```
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] ← {i, k};
            activate task Diffuse(m)
       else rec_by_i[m] ← rec_by_i[m] ∪ {k}
    end if.

when (|rec_by_i[m]| ≥ t + 1) ∧ (p_i has not yet URB-delivered m): URB_deliver (m).

task Diffuse(m):
    repeat forever
      for each j ∈ {1, ..., n} \ rec_by_i[m] do send MSG (m) to p_j end for
    end repeat.
```

Figure 4.1: Uniform reliable broadcast in $\mathcal{AS}\_\mathcal{F}_{n,t}[t < n/2]$ (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* $\mathcal{AS}\_\mathcal{F}_{n,t}[t < n/2]$.

Proof (The proof of this construction is a simplified version of the construction given in Section 4.4.) The validity property (no creation or alteration of application messages) and the integrity property (an application message is received at most once) of the URB abstraction follow directly from the text of the construction. So, we focus here on the proof of the termination property of the URB abstraction.

Let us first consider a non-faulty process p_i that URB-broadcasts a message m. We have to show that all the non-faulty processes URB-deliver m. (Let * denote this point in the proof, see below.) As p_i is non-faulty, it activates the task *Diffuse*(m) and sends MSG (m) to every other process p_j an infinite number of times. As the channels are fair, it follows that each non-faulty process receives MSG (m) and activates the task *Diffuse*(m) the first time it receives MSG (m). Hence, each non-faulty process infinitely often sends MSG (m) to every process. As there is a majority of non-faulty processes and the channels are fair, it follows that the set $rec\_by_i[m]$ eventually contains $(t+1)$ process identities. Hence, the URB-delivery condition of m becomes eventually true at every non-faulty process, which proves the theorem for the case of a non-faulty process that URB-broadcasts a message.

We have now to prove the second case of the termination property, namely, if a (non-faulty or faulty) process p_x URB-delivers a message m, then every non-faulty process URB-delivers m. If p_x URB-delivers a message m, we have $|rec\_by_x[m]| \geq t+1$, which means that at least one non-faulty process p_i has received the protocol message MSG (m). When that non-faulty process p_i has received MSG (m) for the first time, it has activated the task *Diffuse*(m). The proof is then the same as in the part of the previous case that starts at (*). $\square_{Theorem\ 4.1}$

4.2.2 AN IMPOSSIBILITY RESULT

This section shows that the assumption $t < n/2$ is a necessary requirement when one wants to construct URB in $\mathcal{AS}\_\mathcal{F}_{n,t}[\emptyset]$. The proof of this impossibility is based on an "indistinguishability" argument. (It is close to the proof of Theorem 3 of Chapter 2.)

Theorem 4.2 *There is no algorithm that constructs an URB abstraction in* $\mathcal{AS}\_\mathcal{F}_{n,t}[t \geq n/2]$.

Proof The proof is by contradiction. Let us assume that there is an algorithm A that constructs the URB abstraction in $\mathcal{AS}\_\mathcal{F}_{n,t}[t \geq n/2]$. Given $t \geq n/2$, let us partition the processes into 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 consider the following executions E_1 and E_2.

- Execution E_1. In that execution, all the processes of $P2$ crash initially, and all the processes in $P1$ are non-faulty. Moreover, a process $p_x \in P1$ issues URB_broadcast (m). Due to the very existence of the algorithm A, every process of $P1$ URB-delivers m.
- Execution E_2. In that execution, the processes of $P2$ are non-faulty, and no process of $P2$ ever issues URB_broadcast (). The processes of $P1$ behave as in $E1$: p_x issues URB_broadcast (m), and they all URB-deliver m. Moreover, after it has URB-delivered m, each process of $P1$

crashes, and all protocol messages ever sent by a process of $P1$ are lost (and consequently are never received by a process of $P2$). It is easy to see that this is possible as no process of $P1$ can distinguish this run from E_1.

Let us observe that the fact that no message sent by a process of $P1$ is ever received by any process of $P2$ is possible because the termination property associated with the fair channels that connect the processes of $P1$ to the processes of $P2$ requires that the sender of a protocol message be non-faulty to have the certainty that this message is ever received. (There is no reception guarantee for a message that is sent an arbitrary but finite number of times.)

In that execution, as no process of $P2$ ever receives a message from a process of $P1$, none of these processes can ever URB-deliver m, which completes the proof of the theorem.

□*Theorem* 4.2

Remark Let us observe that the previous impossibility result is due to the *uniformity* requirement stated in the termination property of the URB abstraction. More precisely, this property states that, if a process p_i URB-delivers a message m, then every non-faulty process has to URB-deliver m. The fact that the process p_i can be a faulty or a non-faulty process defines the uniformity requirement.

If this property is weakened into "if a non-faulty process p_i URB-delivers a message m, then all the non-faulty processes URB-deliver m", then we have the simple reliable broadcast, and the impossibility result does no longer hold. When we look at the construction of Figure 4.1, the predicate $|rec\_by_i[m]| \geq t + 1$ is used to ensure the uniformity requirement. It ensures that, when a message is URB-delivered, at least one non-faulty process has a copy of it.

4.3 THE FAILURE DETECTOR CLASS Θ

The previous impossibility result gives rise to the following natural question is the following: Which information on failures do the processes have to be provided with in order for the URB abstraction to be built whatever the value of t? This section presents the failure detector class, denoted Θ, which is the weakest failure detector class among all the failure detector classes that answer the previous question. This means that any failure detector of a class D, such that the URB abstraction can be built in $\mathcal{AS}\_\mathcal{F}_{n,t}[D]$, can be used to build a failure detector of the class Θ. Said in another way, the information on failures provided by D "contains" the information on failures provided by Θ (as seen at the end of Chapter 2, an algorithm -usually pretty sophisticated- can be needed to extract this information).

4.3.1 DEFINITION OF Θ

A failure detector of the class Θ provides each process p_i with a read-only local variable, a set denoted $trusted_i$, that satisfies the following properties. Let $trusted_i^\tau$ denote the value of $trusted_i$ at time τ. Let us remember that this notion of time is with respect to an external observer: no process has access to it. Let us also remember that $Correct(F)$ denotes the set of processes that are non-faulty in

that run. Given a run with the failure pattern F, Θ is defined by the following properties (using the notation introduced in the section devoted to a formal definition of failure detectors in Chapter 2, we have $trusted_i^\tau = H(i,\tau)$):

- Accuracy. $\forall i \in \Pi : \forall \tau \in \mathbb{N}: \big(trusted_i^\tau \cap Correct(F)\big) \neq \emptyset$.
- Liveness. $\exists \tau \in \mathbb{N}: \forall \tau' \geq \tau : \forall i \in Correct(F) : trusted_i^{\tau'} \subseteq Correct(F)$.

The accuracy property is a perpetual property that states that, at any time, any set $trusted_i$ contains at least one non-faulty process. Let us notice that this process is not required to be always the same, it can change with time. The liveness property states that, after some time, the set $trusted_i$ of any non-faulty process p_i contains only non-faulty processes.

4.3.2 URB IN $\mathcal{AS}\_\mathcal{F}_{n,t}[\Theta]$

A construction Constructing an URB abstraction in the system model $\mathcal{AS}\_\mathcal{F}_{n,t}[\emptyset]$ enriched with a failure detector of the class Θ is particularly easy. The only modification to do to the construction described in Figure 4.1 consists in replacing the "$t < n/2$" assumption by the use of the Θ failure detector. From an operational point of view, this amounts to replacing the condition to URB-deliver a message m used in Figure 4.1, namely,

$$(|rec\_by_i[m]| \geq t+1) \wedge (p_i \text{ has not yet URB-delivered } m),$$

by

$$(trusted_i \subseteq rec\_by_i[m]) \wedge (p_i \text{ has not yet URB-delivered } m).$$

The accuracy property of Θ guarantees that, when p_i URB-delivers m, at least one non-faulty process has a copy of m. As we have seen in the construction of Figure 4.1, this guarantees that the application message m that is URB-delivered can no longer be lost). The liveness property of Θ guarantees that eventually m can be locally URB-delivered (if a faulty process could remain forever in $trusted_i$, it could prevent the predicate $trusted_i \subseteq rec\_by_i[m]$) from ever becoming true).

4.3.3 COMPARING Θ AND Σ

Θ when $t < n/2$ The URB communication abstraction can be implemented in $\mathcal{AS}\_\mathcal{F}_{n,t}[t < n/2]$ without the additional power of a failure detector, and (as we have just seen) can be implemented in $\mathcal{AS}\_\mathcal{F}_{n,t}[\Theta]$, whatever the value of t. So, a natural question is the following: Can Θ be implemented in $\mathcal{AS}\_\mathcal{F}_{n,t}[t < n/2]$?

Actually, the same algorithm as the one that is described in Figure 6 of Chapter 2 (that builds Σ in $\mathcal{AS}_{n,t}[t < n/2]$) builds a failure detector of the class Θ in $\mathcal{AS}\_\mathcal{F}_{n,t}[t < n/2]$. This follows from the fact that, when $t < n/2$, any majority of processes contains always a non-faulty process, and as every non-faulty process sends forever ALIVE () messages, the fair channels connecting each pair of non-faulty processes simulate reliable channels.

Θ with respect to Σ The class of quorum failure detectors (Σ) has been introduced in Chapter 2, where it has been shown to be the weakest failure detector class that allows circumventing the impossibility to construct an atomic register in $\mathcal{AS}_{n,t}[\emptyset]$. Let us remember that Σ provides each process p_i with a set $sigma_i$ such that, after some finite time, $sigma_i$ contains only non-faulty processes, and any two quorums $sigma_i$ and $sigma_j$, each taken at any time, do intersect.

Let us consider any system where neither Σ nor Θ can be built. The following theorem shows that Θ can be built in any such system enriched with Σ, while the converse is not true (Σ cannot be built in any such system enriched with Θ).

Theorem 4.3 *In any system where $t \geq n/2$, Σ is strictly stronger than Θ (i.e., Θ can be built from Σ, but Σ cannot be built from Θ).*

Proof Let us first observe that it follows from their definitions that Σ is at least as strong as Θ. This follows from the following two observations. First, their liveness properties are the same. Second, the combination of the intersection and liveness properties of Σ implies that any set $sigma_i$ contains a correct process, which is the accuracy property of Θ. (Let us observe that this is independent of the value of t.)

The rest of the proof shows that, when $t \geq n/2$, the converse is not true, from which follows that Σ is strictly stronger than Θ in systems where $t \geq n/2$.

The proof is by contradiction. Let us assume that there is an algorithm A that, accessing any failure detector of the class Θ, builds a failure detector of the class Σ. Let us partition the processes into 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 FD be a failure detector such that, in any failure pattern in which at least one process $p_x \in P1$ (resp., $p_y \in P2$) is non-faulty, outputs p_x (resp. p_y) at all the processes of $P1$ (resp., $P2$). Moreover, in the failure patterns in which all the processes of $P1$ (resp., $P2$) are faulty, FD outputs at all the processes the same non-faulty process $\in P2$ (resp., $P1$).

It is easy to see that FD belongs to the class Θ: no faulty process is ever output (hence we have the liveness property), and at least one non-faulty process is always output at any non-faulty process (hence we have the accuracy property).

