PART I

The Register Abstraction

CHAPTER 1
The Atomic Register Abstraction

This chapter introduces the concepts of a regular register and of an atomic register. An informal presentation is first given in Section 1.1. Then, Section 1.2 provides the reader with a formal approach of the atomicity notion. It also discusses a fundamental property provided by the atomicity consistency criterion, namely locality.

1.1 THE REGISTER ABSTRACTION

1.1.1 CONCURRENT OBJECTS AND REGISTERS

Concurrent object A *concurrent object* is an object that can be accessed concurrently by two or more sequential processes. As it is sequential, a process that has invoked an operation on an object has to wait for a corresponding response before invoking another operation on the same or another object. When this occurs, we say that the operation is *pending*.

While each process can access at most one object at a time, an object can be simultaneously accessed by several processes. This occurs when two or more processes have pending invocations on the same object, hence the name “concurrent object”.

Register object One of the most fundamental concurrent objects is the shared register object (in short, *register*). Such an object abstracts physical objects such as a word (or a set of words) of a shared memory, a shared disk, etc. A register R provides the processes with an interface made up of two operations denoted R.read() and R.write(). The first allows the invoking process to obtain the value of the register R, while the second allows it to associate a new value with the register.

Type of register According to the value that can be returned by a read operation, several types of registers can be defined. We consider here two types of registers: the *regular* register and the *atomic* register. In both cases, the value returned by a read operation is a value that has been written by a write operation. The actual value returned depends on the concurrency pattern in which is involved the corresponding read operation. Regularity and atomicity differ in the way each addresses (and captures) concurrency.

A register type defines which are the correct behaviors of a register of that type. Hence, the regular (resp. atomic) type defines *regularity* (resp. *atomicity*) consistency condition.

Underlying time notion The definitions that follow refer to a notion of *time*. This time notion can be seen as given by an imaginary clock that models the progress of a computation as perceived by an external omniscient observer. It is accessible neither to the processes, nor to the register objects. Its aim is to capture the fact that, from the point of view of an omniscient external observer, the flow of operations is such that (1) some operation invocations have terminated while others have not yet started, and (2) some operation invocations overlap in time (they are concurrent). (These notions will be formally defined in Section 1.2.)

1.1.2 REGULAR REGISTER

Definition A *regular register* is a one-writer/multi-reader (1WMR) register, i.e., a read/write object that can be written by a single process and read by any number of processes. The definition of a regular register assumes a single writer in order to prevent write conflicts. More precisely, as the writer is sequential, the write operations are totally ordered (the corresponding sequence of write operations is called the *write sequence*). The value returned by a read is defined as follows:

- If no write is concurrent with the read, the read returns the current value of the register (i.e., the value written by the last write in the current write sequence).
- If writes are concurrent with the read, the read returns the value written by one of these writes or the last value of the register before these writes.

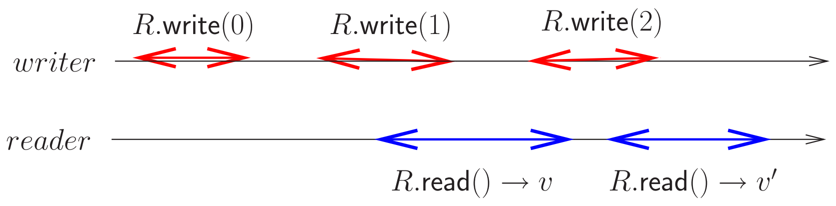

Figure 1.1: Possible behaviors of a regular register

Examples The definition of the regularity consistency condition is illustrated in Figure 1.1 where a writer process and a single reader process are considered. The notation "R.read() $\to v$" is used to indicate that the read operation returns the value v. As far as concurrency patterns are concerned, the durations of each operation are indicated on the figure with double-arrow segments.

The writer issues three write operations that sequentially write into the register R the values 0, 1 and 2. On its side, the reader issues two read operations; the first obtains the value v, while the second obtains the value v'. The first read is concurrent with the writes of the values 1 and 2 (their executions overlap in time). According to the definition of regularity, it can return for v any of the values 0, 1 or 2. The second read is concurrent only with the write of the value 2. It can, consequently, return for v' the value 1 or the value 2.

So, in this example, the regular type allows the second read to return 1 (which has been written before the value 2), while the first read (that precedes it) is allowed to return the value 2 (which has been written after the value 1). This is called a *new/old inversion*: in presence of read/write concurrency, a sequence of read operations is not required to return a sequence of values that complies with the sequence of write operations. It is interesting to notice that, if we suppress $R.\mathsf{write}(2)$ from the figure, v is restricted to be 0 or 1, while v' can only be the value 1 (and, as we are about to see, the register then behaves as if it was atomic).

Observation An object is defined by a *sequential specification* when its correct behaviors can be defined by sequences (traces) on its operations. As an example, let us consider an infinite queue object Q defined by the classical operations $Q.\mathsf{enqueue}()$ and $Q.\mathsf{dequeue}()$. Its sequential specification includes all the sequences (involving only these two operations) such that, for any prefix of any sequence, the number of $Q.\mathsf{dequeue}()$ operations is never more than the number of $Q.\mathsf{enqueue}()$ operations.

It is easy to see that, due to the possibility of new/old inversions, a regular register cannot have a sequential specification.

1.1.3 ATOMIC REGISTERS

Definition There are two main differences between regularity and atomicity, namely, an atomic register (1) can be a multi-writer/multi-reader (MWMR) register and (2) does not allow for new/old inversions (i.e., it has a sequential specification). Let us notice that a 1WMR atomic register is also regular. More precisely, an atomic register is defined by the following properties.

- All the read and write operations appear as if they have been executed sequentially,
- This sequence respects the time order of the operations (i.e., if op_1 has terminated before op_2 has started, then op_1 appears in the sequence before op_2),
- Each read returns the value written by the closest preceding write in this sequence (or the initial value if there is no preceding write operation).

The corresponding sequence of operations is called a *witness* sequence. Let us notice that the concurrent operations can be ordered arbitrarily as long as the sequence obtained is a witness sequence. Basically, this definition states that everything has to appear as if each operation has been executed instantaneously at some point of the time line (of the omniscient external observer) between its invocation (start event) and its termination (end event).

Examples An example of an execution of a MWMR atomic register accessed by three processes is described in Figure 1.2. (Two dotted arrows are associated with each operation invocation. They meet at a point of the "real time" line at which the corresponding operation could have instantaneously occurred. These points of the time lime defines a sequence on the operation invocations that belongs to the specification of the object.) In that example, everything appears as if the operations have been

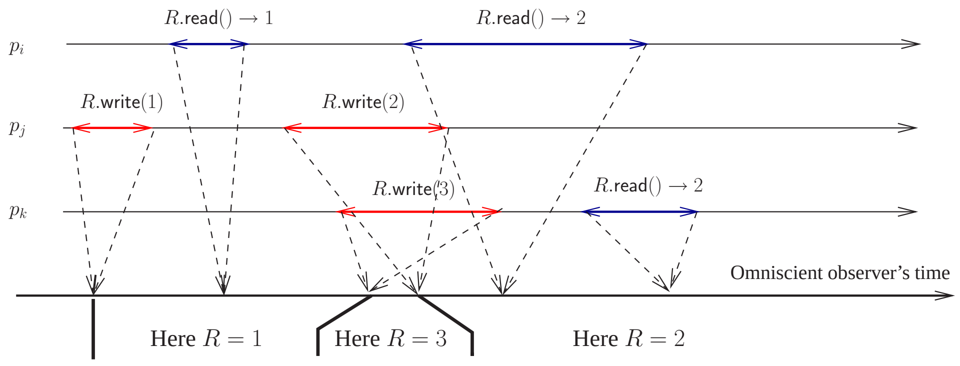

Figure 1.2: Behavior of an atomic register

executed according to the following witness sequence:

$$R.\mathsf{write}(1),\ R.\mathsf{read}() \to 1,\ R.\mathsf{write}(3),\ R.\mathsf{write}(2),\ R.\mathsf{read}() \to 2,\ R.\mathsf{read}() \to 2.$$

The concurrent operations $R.\mathsf{write}(3)$ and $R.\mathsf{write}(2)$ could have been ordered the other way. In that case, the last two read operations would have to return the value 3 in order that the register R behaves atomically.

When we consider the example described in Figure 1.1 with $v = 2$ and $v' = 1$, there is a new/old inversion, and, consequently, the register R does not behave atomically. Differently, if (1) either $v = 0$ or 1 and $v' = 1$ or 2, (2) or $v = v' = 2$, there is no new/old inversion, and, consequently, the behavior of the register is atomic.

Why atomicity is important The atomicity consistency condition is fundamental. This is because atomicity allows the composition of atomic objects without losing the benefit of atomicity. That is why atomicity is called a *local* consistency condition. A consistency condition C is local if, when considering several concurrent objects (here the registers) as a single object O, the concurrent object O satisfies the condition C when each object taken alone satisfies C (see section 1.2.4).

This is very important both when one has to reason about multiprocess programs that access shared registers, and when one has to implement shared registers.

- From a theoretical point of view, locality means that we can keep *reasoning sequentially* whatever the number of atomic registers involved in the computation. Locality allows us to reason on a set of atomic registers as if they were a single atomic object. We can reason in terms of witness sequences, not only for each register separately, but also on all the registers as if they were a single atomic object.

  As an example, let us consider an application composed of processes that share two atomic registers $R1$ and $R2$. Then, the composite object $[R1, R2]$, that provides the processes with the four operations: $R1.\mathsf{write}()$, $R1.\mathsf{read}()$, $R2.\mathsf{write}()$, and $R2.\mathsf{read}()$, behaves atomically (everything appears as if one operation at a time was executed, and the projection of this global sequence on the operations of $R1$ -resp. $R2$- is a witness sequence for $R1$ -resp. $R2$-).

- From a practical point of view, locality means *modularity*. This has several advantages. On the one side, each atomic register can be implemented in its own way: the implementation of one atomic register is not required to interfere with the implementation of the other registers.

  On the other side, as soon as we have an algorithm that implements an atomic register (e.g., in a message-passing system as we will see in the next chapter), we can use multiple independent instantiations of it, one for each register, and the system will behave correctly without any additional control or synchronization.

To summarize, locality means that atomic registers compose for free (i.e., their composition is at no additional cost).

1.2 A FORMAL APPROACH TO ATOMICITY

1.2.1 PROCESSES, OPERATIONS AND EVENTS

Processes and operations A multiprocess program consists of a finite set of n (application) processes, denoted $p_1, \ldots, p_n$ that cooperate through a set of registers. As already indicated, each register R provides the processes with two operations $R.\mathsf{write}()$ and $R.\mathsf{read}()$. The notation $R.\mathsf{op}(arg)(res)$ is used denote any operation on a register R, where arg is the input parameter (empty for a read, and the value v to be written for a write), and res is the response returned by the operation (*ok* for a write, and the value v obtained from R for a read operation).

Events The execution of an operation $\mathsf{op}()$ on a register R by a process p_i is modeled by two events, namely, the events denoted $inv[R.\mathsf{op}(arg)$ by $p_i]$ that occurs when p_i invokes the operation (invocation event), and the event denoted $resp[R.\mathsf{op}(res)$ by $p_i]$ that occurs when the operation terminates. (When there is no ambiguity, we talk about operations where we should be talking about operation executions.) We say that these events are generated by process p_i and associated with register R. Given an operation $R.\mathsf{op}(arg)(res)$, the event $resp[R.\mathsf{op}(res)$ by $p_i]$ is called the *reply event* matching the *invocation event* $inv[R.\mathsf{op}(arg)$ by $p_i]$.

1.2.2 HISTORIES

Representing an execution as a history of events This paragraph formalizes what we usually intend when we use the word *execution* or *run*.

As simultaneous (invocation and reply) register events generated by sequential processes are independent, it is always possible to order simultaneous events in an arbitrary way without altering the behavior of an execution. This makes it possible to consider a total order relation (denoted $<_H$ in the following) on the events that abstracts the time order in which the events do actually occur (i.e., the time of the omniscient external observer). This is precisely how executions are formally captured.

Hence, the interaction between a set of sequential processes and a set of shared registers is modeled by a sequence of invocation and reply events, called a *history* (sometimes also called a *trace*), and denoted $\widehat{H} = (H, <_H)$ where H is the set of events generated by the processes and $<_H$ a total order on these events.

The notation $\widehat{H}|p_i$ ($\widehat{H}$ at p_i) denotes the subsequence of $\widehat{H}$ made up of all the events generated by process p_i. It is called the *local* history at p_i.

As a simple example, Figure 1.3 describes the history (the sequence of 12 events $e_1 \ldots, e_{12}$) associated with the execution depicted in Figure 1.2. (Only the first four events are described explicitly.)

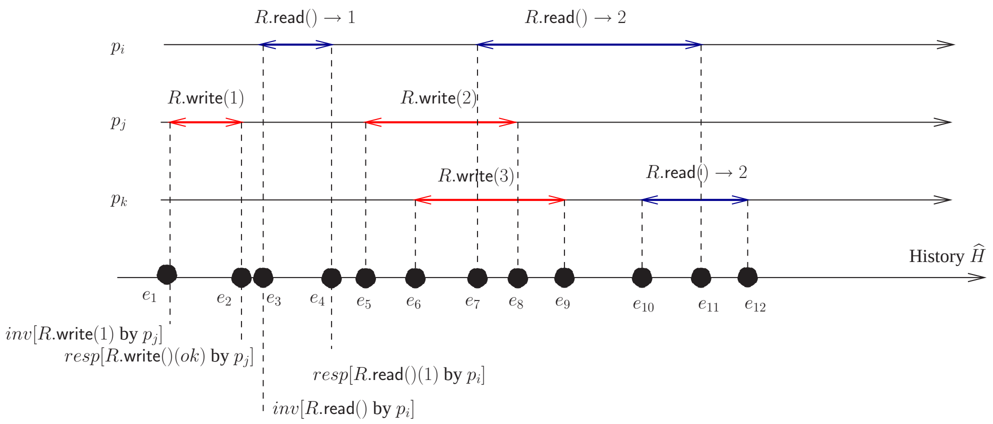
R.read() → 1
R.read() → 2
p_i
R.write(1)
R.write(2)
p_j
R.write(3)
R.read() → 2
p_k
History $\widehat{H}$
e_1 e_2 e_3 e_4 e_5 e_6 e_7 e_8 e_9 e_{10} e_{11} e_{12}
$inv[R$.write(1) by $p_j]$
$resp[R$.write()(ok) by $p_j]$
$resp[R$.read()(1) by $p_i]$
$inv[R$.read() by $p_i]$

Figure 1.3: Example of a history

Equivalent histories Two histories $\widehat{H}$ and $\widehat{H'}$ are said to be *equivalent* if they have the same local histories, i.e., for each p_i, $\widehat{H}|p_i = \widehat{H'}|p_i$. That is, equivalent histories are built from the same set of events (remember that an event includes the name of an object, the name of a process, the name of an operation and its input or output parameter).

Well-formed histories As we consider histories generated by sequential processes, we restrict our attention to the histories $\widehat{H}$ such that, for each process p_i, $\widehat{H}|p_i$ (local history at p_i) is sequential: it starts with an invocation, followed by its matching reply, followed by another invocation, etc. We say in this case that $\widehat{H}$ is *well-formed*.

Partial order on operations A history $\widehat{H}$ induces an irreflexive partial order on its operations as follows. Let op$= X$.op1 () by p_i and op' $= Y$.op2() by p_j be two operations. Operation op precedes operation op' (denoted op$\rightarrow_H$op') if op terminates before op' starts, where "terminates" and "starts" refer to the time line abstracted by the $<_H$ total order relation. More formally:

$$(\mathsf{op} \rightarrow_H \mathsf{op'}) \stackrel{\text{def}}{=} (resp[\mathsf{op}] <_H inv[\mathsf{op'}]).$$

Two operations op and op' are said to *overlap* (as already seen, we also say they are *concurrent*) in a history $\widehat{H}$ if neither $resp[\mathsf{op}] <_H inv[\mathsf{op'}]$, nor $resp[\mathsf{op'}] <_H inv[\mathsf{op}]$. Notice that two overlapping operations are such that $\neg(\mathsf{op} \rightarrow_H \mathsf{op'})$ and $\neg(op' \rightarrow_H \mathsf{op})$.

The partial order generated by the execution described in Figure 1.2 is given in Figure 1.4.

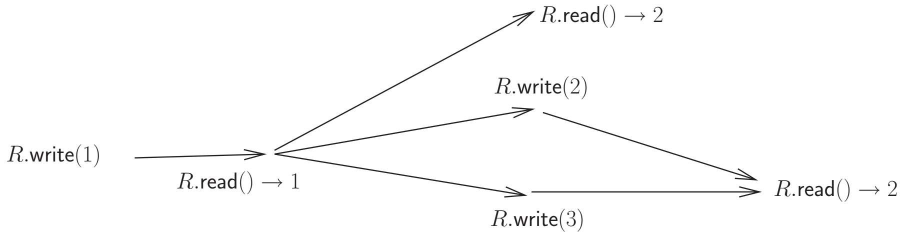

Figure 1.4: Partial order on the operations

Sequential history A history $\widehat{H}$ is *sequential* if its first event is an invocation, and then (1) each invocation event is immediately followed by the matching reply event, and (2) each reply event (except possibly the last if the execution is infinite) is immediately followed by an invocation event. If $\widehat{H}$ is a sequential history, it has no overlapping operations, and, consequently, the order $\rightarrow_H$ on its operations is a total order. A history $\widehat{H}$ that is not sequential is *concurrent*.

A sequential history models a sequential multiprocess computation (there are no overlapping operations in such a computation), while a concurrent history models a concurrent multiprocess computation (there are at least two overlapping operations in such a computation). The very important point is that, with a sequential history, one can thus reason about executions at the granularity of operations invoked by processes, instead of the granularity of underlying events.

1.2.3 ATOMICITY

The motivation that underlies the definition of atomicity is pretty simple, namely, its aim is (1) to permit one to consider an execution of a set of processes accessing a set of shared objects as if only one operation at a time occurs, (2) while being independent of the number of atomic objects (this point addresses "object composition"): several atomic objects have to behave as if they were a single "big" atomic object. Point (1) is addressed in this section, while point (2) is addressed in the following one.

Legal history Given a sequential history $\widehat{S}$ and a register R, let $\widehat{S}|R$ ($\widehat{S}$ at R) denote the subsequence of $\widehat{S}$ made up of all events involving only register R. (Notation $\widehat{S}|R$ is -voluntarily- similar to $\widehat{S}|p_i$: in both cases, it denotes the subsequence of $\widehat{S}$ made up of all events involving only register R or process p_i.) Let us notice that, as $\widehat{S}$ is a sequential history, each $\widehat{S}|R$ is also a sequential history.

We say that a sequential history $\widehat{S}$ is *legal* if, for each register R, the sequence $\widehat{S}|R$ is such that each of its read operations returns the value written by the closest preceding write in $\widehat{S}|R$ (or the initial value of R if there is no preceding write).

Atomic history We define here atomicity for histories without pending operations, i.e., each invocation event of $\widehat{H}$ has a matching reply event in $\widehat{H}$. (Extending the definition to histories with pending operations is left as an exercise.)

Definition A history $\widehat{H}$ is *atomic* if there is a "witness" history $\widehat{S}$ such that:

1. $\widehat{H}$ and $\widehat{S}$ are equivalent,
2. $\widehat{S}$ is sequential and legal, and
3. $\rightarrow_H \subseteq \rightarrow_S$.

The definition above states that for a history $\widehat{H}$ to be atomic, there must exist a permutation $\widehat{S}$ (witness history) of $\widehat{H}$, which satisfies the following requirements. First, $\widehat{S}$ has to be composed of the same set of events as $\widehat{H}$ [item 1]. Second, $\widehat{S}$ has to be sequential (i.e., an interleaving of the process histories at the granularity of complete operations) and legal (i.e., it has to respect the sequential specification of each register) [item 2]. Notice that, as $\widehat{S}$ is sequential, $\rightarrow_S$ is a total order. Finally, $\widehat{S}$ has also to respect the occurrence order of the operations as defined by $\rightarrow_H$ [item 3]. $\widehat{S}$ represents a history that could have been obtained by executing all the operations, one after the other, while respecting the occurrence order of all the non-overlapping operations. Such a sequential history $\widehat{S}$ is sometimes called a *linearization* of $\widehat{H}$.

