminus \{p\}$;

upon event ⟨ onar, Read ⟩ do
    reading := TRUE;
    readval := val;
    trigger ⟨ beb, Broadcast | [WRITE, $ts, val$] ⟩;

upon event ⟨ onar, Write | v ⟩ do
    trigger ⟨ beb, Broadcast | [WRITE, $ts + 1, v$] ⟩;

upon event ⟨ beb, Deliver | p, [WRITE, $ts', v'$] ⟩ do
    if $ts' > ts$ then
        $(ts, val) := (ts', v')$;
    trigger ⟨ pl, Send | p, [ACK] ⟩;

upon event ⟨ pl, Deliver | p, [ACK] ⟩ then
    writeset := writeset $\cup \{p\}$;

upon correct $\subseteq$ writeset do
    writeset := $\emptyset$;
    if reading = TRUE then
        reading := FALSE;
        trigger ⟨ onar, ReadReturn | readval ⟩;
    else
        trigger ⟨ onar, WriteReturn ⟩;
```

4.3.4 Fail-Silent Algorithm: Read-Impose Write-Majority $(1, N)$ Atomic Register

In this section, we consider a fail-silent model. We describe an extension of our "Majority Voting" $(1, N)$ regular register algorithm (Algorithm 4.2) to implement a $(1, N)$ atomic register.

Algorithm 4.6: Read-Impose Write-Majority (part 1, read)

```
Implements:
    (1, N)-AtomicRegister, instance onar.

Uses:
    BestEffortBroadcast, instance beb;
    PerfectPointToPointLinks, instance pl.

upon event ⟨ onar, Init ⟩ do
    $(ts, val) := (0, \perp)$;
    $wts := 0$;
    $acks := 0$;
    $rid := 0$;
    $readlist := [\perp]^N$;
    $readval := \perp$;
    $reading :=$ FALSE;

upon event ⟨ onar, Read ⟩ do
    $rid := rid + 1$;
    $acks := 0$;
    $readlist := [\perp]^N$;
    $reading :=$ TRUE;
    trigger ⟨ beb, Broadcast | [READ, rid] ⟩;

upon event ⟨ beb, Deliver | p, [READ, r] ⟩ do
    trigger ⟨ pl, Send | p, [VALUE, r, ts, val] ⟩;

upon event ⟨ pl, Deliver | q, [VALUE, r, ts', v'] ⟩ such that $r = rid$ do
    $readlist[q] := (ts', v')$;
    if $\#(readlist) > N/2$ then
        $(maxts, readval) := highest(readlist)$;
        $readlist := [\perp]^N$;
        trigger ⟨ beb, Broadcast | [WRITE, rid, maxts, readval] ⟩;
```

The algorithm is called “Read-Impose Write-Majority” and shown in Algorithm 4.6–4.7. The implementation of the write operation is similar to that of the “Majority Voting” algorithm: the writer simply makes sure a majority adopts its value. The implementation of the read operation is different, however. A reader selects the value with the largest timestamp from a majority, as in the “Majority Voting” algorithm, but now also imposes this value and makes sure a majority adopts it before completing the read operation: this is the key to ensuring the *ordering* property of an atomic register.

The “Majority Voting” algorithm can be seen as the combination of the “Read-Impose Write-Majority” algorithm with the two ideas that are found in the two-step transformation from $(1, N)$ regular registers to $(1, N)$ atomic registers (Algorithms 4.3 and 4.4): first, the mechanism to store the value with the highest timestamp that was read so far, as in Algorithm 4.3; and, second, the approach of the read implementation to write the value to all other processes before it is returned, as in Algorithm 4.4.

Algorithm 4.7: Read-Impose Write-Majority (part 2, write and write-back)

```
upon event ⟨ onar, Write | $v$ ⟩ do
    $rid := rid + 1$;
    $wts := wts + 1$;
    $acks := 0$;
    trigger ⟨ beb, Broadcast | [WRITE, $rid, wts, v$] ⟩;

upon event ⟨ beb, Deliver | $p$, [WRITE, $r, ts', v'$] ⟩ do
    if $ts' > ts$ then
        $(ts, val) := (ts', v')$;
    trigger ⟨ pl, Send | $p$, [ACK, $r$] ⟩;

upon event ⟨ pl, Deliver | $q$, [ACK, $r$] ⟩ such that $r = rid$ do
    $acks := acks + 1$;
    if $acks > N/2$ then
        $acks := 0$;
        if $reading$ = TRUE then
            $reading$ := FALSE;
            trigger ⟨ onar, ReadReturn | $readval$ ⟩;
        else
            trigger ⟨ onar, WriteReturn ⟩;
```

Correctness. The *termination* and *validity* properties are ensured in the same way as in Algorithm 4.2 ("Majority Voting"). Consider now the *ordering* property. Suppose that a read operation o_r by process r reads a value v from a write operation o_w of process p (the only writer), that a read operation $o_{r'}$ by process r' reads a different value v' from a write operation $o_{w'}$, also by process p, and that o_r precedes $o_{r'}$. Assume by contradiction that $o_{w'}$ precedes o_w. According to the algorithm, the timestamp ts_v that p associated with v is strictly larger than the timestamp $ts_{v'}$ that p associated with v'. Given that the operation o_r precedes $o_{r'}$, at the time when $o_{r'}$ was invoked, a majority of the processes has stored a timestamp value in ts that is at least ts_v, the timestamp associated to v, according to the write-back part of the algorithm for reading v. Hence, process r' cannot read v', because the timestamp associated to v' is strictly smaller than ts_v. A contradiction.

Performance. Every write operation requires two communication steps corresponding to one roundtrip exchange between p and a majority of the processes, and $O(N)$ messages are exchanged. Every read requires four communication steps corresponding to two roundtrip exchanges between the reader and a majority of the processes, or $O(N)$ messages in total.

4.4 (N, N) Atomic Register

4.4.1 Multiple Writers

All registers discussed so far have only a single writer. That is, our specifications of regular and atomic registers introduced in the previous sections do not provide any