Let us consider a failure pattern F where some process $p_x \in P1$ is non-faulty, and FD outputs $trusted_x = \{x\}$, and some process $p_y \in P2$ is non-faulty, and FD outputs $trusted_y = \{y\}$. The process p_x cannot distinguish the failure pattern F from the failure pattern in which all the processes of $P2$ are faulty. Similarly, p_y cannot distinguish the failure pattern F from the failure pattern in which all the processes of $P1$ are faulty. It follows from these observations and the fact that $trusted_x \cap trusted_y = \emptyset$, that the intersection of Σ cannot be ensured, which concludes the proof of the theorem. $\square_{Theorem\ 4.3}$

The previous theorem actually shows that Σ is Θ + the non-empty intersection of any two sets that are output by Θ.

4.3.4 THE ATOMIC REGISTER ABSTRACTION VS THE URB ABSTRACTION

The atomic register abstraction is strictly stronger than the URB abstraction An immediate consequence of Theorem 4.3 is that, whatever the value of $t \geq n/2$, Θ can be built in $\mathcal{AS}_{n,t}[\Sigma]$ and $\mathcal{AS}\_\mathcal{F}_{n,t}[\Sigma]$ while Σ can be built neither in $\mathcal{AS}_{n,t}[\Theta]$ nor in $\mathcal{AS}\_\mathcal{F}_{n,t}[\Theta]$.

On another side, as we have seen in Chapter 2, when $t \geq n/2$, Σ is the weakest failure detector class that needs to be added to $\mathcal{AS}_{n,t}[\emptyset]$ in order to be able to build an atomic register. On its side, Θ is the weakest failure detector class that allows the construction of the URB communication abstraction in this type of systems.

This means that, when looking from a *failure detector class point of view*, as the atomic register abstraction requires a stronger failure detector class than the one required by URB, it is a problem strictly stronger than the URB abstraction. This is depicted in Figure 4.2 where an arrow from X to Y means that Y can be built on top of X.

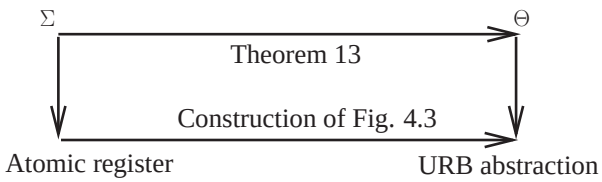

Figure 4.2: From the failure detector class Σ to the URB abstraction ($1 \leq t < n$)

A direct construction We now present a direct construction of the URB abstraction in any system where the atomic register abstraction can be built. This construction corresponds to the bottom left-to-right arrow of Figure 4.2. This shows that, in crash-prone systems, the atomic register is a strictly stronger abstraction than the URB abstraction.

```
operation URB_broadcast (m):
   sent_i ← sent_i ⊕ m; REG[i].write(sent_i).

background task T:
   repeat forever
      for each j ∈ {1, ..., n} do
         reg_i[j] ← REG[j].read();
         for each m ∈ reg_i[j] not yet URB-delivered do URB_deliver (m) end for
      end for
   end repeat.
```

Figure 4.3: From atomic registers to URB (code for p_i)

The construction uses an array of 1WMR atomic registers $REG[1..n]$ such that $REG[i]$ can be read by any process but written only by p_i. Moreover, each process p_i manages a local variable denoted $sent_i$ and an array $reg_i[1..n]$. Each atomic register $REG[x]$, and each local variable $sent_x$ or $reg_i[x]$ contains a sequence of messages. Each is initialized to the empty sequence; $\oplus$ denotes message concatenation.

To URB-broadcast a message m a process p_i appends m to the local sequence $sent_i$ and writes its new value into $REG[i]$. The URB-deliveries occur in a background task T. This task is an infinite loop that reads all the atomic registers $REG[j]$ and URB-delivers once all the messages that they contain.

Theorem 4.4 *The algorithm described in Figure 4.3 constructs a URB communication abstraction in any system in which atomic registers can be built.*

Proof As the algorithm does not forge new messages, the validity property of URB is trivial. Similarly, it follows directly from the text of the algorithm that a message is URB-delivered at most once, hence the integrity property.

For the termination property, let us observe that a non-faulty process p_i that URB-broadcasts a message m adds this message to the sequence of messages contained in $REG[i]$. Then, when p_i executes the background task T, it reads $REG[i]$, and, consequently, $reg_i[i]$ contains m. According to the text of the algorithm, p_i eventually URB-delivers m.

The previous observation has shown that, if a non-faulty process URB-broadcasts a message m, it eventually URB-delivers it. It remains to show that, if any process URB-delivers a message m, then every non-faulty process URB-delivers m.

So, let us assume that a (faulty or non-faulty) process p_x URB-delivers a message m. It follows that p_x has read m from an atomic register $REG[j]$. Due to the atomicity property of $REG[j]$, (1) the process p_j has executed a $REG[j].\mathsf{write}(sent_j)$ operation such that $sent_j$ contains m, and (2) each $REG[j].\mathsf{read}()$ operation issued after this write operation obtains a sequence that contains m. As any non-faulty process p_y reads the atomic registers infinitely often, it will obtain infinitely often m from $REG[j].\mathsf{read}()$ and will URB-deliver it, which concludes the proof of the theorem. $\square_{Theorem\ 4.4}$

4.4 QUIESCENT UNIFORM RELIABLE BROADCAST

After having introduced the quiescence property, this section introduces three failure detector classes that can be used to obtain quiescent uniform reliable broadcast algorithms. The first one is the class of *perfect* failure detectors (denoted P), the second one is the class of *eventually perfect* failure detectors (denoted $\diamond P$), and the third one is the class of *heartbeat* failure detectors (denoted HB).

It is shown that P ensures more than the quiescence property (namely, it also ensures termination, which means that there is a time after which a process knows it will never have to send more messages). The class $\diamond P$ is the weakest class of failure detectors (with bounded outputs) that allows for the construction of quiescent uniform reliable broadcast. Unfortunately, no failure detector of

the classes P and $\diamond P$ can be implemented in a pure asynchronous system (see Chapter 7). Finally, the class HB allows quiescent uniform reliable broadcast to be implemented. The failure detectors of that class have unbounded outputs, but they can be implemented in pure asynchronous systems.

4.4.1 THE QUIESCENCE PROPERTY

An infinity of protocol messages In the previous URB constructions, a correct process is required to send protocol messages forever. This is highly undesirable. This problem can easily be solved by using acknowledgments, in asynchronous systems where every channel is fair and no process ever crashes. Each time a process p_k receives a protocol message MSG (m) from a process p_i, it sends back ACK (m) to p_i, and when p_i receives this acknowledgment message it adds k to $rec\_by_i[m]$. Moreover, a process p_i keeps on sending MSG (m) only to the processes that are not in $rec\_by_i[m]$. Due to the fairness of the channels and the fact that no process crashes; eventually, $rec\_by_i[m]$ contains all the process identities, and, consequently, p_i will stop sending MSG (m).

Unfortunately, (as indicated in Section 4.2) this classical "retransmission + acknowledgment" technique does not work when processes may crash. This is due to the trivial observation that a crashed process cannot send acknowledgments, and (due to asynchrony) a process p_i cannot distinguish a crashed process from a very slow process or a process with which the communication is very slow.

The previous problem has been given the name *quiescence problem*, and solving it requires appropriate failure detectors.

Quiescent construction An algorithm that implements a communication abstraction is *quiescent* (or "satisfies the quiescence property") if each application message it has to transfer to its destination process(es) gives rise to a finite number of protocol messages.

It is important to see that the quiescence property is not a property of a communication abstraction (it does not belong to its definition). It is a property of its construction. Hence, among all the constructions that correctly implement a communication abstraction, some are quiescent while others are not.

4.4.2 QUIESCENT URB BASED ON A PERFECT FAILURE DETECTOR

This section introduces the class of perfect failure detectors, denoted P, and shows how it can be used to design a quiescent URB construction.

The class P of perfect failure detectors This failure detector class provides each process p_i with a set variable $suspected_i$ that p_i can only read. The range of this failure detector class is the set of process identities. Intuitively, at any time, $suspected_i$ contains the identities of the processes that p_i considers as having crashed.

More formally (as defined in Section 4.2 of Chapter 2), a failure detector of the class P satisfies the following properties. Let us remember that, given a failure pattern F, $F(\tau)$ denotes the set of processes that have crashed at time τ, $Correct(F)$ the set of processes that are non-faulty in the failure pattern F and $Faulty(F)$ the set of processes that are faulty in F. ($Correct(F)$ and

$Faulty(F)$ define a partition of $\Pi = \{1, \ldots, n\}$.) Moreover, let $Alive(\tau) = \Pi \setminus F(\tau)$ (the set of processes not crashed at time τ). Finally, $suspected_i^\tau$ denotes the value of $suspected_i$ at time τ.

- Completeness. $\exists \tau \in \mathbb{N}: \forall \tau' \geq \tau: \forall i \in Correct(F), \forall j \in Faulty(F): j \in suspected_i^{\tau'}$.
- Strong accuracy. $\forall \tau \in \mathbb{N}: \forall i, j \in Alive(\tau): j \notin suspected_i^\tau$.

The completeness property is an eventual property that states that there is a finite but unknown time (τ) after which any faulty process is definitely suspected by any non-faulty process. The strong accuracy property is a perpetual property that states that no process is suspected before it crashes.

It is trivial to implement a failure detector satisfying either the completeness or the strong accuracy property. Defining permanently $suspected_i = \{1, \ldots, n\}$ satisfies completeness, while always defining $suspected_i = \emptyset$ satisfies strong accuracy. The fact that, due to the asynchrony of processes and messages, a process cannot distinguish if another process has crashed or is very slow, makes it impossible to implement a failure detector of the class P without enriching the underlying unreliable asynchronous system with synchrony-related assumptions (this issue will be addressed in a deeper way Chapter 7).

P with respect to Θ A failure detector of the class Θ can easily be built in $\mathcal{AS}\_\mathcal{F}_{n,t}[P]$. This can be done by defining $trusted_i$ as being always equal to the current value of $\{1, \ldots, n\} \setminus suspected_i$.

Differently, a failure detector of the class P cannot be built in $\mathcal{AS}\_\mathcal{F}_{n,t}[\Theta]$, from which it follows that P is a failure detector class strictly stronger than Θ. This means that $\mathcal{AS}\_\mathcal{F}_{n,t}[\Theta, P]$ and $\mathcal{AS}\_\mathcal{F}_{n,t}[P]$ have the same computability power.

A quiescent URB construction in $\mathcal{AS}\_\mathcal{F}_{n,t}[\Theta, P]$ In this model, each process p_i has a read access to $trusted_i$ and $suspected_i$. Despite the fact that P is stronger than Θ, the construction that follows uses both. This is for a modularity and separation of concerns purpose.

- As we have already seen, Θ is used to ensure the second part of the termination property, namely, if a process URB-delivers a message m, any non-faulty process URB-delivers it. Hence, the "uniformity" of the reliable broadcast is obtained thanks to Θ.
- P is used to obtain the quiescence property. In later sections, P will be replaced by a weaker failure detector class.

The quiescent URB construction for $\mathcal{AS}\_\mathcal{F}_{n,t}[\Theta, P]$ is described in Figure 4.4. It is the same as the one described in Figure 4.1 (where the predicate $|rec\_by_i[m]| \geq t + 1$ is replaced by $trusted_i \subseteq rec\_by_i[m]$ to benefit from Θ) enriched with the following additional statements.