Remark on non-determinism It is important to notice that the notion of atomicity includes inherently a form of non-determinism in the sense that, given a history $\widehat{H}$, several linearizations of $\widehat{H}$ might exist.

Linearization point The very existence of a linearization of an (atomic) history $\widehat{H}$ means that each operation of $\widehat{H}$ could have been instantaneously executed at a point of the time line (as defined by the total order $<_H$) that lies between its invocation and reply time events. Such a point is called the *linearization point* of the corresponding operation.

Let us notice that one way to prove that all the histories generated by an algorithm are atomic, consists in identifying a linearization point for each of its operations. These points have to (1) respect the time occurrence order of the non-overlapping operations and (2) be consistent with the sequential specification of the object.

1.2.4 ATOMICITY IS A LOCAL PROPERTY

Let P be any property defined on a set of objects. As already indicated, P is *local* if the set of objects as a whole satisfies P whenever each object taken alone satisfies P. As already discussed, locality is an important concept that states that objects can be composed for free.

This section proves that the atomicity consistency criterion is a local property. Intuitively, the fact that atomicity is local comes from the fact that it involves the time occurrence order on non-concurrent operations whatever the objects and the processes concerned by these operations are. We will rely on this aspect in the proof of the following theorem. As the following theorem is correct not only for the atomic registers, but more generally for any object that has a sequential specification, it is formulated and proved on an object basis (as we have previously seen, an atomic register is a particular object that provides the processes with a read and a write operations and is defined by a sequential specification).

Theorem 1.1 *A history $\widehat{H}$ is atomic if, and only if, for each object X involved in $\widehat{H}$, $\widehat{H}|X$ is atomic.*

Proof The "$\Rightarrow$" direction (only if) is an immediate consequence of the definition of atomicity: if $\widehat{H}$ is atomic then, for each object X involved in $\widehat{H}$, $\widehat{H}|X$ is atomic. So, the rest of the proof is restricted to the "$\Leftarrow$" direction.

Given an object X, let $\widehat{S_X}$ be a linearization of $\widehat{H}|X$. It follows from the definition of atomicity that $\widehat{S_X}$ defines a total order on the operations involving X. Let $\rightarrow_X$ denote this total order. We construct an order relation $\rightarrow$ defined on the whole set of operations of $\widehat{H}$ as follows:

1. For each object X: $\rightarrow_X \;\subseteq\; \rightarrow$,
2. $\rightarrow_H \;\subseteq\; \rightarrow$.

Basically, "$\rightarrow$" totally orders all operations on the same object X, according to $\rightarrow_X$ (item 1), while preserving $\rightarrow_H$, i.e., the real-time occurrence order on operations (item 2).
Claim C. "$\rightarrow$ is acyclic". This means that $\rightarrow$ indeed defines a partial order on the set of all the operations of $\widehat{H}$. Assuming this claim, it is thus possible to construct a sequential history $\widehat{S}$ including all events of $\widehat{H}$ and respecting $\rightarrow$. We trivially have $\rightarrow \subseteq \rightarrow_S$ where $\rightarrow_S$ is the total order on the operations defined from $\widehat{S}$. We have the three following conditions: (1) $\widehat{H}$ and $\widehat{S}$ are equivalent (they contain the same events); (2) $\widehat{S}$ is sequential (by construction) and legal (due to item 1 above); and (3) $\rightarrow_H \subseteq \rightarrow_S$ (due to item 2 above and $\rightarrow \subseteq \rightarrow_S$). It follows that $\widehat{H}$ is linearizable.

Proof of claim C. We show (by contradiction) that $\rightarrow$ is acyclic. Assume first that $\rightarrow$ induces a cycle involving the operations on a single object X. Indeed, as $\rightarrow_X$ is a total order, in particular transitive, there exist two operations op_i and op_j on X such that $\mathsf{op}_i \rightarrow_X op_j$ and $\mathsf{op}_j \rightarrow_H op_i$. We have the following.

- $\mathsf{op}_i \rightarrow_X \mathsf{op}_j \Rightarrow inv[\mathsf{op}_i] <_H resp[\mathsf{op}_j]$ because X is atomic, and
- $\mathsf{op}_j \rightarrow_H \mathsf{op}_i \Rightarrow resp[\mathsf{op}_j] <_H inv[\mathsf{op}_i]$,

which shows a contradiction, as $<_H$ is a total order on the whole set of events.

It follows that any cycle must involve at least two objects. To obtain a contradiction, we show that, in that case, a cycle in $\rightarrow$ implies a cycle in $\rightarrow_H$ (which is acyclic). Let us examine the way the cycle could be obtained. If two consecutive edges of the cycle are due to either some $\rightarrow_X$ (because of an object X), or $\rightarrow_H$ (due the total order $<_H$), then the cycle can be shortened as any of these relations is transitive. Moreover, $\mathsf{op}_i \rightarrow_X \mathsf{op}_j \rightarrow_Y \mathsf{op}_k$ is not possible for $X \neq Y$, as each operation is on one object only ($\mathsf{op}_i \rightarrow_X \mathsf{op}_j \rightarrow_Y \mathsf{op}_k$ would imply that op_j is on both X and Y).

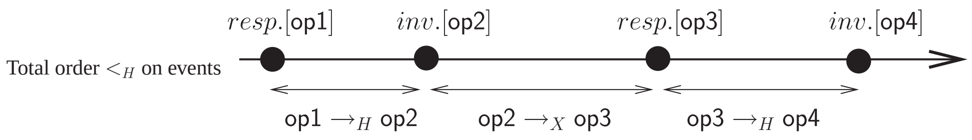

Figure 1.5: Developing $\mathsf{op1} \rightarrow_H \mathsf{op2} \rightarrow_X \mathsf{op3} \rightarrow_H \mathsf{op4}$

So, let us consider any sequence of edges of the cycle such that: $\mathsf{op1} \rightarrow_H \mathsf{op2} \rightarrow_X \mathsf{op3} \rightarrow_H \mathsf{op4}$. We have (see Figure 1.5):

1. $\mathsf{op1} \rightarrow_H \mathsf{op2} \Rightarrow resp[\mathsf{op1}] <_H inv[op2]$ (definition of $\mathsf{op1} \rightarrow_H \mathsf{op2}$),
2. $\mathsf{op2} \rightarrow_X \mathsf{op3} \Rightarrow inv[\mathsf{op2}] \quad <_H resp[\mathsf{op3}]$ (as X is atomic),
3. $\mathsf{op3} \rightarrow_H \mathsf{op4} \Rightarrow resp[\mathsf{op3}] <_H inv[\mathsf{op4}]$ (definition of $\mathsf{op3} \rightarrow_H \mathsf{op4}$).

Combining these statements, we obtain $resp[\mathsf{op1}] <_H inv[\mathsf{op4}]$ from which we can conclude that $\mathsf{op1} \rightarrow_H \mathsf{op4}$. It follows that all the edges due to the relations $\rightarrow_X$ (associated with every object X) can be suppressed, and, consequently, any cycle in $\rightarrow$ can be reduced to a cycle in $\rightarrow_H$, which is a contradiction as $\rightarrow_H$ is an irreflexive partial order. End of proof of claim C. $\square_{Theorem\ 1.1}$

As already indicated, atomicity allows the programmer to reason as if the operations issued by processes on the objects were executed one at a time. The previous theorem is fundamental. It states that to reason about sequential processes that access atomic registers, one can reason on a per object basis without losing the atomicity property of the whole computation.

1.3 BIBLIOGRAPHIC NOTES

- The notion of a regular register has been introduced by Lamport [110]. The notion of an atomic read/write object (register), as studied here, has been investigated and formalized by Lamport in the same paper. (Lamport has also introduced the notion of a *safe* register that is a notion weaker than a regular register. This notion has not been addressed and developed here because its interest is very limited in the context of message passing systems.)

  A more hardware-oriented investigation of atomic registers has been given by Misra [123]. An extension of the regularity condition to MWMR registers is described in [155].
- The generalization of the atomicity consistency condition to any object defined by a sequential specification (set of traces) has been developed by Herlihy and Wing under the name linearizability [98].
- The notion of local property (composition) and the theorem stating that atomicity is a local property are due to Herlihy and Wing [98].

- It is important to notice that, differently from atomicity, sequential consistency [109] and the major part of the consistency conditions encountered in database concurrency control [23, 141] are consistency conditions that do not satisfy the locality property.

CHAPTER 2

Implementing an Atomic Register in a Crash-Prone Asynchronous System

This chapter is on the implementation (we also say construction) of an atomic register in a message-passing system prone to process crashes. It first presents, in an incremental way, the construction of a multi-writer/multi-reader (MWMR) atomic register. This construction assumes a majority of non-faulty processes. Then, after having shown that this assumption is a necessary requirement for such a construction, the chapter considers a failure detector approach in order to circumvent this necessity.

2.1 THE UNDERLYING SYSTEM MODEL

Processes The system on top of which we want to build an atomic register is made up of a set of n sequential processes denoted $\Pi = \{p_1, \dots, p_n\}$ (we also sometimes use Π to denote the set of identities $\{1, \dots, n\}$). Each process p_i is an automaton. It has an initial state and executes a sequence of steps defined by a transition function called the *local algorithm* assigned to that process. A step is atomic (i.e., executed entirely or not at all). A step corresponds to the execution of an internal statement (i.e., a statement involving only the local state automaton), the sending of a message or the reception of a message. The set of these local algorithms is called a *distributed algorithm*.

Failure model A process can crash. After it has crashed, a process executes no more steps. There is no recovery. A process that crashes in a run is *faulty* in that run; otherwise, it is *non-faulty*.

The model parameter t $(0 \leq t < n)$ denotes the maximum number of processes that may crash during an execution. We are interested in t-*resilient* algorithms, i.e., in algorithms that have to always behave correctly when at most t processes crash. Conversely, when more than t processes crash, there is no guarantee on their behavior.

Communication medium Each pair of processes is connected by a bidirectional *link* (sometimes also called *channel*). In order to simplify the description of the algorithms, we sometimes consider that a process can also send a message to itself.

To send a message m to a process p_j, a process p_i invokes the operation "send m to p_j". To receive a message from p_j, it invokes " receive () from p_j" (or " receive m from p_j" where m will contain the received message); p_i can invoke " receive ()" where the sender of the message can be any process. When a process invokes a receive operation, it is blocked until a message arrives.

Each communication channel is assumed to be reliable: there are neither creation, modification, nor loss of messages. The notation " broadcast m" is used as a shortcut for

$$\textbf{for each } j \in \{1, \ldots, n\} \textbf{ do } \texttt{send } m \texttt{ to } p_j \textbf{ end for}$$

where $j \in \{1, \ldots, n\}$ means that the order in which the message m is sent to every process is arbitrary. It is important to see that " broadcast m" is not an all-or-nothing operation: if p_i crashes while executing it, an arbitrary subset of the processes receive message m. Moreover, when it executes " broadcast m", a process p_i sends a message to itself. This allows us to write more concise algorithms. (Of course, simple modifications of these algorithms allow the sending of such messages to be suppressed in practical implementations.)

Timing model The processes are asynchronous. This means that each process proceeds at its own speed. There is no assumption on the relative speed of one process with respect to another. The only assumption is that, until it crashes -if it ever crashes-, the speed of a process is always positive: a non-faulty process eventually executes the next step of its algorithm.

At the communication level, there is no bound on message transfer delays. The only guarantee is that, if a process returns from the send of a message and the destination process is non-faulty, then the destination process eventually receives the message (because communication channels are assumed to be reliable) if it executes " receive ()"operation infinitely often.

This message-passing system model is called the *asynchronous* model. It is also sometimes called *time-free* model. It is important to notice that the processes are not provided with a notion of global time. If a process uses a local physical clock, the dates and durations measured by its clock are meaningful only for itself, which means that they have no meaning for the other processes.

Notation In the following, we use the notation $\mathcal{AS}_{n,t}[\emptyset]$ to denote the previous message-passing crash-prone asynchronous system model.

Let P be a predicate on the model parameters n and t, e.g., $P \equiv (t < n/2)$. $\mathcal{AS}_{n,t}[P]$ denotes the system model that is the restriction of $\mathcal{AS}_{n,t}[\emptyset]$ to the runs where P is always satisfied. As an example, $\mathcal{AS}_{n,t}[t < n/2]$ denotes any system where, in each run (history), a majority of processes do not crash.

Finally, $\mathcal{AS}_{n,t}[T]$, where T is the type of a distributed object, denotes the system model $\mathcal{AS}_{n,t}[\emptyset]$, where the processes can additionally access objects of the type T to cooperate. An example (that will be studied in Section 2.4), $\mathcal{AS}_{n,t}[\Sigma]$ denotes a system model where the processes have access to a failure detector object of the class (type) Σ.

More generally, $\mathcal{AS}_{n,t}[P, T]$, where T is an object type and P a predicate (e.g., $t < n/2$), denotes the system model $\mathcal{AS}_{n,t}[\emptyset]$ where the processes can access objects of type T, and all the executions (histories) satisfy predicate P.

2.2 BUILDING AN MWMR ATOMIC REGISTER IN $\mathcal{AS}_{n,t}[t < n/2]$

2.2.1 PROBLEM SPECIFICATION

Specification The MWMR atomic register R is implemented collectively by the n processes. To that end, each process executes an appropriate algorithm such that the safety and liveness properties of R.read() and R. write() operations associated with R are satisfied. These properties are the following:

- Safety. This property is nothing else than the atomicity consistency condition: any execution of R.read() and R.write() operations is such that these operations (except possibly the last operation issued by faulty processes) appear as if they have been executed sequentially. This total order complies with the operations' real time order (see the previous chapter for a formal statement of atomicity).
- Liveness. This property states that, whatever the operation (R.read() or R.write()), if the invoking process is non-faulty, then that operation terminates (the invoking process obtains a response).

An incremental construction The algorithm building an atomic MWMR register is presented incrementally. An algorithm building a 1WMR regular register is first introduced. It is then enriched to prevent new/old inversions in order to obtain a 1WMR atomic register. Finally, a simple modification extends it to go from a 1WMR atomic register to an MWMR atomic register. All these constructions require a majority of processes to be non-faulty, so they assume that the underlying system model is $\mathcal{AS}_{n,t}[t < n/2]$.

2.2.2 IMPLEMENTING A 1WMR REGULAR REGISTER IN $\mathcal{AS}_{n,t}[t < n/2]$

Underlying principle The idea that underlies the construction is quite simple. Let p_w denote the writer process. On the one hand, p_w associates a sequence number with each of its write operations and broadcasts the pair (new value, sequence number). On the other hand, every process p_i keeps in its local memory the pair with the highest sequence number it has ever seen.

Both the safety property (regularity) and the liveness property associated with a regular register are obtained from the "majority of correct processes" assumption ($t < n/2$). This is because this assumption allows a process to always communicate with a majority of processes (i.e., with at least one non-faulty process) before terminating its current read or write operation.

Local variables Each process p_i manages the following local variables.

- cur_val_i is a data variable that contains the current value of the register R (as known by p_i).
- w_sn_i is a control variable that keeps the sequence number associated with the value currently kept in cur_val_i. As far as p_w is concerned, w_sn_w is also used to generate the increasing sequence numbers associated with the values written into R.

- req_sn_i is a control variable containing the sequence number that p_i has associated with its last read of R. (These sequence numbers allow every ACK () to be correctly associated with the request that gives rise to its sending.)

All the local variables used to generate a sequence number are initialized to 0. The register R is assumed to be initialized to some value (say v_init). Consequently, all the local variables cur_val_i are initialized to v_init.

The construction The algorithm that builds a regular 1WMR register R is described in Figure 2.1. The statement "**wait until** $\big($TAG(sn) received$\big)$" means that the invoking process is blocked until its input buffer contains a message whose type is TAG and that carries the value sn. When the wait statement terminates the message is consumed and suppressed from the input buffer.

When p_w invokes R.write (v), it computes the next sequence number w_sn_w, broadcasts the message WRITE(v, w_sn_w), and waits for corresponding acknowledgments from a majority of processes before terminating the write operation. When a process p_i receives such a message, it updates its current pair (cur_val_i, w_sn_i) if $w\_sn_w \geq w\_sn_i$, and sends back to p_w an acknowledgment ACK_WRITE_REQ(w_sn) to p_w. Otherwise, the message WRITE(v, w_sn_w) is an old message and p_i discards it.

When a process p_i invokes R.read (), it broadcasts an inquiry message READ_REQ (req_sn_i) where req_sn_i is a sequence number used to identify each read request of p_i. When a process p_k receives such a message it sends back its current value of the register. Then, when p_i has received ACK_READ_REQ $(req\_sn_i, -, -)$ messages from a majority of processes, it returns the value v it has received that is associated with the greatest sequence number.

Remarks When it receives a WRITE (val, w_sn) message from the writer p_w, a process p_i evaluates the predicate $w\_sn \geq w\_sn_i$. Actually this predicate could be strengthened in $w\_sn > w\_sn_i$ for a process $p_i \neq p_w$. Using the predicate $w\_sn \geq w\_sn_i$ allows us not to distinguish p_w from the other processes.

The code of the algorithm can be easily modified to save a few messages. When p_w executes R.write (), it is not necessary to send a message to itself. It can instead write directly v into cur_val_w. Moreover, when p_w wants to read R, it can directly return the current value of cur_val_w. In the same vein, when a process p_i $(i \neq w)$ invokes R.read (), it can save the sending of a message to itself as long as, in addition to the acknowledgment messages it receives, it also considers its own pair (w_sn_i, cur_val_i) when it determines the value to be returned. Finally, when it waits for acknowledgments, a process has now to wait for messages from a majority of processes minus one.

Cost It is easy to see that the cost of a read or a write operation is $2n$ messages. As far as the time complexity is concerned, let us assume that (a) local computation durations are negligible when compared to message transit delays, and (b) every message takes one time unit. The number of "time units" that are needed by an operation is actually counts the number of sequential communication steps this operation gives rise to.

```
operation R.write (v): % This code is only for the single writer $p_w$ %
    $w\_sn_w \leftarrow w\_sn_w + 1$;
    broadcast WRITE $(v, w\_sn_w)$;
    wait until ( ACK_WRITE $(w\_sn_w)$ received from a majority of processes );
    return $(ok)$.
--------------------------------------------------------------------------------
% The code snippets that follow are for every process $p_i$ $(i \in \{1, \ldots, n\})$ %

operation R.read (): % This code is for any process $p_i$ %
    $req\_sn_i \leftarrow req\_sn_i + 1$;
    broadcast READ_REQ $(req\_sn_i)$;
    wait until ( ACK_READ_REQ $(req\_sn_i, -, -)$ received from a majority of processes );
    let $max\_sn$ be the greatest sequence number $(w\_sn)$ received in
                an ACK_READ_REQ $(req\_sn_i, w\_sn, -)$ message;
    let $v$ be such that ACK_READ_REQ $(req\_sn_i, max\_sn, v)$ has been received;
    return $(v)$.

when WRITE $(val, w\_sn)$ is received from $p_w$:
    if $(w\_sn \geq w\_sn_i)$ then $cur\_val_i \leftarrow val$; $w\_sn_i \leftarrow w\_sn$ end if;
    send ACK_WRITE $(w\_sn)$ to $p_w$.

when READ_REQ $(r\_sn)$ is received from $p_j$ $(j \in \{1, \ldots, n\})$:
    send ACK_READ_REQ $(r\_sn, w\_sn_i, cur\_val_i)$ to $p_j$.
```

Figure 2.1: An algorithm that constructs a regular 1WMR register in $\mathcal{AS}_{n,t}[t < n/2]$

An operation thus takes 2 time units (let us remember that the communication graph is complete, i.e., each pair of processes is connected by an independent bidirectional channel). Hence, the time complexity (number of sequential communication steps) does not depend on n.

When the communication graph is not complete The algorithm described in Figure 2.1 is based on the assumption that the underlying communication graph is completely connected: any pair of processes is connected by a reliable channel.

So, an interesting question is "What does happen when this is not the case?" It is easy to see that the algorithm can be easily modified in order to work in all the runs for which the communication graph connecting the non-faulty processes remains strongly connected (i.e., any pair of non-faulty processes is connected by a path of non-faulty processes). The only modification of the algorithm consists in adding an appropriate routing for the messages. In that case, both the message complexity and the time complexity depend on the communication graph.