- Each time a process p_i receives a protocol message MSG (m), it systematically sends back an acknowledgment to its sender. This acknowledgment message is denoted ACK (m). Moreover, when a process p_i receives ACK (m) from a process p_k, it knows that p_k has a copy of the application message m and it consequently adds k to $rec\_by_i[m]$. (Let us observe that this would be sufficient to obtain a quiescent URB construction if no process ever crashes.)
- In order to prevent a process p_i from sending forever protocol messages to a crashed process p_j, the task *Diffuse*(m) is appropriately modified. A process p_i repeatedly sends the protocol message MSG (m) to a process p_j only if $j \notin (rec\_by_i[m] \cup suspected_i)$.

```
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] ← {i, k};
           activate task Diffuse(m)
      else rec_by_i[m] ← rec_by_i[m] ∪ {k}
    end if;
    send ACK (m) to p_k.

when ACK (m) is received from p_k: rec_by_i[m] ← rec_by_i[m] ∪ {k}.

when (trusted_i ⊆ rec_by_i[m]) ∧ (p_i has not yet URB-delivered m): URB_deliver (m).

task Diffuse(m):
    repeat
      for each j ∈ {1, ..., n} \ rec_by_i[m] do
        if (j ∉ suspected_i) then send MSG (m) to p_j end if
      end for
    until (rec_by_i[m] ∪ suspected_i) = {1, ..., n} end repeat.
```

Figure 4.4: Quiescent uniform reliable broadcast in $\mathcal{AS}\_\mathcal{F}_{n,t}[\Theta, P]$ (code for p_i)

Due to the completeness property of the failure detector class P, j will eventually appear in $suspected_i$ if p_j crashes. Moreover, due to the strong accuracy property of the failure detector class P, j will not appear in $suspected_i$ before p_j crashes (if it ever crashes).

The proof that this algorithm is a quiescent construction of the URB abstraction is similar to the proof (given below) of the construction given in Figure 4.5 for the system model $\mathcal{AS}\_\mathcal{F}_{n,t}[\Theta, HB]$. It is consequently left to the reader.

Terminating construction Let us observe that the construction of Figure 4.4 is not only quiescent but also *terminating*. Termination is a property stronger than quiescence.

More precisely, for each application message m, task $Diffuse(m)$ not only stops sending messages, but eventually terminates. This means that there is a finite time after which the predicate $(rec\_by_i[m] \cup suspected_i) = \{1, \ldots, n\}$ that control the exit of the repeat loop becomes satisfied. When this occurs, the task $Diffuse(m)$ has no longer to send protocol messages and can consequently terminate.

This is due to the properties of the failure detector class P from which we can conclude that (1) the predicate $rec\_by_i[m] \cup suspected_i = \{1, \ldots, n\}$ becomes eventually true, and (2), when it becomes true, the set $suspected_i$ contains only crashed processes (no non-faulty process is mistakenly considered as crashed by the failure detector).

As we are about to see below, the termination property can no longer be guaranteed when a failure detector of the class $\diamond P$ or HB (defined below) is used instead of a failure detector of the class P.

The class $\diamond P$ of eventually perfect failure detectors This class is the class of eventually perfect failure detectors. As the class P, that class provides each process p_i with a set $suspected_i$ that satisfies the following property: the sets $suspected_i$ can output arbitrarily values during a finite but unknown period of time, after which their outputs are the same as the ones of a perfect failure detector. More formally, $\diamond P$ includes all the failure detectors that satisfy the following properties.

- Completeness. $\exists \tau \in \mathbb{N}: \forall \tau' \geq \tau: \forall i \in Correct(F), \forall j \in Faulty(F): j \in suspected_i^{\tau'}$.
- Eventual strong accuracy. $\exists \tau \in \mathbb{N}: \forall \tau' \geq \tau: \forall i, j \in Alive(\tau'): j \notin suspected_i^{\tau'}$.

The completeness property is the same as for P: every process that crashes is eventually suspected by every non-faulty process. The accuracy property is weaker than the accuracy property of P. It requires only that there is a time after which only the faulty processes are suspected. Hence, the set $suspected_i$ of each non-faulty process eventually contains all the crashed processes (completeness) and only them (eventual strong accuracy).

As we can see, both properties are eventual properties. There is a finite anarchy period during which the values read from the sets $\{suspected_i\}_{1 \leq i \leq n}$ can be arbitrary (e.g., a non-faulty process can be mistakenly suspected, in a permanent or intermittent manner, during that arbitrarily long period of time). The class P is strictly stronger than the class $\diamond P$. It is easy to see that the classes $\diamond P$ and Θ cannot be compared, and $\diamond P$ and Σ cannot be compared either. This is because, on the one side, the definition of both Θ and Σ involves a perpetual property -a property that has to be always satisfied while the definition of $\diamond P$ involves only eventual properties, and on the other side, some correct processes can never appear in a set $trusted_i$ or $sigma_i$.

$\diamond P$-based quiescent URB A quiescent URB construction that works in $\mathcal{AS}\_\mathcal{F}_{n,t}[\Theta, \diamond P]$ is obtained by replacing the predicate that controls the termination of the task $Diffuse(m)$ in Figure 4.4, by the following weaker predicate $rec\_by_i[m] = \{1, \ldots, n\}$. This modification is due to the fact that a set $suspected_i$ no longer guarantees permanently that all the process it contains have crashed. During the finite but unknown anarchy period, these sets can contain arbitrary values. But, interestingly, despite the possible bad behavior of the sets $suspected_i$, the test $j \notin suspected_i$ (that controls the sending of a protocol message to p_j in the task $Diffuse(m)$) is still meaningful. This is due to the fact that we know that, after some finite time, $suspected_i$ will contain only crashed processes and will eventually contain all the crashed processes. It follows from the previous observation that the construction for $\mathcal{AS}\_\mathcal{F}_{n,t}[\Theta, \diamond P]$ cannot be terminating (according to the failure pattern, it is possible that the termination predicate $rec\_by_i[m] = \{1, \ldots, n\}$ be never satisfied).

4.4.3 THE CLASS HB OF HEARTBEAT FAILURE DETECTORS

The weakest class of failure detectors for quiescent communication The range of the failure detector classes P and $\diamond P$ is 2^{Π} (the value of $suspected_i$ is a set of process identities). So, their outputs are

bounded. It has been shown that $\diamond P$ is the weakest class of failure detectors (with bounded outputs) that can be used to implement quiescent reliable communication in asynchronous systems prone to process crashes and the channels of which are unreliable but fair. Unfortunately, it is impossible to implement a failure detector of the class $\diamond P$ in $\mathcal{AS}_{n,t}[\emptyset]$ and, consequently, in $\mathcal{AS}\_\mathcal{F}_{n,t}[\emptyset]$ either. To be possible, such an implementation requires that the underlying system satisfies additional synchrony assumptions.

The class HB of heartbeat failure detectors In fact, the uniformity and quiescence properties can be obtained in $\mathcal{AS}\_\mathcal{F}_{n,t}[\emptyset]$ as soon as this system is enriched with the following:

1. Either the assumption $t < n/2$ or a failure detector of the class Θ. This is used to ensure the uniformity property of message delivery (if a message is delivered by a process, it will be delivered by every non-faulty process).
2. A failure detector (of a class denoted HB) that provides each process p_i with an array of counters $HB_i[1..n]$ such that $HB_i[j]$ stops increasing only if p_j crashes. This failure detector is used to ensure the quiescence property.

Formally, a failure detector of the class HB (heartbeat) is defined by the following two properties where $HB_i^\tau[j]$ is the value of $HB_i[j]$ at time τ.

- Completeness. $\forall i \in Correct(F), \forall j \in Faulty(F): \exists K: \forall \tau \in \mathbb{N}: HB_i^\tau[j] < K$.
- Liveness.
  1. $\forall i, j \in \Pi: \forall \tau \in \mathbb{N}: HB_i^\tau[j] \leq HB_i^{\tau+1}[j]$,
  2. $\forall i, j \in Correct(F): \forall K: \exists \tau \in \mathbb{N}: HB_i^\tau[j] > K$.

The range of each entry of the array HB is the set of positive integers. Differently from $\diamond P$, this range is not bounded. The completeness property states that the heartbeat counter at p_i of a crashed process p_j (i.e., $HB_i[j]$) stops increasing, while the liveness property states that such a heartbeat counter $HB_i[j]$ (1) never decreases and (2) increases without bound if both p_i and p_j are non-faulty.

Let us observe that the counter of a faulty process increases during a finite but unknown period, while the speed at which the counter of a non-faulty process increases is arbitrary (this speed is "asynchronous"). Moreover, the values of two local counters $HB_i[j]$ and $HB_k[j]$ are not related.

Implementing HB There is a trivial implementation of a failure detector of the class HB in $\mathcal{AS}\_\mathcal{F}_{n,t}[\emptyset]$. Each process p_i manages its array $HB_i[1..n]$ (initialized to $[0, \ldots, 0]$) as follows. On the one side, p_i repeatedly sends the message HEARTBEAT (i) to each other process. On the other side, when it receives HEARTBEAT (j), p_i increases $HB_i[j]$. This very simple implementation is not quiescent. It requires the alive processes to send messages forever.

This means that HB has to be considered as a "black box" (i.e., we do not look at the way it is implemented) when we say that quiescent communication can be realized in $\mathcal{AS}\_\mathcal{F}_{n,t}[HB]$. In fact, a failure detector of a class such as P, $\diamond P$, Θ or Σ, provides a system with additional

computational power. Differently, a failure detector of a class HB constitutes an abstraction that "hides" implementation details (all the non-quiescent part is pieced together in a separate module, namely, the heartbeat failure detector).

A remark on oracles The notion of *oracle* has first been introduced as a language whose words can be recognized in one step from a particular state of a Turing machine. The main feature of such oracles is to *hide* a sequence of computation steps in a single step, or to *guess* the result of a non-computable function. They have been used to define (a) equivalence classes of problems and (b) hierarchies of problems when these problems are considered with respect to the assumptions they require to be solved.

In our case, failure detectors are oracles that provide the processes with information that depends only on the failure pattern that affects the execution in which they are used. It is important to remember that the outputs of a failure detector never depend on the computation. According to the previous terminology, we can say that the classes such as P, $\diamond P$, Θ, or Σ are classes of "guessing" failure detectors, while HB is a class of "hiding" failure detectors.

A URB construction in $\mathcal{AS}\_\mathcal{F}_{n,t}[\Theta, HB]$ A quiescent URB construction for $\mathcal{AS}\_\mathcal{F}_{n,t}[\Theta, HB]$ is described in Figure 4.5. It is nearly the same as the one for $\mathcal{AS}\_\mathcal{F}_{n,t}[\Theta, P]$ described in Figure 4.4. It differs only in the task $Diffuse(m)$. Basically, a process p_i sends the protocol message MSG (m) to a process p_j only if $j \notin rec\_by_i[m]$ (from p_i's point of view, p_j has not yet received that message), and $HB_i[j]$ has increased since the last test (from p_i'point of view, p_j is alive). The local variables $prev\_hb_i[m][j]$ and $cur\_hb_i[m][j]$ are used to keep the two last values read from $HB_i[j]$.

Theorem 4.5 *The algorithm described in Figure 4.5 is a quiescent construction of the URB communication abstraction in $\mathcal{AS}\_\mathcal{F}_{n,t}[\Theta, HB]$.*

Proof The proof of the URB validity property (no creation of application messages) and URB integrity property (an application message is delivered at most once) follow directly from the text of the construction. Hence, the rest of the proof addresses the URB termination property and the quiescence property. It is based on two preliminary claims. Let us first observe that, once added, an identity j is never withdrawn from $rec\_by_i[m]$.

Claim C1. If a non-faulty process activates $Diffuse(m)$, all the non-faulty processes activate $Diffuse(m)$.

Proof of claim C1. Let us consider a non-faulty process p_i that activates $Diffuse(m)$. It does it when it receives MSG (m) for the first time. Let p_j be a non-faulty process. There are two cases.

- There is a time after which $j \in rec\_by_i[m]$. The process p_i has added j to $rec\_by_i[m]$ because it has received MSG (m) or ACK (m) from p_j. It follows that p_j has received MSG (m). The first time it has received this protocol message, it has activated $Diffuse(m)$, which proves the claim for that case.

```
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], prev_hb_i[m], cur_hb_i[m];
           rec_by_i[m] ← {i, k};
           activate task Diffuse(m)
      else rec_by_i[m] ← rec_by_i[m] ∪ {k}
    end if;
    send ACK (m) to p_k.

when ACK (m) is received from p_k: rec_by_i[m] ← rec_by_i[m] ∪ {k}.

when (trusted_i ⊆ rec_by_i[m]) ∧ (p_i has not yet URB-delivered m): URB_deliver (m).

task Diffuse(m):
    prev_hb_i[m] ← [−1, ..., −1];
    repeat
      cur_hb_i[m] ← HB_i;
      for each j ∈ {1, ..., n} \ rec_by_i[m] do
       if (prev_hb_i[m][j] < cur_hb_i[m][j]) then send MSG (m) to p_j end if
      end for;
      prev_hb_i[m] ← cur_hb_i[m]
    until rec_by_i[m] = {1, ..., n} end repeat.
```

Figure 4.5: Quiescent uniform reliable broadcast in $\mathcal{AS}\_\mathcal{F}_{n,t}[\Theta, HB]$ (code for p_i)

- The identity j is never added to $rec\_by_i[m]$. As p_j is non-faulty, it follows from the liveness of HB that $HB_i[j]$ increases forever, from which follows that the predicate $(prev\_hb_i[m][j] < cur\_hb_i[m][j])$ is true infinitely often. It then follows that p_i sends infinitely often MSG (m) to p_j. Due to the termination property of the fair channel connecting p_i to p_j, p_j receives MSG (m) infinitely often from p_i. The first time it receives it, it activates the task $Diffuse(m)$, which concludes the proof of the claim. End of the proof of claim C1.

Claim C2. If all the non-faulty processes activate $Diffuse(m)$, they all eventually execute URB_deliver (m).

Proof of the claim C2. Let p_i and p_j be any pair of non-faulty processes. As p_i executes $Diffuse(m)$ and p_j is non-faulty, p_i sends MSG (m) to p_j until $j \in rec\_by_i[m]$. Let us observe that, due to the systematic sending of acknowledgments and the termination property of the channels, we eventually have $j \in rec\_by_i[m]$. It follows that $rec\_by_i[m]$ eventually contains all the non-faulty processes.

On another side, it follows from the liveness property of Θ that there is a finite time from which $trusted_i$ contains only non-faulty processes.

It follows from the two previous observations that, for any non-faulty process p_i, there is a finite time from which the predicate $(trusted_i \subseteq rec\_by_i[m])$ becomes true forever, and,

consequently, p_i can URB-delivers m. End of the proof of claim C2.

Proof of the termination property. Let us first show that, all the non-faulty processes URB-deliver the application message m when a non-faulty process p_i invokes URB_broadcast (m). As p_i is non-faulty, it sends the protocol message MSG (m) to itself and (by assumption) receives it. It then activates the task *Diffuse*(m). It then follows from the Claim C1 that all the non-faulty processes activate *Diffuse*(m), and from the Claim C2 that they all URB-deliver m.

Let us now show that if a (faulty or non-faulty) process p_i URB-delivers the application m, then all the non-faulty processes URB-deliver m. As p_i URB-delivers m, we have $trusted_i \subseteq rec\_by_i[m]$. Due to the accuracy property of the underlying failure detector of the class Θ, $trusted_i$ contains always a non-faulty process. Let p_j be a non-faulty process such that $j \in trusted_i$ when the delivery predicate $trusted_i \subseteq rec\_by_i[m]$ becomes true. As $j \in rec\_by_i[m]$, it follows that p_j has received MSG (m) (see the first item of the proof of Claim C1). The first time it has received such a message, p_j has activated *Diffuse*(m). It then follows from Claim C1 that all the non-faulty processes activate *Diffuse*(m), and from Claim C2 that they all URB-deliver m.

Proof of the quiescence property. We have to prove here that any application message m give rise to a finite number of protocol messages. This proof relies only on the underlying heartbeat failure detector and the termination property of the underlying fair channels.

Let us first observe that the reception of a protocol message ACK (m) never entails the sending of protocol messages. So, the proof amounts to show that the number of protocol messages of the type MSG (m) is finite. Moreover, a faulty process sends a finite number of protocol messages MSG (m), so we have only to show that the number of MSG (m) messages sent by each non-faulty process p_i is finite. Such messages are sent only inside the task *Diffuse*(m). Let p_j be a process to which the non-faulty process p_i sends MSG (m). If there is a time from which $j \in rec\_by_i[m]$ holds, p_i stops sending MSG (m) to p_j. So, let us consider that $j \in rec\_by_i[m]$ remains forever false. There are two cases.

- Case p_j is faulty. In that case, there is a finite time after which, due to the completeness property of HB, $HB_i[j]$ no longer increases. It follows that there is a finite time from which the predicate $(prev\_hb_i[m][j] < cur\_hb_i[m][j])$ remains false forever. When this occurs, p_i stops sending messages MSG (m) to p_j, which proves the case.

- Case p_j is non-faulty. We show a contradiction. In that case, the predicate $(prev\_hb_i[m][j] > cur\_hb_i[m][j])$ is true infinitely often. It follows that p_i sends infinitely often MSG (m) to p_j. Due to the termination property of the fair channel from p_i to p_j, the process p_j receives MSG (m) from p_i an infinite number of times. Consequently, it sends back ACK (m) to p_i an infinite number of times, and due to the termination property of the channel from p_j to p_i, p_i receives this protocol message an infinite number of times. At the first reception of ACK (m), p_i adds j to $rec\_by_i[m]$. As no process identity is ever withdrawn from $rec\_by_i[m]$, the

predicate $j \in rec\_by_i[m]$ remains true forever, contradicting the initial assumption, which concludes the proof of the quiescence property.

□ *Theorem* 4.5

Quiescence vs termination Differently from the quiescent URB construction for $\mathcal{AS}\_\mathcal{F}_{n,t}[\Theta, P]$ (given in Figure 4.4), but similarly to the quiescent construction for $\mathcal{AS}\_\mathcal{F}_{n,t}[\Theta, \diamond P]$, the construction described in Figure 4.5 for $\mathcal{AS}\_\mathcal{F}_{n,t}[\Theta, HB]$ is not terminating. It is easy to see that it is possible that the task *Diffuse*(m) never terminates. In fact, while quiescence concerns only the activity of the underlying network activity (due to message transfers), termination is a more general property that is concerned by the activity of both message transfers and processes.

This is due to the fact that the properties of both $\diamond P$ and HB are eventual. When $HB_i[j]$ does not change, we do not know if it is because p_j has crashed or because its next increase is arbitrarily delayed. This uncertainty is due to the net effect of asynchrony and failures. When the failure detector is perfect (class P), the "due to failures" part of this uncertainty disappears (as when a process is suspected we know for sure that it has crashed), and, consequently, a P-based construction has to cope only with asynchrony.

4.5 URB IN A SYSTEM WITH VERY WEAK CONNECTIVITY

4.5.1 THE SYSTEM MODEL $\mathcal{AS}\_\mathcal{W}_{n,t}[\emptyset]$

This section investigates a quiescent construction of the URB abstraction in a system with very weak connectivity. Basically, from the channel assumption point of view, this type of network connectivity is the weakest in the sense that it prevents the system from partitioning while allowing channels to behave unreliably.

Fair channels and unreliable channels The processes are asynchronous and may crash (as before). On the network side, we have the following. Each directed pair of processes is connected by a channel that is either fair or unreliable. An unreliable channel is similar to a fair channel as far as the validity and integrity properties are concerned, but it has no termination property. Whatever the number of times a message is sent (even an infinite number of times), the channel can lose all its messages. So, if an unreliable channel connects p_i to p_j, it is possible that no message sent by p_i is ever received by p_j on this channel, exactly as if this channel was missing.

An example of such a network is represented in Figure 4.6. A black or white big dot represents a process. A simple arrow from a process to another process represents a fair unidirectional channel. A double arrow indicates that both unidirectional channels connecting the two processes are fair. All the other channels are unreliable (in order not to overload the figure they are not represented).

Channel failure pattern Given a run, the notion of a *channel failure pattern* (denoted F_c) associated with that run is defined as follows. F_c is the set of the channels that are not fair in that run. When considering Figure 4.6, the channel failure pattern includes all the unidirectional channels that are not explicitly represented.

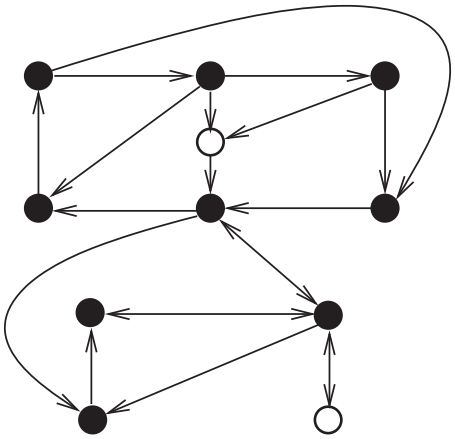

Figure 4.6: An example of a network with fair paths

Notion of fair path In order to be able to construct a communication abstraction that, in any run, allows any pair of processes that are non-faulty (in that run) to communicate, basic assumptions on the connectivity of the non-faulty processes are required. Such an assumption is based on the notion of fair path.

More precisely, we assume that, in any run characterized by the process failure pattern F and channel failure pattern F_c, every directed pair of non-faulty processes is connected by a directed path made up of non-faulty processes and fair channels. Such a path is called a *fair path*.

When considering Figure 4.6, let the black dots denote the non-faulty processes, and the white dots denote the faulty ones. One can check that every directed pair of non-faulty processes is connected by a fair path.

Notation The previous system model made up of asynchronous crash-prone processes, fair or unreliable unidirectional channel connecting any directed pair of processes, and the fair path assumption for any run, is denoted $\mathcal{AS}\_\mathcal{W}_{n,t}[\emptyset]$.

4.5.2 IMPLEMENTING HB IN $\mathcal{AS}\_\mathcal{W}_{n,t}[\emptyset]$

This section presents and proves correct a construction of a heartbeat failure detector HB in $\mathcal{AS}\_\mathcal{W}_{n,t}[\emptyset]$. It is presented before a quiescent URB construction in $\mathcal{AS}\_\mathcal{W}_{n,t}[\Theta, HB]$ because it is much simpler and is based on techniques that are re-used in the quiescent URB construction.

A forwarding technique Each process p_i periodically sends a HEARTBEAT (i) message to all processes (but itself). The fair path assumption is used to cope with the unreliability of some channels in the following way. Whenever a process receives a message HEARTBEAT (i), it forwards it to all processes (but itself and p_i). While this will ensure that, for any pair of correct processes p_i and p_j, $HB_i[j]$ will increase forever, this technique suffers a redhibitory drawback. The processes can

forward forever the very same HEARTBEAT (i) message, while p_i sent HEARTBEAT (i) only a finite number of times before crashing.