Theorem 2.1 *The algorithm described in Figure 2.1 constructs a 1WMR regular register in $\mathcal{AS}_{n,t}[t < n/2]$ (asynchronous message-passing system prone to up to t crashes with $t < n/2$).*

Proof

Proof of the liveness property. We have to prove here that any operation invoked by a non-faulty process terminates. Let us notice that the only statement where a process can block forever is a **wait until** statement. The fact that no process blocks forever in such a statement follows directly from the four following observations: (1) a process broadcasts an inquiry message (identified with a sequence number) before waiting for acknowledgments from a majority of processes, (2) every inquiry message is systematically answered by every non-faulty process, (3) there is a majority of non-faulty processes, and (4) the channels are reliable.

Proof of the safety property. Let us first observe that, as there is a single writer, write operations are totally ordered. Moreover, every write operation is identified with a sequence number.

To prove the safety property that defines a regular register, we have to prove that, when a process p_i invokes R.read (), it obtains either the last value written before the read operation was invoked, or a value that is written by a concurrent write operation.

Let rsn be the sequence number associated with the value returned by p_i. Let $x \geq 0$ be the sequence number of the last value written before the operation R.read () is invoked, and $x + 1$, ..., $x + y$ be the sequence numbers of the write operations, if any, that are concurrent with R.read (). ($y = 0$ corresponds to the case where there is no write concurrent with the read.) The proof consists in showing that $rsn \in \{x, \ldots, x + y\}$.

As the write of the value associated with x is terminated, it follows from the algorithm that (at least) a majority of processes p_k are such that $w\_sn_k \geq x$. As the R.read () obtains sequence numbers from a majority of processes, it obtains at least one sequence number $\geq x$, and we consequently have $rsn \geq x$.

On the other hand, due to its definition, the read operation is not concurrent with write operations whose sequence numbers are greater than $x + y$, which means that the read operation has terminated when the write numbered $x + y + 1$ is issued by the writer (if such a write is ever issued). Consequently, we have $rsn \leq x + y$, which concludes the proof of the safety property. $\Box_{Theorem}$ 2.1

When the writer crashes If the writer crashes outside the write operation, the processes will obtain the last value it has written. The case where it crashed while executing the write operation is more "interesting". It is possible that it crashes after sending its new value to less than a majority of processes. In that case, depending on asynchrony and the actual crash pattern, it is possible that, when some processes read, they will always obtain the new value, while others always obtain the previous value. This is not in contradiction with the definition of a regular register. Actually, if the writer process crashes during a write operation, that operation may never terminate (it is then concurrent with all the future read operations).

It is easy to see that the crash of a process during a read operation has no effect on the behavior of other processes. This is because a read operation does not entail modifications on local variables of other processes.

2.2.3 FROM REGULARITY TO ATOMICITY

The previous algorithm does not ensure atomicity Let us consider the scenario described in Figure 2.2. There are 5 processes, and none of them has crashed. The numbers on a process axis are sequence numbers. The bold line (cutting the axes of all the processes) is the "write line" associated with the write of the value with sequence number 15. As an example, let us consider the process p_i: before the cut by the write line, cur_val_i contains the value whose sequence number is 14, and contains the value whose sequence number is 15 just after. As far as p_j is concerned, this process receives the message WRITE(−, 15) before the ones carrying the sequence numbers 11 to 14. Due to asynchrony these messages are late (they have been bypassed by the message WRITE(−, 15)) and will be discarded by p_j when they arrive. Let us remember that the channels are reliable but are not required to be "first in, first out".

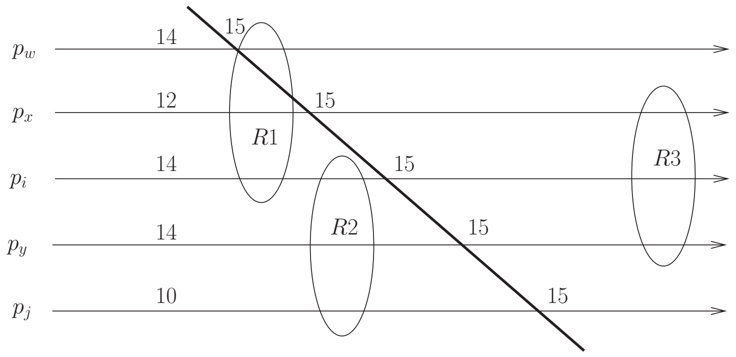

Figure 2.2: Regular is not atomic

An ellipsis corresponds to a read operation, so there are three reads denoted $R1$, $R2$ and $R3$. Let us assume that the read $R1$ is issued by p_i. It obtains the values and the sequence numbers of the set of the three processes p_w, p_x and itself, that constitutes a majority. It follows that $R1$ returns the value whose associated sequence number is 15. If we consider $R3$, it is easy to see that it returns the value whose sequence number is 15. Let us now consider $R2$. It obtains the sequence numbers 14, 14 and 10, and consequently returns the value associated with the sequence number 14.

When we look at Figure 2.2 from an operation duration point of view, we see that, while $R1$ terminates before $R2$ starts, it obtains the new value while $R2$ obtains the old value. There is a new/old inversion. Consequently, the algorithm described in Figure 2.1 does not ensure the atomicity consistency criterion.

The key to obtain atomicity: force a read to write An easy way to enrich the previous algorithm to obtain an algorithm that guarantees atomicity consists in preventing new/old inversions. This can be easily realized by forcing a read operation to write the value v it is about to return. This ensures that, when the read terminates, a majority of the processes have a value as recent as v in their local memory. The parts of the algorithm described in Figure 2.1 that are modified to go from regularity to atomicity are described in Figure 2.3.

Thanks to this embedded write of the read value, if the invoking process p_i does not crash while executing the read, a majority of the processes will have a value with a sequence number greater than or equal to sn, where sn is the sequence number of the value it is about to return. It is easy to see that this prevents new/old inversions from occurring. If p_i crashes before returning from the read operation, the WRITE() message it has sent to p_j (if any) is taken into account by p_j only if it carries a value not older than the one kept in cur_val_j. It follows that a process that crashes during a read cannot create inconsistency. Its only possible effect is to refresh the content of local variables with more up to date values.

Finally, as now not only the writer but any process can send WRITE() messages, the processing of these messages has to be slightly modified: the ACK_WRITE_REQ() message has to be sent back to the sender of the WRITE() message.

```
operation $R$.read ():
    $req\_sn_i \leftarrow req\_sn_i + 1$;
    broadcast READ_REQ ($req\_sn_i$);
    wait until ( ACK_READ_REQ ($req\_sn_i, -, -$) received from a majority of processes );
    let $max\_sn$ be the greatest sequence number ($w\_sn$) received in
                    an ACK_READ_REQ ($req\_sn_i, w\_sn, -$) message;
    let $v$ be such that ACK_READ_REQ ($max\_sn, v$) has been received;
    broadcast WRITE($v, max\_sn$);
    wait until ( ACK_WRITE ($max\_sn$) received from a majority of processes );
    return ($v$).

when WRITE ($val, w\_sn$) is received from $p_j$ ($j \in \{1, \ldots, n\}$):
    if ($w\_sn \geq w\_sn_i$) then $cur\_val_i \leftarrow val$; $w\_sn_i \leftarrow w\_sn$ end if;
    send ACK_WRITE ($w\_sn$) to $p_j$.
```

Figure 2.3: Modifying the R.read () operation to obtain atomicity

2.2.4 FROM A 1WMR ATOMIC REGISTER TO AN MWMR ATOMIC REGISTER

Replacing sequence numbers by timestamps To go from a 1WMR atomic register to an MWMR atomic register, the new problem to solve is to allow the processes to share a single sequence number generator for the values they write into R. A simple way to do it is to use the set of local variables $\{w\_sn_i\}_{1 \leq i \leq n}$ as follows.

When a process p_i wants to write, it broadcasts a message WRITE_REQ(req_sn_i) in order to obtain the current sequence numbers w_sn_j from a majority of processes. It then adds 1 to the maximal value it has received and associates this new sequence number with the value v it wants to write. Let us observe that the same local variable req_sn_i is used now to associate an identity to both its write requests and its read requests.

Of course, this does not prevent several processes from associating the same sequence number with their writes. (Let us notice that, when this occurs, the corresponding writes are concurrent.) This can easily be solved, by associating a timestamp (instead of a "unidimensional" sequence number) with each write operation.

A *timestamp* is a pair (sequence number, process identity). The two fundamental properties of timestamps are that they evolve with the progress of the processes and define a total order consistent with this progress. Let $(sn1, i)$ and $(sn2, j)$ be two timestamps. The timestamp total order is defined as follows (lexicographical ordering):

$$(sn1, i) < (sn2, j) \equiv \big((sn1 < sn2) \vee (sn1 = sn2 \wedge i < j)\big).$$

The final construction The algorithm building an MWMR atomic register in $\mathcal{AS}_{n,t}[t < n/2]$ is described in Figure 2.4. Each process manages a new local variable $last\_writer_i$ that contains the identity of the process that issued the write of the value currently kept in cur_val_i ($last\_writer_i$ can be initialized to any process identity, e.g., 1).

The timestamp of the value in cur_val_i is consequently the pair $(w\_sn_i, last\_writer_i)$. The code associated with the reception of a WRITE(val, w_sn) message takes now into account the timestamp of the value that is about to be written, instead of only its sequence number.

Let us also notice that now, not only the read/write request messages and their acknowledgments are tagged with a request sequence number defined by the requesting process, but the write messages also are tagged the same way. This allows for an unambiguous identification of the write acknowledgments sent to a writer.

On two-phase algorithms The algorithms implementing the R.write () and R.read () operations have exactly the same structure: they first broadcast a request to obtain the more recent control information, do local computation, and finally issue a second broadcast to write a value.

This structure is encountered in a lot of distributed algorithms called *distributed two-phase algorithms*. These phases refer to communication. The first phase consists in acquiring information on the system state, while (according to the information obtained and some local computation) the second phase consists in updating the system state.

2.2.5 CORRECTNESS PROOF OF THE MWMR ATOMIC REGISTER CONSTRUCTION

Lemma 2.2 *The execution of an* R.write () *or* R.read () *operation by a non-faulty process always terminates.*

Proof The reasoning is exactly the same as the one stated in the proof of Theorem 2.1 where only the case of the 1WMR regular register was considered. We repeat it here only to make the reading

```
operation $R$.write ($v$):
    $req\_sn_i \leftarrow req\_sn_i + 1$;
    % Phase 1: acquire information on the system state %
    broadcast WRITE_REQ ($req\_sn_i$);
    wait until ( ACK_WRITE_REQ ($req\_sn_i, -$) received from a majority of processes );
    let $max\_sn$ be the greatest sequence number ($w\_sn$) received in
                an ACK_WRITE_REQ ($req\_sn_i, w\_sn$) message;
    % Phase 2 : update system state %
    broadcast WRITE ($req\_sn_i, v, max\_sn + 1, i$);
    wait until ( ACK_WRITE ($req\_sn_i$) received from a majority of processes );
    return ($ok$).

operation $R$.read ():
    $req\_sn_i \leftarrow req\_sn_i + 1$;
    % Phase 1: acquire information on the system state %
    broadcast READ_REQ ($req\_sn_i$);
    wait until ( ACK_READ_REQ ($req\_sn_i, -, -, -$) received from a majority of processes );
    let ($max\_wsn$,$max\_lw$) be the greatest timestamp ($w\_sn, last\_writer$) received in
                an ACK_READ_REQ ($req\_sn_i, w\_sn, last\_writer, -$) message;
    let $v$ be such that ACK_READ_REQ ($req\_sn_i, max\_wsn, max\_lw, v$) has been received;
    % Phase 2 : update system state %
    broadcast WRITE ($req\_sn_i, v, max\_wsn, max\_lw$);
    wait until ( ACK_WRITE ($req\_sn_i$) received from a majority of processes );
    return ($v$).

when WRITE ($r\_sn, val, w\_sn, last\_writer$) is received from $p_j$ ($j \in \{1, \ldots, n\}$):
    if($w\_sn, last\_writer$) $\geq$ ($w\_sn_i, last\_writer_i$)
        then $cur\_val_i \leftarrow val$; $w\_sn_i \leftarrow w\_sn$; $last\_writer_i \leftarrow last\_writer$ end if;
    send ACK_WRITE($r\_sn$) to $p_j$.

when READ_REQ ($r\_sn$) is received from $p_j$ ($j \in \{1, \ldots, n\}$):
    send ACK_READ_REQ ($r\_sn, w\_sn_i, last\_writer_i, cur\_val_i$) to $p_j$.

when WRITE_REQ ($r\_sn$) is received from $p_j$ ($j \in \{1, \ldots, n\}$):
    send ACK_WRITE_REQ($r\_sn, w\_sn_i$) to $p_j$.
```

Figure 2.4: An algorithm that constructs an atomic MWMR register in $\mathcal{AS}_{n,t}[t < n/2]$ (code for any p_i)

easier. The fact that no process blocks forever in a **wait until** statement follows directly from the four following observations: (1) a process broadcasts an inquiry message (identified with a proper sequence number) before waiting for acknowledgments from a majority of processes, (2) every inquiry message is systematically answered by every non-faulty process, (3) there is a majority of non-faulty processes, and (4) the channels are reliable. $\square_{Lemma\ 2.2}$

Definition 2.3 An *effective write* operation is such that at least one process has received its WRITE () message. An *effective read* operation is such that the invoking process does not crash while executing it.

The timestamp of an effective R.write () operation is the timestamp it associates with the value it writes. The timestamp of an effective R.read () operation is the timestamp associated with the value it returns.

An effective write is a write whose value is taken into account by at least one process. Let us observe that all write operations issued by non-faulty processes are effective. On the other hand, some of write operations whose invoking process crashes during the invocation are effective, while others are not.

Lemma 2.4 *Let* w1 *and* w2 *be two effective write operations time stamped* $(sn1, id1)$ *and* $(sn2, id2)$, *respectively.* w1 $\neq$ w2 $\Rightarrow$ $(sn1, id1) \neq (sn2, id2)$,

Proof Let us first observe that if w1 and w2 are issued by different processes, the second field of their timestamps are different, and the lemma follows. So, let us consider the case where w1 and w2 are issued by the same process p_i. Let $(sn1, i)$ the timestamp of w1 and $(sn2, i)$ the timestamp of w2.

Without loss of generality, let us assume that w1 is executed first. As p_i is sequential, it follows that w1 was terminated when p_i issues w2, from which we conclude that a majority of processes p_j are such that $(w\_sn_j, last\_writer_j) \geq (sn1, i)$ when w1 terminates.

Let us now consider the first phase of w2. During this phase, p_i collect sequence numbers from a majority of processes. As any two majorities intersect, it follows that at least one of these sequence numbers is greater than or equal to $sn1$. Finally, the lemma follows from the fact that $sn2$ is set to a value greater than the greatest sequence number received. $\square_{Lemma\ 2.4}$

Lemma 2.5 *Let* op1 *and* op2 *be two effective operations time stamped* $(sn1, id1)$ *and* $(sn2, id2)$, *respectively, such that* op1 *terminates before* op2 *starts. We have the following:*
If op1 *is a read or a write operation and* op2 *is a read operation, then* $(sn1, id1) \leq (sn2, id2)$.
If op1 *is a read or a write operation and* op2 *is a write operation, then* $(sn1, id1) < (sn2, id2)$.

Proof The proof of this lemma uses Lemma 2.4 and is similar to it. The only difference is that, while a write operation increases a timestamp value, a read operation does not. A development of a complete proof is left to the reader as an exercise. $\square_{Lemma\ 2.5}$

Lemma 2.6 *There is a total order* $\widehat{S}$ *on all the effective operations* (i) *that respects their real time occurrence order, and* (ii) *is such that any read operation obtains the value written by the last write operation that precedes it in* $\widehat{S}$.

Let us remember that the notion of "time occurrence order" has been defined in the previous chapter. An operation op1 precedes an operation op2 if the response event of op1 appears before the invocation event of `op2` in the event history $\widehat{H} = (H, <_H)$ that models the execution.

Proof Let us consider the total order $\widehat{S}$ on all the effective operations defined as follows. The operations are first ordered in $\widehat{S}$ according to their timestamps. As all the write operations are totally ordered by their timestamps (Lemma 2.4), it follows that, if two operations have the same timestamps, one of them is necessarily a read operation. If a read and a write have the same timestamps, the write is ordered in $\widehat{S}$ before the read. If two reads have the same timestamp, the one that starts first, is ordered in $\widehat{S}$ before the other one. (The first is the one whose invocation event appears first in the associated history $\widehat{H}$).

The total order $\widehat{S}$ being defined, we are now able to show that it is a witness sequence (or linearization) of the execution.

- Proof of property (i). Let `op1` (time stamped $(sn1, id1)$) and `op2` (time stamped $(sn2, id2)$) be effective read or write operations such that `op1` terminates before `op2` starts. Due to Lemma 2.5, we have $(sn1, id1) \leq (sn2, id2)$ if `op2` is a read operation, and $(sn1, id1) < (sn2, id2)$ if `op2` is a write operation. We conclude from the way $\widehat{S}$ is built (from both the order on the operations defined by their timestamps and the order on the response/invocation events), that `op1` is ordered before `op2` in $\widehat{S}$.
- Proof of property (ii). Let `read` be a read operation that returns a value v time stamped (sn, j). We conclude that v has been written by p_j after it has computed the write sequence sn. The fact that `read` obtains the value of the last preceding write in $\widehat{S}$ follows directly from the way $\widehat{S}$ is built and the fact that no two written values have the same timestamp (Lemma 2.4).

$\square_{Lemma\ 2.6}$

Theorem 2.7 *The algorithm described in Figure 2.4 constructs an atomic MWMR register in* $\mathcal{AS}_{n,t}[t < n/2]$.

Proof The proof follows from Lemma 2.2 (liveness) and Lemma 2.6 (safety). $\square_{Theorem\ 2.7}$

2.3 AN IMPOSSIBILITY THEOREM

The previous constructions of a 1WMR and a MWMR atomic registers are t-resilient (i.e., they cope with up to t process crashes) where $t < n/2$. So, a natural question that comes immediately to mind is the following one: "Is it possible to design atomic register algorithms for any value of t, or is $t < n/2$ a fundamental barrier that cannot be bypassed when one has to cope with the net effect of asynchrony and process failures?"

This section answers the previous fundamental question by showing that it is impossible to design a distributed algorithm that builds a register in $\mathcal{AS}_{n,t}[\emptyset]$. Interestingly, this proof is based on an *indistinguishability argument* that is common to several impossibility results, namely the fact that some processes cannot distinguish one execution from another one. In that sense, although it is very simple, this proof depicts an essential feature that lies at the core of fault-tolerant distributed computing.

Theorem 2.8 *There is no algorithm that constructs an atomic register in $\mathcal{AS}_{n,t}[t \geq n/2]$.*

Proof Given $t \geq n/2$, let us partition the processes in two subsets $P1$ and $P2$ (i.e., $P1 \cap P2 = \emptyset$ and $P1 \cup P2 = \{p_1, \ldots, p_n\}$) such that $|P1| = \lceil n/2 \rceil$ and $|P2| = \lfloor n/2 \rfloor$. Let us observe that $|P1| \leq t$ and $|P2| \leq t$, which means that the system model accepts executions in which all the processes of $P1$ crash, and executions in which all the processes of $P2$ crash.

The proof is by contradiction. Let us assume that there is an algorithm A that builds an atomic register R for $t \geq n/2$. Let 0 be the initial value of R. Let us define the following executions (depicted in Figure 2.5 where $n = 5$ and $t = 3$). Let us repeat that, according to the system model and the previous assumptions, these executions can happen.

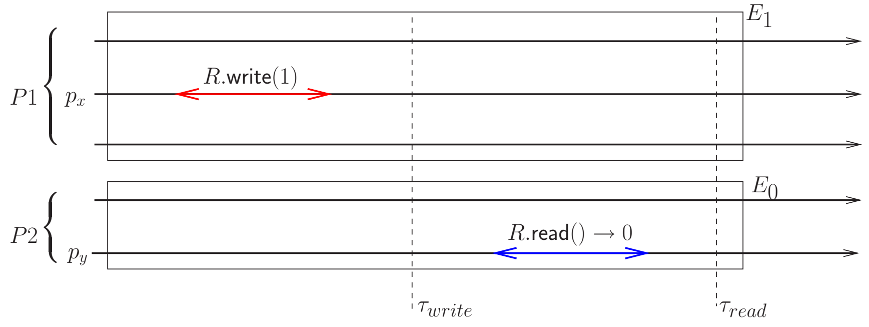

Figure 2.5: There is no register algorithm in $\mathcal{AS}_{n,t}[\emptyset]$

- Execution E_1. In this execution, all the processes of $P2$ crash initially (so no process of $P2$ ever executes a step in E_1), and all the processes in $P1$ are non-faulty. Moreover, a process $p_x \in P1$ issues R.write (1), and no other process of $P1$ invokes an operation. As the algorithm A is correct (assumption), it satisfies the liveness property and, consequently, this write operation does terminate. Let τ_{write} be a (finite) time after it has terminated.
- Execution E_0. In this execution, all the processes of $P1$ crash initially, the processes of $P2$ are non-faulty and do nothing until τ_{write}. Let us observe that, due to asynchrony, this is possible. After τ_{write}, a process $p_y \in P2$ issues R.read (), and no other process of $P2$ invokes an operation. As the algorithm A is correct, this read operation terminates and returns the initial value 0 to p_y. Let τ_{read} be a (finite) time after which this read operation has terminated. (In the following "the same as" means that in both executions, the processes issues the same operations (and receive the same results) at the very same time.)
- Execution E_{10}. That execution is defined as follows.