One could think that the previous problem can be solved by associating a sequence number sn with every HEARTBEAT (i) message sent by p_i. A process p_j forwards HEARTBEAT (i, sn) only if it has not yet received a heartbeat message that originated at p_i with a greater or equal sequence number. Unfortunately, this tentative solution is not satisfactory because the size of the heartbeat messages sent by the correct processes increases without bound. Even worst, it does not work in the fair channel model because a non-faulty process will not send an infinite number of times the very same message HEARTBEAT (i), but an infinite sequence of *different* messages, namely HEARTBEAT $(i, 1)$, HEARTBEAT $(i, 2)$, etc. As the reader can see, the channels should then be "fair lossy" instead of "fair" in order for the technique based on sequence numbers to work.

Hence, in order to have a construction based on fair channels, we are interested here in a construction in which the size of all messages sent by processes is bounded. This has a very interesting advantage: the message size is independent on the duration of runs (as we are about to see, this size depends only on the number of processes).

The construction A bounded construction is described in Figure 4.7. The processes send messages of the form HEARTBEAT $(path)$ whose content $path$ is a sequence of process identities. The operator $\oplus$ is used to add a process identity to the end of a sequence $path$. To ease the presentation, such a sequence $path$ is sometimes used as a set.

```
background task T:
   repeat forever
      for each j ∈ {1, ..., n} \ {i} do send HEARTBEAT (⟨i⟩) to p_j end for
   end repeat.

when HEARTBEAT (path) is received:
   for each j ∈ path do HB_i[j] ← HB_i[j] + 1 end for;
   for each j ∈ {1, ..., n} \ (path ∪ {i}) do send HEARTBEAT (path ⊕ i) to p_j end for.
```

Figure 4.7: A bounded construction of HB in $\mathcal{AS}\_\mathcal{W}_{n,t}[\emptyset]$ (code for p_i)

Each process p_i periodically sends to all the other processes a message HEARTBEAT $(path)$, where $path$ contains only its identity i. When a process p_i receives a message HEARTBEAT $(path)$, it first increases $HB_i[j]$ for each process p_j such that $j \in path$. Let $path = \langle a, b, \ldots, x \rangle$. As we can see from the text of the construction, we deduce from the message content $path = \langle a, b, \ldots, x \rangle$, that a message HEARTBEAT $(\langle a \rangle)$ originated at p_a, was then received by p_b that expanded it to HEARTBEAT $(\langle a, b \rangle)$, that was then received by p_c, etc. Hence, the previous increases of the appropriate entries of HB_i encode the fact that every process in the sequence $path$ was not crashed when it expanded and forwarded the heartbeat message.

After it has increased the appropriate entries of HB_i, p_i forwards the updated heartbeat message HEARTBEAT $(path \oplus i)$ to all the processes except itself and the processes already present in $path$ (these processes have been visited by prefixes of the message HEARTBEAT $(path \oplus i)$ that originated at p_a where $path = \langle a, b, \ldots, x \rangle$).

Notation The directed channel from p_i to p_j is denoted $\langle i, j \rangle$. A fair path of process identities $\langle a, b, \ldots, x \rangle$ is *simple* if no identity appears more than once in the path. For every fair path, there is a simple path with same source and same destination.

Theorem 4.6 *The algorithm described in Figure 4.7 is a correct construction of a failure detector of the class HB in $\mathcal{AS}\_\mathcal{W}_{n,t}[\emptyset]$. Moreover, this construction uses bounded messages.*

Proof Proof of the liveness property. The proof that, $\forall i, j$, $HB_i[j]$ never decreases, follows immediately from the text of the algorithm: the only statement where $HB_i[j]$ is updated increases its value.

To prove that, $\forall a, x \in Correct(F)$, $HB_x[a]$ is unbounded, let us first observe that, while the channel $\langle a, x \rangle$ can be unreliable, there is (by assumption) a simple fair path $\langle a, b, \ldots, w, x \rangle$. The process p_a sends infinitely often the heartbeat message HEARTBEAT $(\langle a \rangle)$ to each other process. As the channel $\langle a, b \rangle$ is fair and process p_b is non-faulty, it receives the heartbeat message HEARTBEAT $(\langle a \rangle)$ an infinite number of times. The process p_b expands each of them into HEARTBEAT $(\langle a, b \rangle)$ and forwards this heartbeat message an infinite number of times to p_c, etc., until p_x that receives an infinite number of times the message HEARTBEAT $\langle a, b, \ldots, w \rangle$. It follows that p_x increases $HB_x[a]$ an infinite number of times.

Proof of the completeness property. In order to prove this property, we first state three claims whose proofs are obvious from the text of the algorithm and, for C3, the channel integrity property.
Claim C1. If a process p_i sends HEARTBEAT $(path)$, i is the last identity in the sequence $path$.
Claim C2. Given any HEARTBEAT $(path)$ message, no process identity appears twice in $path$.
Claim C3. Let p_i and p_j be two processes and $path$ a non-empty sequence of process identities. If p_i receives the message HEARTBEAT $(path \oplus j)$ an infinite number of times, then p_j receives the message HEARTBEAT $(path)$ an infinite number of times.

We have to prove that, if p_i is a non-faulty-process and p_j is a faulty process, eventually $HB_i[j]$ stops increasing. The proof is by contradiction. Let us assume that $HB_i[j]$ increases unboundedly. This means that p_i receives an infinite sequence of messages HEARTBEAT $(path_1)$, HEARTBEAT $(path_2)$, etc., such that $\forall x$: $j \in path_x$. It follows from the claim C2 that the whole set of paths sent in all the heartbeat messages is finite, from which we conclude that there exists $path$ such that $j \in path$ and p_i receives HEARTBEAT $(path)$ an infinite number of times. Let $path = \langle i_1, i_2, \ldots, i_x \rangle$. Then, for some $k \leq x$, we have $i_k = j$. There are two cases.

- $k = x$ (j is the last process identity in $path$). It then follows from the claim C1 that p_j sent HEARTBEAT ($path$), and from the integrity property of the channel $\langle j, i \rangle$ that p_j sent infinitely often this message. This contradicts the fact that p_j is faulty (i.e., it eventually crashes).
- $k < x$ (j is not the last process identity in $path$). In that case, by repeatedly applying the claim C3 to the pairs of processes (p_{i_x}, p_i), $(p_{i_{x-1}}, p_{i_x})$, etc., until $(p_{i_k}, p_{i_{k+1}})$, we conclude that $p_{i_{k+1}}$ receives infinitely often the message HEARTBEAT ($\langle i_1, i_2, \ldots, i_k \rangle$).

  As in the previous item, it then follows from the claim C1 that p_{i_k} (i.e., p_j) is the sender of this message HEARTBEAT ($\langle i_1, i_2, \ldots, i_k \rangle$), and from the integrity property of the channel $\langle i_k, i_{k+1} \rangle$, that it sent it infinitely often to $p_{i_{k+1}}$. This contradicts the fact that p_j is faulty (i.e., it eventually crashes).

Proof of the message boundedness property. Finally, the fact that the size of the messages used in the construction is bounded follows from the claim C2. □$_{Theorem\ 4.6}$

4.5.3 URB IN $\mathcal{AS}\_\mathcal{W}_{n,t}[\Theta, HB]$

This section presents a quiescent URB construction suited to $\mathcal{AS}\_\mathcal{W}_{n,t}[\Theta, HB]$. This construction, described in Figure 4.10, uses the same local variables as the previous constructions. The part related to Θ is exactly the same as the construction presented in Figures 4.4 and 4.5; namely, $trusted_i$ is used to ensure that an application message can be URB-delivered only when a non-faulty process has a copy of it. As we have already seen, the simple replacement of the delivery predicate $trusted_i \subseteq rec\_by_i[m]$ by $|rec\_by_i[m]| \geq t + 1$ gives a construction that works in $\mathcal{AS}\_\mathcal{W}_{n,t}[t < n/2, HB]$.

How to achieve quiescence? When compared to the previous constructions that rely on fair channels, the main issue of the construction that relies on fair paths is the following. As some channels are not fair, we have to ensure that each message m URB-broadcast by a non-faulty process p_i is appropriately conveyed to every non-faulty process. To that end, taking into account the fact that there is a (unknown) fair path connecting every directed pair of non-faulty processes, p_i can repeatedly send MSG (m) to all processes (but itself), and each time a process receives MSG (m) it forwards it, etc. This flooding technique will guarantee that all the non-faulty processes will receive a copy of the message.

In order to obtain a quiescent construction, this repeated send (along the unknown fair paths) has to eventually stop. One way might be to use acknowledgment messages (as done in Figures 4.4 and 4.5). The acknowledgments sent by p_j have to be transmitted along the (unknown) fair path that connects p_j to p_i using a flooding technique similar to the one use for the messages MSG (m). The question is then: How to stop the repeated sending of these acknowledgment messages in order to obtain the quiescence property? More precisely, how to stop the permanent forwarding of ACK (j, m) messages between two processes p_x to p_y that are on the fair path connecting process p_j, that initiates the sending of the ACK (j, m) messages, and process p_i that is their final destination? A corresponding scenario is depicted in Figure 4.8 (where only fair channels between

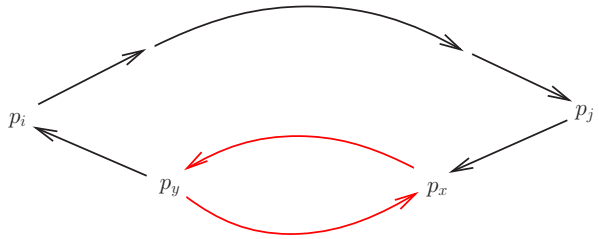

Figure 4.8: Permanent forwarding of ACK (j, m) messages between p_x and $_y$

p_i and p_j are depicted).

The solution to this problem is based on an enrichment of the protocol messages MSG () that makes useless the explicit use of acknowledgment messages. This enrichment is done as follows. Each message MSG () carries now three values. The first one is the message m itself (as in figures 4.4 and 4.5), the second one (denoted rec_by) is a set of processes that have received a copy of m, and its third field is the sequence of process identities (denoted $path$) followed by that copy of m (this is the same as the $path$ field of the heartbeat messages of Figure 4.7).

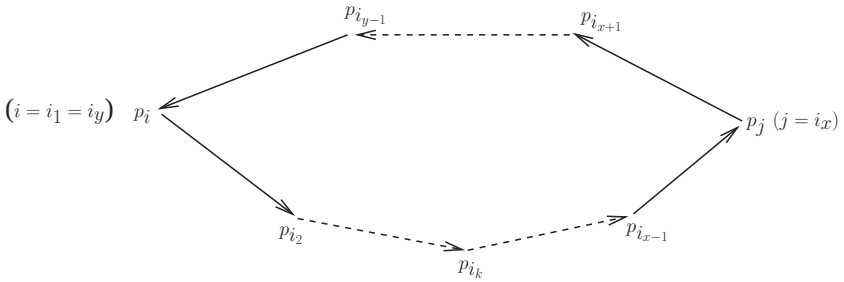

Figure 4.9: An example of fair paths connecting p_i and p_j

Let us consider Figure 4.9 to illustrate the previous items. This figure considers that the channels from p_i to p_j are not fair. Two simple fair paths are represented. Let $i_1 = i_y = i$ and $i_x = j$. A simple fair path from p_i to p_j is the sequence $\langle i_1, i_2, \ldots, i_k, \ldots, i_{x-1}, i_x \rangle$, while a simple fair from p_j to p_i is the sequence $\langle i_x, i_{x+1}, \ldots, i_{y-1}, i_y \rangle$. Let us notice that, as each fair path is simple, a process identity appears no more than once in each of them. Let us also notice that the fair path joining $p_{i_{x+1}}$ to $p_{i_{y-1}}$ can go through a process p_{i_k} that belongs to the simple fair path joining p_i to p_j. It follows that any process identity is contained at most twice in the fair path from p_i to itself $\langle i_1, \ldots, i_{x-1}, i_x, i_{x+1}, \ldots, i_y \rangle$ (that is the concatenation of the two previous simple fair paths).

The idea is that copies of m are forwarded from p_i to p_j through the simple fair path $\langle i_1, \ldots, i_x \rangle$, and their acknowledgments are forwarded from p_j to p_i through the simple fair path $\langle i_x, \ldots, i_y \rangle$. As both the copies of m and their acknowledgments are carried by MSG () messages, these messages are launched at p_i, have their rec_by and $path$ fields enriched while they progress on the concatenation of the fair paths, and return to p_i, thereby acknowledging the receipt of m by p_j.

```
operation URB_broadcast (m): send MSG (m, ∅, ϵ) to p_i.

when MSG (m, rec_by, path) is received from p_k:
   if (first reception of m)
      then allocate rec_by_i[m], prev_hb_i[m], cur_hb_i[m];
           rec_by_i[m] ← {i};
           activate task Diffuse(m)
   end if;
   rec_by_i[m] ← rec_by_i[m] ∪ rec_by;
   dest_i ← {x | x ∈ (Π \ {i}) ∧ x appears at most once in path};
   for each j ∈ dest_i do send MSG (m, rec_by_i[m], path ⊕ i) to p_j end for.

when (trusted_i ⊆ rec_by_i[m]) ∧ (p_i has not yet URB-delivered m): URB_deliver (m).

task Diffuse(m):
   prev_hb_i[m] ← [−1, ..., −1];
   repeat
     cur_hb_i[m] ← HB_i;
     if (∃k : k ∉ rec_by_i[m]) ∧ (prev_hb_i[m][k] < cur_hb_i[m][k]) then
      for each j such that (prev_hb_i[m][j] < cur_hb_i[m][j])
                do send MSG (m, rec_by_i[m], ⟨i⟩) to p_j
      end for;
      prev_hb_i[m] ← cur_hb_i[m]
     end if
   until rec_by_i[m] = {1, ..., n} end repeat.
```

Figure 4.10: Quiescent uniform reliable broadcast in $\mathcal{AS}\_\mathcal{W}_{n,t}[\Theta, HB]$ (code for p_i)

The construction As already indicated, the construction is described in Figure 4.10. Its skeleton is the same as the one of the previous constructions. As before, $\oplus$ denotes concatenation of a new process identity at the end of a sequence (path), and ϵ denotes the empty sequence. We explain here only the behavior of a process p_i when it receives a protocol message MSG $(m, rec\_by, path)$, and the behavior of the task $Diffuse(m)$.

In the task $Diffuse(m)$, p_i periodically checks if there is a process p_k that, from its local point of view, has not yet received a copy of m (predicate $k \notin rec\_by_i[m]$) and is not crashed (predicate $prev\_hb_i[m][k] < cur\_hb_i[m][k]$). If there is a such a process, p_i sends MSG $(m, rec\_by_i[m], \langle i \rangle)$ to each process p_j that it considers as not crashed. Let us notice that the $path$ field of this message is the sequence $\langle i \rangle$, expressing the fact that this message originated at p_i. It is important to notice

that p_i sends MSG $(m, rec\_by_i[m], \langle i \rangle)$ to all the processes that it considers alive and not only to the processes p_k such that $k \notin rec\_by_i[m]$. As we will see in the proof, this is done in order to benefit from the termination property of fair channels (that requires a sender to send an infinite number of times the very same message in order that the destination process receives it).

When p_i receives MSG $(m, rec\_by, path)$ for the first time, it initializes $rec\_by_i[m]$ to $\{i\}$ and activates the task $Diffuse(m)$. In all cases, p_i updates $rec\_by_i[m]$ with what it learns from the message field rec_by. Then, after it has computed the set $dest_i$ of the processes to which it has to forward MSG $(m, rec\_by_i[m], path \oplus i)$, p_i forwards that enriched message. The set $dest_i$ is made up of all the processes but p_i itself and the processes that appear twice or more times in $path$ (let us remember that a process identity can appear at most twice in the concatenation of two simple fair paths that connect two processes, one path in each direction; see our previous discussion concerning Figure 4.9).

4.5.4 PROOF OF THE CONSTRUCTION FOR $\mathcal{AS}\_\mathcal{W}_{n,t}[\Theta, HB]$

This section proves that the algorithm given in Figure 4.10 constructs a quiescent URB abstraction in the system model $\mathcal{AS}\_\mathcal{W}_{n,t}[\Theta, HB]$. The structure of the proof is similar to the structure of the proof of Theorem 4.5. The new technical issue consists in replacing the "fair channel" assumption by the "fair path" assumption. Lemmas that capture fundamental properties of the construction are first given.

Lemma 4.7 $\forall(m, path)$: *there is a finite number of distinct messages* MSG $(m, -, path)$.

Proof The field rb in MSG $(m, rb, path)$ is a set of process identities. As there are n processes, the number of possible sets rb is bounded. $\square_{Lemma\ 4.7}$

Lemma 4.8 *If* p_i *sends* MSG $(m, -, path)$, *no process identity appears more than twice in* $path$.

Proof The lemma follows from the following observations: (a) a process forwards a message only to processes whose identity appears at most once in the field $path$ and (b) a process that sends a message adds it identity to the end of the field $path$ of the message it sends. $\square_{Lemma\ 4.8}$

Lemma 4.9 $\forall i, j, m$: *(1) the first value of* $rec\_by_i[m]$ *is* $\{i\}$; *(2) if* j *is ever added to* $rec\_by_i[m]$, *it is never withdrawn thereafter; (3) if* $j \in rec\_by_i[m]$, *then* p_j *has received a copy of* m.

Proof The lemma follows directly from the text of the algorithm. $\square_{Lemma\ 4.9}$

Lemma 4.10 *If a non-faulty process* p_i *activates* $Diffuse(m)$, *every non-faulty process* p_j *activates* $Diffuse(m)$.

Proof We consider two cases according to the fact that there is a fair channel from p_i to p_j or not.

Case 1: There is a fair channel from p_i to p_j. Let us assume by contradiction that p_j never activates *Diffuse*(m). As p_j adds (for the first time) j into $rec\_by_j[m]$ only when it receives MSG $(m, -, -)$, and as p_j does not invoke *Diffuse*(m), it follows that we cannot have $j \in rec\_by_j[m]$. Hence, no message MSG $(m, rec\_by, -)$ can then be such that $j \in rec\_by$. It follows that we can never have $j \in rec\_by_i[m]$. As both p_i and p_j are non-faulty, it follows from the liveness property of the underlying heartbeat failure detector that the predicate $prev\_hb_i[m][j] < cur\_hb_i[m][j]$ is true infinitely often. Consequently, p_i sends to p_j messages MSG $(m, -, \langle i \rangle)$ an infinite number of times. As, due to Lemma 4.7, there is a finite number of such messages, it follows that there is a set rb such that p_i sends the message MSG $(m, rb, \langle i \rangle)$ to p_j an infinite number of times. By the termination property of the fair channel $\langle i, j \rangle$, p_j eventually receives such a message and activates the task *Diffuse*(m), which shows a contradiction.

Case 2: There is no fair channel from p_i to p_j. Let $\langle i_1, i_2, \ldots, i_{x-1}, i_x \rangle$ (with $i_1 = i$ and $i_x = j$) be a fair path from p_i to p_j. It follows from the previous case, that p_{i_2} activates *Diffuse*(m). Considering the fair channel $\langle i_2, i_3 \rangle$, it follows that p_{i_3} activates *Diffuse*(m), etc., until p_j that activates *Diffuse*(m). □*Lemma* 4.10

Lemma 4.11 *Let p_i and p_j be two non-faulty processes. If p_i invokes Diffuse(m), then eventually the predicate $j \in rec\_by_i[m]$ holds forever.*

Proof Let p_i and p_j be two non-faulty processes. Let us first notice that, due to part (2) of Lemma 4.9, if j is added to $rec\_by_i[m]$, the predicate $j \in rec\_by_i[m]$ holds forever. Hence, the proof consists in showing that j is eventually added to $rec\_by_i[m]$.

The proof is by contradiction. Let us suppose that j is never added to $rec\_by_i[m]$. The notations are exactly the same as in Figure 4.9, namely: $i = i_1 = i_y$ and $i_x = j$; $\langle i_1, \ldots, i_x \rangle$ is a simple fair path from p_i to p_j, and $\langle i_x, \ldots, i_y \rangle$ is a simple fair path from p_j to p_i. Moreover, for $1 \leq \alpha < y$, let $P_\alpha = \langle i_1, \ldots, i_\alpha \rangle$. Let us notice that, as both paths $\langle i_1, \ldots, i_x \rangle$ and $\langle i_x, \ldots, i_y \rangle$ are simple, no process identity can appear more than twice in P_α. This allows us to conclude that the process identity $i_{\alpha+1}$ appears at most once in P_α.

Claim C. For each $\alpha \in \{1, \ldots, y-1\}$, there is a set rb_{i_α}, containing $\langle i_1, \ldots, i_\alpha \rangle$, such that p_{i_α} sends MSG $(m, rb_{i_\alpha}, P_\alpha)$ to $p_{i_{\alpha+1}}$ an infinite number of times.

Assuming the claim C is true, let us consider $\alpha = y - 1$. It follows from the claim and the termination property of the fair channel from p_{i_α} $(= p_{i_{y-1}})$ to p_{i_y} $(= p_i)$ that p_i eventually receives a message MSG $(m, rb_{i_{y-1}}, P_{y-1})$. When p_i receives such a message, it adds $rb_{i_{y-1}}$ to $rec\_by_i[m]$. As $rb_{i_{y-1}}$ contains $\{i_1, \ldots, i_{y-1}\}$, and $i_x \in \{i_1, \ldots, i_{y-1}\}$ (see Figure 4.9), we have $i_x \in rec\_by_i[m]$, i.e., $j \in rec\_by_i[m]$, which contradicts the assumption that j never belongs to $rec\_by_i[m]$ and proves the lemma.

Proof of the claim C. The claim is proved by induction on α. The base case involves p_{i_1} $(= p_i)$ and p_{i_2}. Let us remember that both p_{i_2} and p_j are non-faulty. We consequently have the following:
(1) the predicate $(j \notin rec\_by_i[m]) \wedge (prev\_hb_i[m][j] < cur\_hb_i[m][j])$ is true infinitely often ($j \notin rec\_by_i[m]$ comes from the contradiction assumption, while $prev\_hb_i[m][j] < cur\_hb_i[m][j]$ comes from the accuracy of the underlying heartbeat failure detector),
(2) the predicate $prev\_hb_i[m][i_2] < cur\_hb_i[m][i_2]$ is true infinitely often (this comes from the accuracy of the underlying heartbeat failure detector; let us observe that i_2 can be such that $i_2 \in rec\_by_i[m]$),
(3) the channel from p_{i_1} $(= p_i)$ to p_{i_2} is fair.
It follows from these observations that p_{i_1} $(= p_i)$ sends messages of the form MSG $(m, -, p_{i_1})$ to p_{i_2} an infinite number of times. Hence, due to Lemma 4.7, there is a set rb_{i_1} such that p_i sends MSG (m, rb_{i_1}, i_1) to p_{i_2} an infinite number of times. Finally, due to the parts (1) and (2) of Lemma 4.9, we have $i_1 \in rb_{i_1}$, which concludes the proof of the base case.