- No process crashes.
- E_{10} is the same as E_1 until τ_{write}.
- E_{10} is the same as E_0 until the time τ_{read}.
- If any, the messages that the processes of $P1$ send to the processes of $P2$ are delayed to be received after time τ_{read}. Similarly, if any, the messages that the processes of $P2$ send to the processes of $P1$ are delayed to be received after time τ_{read}. (Let us remember that, due the system asynchrony, messages can be delayed in that way.)

Let us consider the process $p_y \in P2$. This process cannot distinguish between E_0 and E_{10} until τ_{read}. Hence, as it reads 0 in E_0, it has to read the same value in E_{10}. But, as the algorithm A ensures atomicity, p_y should read 1 in E_{10} (the last write that precedes the read operation has written the value 1). We obtain a contradiction, from which we conclude that there is no algorithm A with the required properties. $\square_{Theorem\ 2.8}$

2.4 THE FAILURE DETECTOR APPROACH TO CIRCUMVENT IMPOSSIBILITIES

2.4.1 WHAT A FAILURE DETECTOR IS

The concept of *failure detector* is one of the main approaches that has been proposed to circumvent impossibility results in fault-tolerant asynchronous distributed computing (i.e., in the system model $\mathcal{AS}_{n,t}[\emptyset]$). From an operational point of view, a failure detector can be seen as an oracle made up of several modules, each associated with a process. The module attached to process p_i provides it with hints concerning which processes have failed. Failure detectors are divided into classes, based on the particular type of information they provide on failures. Different problems may require different classes of failure detectors in order to be solved in an otherwise fault-prone asynchronous distributed system. One can identify two main characteristics of the failure detector approach, namely, one associated with its software engineering feature, the other associated with its computability dimension.

The software engineering dimension of failure detectors A failure detector class is defined by a set of abstract properties. This way, a failure detector-based distributed algorithm relies only on the properties that define the failure detector class, regardless of the way they are implemented in a given system. This *software engineering dimension* of the failure detector approach favors algorithm design, algorithm proof, modularity, and portability.

Similarly to a stack and a queue that are defined by their specification, and can have many different implementations, a failure detector of a given class can have many different implementations each taking into account appropriate features of a particular underlying system (such as its topology, local clocks, distribution of message delays, timers, etc.). Due to the fact that a failure detector is defined by abstract properties and not in terms of a particular implementation, an algorithm that uses it does not have to be rewritten when the underlying system is modified.

It is important to notice that, in order for a non-trivial failure detector to be implementable, the underlying system has to satisfy additional behavioral properties (that in some sense restrict its asynchrony). (If not, the impossibility result -that the considered failure detector allows to circumvent- would no longer be impossible.)

Let Pb be a problem that can be solved with the help of a failure detector class *FD*. (With the previous notation, this means that P can be solved in $\mathcal{AS}_{n,t}[FD]$.) The failure detector concept favors separation of concerns as follows.

- Design and prove correct an *FD*-based distributed algorithm that solves Pb.
- Independently of the previous item, investigate the system behavioral properties that have to be satisfied for *FD* to be implementable, and provide an implementation of *FD* for these systems.

The computability dimension of failure detectors Given a problem Pb that cannot be solved in an asynchronous system prone to failures (e.g., the construction of an atomic register in $\mathcal{AS}_{n,t}[\emptyset]$), the failure detector approach allows us to investigate and state the minimal assumptions on failures the processes have to be provided with, in order for the problem Pb to be solved. This is the *computability dimension* of the failure detector approach.

An interesting side of this computability dimension lies in the ranking of problems it allows. This ranking is based on the weakest failure detectors these problems require to be solved. (The notion of "weakest" failure detector for the register problem will be discussed in Section 2.5, and addressed in a general way in the last chapter). This provides us with a failure detector-based method to establish a hierarchy among distributed computing problems.

2.4.2 FORMAL DEFINITIONS

Failure pattern A failure pattern defines a possible set of failures, with their occurrence times, that can occur during an execution. Formally, a failure pattern is a function $F : \mathbb{N} \rightarrow 2^{\Pi}$, where $\mathbb{N}$ is the set of natural numbers (time domain), and 2^{Π} is the power-set of Π (the set of all sets of process identities). $F(\tau)$ denotes the set of processes that have crashed up to time τ. As a crashed process does not recover, we have $F(\tau) \subseteq F(\tau + 1)$. Let $Faulty(F)$ be set of processes that crash in an execution with failure pattern F. Let τ_{max} denote the end of that execution. We have $Faulty(F) = F(\tau_{max})$. As τ_{max} is not known and depends on the execution, and we want to be as general as possible (and not tied to a time-specific class of executions), we (conceptually) consider that an execution never ends, i.e., we consider that $\tau_{max} = +\infty$. We have accordingly $Faulty(F) = \cup_{1 \leq \tau < +\infty} F(\tau) = \lim_{\tau \rightarrow +\infty} F(\tau)$. Let $Non\text{-}faulty(F) = \Pi - Faulty(F)$ (the set of processes that do not crash in F). $Correct(F)$ is used as a synonym of $Non\text{-}faulty(F)$.

It is important to notice that the notion of faulty and correct are defined with respect to a failure pattern, i.e., to the failure pattern that occurs in a given execution.

Failure detector history with range $\mathcal{R}$ A *failure detector history with range* $\mathcal{R}$ describes the behavior of a failure detector during an execution. $\mathcal{R}$ defines the type of information on failures provided to the processes. We consider here failure detectors whose range $\mathcal{R}$ is 2^{Π} (i.e., a value of $\mathcal{R}$ is a set of process identities). This means that the information a process obtains from its failure detector module is always a set of processes.

Formally, a failure detector history is a function $H : \Pi \times \mathbb{N} \rightarrow \mathcal{R}$, where $H(p_i, \tau)$ is the value of the failure detector module of process p_i at time τ.

Failure detector class FD with range $\mathcal{R}$ A *failure detector class FD with range* $\mathcal{R}$ is a function that maps each failure pattern F to a set of failure detector histories with range $\mathcal{R}$. This means that $FD(F)$ represents the whole set of possible behaviors that the failure detector FD can exhibit when the actual failure pattern is F.

In the following, we sometimes say "a failure detector *FD*" as a shortcut for "a failure detector of the class *FD*".

Remark It is important to notice that the output of a failure detector does not depend on the computation, it depends only on the actual failure pattern. On another hand, a given failure detector might associate several histories with each failure pattern. Each history represents a possible sequence of outputs for the same failure pattern. This feature captures the inherent non-determinism of a failure detector.

Remark The failure detector classes presented in this book do not appear in their "historical order" (the order in which they have been chronologically introduced in research articles). They are introduced according to the order in which this book presents problems they allows us to solve. Hence, as it allows for an optimal implementation of a register whatever the number of process that may crash, the first class of failure detector that is presented is the class Σ. (The optimality notion cited above is with respect to the information on failure that is needed to solve a given problem.)

2.4.3 THE CLASS Σ OF QUORUM FAILURE DETECTORS

Definition A quorum is a non-empty set of processes. The class of *quorum failure detectors*, that is denoted Σ, contains all the failure detectors that provide each process p_i with a quorum local variable, denoted $sigma_i$, that p_i can only read, and such that the set of local variables $\{sigma_i\}_{1 \le n}$ collectively satisfy the intersection and liveness properties stated below.

Let us denote $sigma_i^{\tau}$ the output of Σ at process p_i at time τ (using the formalism introduced in the previous section we have $sigma_i^{\tau} = H(p_i, \tau)$).

- Intersection. $\forall i, j \in \{1, \ldots, n\}$: $\forall \tau, \tau' \in \mathbb{N}$: $sigma_i^{\tau} \cap sigma_j^{\tau'} \neq \emptyset$.
- Liveness. $\exists \tau \in \mathbb{N}$: $\forall \tau' \geq \tau$: $\forall i \in Correct(F)$: $sigma_i^{\tau'} \subseteq Correct(F)$.

The intersection property states that any two quorum values do intersect, whatever the times at which they are output. As it has to be always satisfied, this property in called a *perpetual* property: it is an invariant provided by Σ. A Σ-based algorithm that aims to construct an atomic register can rely on this invariant to prevent partitioning (and consequently prevents occurrence of the bad scenario described in the proof of Theorem 2.8), and thereby guarantee the required atomicity (safety) property of a register.

The second property states that, after some finite time, the quorum values output at any non-faulty process contain only non-faulty processes. These processes are not required to be the same forever. They can change as long as the intersection property remains satisfied. This property is called an *eventual* property: it states that, after some finite time, "something" has to be forever satisfied. Its aim is to allow a Σ-based algorithm to guarantee that the read and write operations issued by the non-faulty processes always terminate.

***Implementing* Σ *when* $t < n/2$** There is a very simple algorithm that builds a failure detector of the class Σ in $\mathcal{AS}_{n,t}[t < n/2]$ (Figure 2.6). Each process p_i manages a queue (denoted $queue_i$) that contains the n process identities. The initial value is any permutation of these identities. Each process broadcasts forever (i.e., until it crashes if it ever crashes) ALIVE () messages to indicate it has not crashed. When a process p_i receives such a message from a process p_j, it moves j in $queue_i$ from its current position to the head of $queue_i$. Finally, it defines the current value of $sigma_i$ as the majority of the processes that are at the head of $queue_i$.

```
background task: repeat forever broadcast ALIVE () end repeat.

when ALIVE () is received from $p_j$ ($j \in \{1, \dots, n\}$:
        suppress $j$ from $queue_i$; add $j$ at the head of $queue_i$;
        $sigma_i \leftarrow$ the $\lceil \frac{n+1}{2} \rceil$ processes at the head of $queue_i$.
```

Figure 2.6: Building a failure detector of the class Σ in $\mathcal{AS}_{n,t}[t < n/2]$

The intersection property trivially follows from the fact that any two majorities do intersect. As far as the liveness property is concerned, let c be the number of correct processes. We have $c > n/2$, i.e., $c \geq \lceil \frac{n+1}{2} \rceil$. Let us observe that, after some time, only the c non-faulty processes send messages, and consequently, only these processes will appear in the first c positions of the queue of any non-faulty process. The liveness follows immediately from $c \geq \lceil \frac{n+1}{2} \rceil$.

Remark As we have seen, it is possible to build an atomic register in $\mathcal{AS}_{n,t}[t < n/2]$, and as we are about to see, it is also possible to build an atomic register in $\mathcal{AS}_{n,t}[\Sigma]$. Hence, it is not counter-intuitive that a failure detector of the class Σ can be built in $\mathcal{AS}_{n,t}[t < n/2]$.

On another hand, thanks to Theorem 2.8, and the fact that Σ allows the construction of an atomic register, we can conclude that it is not possible to build a failure detector of the class Σ in $\mathcal{AS}_{n,t}[\emptyset]$. Such a construction requires additional assumptions that the underlying system has to satisfy.

The fundamental added value supplied by a failure detector The fundamental added value, when considering Σ with respect to the assumption $t < n/2$, is that we *do know* which is the weakest information on failures the processes have to be provided with in order to build an atomic register. "$t < n/2$" is a model assumption, it is not the weakest information on failures that allows the construction of an atomic register.

2.4.4 A Σ-BASED ALGORITHM THAT BUILDS A 1W1R ATOMIC REGISTER

This section presents a Σ-based algorithm that builds a 1W1R atomic register R (i.e., it builds a register in the system model $\mathcal{AS}_{n,t}[\Sigma]$). The algorithm appears in Figure 2.7. Extending this algorithm to build an MWMR atomic register is straightforward. It can be easily done using an incremental construction similar to the one described in Section 2.2.

One writer, one reader, but the other processes have to participate The writer is denoted p_w, while the reader is denoted p_r. It is important to notice that all the processes have to participate in the algorithm. If only p_w and p_r were used to implement the register R, the operation R.write() could block forever if p_r crashes while p_w does not (p_w waiting forever for a message from p_r). And similarly, R.read() could block forever if p_w crashes while p_r does not. This would violate the liveness property of the write (or the read) operation. Each process p_i has consequently to manage a local copy cur_val_i of R, as in the previous algorithms.

The algorithm The code of the algorithm is very close to the previous ones. The local variables have the same meaning as in previous algorithms, and the basic structure is also the same. There are only two differences.

- The first is the use of a quorum failure detector of the class Σ instead of the majority of non-faulty processes assumption. As the value of the quorum failure detector module $sigma_i$ can change forever, its use is encapsulated in a **repeat** statement to express the fact that a process has to wait for messages until it has received a message from each process of a quorum output by Σ. The messages a process is waiting for can be received in any order, and at different time instants.

- The second difference is not related to the use of Σ, but to the fact that there is a single reader. As p_r is the only reader, when it issues a R.read() invocation, it is not necessary for it to execute the second phase of the R.read() operation (write phase), whose aim is to ensure that the value kept in the local memories of the other processes is at least as recent as the value it is about to return (this was required in Figure 2.3 and Figure 2.4 to obtain a 1WMR atomic register and a MWMR atomic register, respectively). As no other process is allowed to read, it is sufficient that p_r keeps a local copy of the value it is about to return, in order to prevent new/old inversions. So, the second phase of a read operation required to guarantee atomicity is now simply a local write (that actually depends on the sequence number of the returned value).

```
% This code is only for the single writer $p_w$ %
operation $R$.write $(v)$:
        $w\_sn_w \leftarrow w\_sn_w + 1$;
        broadcast WRITE $(v, w\_sn_w)$;
        repeat $quorum_i \leftarrow sigma_i$
          until $(\forall j \in quorum_i$: ACK_WRITE $(w\_sn_w)$ received from $p_j)$ end repeat;
        return $(ok)$.
-----------------------------------------------------------------------------
% This code is only for the single reader $p_r$ %
operation $R$.read ():
        $req\_sn_i \leftarrow req\_sn_i + 1$;
        broadcast READ_REQ $(req\_sn_i)$;
        repeat $quorum_i \leftarrow sigma_i$
          until $(\forall j \in quorum_i$: ACK_READ_REQ $(req\_sn_i, -, -)$ received from $p_j)$ end repeat;
        let $max\_sn$ be the greatest sequence number $(w\_sn)$ received in
                      an ACK_READ_REQ $(req\_sn_i, w\_sn, -)$ message;
        if $(max\_sn > w\_sn_i)$ then
          $cur\_val_i \leftarrow v$ such that ACK_READ_REQ $(req\_sn_i, max\_sn, v)$ has been received;
          $w\_sn_i \leftarrow max\_sn$
        end if;
        return $(cur\_val_i)$.
-----------------------------------------------------------------------------
% The code snippets that follow are for every process $p_i, i \in \{1, \ldots, n\}$.

when WRITE $(val, w\_sn)$ is received from $p_w$:
        if $(w\_sn \geq w\_sn_i)$ then $cur\_val_i \leftarrow val$; $w\_sn_i \leftarrow w\_sn$ end if;
        send ACK_WRITE $(w\_sn)$ to $p_w$.

when READ_REQ $(r\_sn)$ is received from $p_r$:
        send ACK_READ_REQ $(r\_sn, w\_sn_i, cur\_val_i)$ to $p_r$.
```

Figure 2.7: An algorithm that constructs an atomic 1W1R atomic register in $\mathcal{AS}_{n,t}[\Sigma]$

The proof is a simplified version of the proof of the algorithm described in Figure 2.4 where the majority of correct processes assumption is replaced by the properties of Σ. It is consequently left to the reader as an exercise.

2.5 Σ IS THE WEAKEST FAILURE DETECTOR CLASS TO BUILD AN ATOMIC REGISTER

2.5.1 WHAT DOES “WEAKEST FAILURE DETECTOR CLASS” MEAN?

Notion of extraction algorithm The previous section has shown that it is possible to build an atomic register in $\mathcal{AS}_{n,t}[\Sigma]$. Said another way, Σ is sufficient to implement an atomic register in an asynchronous system prone to any number of process crashes. This section shows that, as soon as we rely on information on failures when we want to build a register, Σ is also necessary.

Let D be a failure detector class such that it is possible to build a register in $\mathcal{AS}_{n,t}[D]$. Intuitively, "necessary" means that the information on failures provided by D "includes" information on failures provided by Σ. More precisely, let D be any failure detector class such that it is possible to build an atomic register in $\mathcal{AS}_{n,t}[D]$, and A be any algorithm that builds a register in $\mathcal{AS}_{n,t}[D]$. Proving the necessity of Σ to build an atomic register consists in designing an algorithm that, given the previous D-based algorithm A as an input, builds a failure detector of the class Σ. We say that this algorithm *extracts* Σ from the D-based algorithm A (see Figure 2.8).

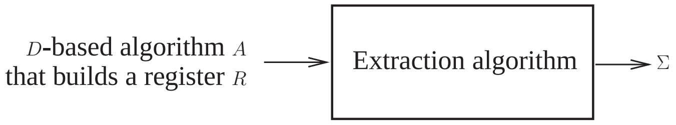

Figure 2.8: Extracting Σ from a register D-based algorithm A

Remark It is important to notice that the notion of "weakest" used here is related to information on failures only. Nothing prevents us from designing an oracle that does not provide processes with hints on failures but with another type of information (e.g., about the synchrony of the system) that would allow the construction of an atomic register despite any number of process crashes. "Weakest" means that any oracle that (1) provides processes only with information on failures (i.e., any failure detector class), and (2) allows processes to build an atomic register, allows the construction of a failure detector of class Σ.

2.5.2 THE EXTRACTION ALGORITHM

Aim As just indicated, the aim is to design an algorithm that emulates the output of Σ at each process p_i. This algorithm uses as a subroutine any algorithm A and failure detector D such that A is a n-process D-based algorithm that implements an atomic register in an n-process asynchronous message-passing system prone to any number of crashes.

An array of atomic registers Let Q be a non-empty set of processes, and $REG_Q[1..n]$ an array of n atomic registers (initialized to $[\bot, \ldots, \bot]$) such that each atomic register $REG_Q[x]$ is implemented by the n-process algorithm A executed only by $|Q|$ threads, each associated with a process of Q.

A simple register-based algorithm (task) Let WR_Q be the following register-based algorithm (also called a task) where each process p_i such that $i \in Q$ executes the following algorithm (where $reg_i[1..n]$ is an array local to p_i):

```
algorithm $WR_Q$:
  $REG_Q[i]$.write($\top$); for each $x \in \{1, \ldots, n\}$ do $reg_i[x] \leftarrow REG_Q[x]$.read() end for.
```

The process p_i first writes the value $\top$ in its entry of the array REG_Q, and then reads asynchronously all its entries. The $REG_Q[i]$.write($\top$) and $REG_Q[x]$.read() operations are provided to the processes by the previous algorithm A. (Let us notice that the value obtained by a read is irrelevant. As we will see, what is important is the fact that $REG_Q[x]$ has been written or not.) A corresponding run (history) of WR_Q is denoted E_Q. In that run, no process outside Q sends or receives messages related to the task WR_Q. When we consider the underlying failure detector-based algorithm A that implements the registers $REG_Q[1..n]$, as the processes that are not in Q do not participate in WR_Q, the messages sent by the processes of Q to these processes are never received, or are delayed for an arbitrarily long period. (Alternatively, we could say that, in WR_Q, the processes of Q "omit" sending messages to the processes that are not in Q.)

Let $\mathcal{C}$ denote the set of non-faulty processes in the run we consider. Let us observe that, as the underlying failure detector-based algorithm A that builds a register is correct, if the set Q contains all the correct processes (i.e., $\mathcal{C} \subseteq Q$), E_Q is such that every correct process terminates the task WR_Q. In the other cases, i.e., for the tasks WR_Q such that $\neg(\mathcal{C} \subseteq Q)$, E_Q is such that a process of Q either terminates WR_Q, or blocks forever, or crashes (this depends on the actual failure pattern, the outputs of the underlying failure detector D used by the algorithm A, and the code of A).

Running concurrently $2^n - 1$ tasks The extraction algorithm considers the $2^n - 1$ distinct tasks WR_Q where Q is a non-empty set such that $Q \in 2^{\Pi}$. To that end, each process p_i manages 2^{n-1} threads, one for each subset Q such that $i \in Q$. Let us notice that the crash of a process p_i entails the crash of all its threads.

The extraction algorithm The algorithm that extracts Σ is described in Figure 2.9. Let us recall that its aim is to provide each process p_i with a local variable $sigma_i$ such that the $(sigma_x)_{1 \leq x \leq n}$ variables satisfy the intersection and liveness properties defined in Section 2.4.3.

To that end, each process p_i manages two local variables: a set of sets of process identities, denoted $quorum\_sets_i$, and a queue denoted $queue_i$. The aim of $quorum\_sets_i$ is to contain all the sets Q such that p_i has terminated WR_Q (task $T1$), while $queue_i$ is managed in such a way that eventually any correct process appears in it before any faulty process (tasks $T2$ and $T3$).

The idea is to select an element of $quorum\_sets_i$ as the current output of $sigma_i$. As we will see in the proof, given any pair of processes p_i and p_j, any quorum in $quorum\_sets_i$ has a non-empty intersection with any quorum in $quorum\_sets_j$, thereby supplying the required intersection property.

The main issue is to ensure the liveness property of $sigma_i$ (eventually $sigma_i$ has to contain only correct processes) while preserving the intersection property. This is realized with the help of the local variable $queue_i$ as follows: the current output of $sigma_i$ is the set (quorum) of $quorum\_sets_i$ that appears as being the "first" in $queue_i$. The formal definition of "first element of $quorum\_sets_i$ with respect to $queue_i$" is stated in the task $T4$. To make it easy to understand, let us consider the following example. Let $quorum\_sets_i = \{\{3, 4, 9\}, \{2, 3, 8\}, \{1, 2, 4, 7\}\}$, and $queue_i =< 4, 8, 3, 2, 7, 5, 9, 1, \cdots >$. The set $S = \{2, 3, 8\}$ is the first set of $quorum\_sets_i$ with respect to $queue_i$ because each of the other sets $\{3, 4, 9\}$ and $\{1, 2, 4, 7\}$ includes an element (e.g., 9 and 7, respectively) that appears in $queue_i$ after the elements of S. (In case several sets are "first", any of them can be selected). The notion of "first quorum in $queue_i$" is used to ensure that Σ_i eventually includes only correct processes.

```
Init: $quorum\_sets_i \leftarrow \{\{1 \ldots, n\}\}$; $queue_i \leftarrow < 1, \ldots, n >$;
      for each $Q \in \big(2^{\Pi} \setminus \{\emptyset, \{1, \ldots, n\}\}\big)$ do
         if $(i \in Q)$ then launch a thread associated with the task $WR_Q$ end if end for.
      % Each process $p_i$ participates concurrently in all the tasks $WR_Q$ such that $i \in Q$ %

Task $T1$: when $p_i$ terminates task $WR_Q$: $quorum\_sets_i \leftarrow quorum\_sets_i \cup \{Q\}$.

Task $T2$: repeat periodically broadcast ALIVE$(i)$ end_repeat.

Task $T3$: when ALIVE $(j)$ is received: suppress $j$ from $queue_i$; enqueue $j$ at the head of $queue_i$.

Task $T4$: when $p_i$ reads $sigma_i$:
   let $m = \min_{Q \in quorum\_sets_i}(\max_{x \in Q}(rank[x]))$ where $rank[x]$ denotes the rank of $x$ in $queue_i$;
   return (a set $Q$ such that $\max_{x \in Q}(rank[x]) = m$).
```

Figure 2.9: Extracting Σ from a failure detector-based algorithm A that implements a register (code for p_i)

Remark Initially $quorum\_sets_i$ contains the set $\{1, \ldots, n\}$. As no set of processes is ever withdrawn from $quorum\_sets_i$ (task $T1$), $quorum\_sets_i$ is never empty. Moreover, it is not necessary to launch the task $WR_{\{1,\ldots,n\}}$ in which all processes participate. This is because, as the underlying failure detector-based algorithm A (that implements a register) is correct, it follows that each correct process decides in task $WR_{\{1,\ldots,n\}}$. This case is directly taken into account in the initialization of $quorum\_sets_i$ (thereby saving the execution of the task $WR_{\{1,\ldots,n\}}$).

2.5.3 PROOF OF CORRECTNESS

Theorem 2.9 *Let A be any failure detector-based algorithm that implements an atomic register in an asynchronous message-passing system prone to any number of process crashes. Given A, the algorithm described in Figure 2.9 is a bounded construction of a failure detector of the class* Σ.

Let us recall that a *bounded* construction is an algorithm whose all variables and messages have a bounded size.

Proof Proof of the intersection property. The proof is by contradiction. Let us first observe that the set $sigma_i$ returned to a process p_i is a set of $quorum\_set_i$ (that contains the set $\{1, \ldots, n\}$ -initial value- plus all the sets Q such that p_i has terminated WR_Q). Let us assume that there are two sets Q_1 and Q_2 such that (1) $Q_1, Q_2 \in \bigcup_{1 \le j \le n}(quorum\_set_j)$, and (2) $Q_1 \cap Q_2 = \emptyset$. The first item means that Q_1 and Q_2 can be returned to some processes as their local value for Σ.

Let p_i be a process that terminates WR_{Q_1} and p_j a process that terminates WR_{Q_2} (due to the "contradiction" assumption, such processes do exist). Using the fact that the message-passing system is asynchronous, let us construct the runs E_{Q_1} and E_{Q_2} associated with WR_{Q_1} and WR_{Q_2} as follows. If any, every message sent by any process in Q_1 to any process in Q_2 (when these processes execute A to implement a register of the array REG_{Q_1}) is delayed until p_i has added Q_1 to $quorum\_set_i$ and p_j has added Q_2 to $quorum\_set_j$. Let us delay similarly messages sent by processes in Q_2 to processes in Q_1 when they execute A for each register of the array REG_{Q_2}.

Let us observe that, in concurrent runs E_{Q_1} and E_{Q_2}, algorithm A, which is executed only by (1) processes of Q_1 in E_{Q_1} to build registers $REG_{Q_1}[1..n]$, and (2) processes of Q_2 in E_{Q_2} to build registers $REG_{Q_2}[1..n]$, is fed with the same outputs of the underlying failure detector D. Due to the fact that (if any) messages from Q_1 to Q_2 and from Q_2 to Q_1 are delayed, we have that p_i reads $\perp$ from $REG_{Q_1}[j]$ in E_{Q_1}, and p_j reads $\perp$ from $REG_{Q_2}[i]$ in E_{Q_2}.

Let us construct a run $E_{Q_{12}}$, where $Q_{12} = Q_1 \cup Q_2$, that is a simple merge of E_{Q_1} and E_{Q_2} defined as follows. In this run, algorithm A (that involves only the processes in Q_{12} and implements the array of registers $REG_{Q_{12}}[1..n]$) is fed with the same failure detector outputs as the ones supplied to the concurrent runs E_{Q_1} and E_{Q_2}. Moreover, messages from Q_1 to Q_2 and from Q_2 to Q_1 are delayed as in E_{Q_1} and E_{Q_2}. So, p_i (resp., p_j) receives the same messages and the same outputs from the underlying failure detector in $E_{Q_{12}}$ and E_{Q_1} (resp., E_{Q_2}).

- On the one hand, we have the following. As process p_i receives the same messages and the same failure detector outputs in $E_{Q_{12}}$ as in E_{Q_1}, arrays $REG_{Q_1}[1..n]$ and $REG_{Q_{12}}[1..n]$ contain the same values. Consequently, p_i reads $\perp$ from $REG_{Q_{12}}[j]$. Similarly, p_j reads $\perp$ from $REG_{Q_{12}}[i]$.
- On the other hand, we have the following. In $E_{Q_{12}}$, process p_i writes $\top$ into $REG_{Q_{12}}[i]$ and the process p_j writes $\top$ into $REG_{Q_{12}}[j]$. Moreover, one of these operations terminates before the other. Without loss of generality, let us assume that the write by p_i terminates before the write by p_j. Consequently, p_j reads $REG_{Q_{12}}[i]$ after it has been written. Due to the atomicity of that register, it follows that p_j obtains the value $\top$ when it reads $REG_{Q_{12}}[i]$.

The second item contradicts the first one. It follows that the initial assumption (namely, the existence of a failure detector-based algorithm A that builds a register, $Q_1, Q_2 \in \bigcup_{1\le j\le n}(quorum\_set_j)$ and $Q_1 \cap Q_2 = \emptyset$) is false, from which we conclude that at least one of the assertions $Q_1, Q_2 \in \bigcup_{1\le j\le n}(quorum\_set_j)$ and $Q_1 \cap Q_2 = \emptyset$ is false, which completes the proof of the intersection property (Corollary 2.10 -stated below- is an immediate consequence of that property).

Proof of the liveness property. As far as the liveness property is concerned, let us consider the task $WR_{\mathcal{C}}$ (recall that $\mathcal{C}$ is the set of correct processes). As the underlying failure detector-based algorithm A that implements the registers $REG_{\mathcal{C}}[1..n]$ is correct by assumption, each correct process p_i terminates its $REG_{\mathcal{C}}[i]$.write($\top$) and $REG_{\mathcal{C}}[x]$.read() operations in $E_{\mathcal{C}}$. Consequently, in the extraction algorithm, the variable $quorum\_set_i$ of each correct process p_i eventually contains the set $\mathcal{C}$.

Moreover, after some finite time, each correct process p_i receives ALIVE(j) messages only from correct processes. This means that, at each correct process p_i, every correct process eventually precedes every faulty process in $queue_i$. Due to the definition of "first set of $quorum\_set_i$ with respect to $queue_i$" stated in task $T4$, it follows that, from the time at which $\mathcal{C}$ has been added to $quorum\_set_i$, the quorum Q selected by the task $T4$ is always such that $Q \subseteq \mathcal{C}$, which proves the liveness property of $sigma_i$.

The construction is bounded. A simple examination of the extraction algorithm shows that (1) both the variables $queue_i$ and $quorum\_sets_i$ are bounded, and (2) messages carry bounded values, from which it follows that the construction is bounded. $\square_{Theorem\ 2.9}$

An additional property The proof of intersection property shows that it is not possible to have two sets Q_1 and Q_2 such that $Q_1 \cap Q_2 = \emptyset$ and at least one process of Q_1 terminates WR_{Q_1} and at least one process of Q_2 terminates WR_{Q_2}. Hence the following corollary.

Property 2.10 Let two sets Q_1 and Q_2 be such that $Q_1 \cap Q_2 = \emptyset$. Then, no process of Q_1 terminates WR_{Q_1} or no process of Q_2 terminates WR_{Q_2} (or both).

2.6 BIBLIOGRAPHIC NOTES

- As indicated in the previous chapter, the notions of regular and atomic registers were introduced by Lamport [110].
- The construction of an atomic register on top of an asynchronous message-passing system prone to process crashes has first been investigated by Attiya, Bar-Noy and Dolev in [15]. Other constructions are described in [14, 16, 117].
- Algorithms that build an atomic register in dynamic systems (i.e., systems where processes can enter and leave) are described in [2, 9, 40, 53, 74, 118]. The case of a regular register is addressed in [17, 153]. The case where registers are network attached disks is analyzed in [8].
- The notion of failure detectors was introduced by Chandra and Toueg in [34].

  Weakest failure detectors to solve several fundamental problems in distributed computing (such as consensus, non-blocking atomic commitment, quittable consensus) are presented in [48].

  It is shown in [103] that any non-trivial distributed computing problem has a weakest failure detector.

  Pedagogic presentations of this concept can be found in [85, 127, 148].
- The notion of quorum has been introduced by Gifford in [79] in the context of duplicated data management. General methods to define quorums can be found in [124, 142]. Quorums

suited to Byzantine failures (that are more severe than crash failures) have been studied by Malkhi and Reiter [120].

The class Σ of quorum failure detectors was introduced by Delporte-Gallet, Fauconnier and Guerraoui [47].

- The first proof that shows that Σ is the weakest class of failure detectors to build a register despite asynchrony and any number of process crashes was given by Delporte-Gallet, Fauconnier and Guerraoui [47, 48].

  The proof presented here is due to Bonnet and Raynal [30]. An extension of the class Σ, where the intersection property is no longer required to be perpetual, is presented in [71].

  A general method to extract quorum failure detectors is presented in [24].

- Lots of advanced algorithms that implement an atomic register in asynchronous message-passing systems prone to crash failures are presented in the literature. Those algorithms investigate mainly lower bounds and efficiency of the read and write operations. Examples of such distributed algorithms can be found in [40, 55, 58, 91].

PART II

The Uniform Reliable Broadcast Abstraction

CHAPTER 3 The Uniform Reliable Broadcast Abstraction

This chapter focuses on the *uniform reliable broadcast* (URB) abstraction and its implementation in an asynchronous message-passing system prone to process crashes. Such a communication abstraction is central in the design and the implementation of fault-tolerant distributed systems, as many non-trivial fault-tolerant distributed applications require communication with provable guarantees on message deliveries.

After having defined the URB abstraction, the chapter presents a construction of it in an asynchronous message passing system prone to process crashes but with reliable channels (i.e., in the system model $\mathcal{AS}_{n,t}[\emptyset]$). The chapter considers then two properties (related to the quality of service) that can be added to URB without requiring to enrich the system model with additional assumptions. These properties are on the message delivery order and are "first in first out" (FIFO) message delivery, and "causal order" (CO) message delivery.

3.1 THE UNIFORM RELIABLE BROADCAST PROBLEM

3.1.1 FROM BEST EFFORT TO GUARANTEED RELIABILITY

The broadcast operation "broadcast (m)", introduced in the previous chapter, was a simple abbreviation used to replace the statement

$$\textbf{for each } j \in \{1, \ldots, n\} \textbf{ do } \mathsf{send}\ m \ \mathsf{to}\ p_j \ \textbf{end for}.$$

In the system model $\mathcal{AS}_{n,t}[\emptyset]$, this operation has a *best effort* semantics in the following sense. If the sender p_i is correct, a copy of the message m is sent to every process, and as the channels are reliable, every process (that has not crashed) receives a copy of the message. As the channels are asynchronous, these copies can be received at distinct independent time instants. Differently, if the sender crashes while executing broadcast m, an arbitrary subset of the processes receives the message m. Hence, in presence of failures, the specification of " broadcast m" provides no indication on which processes will receive the message m and which will not receive it. The aim of this section is to introduce a broadcast operation that provides the processes with stronger message delivery guarantees.

3.1.2 UNIFORM RELIABLE BROADCAST (URB)

URB provides the processes with two communication operations which are denoted "URB_broadcast (m)" and "URB_deliver ()". The first allows a process p_i to send a message m to all the processes (including the sender), while the second one allows a process to deliver a message that has been broadcast. In order to prevent ambiguities, when a process invokes "URB_broadcast m" we say that it "URB-broadcasts the message m", and when it returns from "URB_deliver ()," we say that it "URB-delivers a message" (sometimes, we also suppress the prefix "URB", when clear from the context). Differently, the primitives "send() to" and "receive()" are used for the messages sent and received at the underlying network level.

The specification of these operations assumes that every message that is broadcast is unique. This is easy to implement by associating a unique identity with each message m. Such an identity is made up of a pair ($m.sender$, $m.seq\_nb$) where $m.sender$ is the identity of the sender process, and $m.seq\_nb$ is a sequence number locally generated by $p_{m.sender}$. The sequence numbers associated with the messages broadcast by a process are the natural integers 1, 2, etc.

Definition The specification consists of the three following properties (this means that, to be correct, any implementation of "URB_broadcast (m)" and "URB_deliver ()" has to satisfy these properties):

- Validity. If a process URB-delivers a message m, then m has been previously URB-broadcast (by $p_{m.sender}$).
- Integrity. A process URB-delivers a message m at most once.
- Termination. (1) If a non-faulty process URB-broadcasts a message m, or (2) if a process URB-delivers a message m, then each non-faulty process URB-delivers the message m.

The validity property relates an output (here a message that is delivered) with an input (a message that has been broadcast). Said differently, there is no creation or alteration of messages. The integrity property states that there is no message duplication. Taken together, these two properties define the safety property of the URB problem. Let us observe that they are satisfied even if no message is ever delivered, whatever the messages that have been sent. So, for the specification to be complete, a liveness property is needed, namely, not all the messages can be lost. This is the aim of the termination property: if the process that URB-broadcasts a message is non-faulty, or if at least one process (be it faulty or non-faulty, this is why the abstraction is called *uniform*) URB-delivers a message, then that message has to be URB-delivered (at least) by the non-faulty processes. (Hence, this termination property belongs to the family of "all or none/nothing" properties.)

Remark It is easy to see from the previous specification that, during each execution, (1) the non-faulty processes deliver the same set of messages, (2) set includes all messages broadcast by non-faulty processes, and (3) each faulty process delivers a subset of the messages delivered by non-faulty processes. (Let us observe that two distinct faulty processes may deliver different subsets of messages.)

It is important to see that a message URB-broadcast by a faulty process might be or not delivered. It is not possible to place a requirement on such a message as the sender can crash before the message has been sent to a non-faulty process. The delivery of such messages depends on the execution.

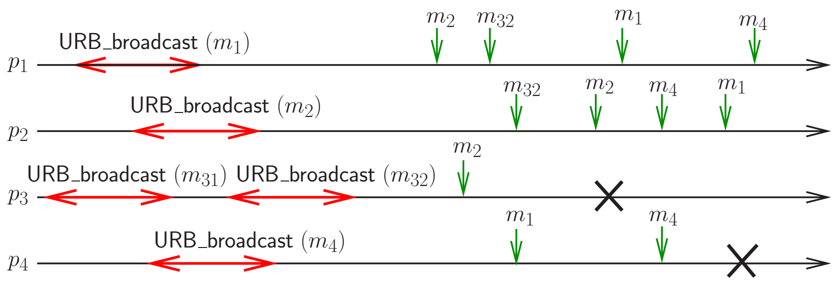

Figure 3.1: An example of the uniform reliable broadcast delivery guarantees

A simple example appears in Figure 3.1. There are four processes that URB-broadcast 5 messages. Processes p_1 and p_2 are non-faulty while p_3 and p_4 crash (crosses in the figure). The message deliveries are indicated with vertical top to bottom arrows on the process axes. Both p_1 and p_2 deliver the same set M of messages, while each faulty process delivers a subset of M. Moreover, not only the message m_{31}, that is sent by a faulty process, is never delivered, but the faulty process p_3 delivers none of the messages m_{31} and m_{32} that it has broadcast. More, the message m_{32} that has been sent by p_3 after m_{31} is delivered by the non-faulty processes, while m_{31} is not. This is due to the net effect of asynchrony and process crashes. It is easy to see that the message deliveries in Figure 3.1 respect the specification of the uniform reliable broadcast.

URB is a paradigm The uniform reliable broadcast problem is a paradigm that captures a family of distributed coordination problems. As an example, "URB_broadcast (m)" and " URB_deliver ()" can be given the meaning "this is an order" and "I execute it", respectively. It follows that non-faulty processes will execute the same set of actions, this set including all the orders issued by the non-faulty processes, plus a subset of orders issued by faulty processes.

Let us notice that URB is a *one-shot* problem. The specification applies to each message that is URB-broadcast separately from the other messages that are URB-broadcast.

Reliable broadcast The *reliable broadcast* communication abstraction is a weakened form of URB. It is defined by the same validity and integrity properties (no message loss, corruption or duplication) and the following weaker termination property:

- Termination. If a non-faulty process (1) URB-broadcasts a message m, or (2) URB-delivers a message m, then each non-faulty process URB-delivers the message m.

This means that a process that is faulty can deliver messages not delivered by the non-faulty processes. This termination property is the URB termination property without its *uniformity* requirement.

Let us observe that the termination property of the reliable broadcast abstraction does not state that the messages delivered by a faulty process have to be a subset of the messages delivered by the non-faulty processes. Hence, reliable broadcast is much less constraining (and consequently less powerful) than uniform reliable broadcast.