For the induction step, let us consider that for $1 \leq \alpha < x - 1$, p_{i_α} sends MSG $(m, rb_{i_\alpha}, P_\alpha)$ to $p_{i_{\alpha+1}}$ an infinite number of times, where rb_{i_α} contains $\{i_1, i_2 \ldots, i_\alpha\}$. As the channel from p_{i_α} to $p_{i_{\alpha+1}}$ is fair, $p_{i_{\alpha+1}}$ receives MSG $(m, rb_{i_\alpha}, P_\alpha)$ from p_{i_α} an infinite number of times. As $p_{i_{\alpha+2}}$ is non-faulty and appears at most once in $P_{\alpha+1}$, it follows that each time $p_{i_{\alpha+1}}$ receives MSG $(m, rb_{i_\alpha}, P_\alpha)$, it sends a message of the form MSG $(m, -, P_{\alpha+1})$ to $p_{i_{\alpha+2}}$. It is easy to see from the text of the algorithm that each message MSG $(m, rb, P_{\alpha+1})$ is such that rb contains rb_{i_α} and $i_{\alpha+1}$. It then follows from Lemma 4.7 that there is a set $rb_{i_{\alpha+1}}$ such that this set contains $\{i_1, \ldots, i_{\alpha+1}\}$, and $p_{i_{\alpha+1}}$ sends MSG $(m, rb_{i_{\alpha+1}}, P_{\alpha+1})$ to $p_{i_{\alpha+2}}$ an infinite number of times, which concludes the proof of the claim. □*Lemma 4.11*

Lemma 4.12 *If all the non-faulty processes activate Diffuse(m), they all eventually execute* URB_deliver *(m).*

Proof Let p_i and p_j be two non-faulty processes. It follows from Lemma 4.11 that we eventually have $j \in rec\_by_i[m]$, from which we conclude that $rec\_by_i[m]$ eventually contains all non-faulty processes. Moreover, it follows from part (2) of Lemma 4.9 that no identity is ever withdrawn from $rec\_by_i[m]$.

On another side, the liveness property of Θ states that there is a finite time from which $trusted_i$ contains only non-faulty processes. It follows that, after some unknown but finite time instant, the delivery predicate $trusted_i \subseteq rec\_by_i[m]$ remains true forever, which entails the URB-delivery of m by p_i. □*Lemma 4.12*

Theorem 4.13 *The algorithm described in Figure 4.10 is a construction of the URB abstraction.*

Proof The proof of the validity property and the integrity property of the URB abstraction (no creation or alteration of application messages for validity, and an application message is URB-delivered at most once for integrity) are obvious. The rest of the proof of the termination property

of the URB abstraction is the same as the proof done in Theorem 4.5. It is repeated here for completeness.

Let us first consider the case of a non-faulty process p_i that URB-broadcasts an application message m. As p_i is non-faulty, it invokes *Diffuse*(m). By Lemma 4.10, all the non-faulty processes eventually invoke *Diffuse*(m), and by Lemma 4.12, they all URB-deliver m.

Let us now show that if a (faulty or non-faulty) process p_i URB-delivers m, then all the non-faulty processes URB-deliver m. As p_i URB-delivers m, we have $trusted_i \subseteq rec\_by_i[m]$. It follows from the accuracy property of Θ (namely, $trusted_i$ always contains a -possibly changing- non-faulty process), that there is non-faulty process p_j such that $j \in rec\_by_i[m]$. It then follows from part (3) of Lemma 4.9 that p_j has a copy of m. The first time it received a message MSG $(m, -, -)$, p_j has activated the task *Diffuse*(m). Then, due to Lemmas 4.10 and 4.12, all the non-faulty processes URB-deliver m, which concludes the proof of the theorem. $\square_{Theorem\ 4.13}$

Theorem 4.14 *The algorithm described in Figure 4.10 is a quiescent construction.*

Proof Let us first show that if a non-faulty process p_i activates the task *Diffuse*(m), it eventually stops sending messages in that task. Let p_j be another process. There are two cases.

- If p_j is non-faulty, due to Lemma 4.11, there is a finite time after which $j \in rec\_by_i[m]$ holds forever.
- If p_j is faulty, due to the completeness property of the underlying heartbeat failure detector, there is a finite time after which the predicate $(prev\_hb_i[m][j] < cur\_hb_i[m][j])$ remains forever false.

Therefore, there is a finite time after which the predicate $(j \notin rec\_by_i[m]) \wedge (prev\_hb_i[m][j] < cur\_hb_i[m][j])$ is always false, which proves that eventually p_i stops sending messages in the task *Diffuse*(m).

We have now to prove that any process eventually stops sending messages of the form MSG $(m, -, -)$. The proof is by contradiction. Let us suppose that there is a process p_i that never stops sending messages of the form MSG $(m, -, -)$. Due to Lemma 4.8 and the fact that the number of processes is bounded, the third component of a message of the form MSG $(m, -, -)$ ranges over a finite set of values. Therefore, there is some fixed $path$ such that p_i sends an infinite number of times the same message MSG $(m, -, path)$. Finally, due to Lemma 4.7, there is a fixed value rb, such that p_i sends an infinite number of times the same message MSG $(m, rb, path)$.

Among all the messages that are sent infinitely often, let $path_0$ be a shortest one (there can be several that have the shortest length), and let p_i be the corresponding sending process. Let us notice that p_i is non-faulty. Due the first part of the proof, there is a time after which p_i stops sending messages in its task *Diffuse*(m). It follows that p_i sends messages of the form MSG $(m, -, path_0)$ only when it receives a message. Let MSG $(m, -, path_1)$ be such a message. Due to the text of the algorithm, we have $path_0 = path_1 \oplus i$. Let $path_1 = \langle \ldots, j \rangle$. As p_i sends an infinity of messages of

the form MSG $(m, -, path_0)$, it has received an infinity of messages of the form MSG $(m, -, path_1)$ from p_j. Due to the integrity property of the channel from p_j to p_i, p_j has sent messages of the form MSG $(m, -, path_1)$ to p_i. But $path_1$ is strictly shorter than $path_0$, which contradicts the minimality of $path_0$ and concludes the proof of the quiescence property. $\square_{Theorem\ 4.14}$

4.6 BIBLIOGRAPHIC NOTES

- The concept of failure detectors has been introduced by Chandra and Toueg in [34] where they have defined, among other failure detector classes, the classes P and $\diamond P$. The class P has been shown to be the weakest class of failure detectors to solve some distributed computing problems in [46, 95].
- The oracle notion in sequential computing is presented in numerous textbooks. Among other books, the reader can consult [78, 100].
- The weakest failure detector class Θ that allows to construct an URB abstraction despite asynchrony, the fact that any number of processes can crash and the channels are only fair, has been proposed by Aguilera, Toueg and Deianov [12].

  The relation between Θ and Σ has been established in [47].
- The notion of quiescent communication and the heartbeat failure detector class have been introduced by Aguilera, Chen and Toueg in [3, 5]. These notions have been investigated in [4] in the context of partitionable networks.

  The very weak communication model and the corresponding quiescent URB construction, that has been presented in the last section, have been introduced in [5].
- When we consider a system as simple as the one made up of two processes connected by a bidirectional channel, there are impossibility results related to the effects of process crashes, channel unreliability, or the constraint to use only bounded sequence numbers. Chapter 22 of Lynch's book [117] presents an in-depth study of the power and limit of unreliable channels.
- The effects of lossy channels on problems in general, in asynchronous systems that are not enriched with failure detectors, is addressed in [20].
- Given two processes that (a) can crash and recover, (b) have access to volatile memory only, and (c) are connected by a (physical) *reliable* channel, let us consider the problem that consists in building a (virtual) reliable channel connecting these two (possibly faulty) processes. Maybe surprisingly, this problem is impossible to solve [62]. This is mainly due to the absence of stable storage.

  It is also impossible to build a reliable channel when the processes are reliable (they never crash) and the underlying channel can duplicate and reorder messages (but cannot create or lose messages), and only bounded sequence numbers can be used [157].

  Differently, if processes do not crash and the underlying channel can lose and reorder messages but cannot create or duplicate messages, it is possible to build a reliable channel, but this construction is highly inefficient [1].

PART III

Agreement Abstractions

CHAPTER 5

The Consensus Abstraction

Consensus is one of the most fundamental problems in fault-tolerant distributed computing. It states that the processes that do not fail have to agree in a non-trivial way. This agreement takes the form of a value (called *decision value*) that has to be one of the values proposed by the processes to the consensus instance.

After having defined the consensus problem, this chapter investigates its use to build a total order uniform reliable broadcast abstraction, and shows that both problems are in fact equivalent. It also shows how, in an appropriate distributed system, it can also help solving the non-blocking atomic commit problem. This problem considers a global computation decomposed into a set of local computations, each associated with a process. After having executed their local computations, the processes have to collectively commit or abort their local computations. To that end, according to its local execution, each process issues a "yes/no" vote. If there is a "no" vote, all local computations have to be aborted. If all processes vote "yes" and there is no failure, each of them has to commit its local computation.

Finally, the chapter shows that consensus cannot be solved in pure asynchronous distributed systems prone to process crashes (i.e., in $\mathcal{AS}_{n,t}[\emptyset]$). This is the famous FLP impossibility result (named after Fischer, Lynch and Paterson who proved it). This impossibility is remarkable in the sense that it holds (1) even if the processes want to agree on a single bit, and (2) for a very benign type of failures (at most one process may fail and the failure consists in a premature stopping; moreover the channels are reliable). Hence, this impossibility exhibits a fundamental limitation of asynchronous message-passing systems in presence of process crash failures. An immediate consequence is that harder problems such as leader election and mutual exclusion cannot be solved either in $\mathcal{AS}_{n,t}[\emptyset]$. The next chapter will address how $\mathcal{AS}_{n,t}[\emptyset]$ has to be enriched in order for consensus to become solvable.

5.1 THE CONSENSUS ABSTRACTION

5.1.1 THE NEED FOR AGREEMENT

A simple agreement problem is the following coordination problem (called *unique action* problem). Each process proposes an action to execute, and they all have to execute once the very same action that has to be one of the actions that has been proposed.

Another example of agreement problem consists in enriching the uniform reliable broadcast (URB) abstraction investigated in the previous chapters with the following additional delivery prop-

erty: the messages have to be delivered in the same order at all the non-faulty processes (this defines a message delivery sequence S), and the sequence of messages delivered by a faulty process has to be a prefix of S. (This is the *total order* URB abstraction informally discussed at the end of Chapter 3).

To attain this goal, the processes have to agree (in one way or another) on a common delivery order. One solution could be to ask one process to act as a server that would establish the common delivery order. But what to do when this process crashes? Elect another process? The processes will then have to execute a leader election protocol to agree on the same new server process, that has to be a non-faulty process. This approach only "moves" the problem of agreeing on a common delivery order to the problem of agreeing on a new server. In both cases, an agreement problem has to solved.

It appears that the unique action problem, the construction of a common message delivery order (and several other coordination problems) are particular instances of a more generic problem, called *consensus* problem. As we are about to see, consensus is one of the most fundamental problems of distributed computing in presence of failures.

5.1.2 THE CONSENSUS PROBLEM

Definition In the consensus problem, each process p_i proposes a value v_i and all processes have to agree (we also say "decide") on the same value that has to be one of the proposed values. In a more precise way, this problem is defined by the following properties.

- Validity. If a process decides a value v, then v has been proposed by some process.
- Integrity. A process decides at most once.
- Agreement. No two processes decide different values.
- Termination. Each correct process decides.

Let A be an algorithm that is assumed to solve the consensus problem. This algorithm is correct if all its runs satisfy the previous properties. Validity, integrity and agreement define the consensus safety properties, while termination defines its liveness property.