In the following, we do not consider this simple reliable broadcast abstraction because it is not useful for practical applications. As it is not known in advance whether a process will crash or not, it is sensible to require a process to behave as if it was non-faulty until it possibly crashes.

3.1.3 BUILDING URB IN $\mathcal{AS}_{n,t}[\emptyset]$

There is a very simple construction of the URB abstraction in the system model $\mathcal{AS}_{n,t}[\emptyset]$. This is due to the fact that the point-to-point channels are reliable. The structure of the construction is given in Figure 3.2.

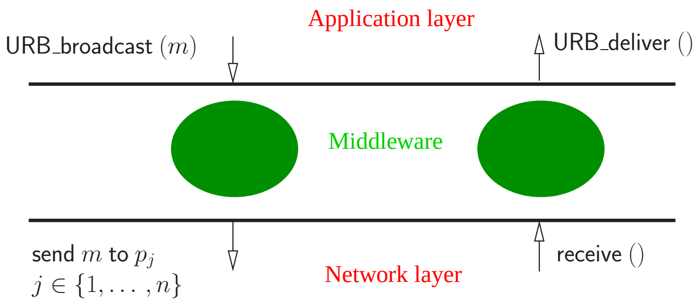

Figure 3.2: URB: architecture view

A simple construction The algorithms implementing URB_broadcast (m) and URB_deliver () are described in Figure 3.3. To broadcast a message m, a process p_i sends m to itself. When a process p_i receives a message, it discards it if it has already received a copy. Thanks to the unique identity $(m.sender, m.seq\_nb)$ carried by each message m, it is easy for p_i to check if m has already been received. If it is the first time it receives m, p_i forwards it to the other processes (but itself and the message sender), and only then delivers m to itself (at the application layer).

It is important to notice that the statement associated with the reception of MSG (m) is not required to be atomic. A process p_i can interleave the execution of several such statements.

Notation Let us notice that a tag MSG is added to each message (this tag will be used in the next sections). A message m is called an *application message*, while a message MSG (m) is called a *protocol message*.

Theorem 3.1 *The algorithm described in Figure 3.3 constructs a uniform reliable broadcast communication abstraction in* $\mathcal{AS}_{n,t}[\emptyset]$.

```
operation URB_broadcast $(m)$:
      send MSG$(m)$ to $p_i$.

when MSG $(m)$ is received from $p_k$:
      if (first reception of $m$) then
         for each $j \in \{1, \ldots, n\} \setminus \{i, k\}$ do send MSG $(m)$ to $p_j$ end for;
          URB_deliver $(m)$ % deliver $m$ to the upper layer application %
      end if.
```

Figure 3.3: Uniform reliable broadcast in $\mathcal{AS}_{n,t}[\emptyset]$ (code for p_i)

Proof The proof of the validity property follows directly from the text of the algorithm that forwards only messages that have been received. The proof of the integrity property follows directly from the fact that a message m is delivered only when it is received for the first time.

The termination property is a direct consequence of the "forward first and then deliver" strategy. Let us first consider a message m broadcast by a non-faulty process p_i. As p_i is non-faulty, it forwards the protocol message MSG (m) to every other process and delivers it to itself. As channels are reliable, each process will eventually receive a copy of MSG (m) and URB-delivers m (the first time it receives MSG (m)).

Let us now consider the case of a (faulty or non-faulty) process p_j that delivers a message m. Before delivering m, p_j has forwarded MSG (m) to all, and the same reasoning as before applies, which completes the proof of the termination property. $\square_{Theorem\ 3.1}$

3.2 IMPROVING THE QUALITY OF SERVICE

Uniform reliable broadcast provides guarantees on which messages are delivered to processes. As we have seen, non-faulty processes deliver the same set S of messages, and each faulty process p_i delivers a subset $S_i \subseteq S$.

FIFO and CO message delivery Some applications are easier to design when processes are provided with stronger guarantees on message delivery. These guarantees are on the order in which messages are delivered to the upper layer application. We consider here two types of such guarantees: the *First In, First Out* (FIFO) property, and the *Causal Order* (CO) property. (A third delivery property, called *Total Order* (TO) will be studied in another chapter.)

A modular view of the FIFO and CO uniform reliable constructions presented in this section is given in Figure 3.4. Each arrow corresponds to a construction. It is important to see that these constructions can be built in any system where the URB abstraction can be built. When compared to URB, neither FIFO-URB nor CO-URB requires additional computability-related assumptions (such as restrictions on the model on top of which URB is built or failure detector-like additional objects).

$$\begin{tikzcd} \boxed{\text{URB}} \arrow[r, "\text{Fig. 3.7}"] \arrow[rr, bend right, "\text{Fig. 3.12}"'] & \boxed{\text{FIFO-URB}} \arrow[r, "\text{Fig. 3.11}"] & \boxed{\text{CO-URB}} \end{tikzcd}$$

Figure 3.4: From URB to CO-URB in $\mathcal{AS}_{n,t}[\emptyset]$

One-shot vs multi-shot problems As we have seen, URB is a one-shot problem. It considers each message separately from the other messages. Differently, the FIFO-URB problem and the CO-URB problem are not one-shot problems. This is because (as we are about to see) their specifications involve all the messages that are broadcast.

3.2.1 "FIRST IN, FIRST OUT" (FIFO) MESSAGE DELIVERY

Definition The FIFO-URB abstraction is made up of two operations denoted "FIFO_broadcast m" and "FIFO_deliver ()". It is the URB abstraction (defined by the validity, integrity and termination properties stated in Section 3.1.2) enriched with the following additional property.

- FIFO message delivery. If a process FIFO-broadcasts a message m and then FIFO-broadcasts a message m', then no process FIFO-delivers m' unless it has FIFO-delivered m before.

This property states that the messages broadcast by each source (taken separately) are delivered according to their sending order. There is no delivery constraint placed on messages broadcast by different sources. It is important to notice that the FIFO property prevents a faulty process from delivering m' while never delivering m. Given any process p_i, a faulty process delivers a prefix of the messages broadcast by p_i.

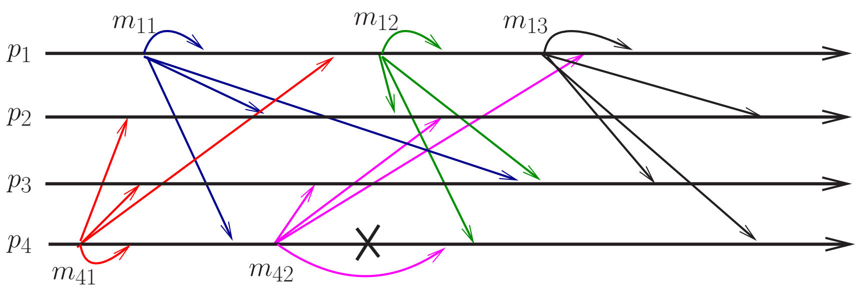

Figure 3.5: An example of FIFO message delivery

An example A simple example is depicted in Figure 3.5 where the transfer of each message is explicitly indicated. Process p_1 FIFO-broadcasts m_{11}, then m_{12}, and finally m_{13}. Process p_4 FIFO-broadcasts m_{41} and then m_{42}. The FIFO message delivery property states that m_{11} has to be delivered before m_{12}, which in turn has to be delivered before m_{13}. Similarly, with respect to the source p_4, no process is allowed to deliver m_{42} before m_{41}. In this example, p_4 crashes before delivering its own message m_{42}.

As the FIFO specification imposes no constraint on the messages broadcast by distinct processes, we can easily see that the delivery of the messages from p_1 and the ones from p_4 can be interleaved differently at distinct receivers.

A simple construction The construction assumes that the underlying communication layer provides processes with a uniform reliable broadcast abstraction as depicted in Figure 3.6.

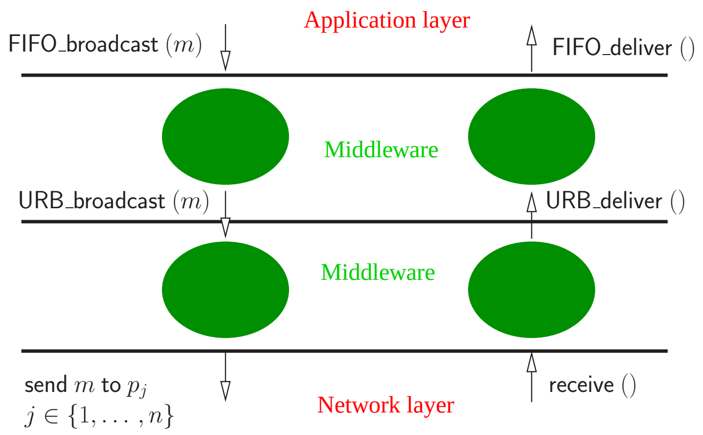

Figure 3.6: FIFO uniform reliable broadcast: architecture view

An easy way to ensure the FIFO message delivery property consists in associating an appropriate predicate with message delivery. While the predicate remains false, the message remains in the input buffer of the corresponding process, and is delivered as soon as the predicate becomes true. The construction for FIFO URB is described in Figure 3.7.

Each process p_i manages two local variables. The set msg_set_i (initialized to $\emptyset$) is used to keep the messages that have been URB-delivered but not yet FIFO-delivered by p_i. The array $next_i[1..n]$ (initialized to $[1, \ldots, 1]$) is such that $next_i[j]$ denotes the sequence number of the next message that p_i will FIFO-deliver from p_j (the sequence number of the first message broadcast by a process p_i is 1, the sequence number of the second message is 2, etc.).

The operation "FIFO_broadcast (m)" consists of a simple invocation of "URB_broadcast (m)". When a message m is URB-delivered by the underlying communication layer, p_i deposits it in the set msg_set_i if m arrives too early with respect to its FIFO-delivery order. Otherwise, p_i FIFO-delivers

```
operation FIFO_broadcast ($m$):
    $m.sender \leftarrow i$; $m.seq\_nb \leftarrow p_i$'s next seq. number (starting from 1);
    URB_broadcast MSG($m$).

when MSG($m$) is URB-delivered: % $m$ carries its identity ($m.sender, m.seq\_nb$) %
    let $j = m.sender$;
    if ($next_i[j] = m.seq\_nb$)
      then FIFO_deliver ($m$);
           $next_i[j] \leftarrow next_i[j] + 1$;
           while ( $\exists m' \in msg\_set_i : (m'.sender = j) \wedge (next_i[j] = m'.seq\_nb)$ )
                do FIFO_deliver ($m'$);
                   $next_i[j] \leftarrow next_i[j] + 1$;
                   $msg\_set_i \leftarrow msg\_set_i \setminus \{m'\}$
           end while
      else $msg\_set_i \leftarrow msg\_set_i \cup \{m\}$
    end if.
```

Figure 3.7: FIFO message delivery in $\mathcal{AS}_{n,t}[\emptyset]$ (code for p_i)

m. After having delivered m, p_i FIFO-delivers the messages from the same sender (if any) whose sequence numbers agree with the delivery order. The processing associated with the URB-delivery of a message m is assumed to be atomic, i.e., a process p_i executes one URB-delivery code at a time.

Theorem 3.2 *The algorithm described in Figure 3.7 constructs a FIFO uniform reliable broadcast communication abstraction in any system in which URB can be built.*

Proof The proof is an immediate consequence of the properties of the underlying URB abstraction (Theorem 3.1) and the use of sequence numbers. $\square_{Theorem\ 3.2}$

3.2.2 "CAUSAL ORDER" (CO) MESSAGE DELIVERY

A partial order on messages Let M be the set of messages that are URB-broadcast during an execution, and $\widehat{M} = (M, \rightarrow_M)$ be the relation where $\rightarrow_M$ is defined on M as follows. Given $m, m' \in M$, $m \rightarrow_M m'$ (and we say that "m causally precedes m'"):

- m and m' have been broadcast by the same process and m has been broadcast before m', or
- m has been delivered by a process p_i before p_i broadcasts m', or
- There is message $m'' \in M$ such that $m \rightarrow_M m''$ and $m'' \rightarrow_M m'$.

Let us notice that, as a message cannot be delivered before being broadcast, $\widehat{M}$ is a partial order.

Causal message delivery The causal order URB abstraction (CO) is made up of two operations denoted "CO_broadcast m" and "CO_deliver ()". It is the URB abstraction (defined by the validity, integrity and termination properties stated in Section 3.1.2) enriched with the following additional property.

- CO message delivery. If $m \rightarrow_M m'$, then no process CO-delivers m' unless it has previously CO-delivered m.

The FIFO delivery order is a restriction of the causal delivery order on a process basis. This means that the causal delivery order generalizes the FIFO delivery order to all the messages whose broadcasts are related by the "message happened before" relation ($\rightarrow_M$), whatever their senders are.

An example An example of CO message delivery is depicted in Figure 3.8. We have $m_{11} \rightarrow_M m_{42}$ and $m_{41} \rightarrow_M m_{42}$. As the messages m_{11} and m_{41} are not "$\rightarrow_M$"-related, it follows that every process can deliver them in any order. Differently, m_{42} has to be delivered at any process after m_{41} (FIFO order is included in CO order), and m_{42} has to be delivered at any process after m_{11} (because p_4 delivers m_{11} before broadcasting m_{42}). So, despite the fact that p_1 and p_2 deliver m_{11} and m_{41} in different order, these messages delivery orders are correct. The message delivery order is also correct at p_3 if m_{42} is delivered according to the plain arrow, and it is not correct if m_{42} is delivered according to the dotted arrow (i.e., before m_{11}).

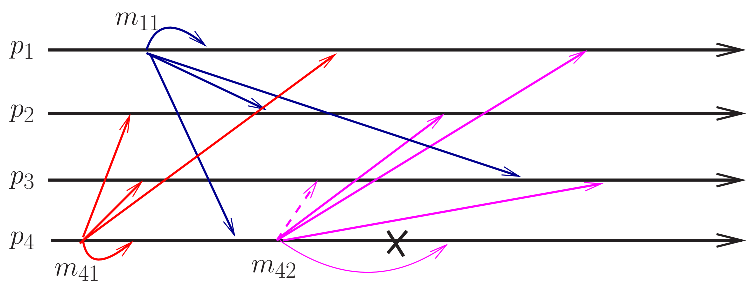

Figure 3.8: An example of CO message delivery

The local order property The definition of this property is motivated by Theorem 3.3 that follows, which gives a characterization of causal order, namely, CO is FIFO + local order.

- Local order. If a process delivers a message m before broadcasting a message m', no process delivers m' unless it has previously delivered m.

Theorem 3.3 *Causal order is equivalent to the combination of FIFO order and local order.*

Proof It follows from its very definition that the causal order property implies the FIFO property and the local order property. Let us show the other direction.

Assuming the FIFO order property and the local order property are satisfied, let m and m' be two messages such that $m \rightarrow_M m'$, and p be a process that delivers m'. The proof consists in showing that p delivers m before m'.

As $m \rightarrow_M m'$, there is a finite sequence of messages $m = m_1, m_2, \ldots, m_{k-1}, m_k = m'$, with $k \geq 2$, that have been broadcast by the processes $q_1, q_2, \ldots, q_k$, respectively, and are such that, $\forall x : 1 \leq x < k$, we have $m_x \rightarrow_M m_{x+1}$ due to the first or the second item of the CO delivery definition (i.e., not taking into account the third item on transitivity). For any x such that $1 \leq x < k$, we have one of the following cases.

- If $q_x = q_{x+1}$: m_x and m_{x+1} are broadcast by the same process. It follows from the FIFO order property that p delivers m_x before m_{x+1}.
- If $q_x \neq q_{x+1}$: m_x and m_{x+1} are broadcast by different processes, and q_{x+1} delivers m_x before broadcasting m_{x+1}. It follows from the local order property that p delivers m_x before m_{x+1}.

It follows that when p delivers $m_k = m'$, it has previously delivered m_{k-1}. Similarly, when it delivers m_{k-1}, it has previously delivered m_{k-2}, etc. until $m_1 = m$. It follows that if p delivers m', it has previously delivered m. $\square_{Theorem\ 3.3}$

Remark Theorem 3.3 is important, from a proof modularity point of view, when one has to prove that an algorithm satisfies CO delivery property. Namely, one has only to show that the algorithm satisfies both the FIFO property and the local order property. It then follows from Theorem 3.3 that the algorithm satisfies to CO delivery property. We will proceed that way in the proof of Theorem 3.4. (A direct proof of the CO delivery property would require a long and tedious induction on the length of "message causality chains" defined by the relation "$\rightarrow_M$".)

3.2.3 FROM FIFO-BROADCAST TO CO-BROADCAST

A simple CO-broadcast construction from URB-broadcast Before presenting a CO-broadcast construction based on the FIFO-broadcast abstraction, this paragraph presents a very simple (but very inefficient) construction of the CO broadcast abstraction on top of the URB abstraction (Figure 3.9). This construction consists in associating with every message all the messages that causally precede it.

To that end, each process p_i manages a local variable, denoted $causal\_pred_i$, that contains the sequence of all the messages m' such that $m' \rightarrow_M m$, where m is the next message that p_i will CO-broadcast. The variable $causal\_pred_i$ is initialized to the empty sequence (denoted ϵ). The operator $\oplus$ denotes the concatenation of a message at the end of $causal\_pred_i$.

When it CO-broadcasts m, p_i URB-broadcasts the protocol message MSG ($causal\_pred_i \oplus m$), and then updates $causal\_pred_i$ to $causal\_pred_i \oplus m$ as, from now on, the application message m belongs to the causal past of the next application messages that p_i will broadcast.

When it URB-delivers MSG ($\langle m_1, \ldots, m_\ell \rangle$), p_i considers, one after the other, each application message m_x of the received sequence. If it has already CO-delivered m_x, it discards it. Otherwise, it CO-delivers it and adds it at the end of $causal\_pred_i$.

Both the code associated with the URB-delivery of a message and the code associated with the CO-broadcast operation are assumed to be executed atomically. This construction is highly inefficient as the size of protocol messages increases forever.

```
operation CO_broadcast ($m$):
    URB_broadcast MSG ($causal\_pred_i \oplus m$);
    $causal\_pred_i \leftarrow causal\_pred_i \oplus m$.

when MSG ($\langle m_1, \ldots, m_\ell \rangle$) is URB-delivered:
    for $x$ from 1 to $\ell$ do
        if ($m_x$ not yet CO-delivered) then
            CO_deliver ($m_x$);
            $causal\_pred_i \leftarrow causal\_pred_i \oplus m_x$
        end if
    end for.
```

Figure 3.9: A simple URB-based CO-broadcast construction in $\mathcal{AS}_{n,t}[\emptyset]$ (code for p_i)

From FIFO-broadcast to CO-broadcast: the construction A FIFO broadcast-based construction of the CO broadcast abstraction is described in Figure 3.11. Its underlying principle is as follows. It uses the FIFO property provided by the underlying FIFO broadcast in order to replace the sequence $causal\_past_i$ used in Figure 3.9 by a suffix of it, denoted $im\_causal\_past_i$. More precisely, while $causal\_past_i$ increases without bound, the FIFO delivery property allows a resetting of $causal\_past_i$ to the empty sequence each time p_i CO-broadcast a new message. This sequence is consequently renamed $im\_causal\_past_i$ and contains only messages that p_i has CO-delivered since its previous CO-broadcast.

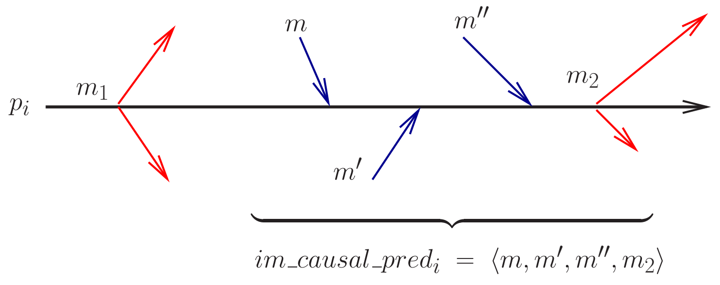

Figure 3.10: How is built the sequence of messages $im\_causal\_pred_i$

As an example illustrating this idea, let us consider Figure 3.10 where the process p_i CO-broadcasts two messages, first m_1 and then m_2. Between these two broadcasts, p_i has CO-delivered the messages m, m' and m'', in that order. Hence, when p_i CO-broadcasts m_2, it actually FIFO-broadcasts the sequence $\langle m, m', m'', m_2 \rangle$, thereby indicating that, if not yet CO-delivered, messages m, m' and m'' have to be CO-delivered before m_2. Hence, we have $im\_causal\_pred_i = \langle m, m', m'' \rangle$ when p_i is about to CO-broadcast m_2.

As before, both the code associated with the FIFO-delivery of a message and the code associated with the CO-broadcast operation are assumed to be executed atomically.

```
operation CO_broadcast ($m$):
    FIFO_broadcast MSG ($im\_causal\_pred_i \oplus m$);
    $im\_causal\_pred_i \leftarrow \epsilon$.

when MSG ($\langle m_1, \ldots, m_\ell \rangle$) is FIFO-delivered:
    for $x$ from 1 to $\ell$ do
        if ($m_x$ not yet CO-delivered) then
            CO_deliver ($m_x$);
            $im\_causal\_pred_i \leftarrow im\_causal\_pred_i \oplus m_x$
        end if
    end for.
```

Figure 3.11: From FIFO URB to CO message delivery in $\mathcal{AS}_{n,t}[\emptyset]$ (code for p_i)

Let us remember that, due to Theorem 3.2, it is possible to build a FIFO reliable broadcast abstraction in any system in which URB can be built. So, the construction of the CO reliable broadcast abstraction on top of the URB abstraction does not require additional assumptions.

Remark The processing associated with a FIFO-delivery is "fast" in the sense that, when a sequence of messages is FIFO-delivered, each message contained in that sequence is CO-delivered (if not yet done). The price that has to be paid to obtain this delivery efficiency property is that the FIFO communication abstraction has to handle "big" protocol messages that are sequences of application messages. Moreover, the underlying FIFO broadcast abstraction cannot enjoy this "fast delivery" property (each process has to manage a local "waiting room" msg_set_i in which messages can be momentarily delayed).

Theorem 3.4 *The algorithm described in Figure 3.11 constructs a CO uniform reliable broadcast communication abstraction in any system in which URB can be built.*

Proof Proof of the validity and integrity properties. Let us first observe that, as "CO_broadcast (m)" is implemented on top of FIFO-broadcast, it directly inherits its validity property (neither creation nor alteration of protocol messages), and integrity property (a protocol message is FIFO-delivered at most once). It follows that no application message m can be loss or modified. It is also clear from the test done before CO-delivering an application message that such a message can be CO-delivered at most once.

Proof of the termination property. As far as termination is concerned, we have the following. When a process CO-broadcasts an application message m, it FIFO-broadcasts a protocol message MSG($seq \oplus m$). Moreover, when a sequence of application messages MSG($\langle m_1, \ldots, m_\ell \rangle$) is

FIFO-delivered, if not yet CO-delivered, each application message m_x, $1 \le x \le \ell$, is CO-delivered without being delayed. Consequently, the CO-broadcast algorithm inherits also the termination property of the underlying FIFO broadcast, from which it follows that each application message that has been CO-broadcast is CO-delivered.

Proof of the CO delivery property. We have to prove that, for any two messages m and m' such that $m \rightarrow_M m'$ (as defined in the previous paragraph), no process CO-delivers m' unless it has previously CO-delivered m_x. This proof is based on three claims.

Claim C1. Let us suppose that a process p_i FIFO-broadcasts $\text{MSG}(seq' \oplus m')$ (where seq' is a sequence of application messages), and either p_i has previously FIFO-broadcast $\text{MSG}(seq \oplus m)$ or $m \in seq'$. Then, no process CO-delivers m' unless it has previously CO-delivered m.
Proof of claim C1. The proof is by contradiction. Let us assume that, while the assumption of the claim is satisfied, some process CO-delivers m' before m. Let τ be the first time instant at which a process CO-delivers m' without having previously CO-delivered m, and let p_j be such a process. We consider two cases, according to what caused p_j to CO-deliver m'.

- Case 1. p_j has FIFO-delivered $\text{MSG}(seq' \oplus m')$. There are two sub-cases (due to the assumption in the claim).
  - Sub-case 1: $m \in seq'$.
  - Sub-case 2: p_i has FIFO-broadcast $\text{MSG}(seq \oplus m)$ before $\text{MSG}(seq' \oplus m')$. It then follows from the FIFO delivery property that p_j has FIFO-delivered $\text{MSG}(seq \oplus m)$ before $\text{MSG}(seq' \oplus m')$.

  It is easy to conclude from the text of the algorithm that, whatever the sub-case, p_j CO-delivers m before m', which contradicts the assumption that p_j CO-delivers m' before m.

- Case 2. p_j has FIFO-delivered a message $\text{MSG}(seq'' \oplus m'')$ such that $m' \in seq''$ and m is not before m' in seq'' (proof hypothesis). Let p_k be the sender of that message. Since $m' \in seq''$, process p_k has CO-delivered m' before FIFO-broadcasting $\text{MSG}(seq'' \oplus m'')$.

  Due to the FIFO order property, p_j has FIFO-delivered all the previous protocol messages FIFO-broadcast by p_k. Since, by assumption, p_j does not CO-deliver m before m', the application message m was not included in any of these FIFO-broadcasts, and m does not appear before m' in seq''. Hence, when p_k CO-delivered m', it has not previously CO-delivered m. Moreover, p_k CO-delivered m' before p_j CO-delivered it. We consequently have $\tau' < \tau$, where τ' is the time instant at which p_k has CO-delivered m'. But this contradicts the definition of τ that states that τ is the first time instant at which a process CO-delivers m' without having previously CO-delivered m.

As both cases lead to a contradiction, the claim follows. **End of proof of claim C1.**

The proof of the CO delivery property follows from the claims C2 and C3 that follow. C2 establishes that the algorithm satisfies the FIFO message delivery property, while C3 establishes that it satisfies the local order property. Once these claims are proved, the CO delivery property is obtained as an immediate consequence of Theorem 3.3 that states the following: FIFO message delivery + local order $\Rightarrow$ CO message delivery.

Claim C2. The algorithm satisfies the FIFO (application) message delivery property.
Proof of claim C2. Let us suppose that p_i CO-broadcasts m before m'. It follows that p_i FIFO-broadcasts $\textsc{msg}(seq \oplus m)$ before $\textsc{msg}(seq' \oplus m')$. Let us consider the channel from p_i to p_j. It follows from Claim C1 that p_j cannot CO-delivers m' unless it has CO-delivered m, which proves the claim. End of proof of claim C2.

Claim C3. The algorithm satisfies the local order property (for application messages).
Proof of claim C3. Let p_i be a process that CO-delivers m before CO-broadcasting a message m', and p_j a process that CO-delivers m'. We must show that p_j CO-delivers m before m'.

Let m'' be the first message that p_i CO-broadcasts after it has CO-delivered m (notice that m'' could be m'). When it CO-broadcasts m'', p_i FIFO-broadcasts $\textsc{msg}(seq'' \oplus m'')$ (for some seq''). Due to the text of the algorithm and the definition of m'', it follows that $m \in seq''$. Due to claim C1, p_i CO-delivers m before m''. If $m'' = m'$, the claim follows. Otherwise, p_i CO-broadcasts m'' before m'. Then due to claim C2, p_j CO-delivers m'' before m'. Consequently, as $m \in seq''$, it follows that p_j CO-delivers m before m', as requested. End of proof of claim C3.

$\square_{Theorem}$ 3.4

3.2.4 FROM URB TO CO-BROADCAST: CAPTURING MESSAGE CAUSAL PAST IN A VECTOR

The delivery condition Differently from the previous one, this construction of the CO broadcast abstraction is built directly on top of the uniform reliable broadcast abstraction (so the layer structure is the same as the one of Figure 3.6 where FIFO is replaced by CO).

Each process p_i manages a vector (vector clock), denoted $last\_sn_i[1..n]$ and initialized to $[0, \ldots, 0]$, such that $last\_sn_i[k]$ contains the sequence number of the last message CO-broadcast by p_k that has been CO-delivered by p_i. Thanks to this control data, each application message m can piggyback a vector of integers denoted $m.last\_sn[1..n]$ such that

$$m.last\_sn[k] = \text{ number of messages } m' \text{ CO-Broadcast by } p_k \text{ such that } m' \rightarrow_M m.$$

Let m be a message that is URB-delivered to p_i. Its CO-delivery condition can be easily stated: m can be CO-delivered if all the messages m' such that $m' \rightarrow_M m$ have already been locally CO-delivered by p_i. Operationally, this is locally captured by the following delivery condition:

$$DC_i(m) \equiv \big(\forall k : \ last\_sn_i[k] \geq m.last\_sn[k]\big).$$

Let us notice that, when a process CO-broadcasts a message m, it can immediately CO-deliver it. This is because, due to the very definition of the causal precedence relation "$\rightarrow_M$", all the messages m' such that $m' \rightarrow m$ have already been CO-delivered, and, consequently, $DC_i(m)$ is then satisfied.

```
operation CO_broadcast ($m$):
    $done_i \leftarrow false$;
    $m.last\_sn[1..n] \leftarrow last\_sn_i[1..n]$;
    $m.sender \leftarrow i$; $m.seq\_nb \leftarrow p_i$'s next seq. number;
    URB_broadcast MSG ($m$);
    wait ($done_i$).

when MSG ($m$) is URB-delivered:
    if $DC_i(m)$
      then CO_deliver ($m$);
           let $j = m.sender$;
           $last\_sn_i[j] \leftarrow m.seq\_nb$;
           $done_i \leftarrow (m.sender = i)$;
           while $(\exists m' \in msg\_set_i : DC_i(m'))$
                 do CO_deliver ($m'$);
                    let $j = m'.sender$;
                    $last\_sn_i[j] \leftarrow m'.seq\_nb$;
                    $msg\_set_i \leftarrow msg\_set_i \setminus \{m'\}$
           end while
      else $msg\_set_i \leftarrow msg\_set_i \cup \{m\}$
    end if.
```

Figure 3.12: From URB to CO message delivery in $\mathcal{AS}_{n,t}[\emptyset]$ (code for p_i)

The construction The construction is described in Figure 3.12. In addition to its sequence number, each message m CO-broadcast by a process p_i, carries in its field $m.last\_sn$ a copy of the local array $last\_sn_i$ (which encodes the causal past of m from the CO-broadcast point of view). The value of $m.last\_sn[k]$ represents the number of messages m' such that $m' \rightarrow_M m$.

To CO-broadcast a message m, first updates the control fields of m, and then URB-broadcasts m and waits until it locally URB-delivers m. The boolean $done_i$ is used to ensure that if m is CO-broadcast by p_i before m', the broadcast of m is correctly encoded in $m'.last\_sn[1..n]$.

As far as the sender p_i of m is concerned, the CO-delivery of m is not done directly within the operation CO_broadcast (m). This is in order to benefit from the properties of the underlying URB broadcast abstraction, namely, if p_i URB-delivers m, we know from the termination property of the URB abstraction that all the non-faulty processes eventually URB-deliver m.

When a message m is URB-delivered by a process p_i, that process tests the delivery condition $DC_i(m)$. If it is false, there are messages m' such that $m' \rightarrow_M m$ that have not yet been CO-delivered. Consequently, m is deposited in the set msg_set_i. If $DC_i(m)$ is true (this is always the case when $m.sender = i$), p_i updates accordingly $last\_sn_i[m.sender]$ to $m.seq\_nb$ (this is where the array

$last\_sn_i$ is updated with the sequence number of the message that is CO-delivered), and sets $done_i$ to *true* if $m.sender = i$.

After it has CO-delivered a message m, process p_i checks if messages in the waiting room msg_set_i can be CO-delivered. If there are such messages, it CO-delivers them, suppresses them from msg_set_i, and updates accordingly $last\_sn_i$.

Except for the wait statement at the end of the operation "`CO_broadcast` (m)", the first three lines of "`CO_broadcast` (m)", on one side, and all the statements associated with the URB-delivery of a message are executed atomically.

Example A simple example of the vector-based CO broadcast construction is described in Figure 3.13. Messages m_1, m_2 and m_3 are such that $m_1.sender = 1$, $m_1.seq\_nb = 1$, $m_2.sender = 2$, $m_2.seq\_nb = 1$, $m_3.sender = 3$, and $m_3.seq\_nb = 1$. Messages m_1 and m_2 have no messages in their causal past (i.e., there is no message m' such that $m' \rightarrow_M m_1$ or $m' \rightarrow_M m_2$, respectively), so we have $m_1.last\_sn = m_2.last\_sn = [0, 0, 0]$. As their broadcasts are not CO-related, these messages can be CO-delivered in different order at different processes. Differently, message m_3 is such that $m_1 \rightarrow_M m_3$. So, $m_3.last\_sn = [1, 0, 0]$ that encodes the fact that the first message CO-broadcast by p_1 (namely m_1) has been CO-delivered by p_3 before p_3 CO-broadcasts m_3.

Consequently, as show in the figure, while m_3 is URB-delivered at p_2 before m_1, its CO-delivery condition forces it to remain in p_2's input buffer msg_set_2 until m_1 has been CO-delivered at p_2 (this is indicated by a dotted arrow in the figure).

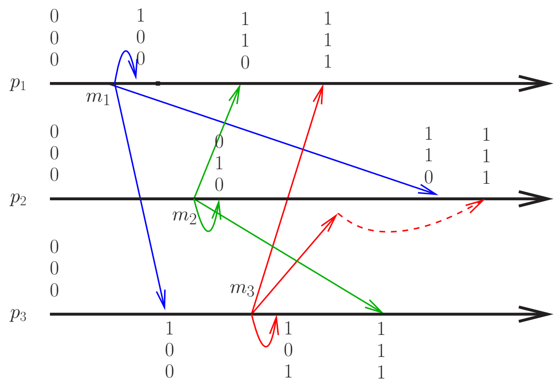

Figure 3.13: How the vectors are used to construct the CO broadcast abstraction

Lemma 3.5 *Let m and m' be two messages.* $(m \rightarrow_M m') \Rightarrow \big(\forall k\ (m.last\_sn[k] \leq m'.last\_sn[k])\big) \wedge \big(\exists k : m.last\_sn[k] < m'.last\_sn[k]\big)$.

Proof Let us remember that, when a process p_i CO-broadcasts a message m, the sequence number associated with m is the rank of m in the sequence of messages CO-broadcast by p_i.

Let us first consider the case where the messages m and m' are CO-broadcast by the same process p_i. Due to the fact the sequence numbers are strictly increasing, it follows from the text of the algorithm that we have $m.last\_sn[i] \leq m.seq\_nb - 1$ when m is CO-broadcast, and $m.seq\_nb \leq m'.last\_sn[i]$ when m' is CO-broadcast, from which we conclude $m.last\_sn[i] < m'.last\_sn[i]$. As far the entries $k \neq i$ are concerned, let us observe that the successive sequence numbers contained in $last\_sn_i[k]$ are increasing, from which we conclude $\forall k : m.last\_sn[k] \leq m'.last\_sn[k]$), which completes the proof for this case.

Let us now consider the case where m and m' are CO-broadcast by different processes. As $m \rightarrow_M m'$, there is a finite chain of messages such that $m = m_0 \rightarrow_M m_1 \rightarrow_M \cdots \rightarrow_M m_z = m'$, such that, for each message m_x, $1 \leq x \leq z$, the process that the process that CO-broadcasts m_x has previously CO-delivered m_{x-1}. (if m_1 is m', the length of the chain is 1.) We claim that $(\forall k \ (m.last\_sn[k] \leq m_1.last\_sn[k])) \ \wedge \ (\exists k : m.last\_sn[k] < m_1.last\_sn[k])$. Then the proof of the lemma follows directly by a simple induction on the length of the message chain.

Proof of the claim. Let p_i be the process that CO-broadcast m, and p_j $(i \neq j)$ the process that CO-delivered m before CO-broadcasting m_1. It follows from the definition of $m \rightarrow_M m_1$, the fact that m is CO-delivered by p_j, and the delivery condition $DC_j(m)$ that $\forall k \ (m.last\_sn[k] \leq last\_sn_j[k])$ just after m is CO-delivered by p_j. On another side, when p_j CO-delivered m, it executed the statement $last\_sn_j[i] \leftarrow m.seq\_nb$. Hence, we have $m.last\_sn[i] < m.last\_sn[i] + 1 = last\_sn_j[i] = m.seq\_nb$, and, consequently, $(\forall k \ (m.last\_sn[k] \leq last\_sn_i[k])) \ \wedge \ (\exists k : m.last\_sn[k] < last\_sn[k]_i)$.

As, due to the algorithm we have $(\forall k \ (m_1.last\_sn[k] \geq last\_sn_i[k])$ when p_j CO-broadcasts m_1, the claim follows. End of proof of the claim. $\square_{Lemma}$ 3.5

Theorem 3.6 *The algorithm described in Figure 3.11 constructs a CO uniform reliable broadcast communication abstraction in any system in which URB can be built.*

Proof Proof of the validity and integrity properties. The validity property follows directly from the validity of the underlying URB broadcast abstraction, and the text of the algorithm (that does not create applications messages).

The integrity property of the underlying URB abstraction guarantees that, for every application message m that is CO-broadcast, a process p_i URB-delivers at most one protocol message MSG (m). If $DC_i(m)$ is satisfied, the message m is immediately CO-delivered. Otherwise, it deposited in msg_set_i, and is suppressed from this set when it is CO-delivered. It follows that no message m can be CO-delivered more than once by each process p_i. Hence the integrity property of the CO abstraction.

Proof of the termination property. The termination property of the underlying URB abstraction guarantees that (1) if a non-faulty process CO-broadcasts a message m (as in that case it URB-broadcasts MSG (m)), or (2) if any process URB-delivers MSG (m), then each non-faulty process URB-delivers MSG (m). It follows that if (1) or (2) occurs, then every non-faulty process p_i either CO-delivers m or deposits m in msg_set_i. Hence, to prove the termination property of the CO broadcast abstraction, we have to show that any non-faulty process p_i eventually CO-delivers all the messages that are deposited in its set msg_set_i. Let us observe that, as no two messages sent by the same process have the same sequence number, for any two messages m and m', we have $m.last\_sn \neq m'.last\_sn$.

Let us assume by contradiction that some messages remain forever in msg_set_i. Let us denote $blocked_i$ this set of messages, and let us totally order the messages $m \in blocked_i$, according to the lexicographic order $<_{lex}$ defined by their vectors $m.last\_sn$. (Let $v = [a, b, c]$ and $v' = [a', b', c']$ be two vectors. Let us remember that $v <_{lex} v'$ if $(a < a') \vee (a = a' \wedge b < b') \vee (a = a' \wedge b = b' \wedge c < c')$.) Finally, let m be the first message of msg_set_i according to lexicographical order. Let p_x be the process that has issued CO_broadcast (m).

As m remains forever in msg_set_i, $DC_i(m)$ remains forever false, and, consequently, there is at least one process identity k such that $last\_sn_i[k] < m.last\_sn[k]$. Let β be the last value taken by $last\_sn_i[k]$, and let $\alpha = m.last\_sn[k]$. Hence, $\beta < \alpha$. Let m' be the last message CO-broadcast by p_k and CO-delivered by p_x, just before it issues CO_broadcast (m). Hence, we have $m.last\_sn[k] = last\_sn_x[k] = \alpha$. This is depicted in Figure 3.14.

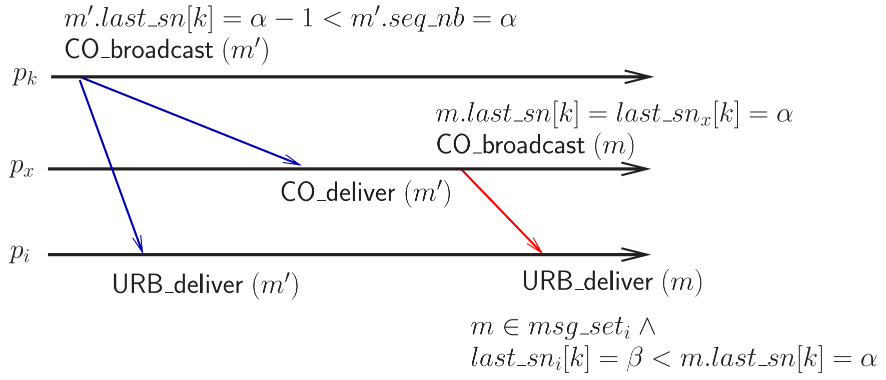
$m'.last\_sn[k] = \alpha - 1 < m'.seq\_nb = \alpha$
CO_broadcast (m')
p_k
$m.last\_sn[k] = last\_sn_x[k] = \alpha$
CO_broadcast (m)
p_x
CO_deliver (m')
p_i
URB_deliver (m')
URB_deliver (m)
$m \in msg\_set_i \wedge$
$last\_sn_i[k] = \beta < m.last\_sn[k] = \alpha$

Figure 3.14: Proof of the CO-delivery property (second construction)

As p_x has CO-delivered m', it has previously URB-delivered MSG (m'). It then follows from the termination property of the URB abstraction, that any non-faulty process (hence p_i) eventually URB-delivers MSG (m'). When p_i URB-delivers MSG (m'), there are two cases.

- $DC_i(m')$ is false and remains false forever. In that case, as we have $m'.last\_sn <_{lex} m.last\_sn$ (consequence of Lemma 3.5), it follows that m is not the first message of msg_set_i according to lexicographical order. A contradiction.
- m' is eventually CO-delivered by p_i. In that case, $last\_sn_i[k]$ becomes equal to α, which contradicts the fact that the last value taken by $last\_sn_i[k]$ is $\beta < \alpha$.

In both cases, we obtain a contradiction, which completes the proof of the termination property.

Proof of the CO delivery property. Let us first consider the case where p_i is not the process that issued CO_broadcast (m). The CO delivery property follows directly from the associated delivery condition $DC_i(m)$: $m.last\_sn[1...n]$ encodes all the messages m' such that $m' \rightarrow_M m$ (Lemma 3.5), while $last\_sn_i[1..n]$ encodes all the messages already CO-delivered by p_i.

Let us now consider the CO-delivery at p_i of a message m that has been CO-broadcast by the same process p_i. The messages m' such that $m' \rightarrow m$ are encoded in the value $last\_sn_i$ at the time at which p_i issues CO_broadcast (m), which means that we then have $m.last\_sn = last\_sn_i$. It follows that the delivery condition $DC_i(m)$ is always satisfied when p_i URB-delivers its own message MSG (m), which completes the proof of the CO delivery property. $\square_{Theorem\ 3.6}$

3.2.5 THE TOTAL ORDER BROADCAST ABSTRACTION REQUIRES MORE

From FIFO/CO to the total order broadcast abstraction It is very important to notice that the message delivery constraints imposed by the previous FIFO and CO communication abstractions do not require the addition of control messages. The delivery constraints are on local variables and control values piggybacked by the messages. Among other features, a message that has been broadcast can be delivered by its sender immediately after it has been broadcast.

This is because the constraints on the delivery order on the messages are defined only from their causal past (which messages have been broadcast before it by the same process for FIFO order, and by any process for CO order). As we will see, this is no longer the case when one has to implement the Total Order (TO) delivery property. In that case, any pair of messages has to be delivered in the same order at any process, even if the broadcast of these messages are neither FIFO-related, nor CO-related.

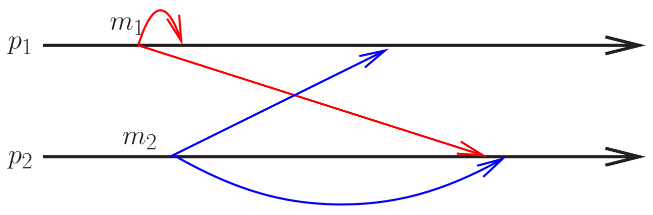

Figure 3.15: Total order message delivery requires cooperation

Let us consider the messages m_1 and m_2 broadcast in Figure 3.15. None of these broadcasts are related to the other (i.e., there is neither a FIFO nor a CO relation linking them). Hence, ensuring the Total Order message delivery property cannot rely only on control information piggybacked by the messages that are broadcast by the application. The processes have to cooperate (exchange additional control messages) to establish a common delivery order. Such an order has to be defined by both p_1 and p_2, and if m_1 is delivered first at p_1, p_2 cannot deliver m_2 just after it has been broadcast.

Actually, as we will see in Chapter 5, it is impossible to construct a total order broadcast abstraction in $\mathcal{AS}_{n,t}[\emptyset]$. This is a fundamental result of fault-tolerant distributed computing. Such a communication abstraction requires a system model strictly stronger than $\mathcal{AS}_{n,t}[\emptyset]$. From a computability point of view, the FIFO and CO broadcast abstractions are weaker than the TO broadcast abstraction: they can be implemented in a weaker system model than the one needed to implement the TO broadcast abstraction.

The FIFO and CO constructions are very general As the previous FIFO and CO reliable broadcast constructions do not use additional control messages, it follows that they work in any system where the URB communication abstraction can be built. They can consequently be used on top of the URB constructions described in the next chapter that addresses the case where, in addition to process crashes, the channels are not reliable, i.e., in systems weaker than $\mathcal{AS}_{n,t}[\emptyset]$.

3.3 BIBLIOGRAPHIC NOTES

- The problem of broadcasting messages in a reliable way in asynchronous systems prone to process failures has given rise to a large literature. Early seminal works can be found in [28, 36, 42, 77, 143]. Recent surveys can be found in [18, 44].

  A nice and very comprehensive presentation of fault-tolerant broadcast problems, their specifications and algorithms that solve them has been given by Hadzilacos and Toueg in [93].

- The causal message delivery property has been introduced by Birman and Joseph [28].

  The construction from FIFO to CO broadcast is due to Hadzilacos and Toueg [93]. The presentation we have followed is theirs.

  The second CO construction is a variant of an algorithm proposed by Raynal, Schiper and Toueg that was designed for asynchronous failure-free systems [149].

  The notion of causal message delivery has been extended to messages that carry data whose delivery is constrained by real-time requirements in [19] and to mobile environment [145].

- The total order broadcast is strongly related to the state machine replication paradigm [107, 154]. Its impossibility in asynchronous systems prone to process crashes is related to the consensus impossibility in these systems [68].

- Different types of broadcast operations are studied in [27, 59]. The books [26, 90] present distributed programming approaches based on reliable broadcast.

- The word "know" has sometimes been used in this chapter (and will be used in the next chapters) with an anthropomorphic meaning where a process "learns" facts and thereafter "knows" them. The interested reader will find formal knowledge-based approaches in [35, 94, 125]. The book [61] is entirely devoted to reasoning about knowledge.

CHAPTER 4

Uniform Reliable Broadcast Abstraction Despite Unreliable Channels

The previous chapter has presented several constructions for the uniform reliable broadcast (URB) abstraction. These constructions consider the underlying system model $\mathcal{AS}_{n,t}[\emptyset]$. They differ in the quality of service they provide to the application processes, this quality being defined with respect to the order in which the messages are delivered (namely, FIFO or CO order). This order restricts message asynchrony.

This chapter presents constructions of the URB abstraction suited to asynchronous systems prone to process crashes and unreliable channels, i.e., asynchronous system models weaker than $\mathcal{AS}_{n,t}[\emptyset]$.

4.1 A SYSTEM MODEL WITH UNRELIABLE CHANNELS

4.1.1 FAIRNESS NOTION FOR CHANNELS

Restrict the type of failures Trivially, if a channel can lose all the messages, it has to transmit from a sender to a receiver, no communication abstraction with provable guarantees can be defined and implemented. So, in order to be able to do useful computation, we need to restrict the type of failures a channel is allowed to exhibit. This is exactly what is addressed by the concept of channel *fairness*.

All the messages transmitted over a channel are *protocol messages* (let us remember that the transmission of an application message gives rise to protocol messages that are sent at the underlying abstraction layer). Several types of protocol messages can co-exist at this underlying layer, e.g., protocol messages that carry application messages and protocol messages that carry acknowledgments. In the following, we consider that each protocol message has a type, and μ is used to denote such a type. Moreover, when there is no ambiguity, the word "message" is used as a shortcut for "protocol message", and "μ-message" is used as a shortcut for "protocol message of type μ".

Fairness with respect to μ-messages Considering a uni-directional channel that allows a process p_i to send message to a process p_j, let us remark that, at the network level, process p_i can send the same message several times to p_j (for example, message retransmission is needed to overcome message losses). This channel is *fair with respect to the message type* μ if it satisfies the three following properties. All the messages that appear in these properties are messages carried by the channel from p_i to p_j.

- Validity. If the process p_j receives a μ-message (on this channel), then this message has been previously sent by p_i to p_j.
- Integrity. If p_j receives an infinite number of μ-messages from p_i, then p_i has sent an infinite number of μ-messages to p_j.
- Termination. If p_i sends an infinite number of μ-messages to p_j, and p_j is non-faulty and executes "receive () from p_i" infinitely often, it receives an infinite number of μ-messages from p_i.

As they capture similar meanings, these properties have been given the same names as in the URB specification stated in the previous chapter. The validity property means that there is neither message creation nor message alteration. The integrity property states that, if a finite number of messages of type μ are sent, the channel is not allowed to duplicate them an infinite number of times (it can nevertheless duplicate them an unknown but finite number of times). Intuitively, this means that the network performs only the retransmissions issued by the sender.

Finally, the termination property states the condition under which the channel from p_i to p_j has eventually to transmit messages of type μ, i.e., the condition under which a μ-message msg cannot be lost. This is the liveness property of the channel. From an intuitive point of view, this property states that if the sender sends "enough" μ-messages, some of these messages will be received. In order to be as less restrictive as possible, "enough" is formally stated as "an infinite number". This is much weaker than a specification such as "every 10 sendings of μ-messages, at least one is received", as this kind of specification would restrict unnecessarily the bad behavior that a channel is allowed to exhibit.

4.1.2 FAIR CHANNEL AND FAIR LOSSY CHANNEL

Fair channel The notion of "*fair* channel" encountered in the literature corresponds to the case where (1) each protocol message defines a specific message type μ, and (2) the channel is fair with respect to all the message types. Hence, the specification of a fair channel is defined by the following properties.

- Validity. If p_j receives a message msg from p_i, then msg has been previously sent by p_i to p_j.
- Integrity. For any message msg, if p_j receives msg from p_i an infinite number of times, then p_i has sent msg to p_j an infinite number of times.
- Termination. For any message msg, if p_i sends an infinite number of times msg to p_j, if p_j is non-faulty and executes "receive () from p_i" infinitely often, it receives msg from p_i an infinite number of times.

As described by the termination property, the only reception guarantee is that each message msg that is sent infinitely often cannot be lost. This means that if a message msg is sent an arbitrary but finite number of times, there is no guarantee on its reception. Let us observe that the requirement "msg sent an infinite number of times" for a message to be received, does not prevent any number of copies of msg to be lost, even an infinite number of copies to be lost (as an example, this is the case when all the even sendings of msg are lost and all the odd sendings are received).

Fair lossy channel The notion of "*fair lossy* channel" encountered in the literature corresponds to the case where all the protocol messages belongs to the same message type. Hence, there is a single message type. Hence, the specification of a fair lossy channel is defined by the following properties.

- Validity. If p_j receives a message from p_i, this message has been previously sent by p_i to p_j.
- Integrity. If p_j receives an infinite number of messages from p_i, then p_i has sent an infinite number of times to p_j.
- Termination. If p_i sends an infinite number of messages to p_j, if p_j is non-faulty and executes "receive () from p_i" infinitely often, it receives an infinite number of messages from p_i.

While the termination property states that the channel transmits messages, it gives no information on which are the messages that are received.

Comparing fair channel and fair lossy channel As we are about to see, given an infinite sequence of protocol messages, the notions of fair channel and fair lossy channel are different, and none of them includes the other one.

To that end, let us consider that the given infinite sequence of protocol messages is the infinite sequence of positive integers 1,2,etc. (each integer being a distinct message). Hence, no two messages sent by p_i are the same. If the channel from p_i to p_j is fair lossy, the termination property guarantees that p_j will receive an infinite sequence of integers (while maybe an infinite number of different integers will never be received). Differently, if the channel is fair, it is possible that no integer is ever received (this is because no integer is sent an infinite number of times).

Let us now consider that the sequence of protocol messages that is sent by p_i is the alternating sequence of 1, 2, 1, 2, 1, etc. If the channel from p_i to p_j is fair, both 1 and 2 are received infinitely often (this is because both integers are sent an infinite number of times). Differently, if the channel is fair lossy, it is possible that p_j receives the integer 1 an infinite number of times and never receives the integer 2 (or receives 2 and never receives 1).

This means that when one has to prove a construction based on unreliable channels, one has to be very careful on the type of these unreliable channels, namely, fair or fair lossy.

From a fair lossy channel to fair channel Given an infinite sequence of protocol messages msg_1, msg_2, msg_3, etc., that p_i wants to send to p_j, it is possible to construct new protocol messages (the ones that are really sent over the channel) such that each message msg_x is eventually received by p_j (if it is non-faulty) under the assumption that the channel is fair lossy.

Let msg_1 be the first protocol message that p_i wants to send to p_j. It actually sends instead the "real" protocol message $\langle msg_1 \rangle$. When it wants to sends the second protocol message msg_2, it actually sends the "real" protocol message made up of the sequence $\langle msg_1, msg_2 \rangle$. Similarly, p_i

sends the sequence $\langle msg_1, msg_2, msg_3\rangle$ when it wants to send its third protocol message msg_3, etc. Hence, the sequence of protocol messages successively sent by p_i to p_j is the sequence $\langle msg_1\rangle$, $\langle msg_1, msg_2\rangle$, $\langle msg_1, msg_2, msg_3\rangle$, etc. It follows that, in the infinite sequence of "real" protocol messages sent by p_i, all "real" protocol messages sent by p_i are different (each being a sequence that is a suffix of the sequence that constitutes the previous message). If p_j is non-faulty and the channel is fair lossy, this simple construction ensures that every msg_x is received infinitely often by p_j. Hence, assuming that p_i sends an infinite number of messages msg_x, this construction simulates a fair channel on top of a fair lossy channel. The price of this construction is the size of protocol messages that increases without bound.

4.1.3 RELIABLE CHANNEL IN PRESENCE OF PROCESS CRASHES

An abstraction for the application layer A *reliable* channel is a communication abstraction stronger than the fair channel abstraction or the fair lossy channel abstraction. Intuitively, a reliable channel neither creates, nor duplicates nor loses messages. Its definition is at the same abstraction level as the definition of the URB abstraction. It is an abstraction offered to the application layer. Consequently, it considers that the messages are application messages and that each message m is unique.

The formal definition of the reliable channel from p_i to p_j is given by the following three properties.

- Validity. If p_j receives a message m from p_i, then m has been previously sent by p_i to p_j.
- Integrity. The process p_j receives a message m at most once.
- Termination. If p_i completes the sending to p_j of k messages, then, if p_j is non-faulty and executes k times "receive () from p_i", p_j receives k messages from p_i.

This definition captures the fact that each message m sent by p_i to p_j is received exactly once by p_j. The words "p_i completes the sending of m" means that, if p_i does not crash before returning from the invocation of that send operation, the "underlying network" (i.e., the implementation of the reliable channel abstraction) guarantees that m will attain p_j. Differently, if p_i crashes during the sending of its kth message to p_j, p_j eventually receives the previous $(k-1)$ messages sent by p_i while there is no guarantee on the reception of the kth message sent by p_i to p_j (this message can be or not received by p_j).

Remark Let us notice that the termination property considers that p_j is non-faulty. This is because, if p_j crashes, due to process and message asynchrony, it is not possible to state a property on which messages can be received by p_j.

Let us also notice that it is not possible to conclude from the previous specification that a reliable channel ensures that the messages are received in their sending order (®FO reception order). This is because, once messages have been given to the "underlying network", nothing prevents it from reordering messages sent by p_i.

Reliable channel vs uniform reliable broadcast As we have seen in the previous chapter, the URB abstraction is defined with respect to the broadcast of a single application message. This means that the broadcast of the message m_1 and the broadcast of the message m_2 constitute two distinct instances of the URB problem. Said in one word, URB is a *one-shot* problem.

Differently, the reliable channel abstraction is not a one-shot problem. Its specification involves all the messages sent by a process p_i to a process p_j. The difference in the specification of both communication abstractions appears clearly in their termination properties.

4.1.4 SYSTEM MODEL

In the rest of this chapter, we consider an asynchronous system made up of n processes prone to process crashes and where each pair of processes is connected by two unreliable but fair channels (one in each direction). This system model is denoted $\mathcal{AS\_F}_{n,t}[\emptyset]$. As in the previous chapters, $\mathcal{AS\_F}_{n,t}[Prop]$ and $\mathcal{AS\_F}_{n,t}[D]$ denote the system model $\mathcal{AS\_F}_{n,t}[\emptyset]$ enriched with the property $Prop$ or a failure detector of the class D, respectively.

4.2 URB IN $\mathcal{AS\_F}_{n,t}[\emptyset]$

This section first presents an URB construction in $\mathcal{AS\_F}_{n,t}[t < n/2]$. It then shows that the property $t < n/2$ is a necessary requirement for such a construction when processes are not provided with information on the actual failure pattern.

4.2.1 A CONSTRUCTION FOR $t < n/2$

Principle Constructing the URB abstraction in $\mathcal{AS\_F}_{n,t}[t < n/2]$ is pretty simple. The construction relies on two simple base techniques.

- First, use the classical retransmission technique in order to simulate a reliable channel on top of a fair channel.
- Second, to ensure the URB termination property, locally URB-deliver an application message m to the upper application layer only when this message has been received by at least one non-faulty process. As $n > 2t$, this means that, without risking to be blocked forever, a process may URB-deliver m as soon as it knows that at least $t + 1$ processes have received a copy of m.

As a message that is URB-delivered by a process is in the hands of at least one correct process, that correct process can transmit it safely to the other processes (by repeated sendings) thanks to the fair channels that connect it to the other processes.

The construction The construction is described in Figure 4.1. When a process p_i wants to URB-broadcast a message m, it sends to itself the protocol message MSG (m) (we assume that the channel of a process to itself is reliable).

The central data structure used in the construction is an array of sets, denoted rec_by_i, where the set $rec\_by_i[m]$ is associated with the application message m. This set contains the identities of all the processes that, to p_i's knowledge, have received a copy of MSG (m).

```
operation URB_broadcast ($m$): send MSG ($m$) to $p_i$.

when MSG ($m$) is received from $p_k$:
        if (first reception of $m$)
            then allocate $rec\_by_i[m]$; $rec\_by_i[m] \leftarrow \{i, k\}$;
                   activate task $Diffuse(m)$
            else $rec\_by_i[m] \leftarrow rec\_by_i[m] \cup \{k\}$
        end if.

when $(|rec\_by_i[m]| \geq t + 1) \wedge$ ($p_i$ has not yet URB-delivered $m$): URB_deliver ($m$).

task $Diffuse(m)$:
        repeat forever
           for each $j \in \{1, \ldots, n\} \setminus rec\_by_i[m]$ do send MSG ($m$) to $p_j$ end for
        end repeat.
```

Figure 4.1: Uniform reliable broadcast in $\mathcal{AS\_F}_{n,t}[t < n/2]$ (code for p_i)

When it receives MSG (m) for the first time, p_i creates the set $rec\_by_i[m]$ and updates it to $\{i, k\}$ where p_k is the process that sent MSG (m). Then p_i activates a task, denoted $Diffuse(m)$. If it is not the first time that MSG (m) is received, p_i only adds k to $rec\_by_i[m]$. $Diffuse(m)$ is the local task that is in charge of retransmitting the protocol message MSG (m) to the other processes in order to ensure the eventual URB-delivery of m.

Finally, when it has received MSG (m) from at least one non-faulty process (this is operationally controlled by the predicate $|rec\_by_i[m]| \geq t + 1$), p_i URB-delivers m, if not yet done.

Let us remember that, as in the previous chapter, the processing associated with the reception of a protocol message is atomic, which means here that the processing of any two messages MSG ($m1$) and MSG ($m2$) are never interleaved, they are executed one after the other. This atomicity assumption, that is on any protocol message reception (i.e., whatever its MSG or ACK type) is valid in all this chapter (ACK protocol messages will be used in Section 4.4). Differently, several tasks $Diffuse(m1)$, $Diffuse(m2)$, etc., are allowed to run concurrently.

Remark It is important to see that the task $Diffuse(m)$ sends forever protocol messages (and consequently, never terminates). The only use of acknowledgments cannot prevent this infinite sending of messages, as shown by the following scenario. Let p_j be a process that has crashed before another process p_i issues URB_broadcast (m). In that case p_j will never acknowledge MSG (m), and, consequently, p_i will retransmit forever MSG (m) to p_j. Preventing these infinite retransmissions requires to provide the processes with appropriate information on failures. This is the topic addressed in Section 4.4 of this chapter.

Theorem 4.1 *The algorithm described in Figure 4.1 constructs an URB abstraction in* $\mathcal{AS\_F}_{n,t}[t < n/2]$.