Binary vs multivalued consensus Let V be the (finite or infinite) set of values that can be proposed by the processes. If $|V| = 2$, only two values (usually denoted 0 and 1) can be proposed, and the consensus is *binary*. When $|V| > 2$, the consensus is *multivalued*.

Consensus object Each instance of the consensus problem defines a concurrent object CS that provides processes with a single operation denoted CS.**propose**(v), where v is the value proposed by the invoking process. That operation returns to the invoking process the value decided by the consensus object. In that sense, a consensus object can be seen as a type of write-once object where the propose(v) operation atomically tries to write v and always returns the value of the object. Only one write succeeds. As a process can invoke C.propose(v) only once, the consensus problem is a *one-shot* agreement problem.

When considering, the "object terminology", the interface of a consensus object is the method propose(). That method returns a value (namely the same value) to every process that involves it.

Remark While consensus can be solved in a trivial way in failure-free asynchronous systems (e.g., deciding the value proposed by the process with the smallest identity), this is no longer the case in failure-prone systems. The uncertainty created by the combined effect of asynchrony, concurrency, and failures makes the design of consensus algorithms in pure asynchronous systems such as $\mathcal{AS}_{n,t}[\emptyset]$ not only very difficult but (as we will see later) impossible.

5.1.3 WHY CONSENSUS IS FUNDAMENTAL

In a lot of distributed computing applications, the processes have to agree on the state of the application (or an appropriate part of it). Consensus can be used to help the processes solve easily this problem as follows.

Each process p_i first computes its local view of the system state. This view can be obtained from messages exchanged with other processes (as an example, in $\mathcal{AS}_{n,t}[\emptyset]$ a process can wait for messages from $n - t$ processes without being permanently blocked). Once computed, its local view of the system state becomes the value it proposes to a consensus instance. Then, thanks to the consensus properties, the very same view is adopted by all the processes that are alive. In that way, all these processes can continue from the same consistent view of the computation. This view is consistent in the sense that (1) it is not arbitrary (it has been computed by a process and is related to the past computation), and (2) no two processes can continue with different views. (If they apply the same deterministic function to that view, they will obtain the same result.)

5.1.4 CONSENSUS AND NON-DETERMINISM

While the specification of the consensus problem states that no two processes can decide different values, it does not state which value is decided. It only states that the decided value is one of the proposed values. Due to the uncertainty created by asynchrony and failures, that value cannot always be computed from a deterministic function on the proposed values. In that sense, a consensus object solves some form of non-determinism.

Actually, due to net effect of asynchrony and failures, the value decided depends on the run. (Differently, fact(n) $= n!$ is deterministic. Given n, whatever the way fact() is implemented and whatever the run, the value of fact(n) is unique.)

5.2 THE TOTAL ORDER URB COMMUNICATION ABSTRACTION

5.2.1 THE TO-URB ABSTRACTION: DEFINITION

Similarly to the FIFO (first in, first out) and CO (causal order) URB communication abstractions, the TO-URB abstraction is the uniform reliable broadcast enriched with a delivery property stating that the processes deliver the messages in the same order (TO stands for "total order").

Definition "TO_broadcast ()" and " TO_deliver ()" being its two communication operations, the TO-URB abstraction is defined by the following properties. Let us remember that, in the family of the URB abstractions, each application message m is unique, and $m.sender$ denotes the identity of the process that has issued the broadcast of m.

- Validity. If a process TO-delivers a message m, then m has been TO-broadcast by some process.
- Integrity. A process TO-delivers a message m at most once.
- Total order message delivery. If a process TO-delivers a message m and then TO-delivers a message m', then no process TO-delivers m' unless it has previously TO-delivered m.
- Termination. (1) If a non-faulty process TO-broadcasts a message m, or (2) if a process TO-delivers a message m, then each non-faulty process TO-delivers the message m.

Hence, TO-URB = URB + TO message delivery. It follows that, as FIFO-URB and CO-URB, TO-URB is not a one-shot problem. Its specification involves all the messages that are TO-broadcast.

Adding FIFO or CO to the TO message delivery property While the URB abstraction requires that any two non-faulty processes deliver the same set S of messages, and that a faulty process delivers a subset of S, the TO-URB abstraction requires that the non-faulty processes deliver the same sequence of messages, and that a faulty process delivers a prefix of that sequence.

As URB can been enriched with FIFO (or CO) message delivery, it is possible to require that (1) the messages be delivered in the same order, (2) this total delivery order respecting the local FIFO order for each sender process, or the global CO order, whose definitions are recalled below.

- FIFO message delivery. If a process FIFO-broadcasts a message m and then FIFO-broadcasts a message m', no process FIFO-delivers m' unless it has FIFO-delivered m before.
- CO message delivery. (Let us remember that "$\rightarrow_M$" denotes the causality precedence relation defined on the messages.) If $m \rightarrow_M m'$, no process CO-delivers m' unless it has previously CO-delivered m.

We then obtain a TO+FIFO-URB communication abstraction or the stronger TO+CO-URB communication abstraction. Algorithms similar to the ones described in Section 2 of Chapter 3 can be designed to build a TO+FIFO-URB abstraction and a TO+CO-URB abstraction from a TO-URB abstraction. These algorithms correspond to the horizontal dotted arrows at the bottom of Figure 5.1. It is also possible to design "direct" constructions for the two dotted vertical arrows.

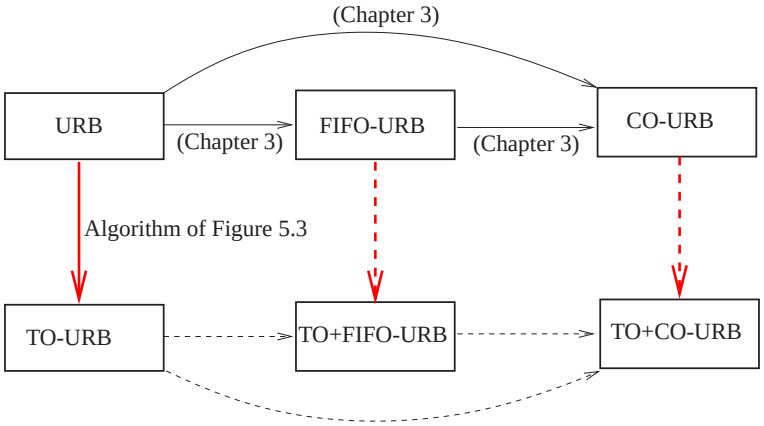

Figure 5.1: Adding total order message delivery to various URB abstractions

The fundamental missing link The important point here is that, unfortunately, going from URB to TO-URB (and more generally designing constructions corresponding to the three top-down arrows of Figure 5.1) cannot be done within $\mathcal{AS}_{n,t}[\emptyset]$. The net effect of asynchrony and crashes makes it impossible. This impossibility will be formally addressed in Section 5.6.

Delivering the messages according to causal order is possible in $\mathcal{AS}_{n,t}[\emptyset]$ because (a coding of) the causal past of each message can be attached to it. This is no longer possible when one wants to deliver the messages according to a total order. Intuitively, this is because ordering in the same way, at each process, the delivery of messages whose broadcast are unrelated requires synchronization that cannot be implemented in presence of asynchrony and failures. Additional computing power from the underlying system is needed, which means that $\mathcal{AS}_{n,t}[\emptyset]$ has to be enriched in order TO-URB can be built. As we are about to see, this enrichment consists in consensus objects.

5.2.2 FROM CONSENSUS TO THE TO-URB ABSTRACTION

This section describes a TO-URB construction that works in $\mathcal{AS}_{n,t}[CONS]$, i.e., $\mathcal{AS}_{n,t}[\emptyset]$ enriched with consensus objects. This is not counter-intuitive as TO-URB pieces together a communication problem (URB abstraction) and an agreement problem (the definition of a common delivery order).

Structure of the construction The structure of the construction is described in Figure 5.2. The middleware layer implementing the construction is defined by the algorithm described in Figure 5.3 that assumes an underlying URB-broadcast abstraction (that, as we have seen in Chapter 3) can be built in $\mathcal{AS}_{n,t}[\emptyset]$ and an unbounded number of consensus objects $CS[1..]$ shared by the processes.

Description of the construction Each process p_i manages three local variables: a set $URB\_delivered_i$ (initially $\emptyset$) containing the messages that have been locally URB-delivered; a fifo queue $TO\_deliverable_i$ (initially the empty sequence denoted ϵ) that contains the sequence of messages

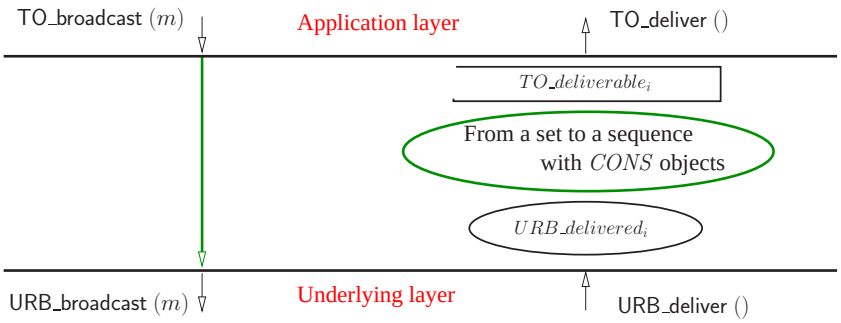

Figure 5.2: Adding total order message delivery to the URB abstraction

that, from the beginning, have been ordered the same way at all the processes; and a sequence number sn_i (initialized to 0) used to address a particular consensus object.

To make the presentation easier, the sequence $TO\_deliverable_i$ is sometimes considered as a set. As all the messages that are TO-broadcast are different, this does not create problems. The operator $\oplus$ denotes sequence concatenation.

init: $sn_i \leftarrow 0$; $TO\_deliverable_i \leftarrow \epsilon$; $URB\_delivered_i \leftarrow \emptyset$.

operation TO_broadcast (m): URB_broadcast (m).

when ($TO\_deliverable_i$ contains messages not yet TO-delivered):
  let m be the first message of $TO\_deliverable_i$ not yet TO-delivered;
  TO_deliver (m).

when m **is URB-delivered**: $URB\_delivered_i \leftarrow URB\_delivered_i \cup \{m\}$.

background task T:
  repeat forever
    wait $((URB\_delivered_i \setminus TO\_deliverable_i) \neq \emptyset)$;
    let $seq_i = (URB\_delivered_i \setminus TO\_deliverable_i)$;
    order (arbitrarily) messages in seq_i;
    $sn_i \leftarrow sn_i + 1$;
    $res_i \leftarrow CS[sn_i]$.propose (seq_i);
    $TO\_deliverable_i \leftarrow TO\_deliverable_i \oplus res_i$
  end repeat.

Figure 5.3: Building a TO-URB abstraction in $\mathcal{AS}_{n,t}[CONS]$ (code for p_i)

When it issues TO_broadcast (m), a process p_i simply URB-broadcasts m. When it URB-delivers a message m, it adds it to its local set $URB\_delivered_i$. To facilitate the presentation, the messages added to $URB\_delivered_i$ and $TO\_deliverable_i$ are never withdrawn. (In a practical setting, a garbage collection mechanism should be added.)

The core of the algorithm is the way messages from $URB\_delivered_i$ are ordered and placed at the tail of the sequence $TO\_deliverable_i$. This is the work of the background task T. Messages are TO-delivered in the order they have been deposited in $TO\_deliverable_i$.

The principle that underlies the task T is the following. This task T is an endless asynchronous distributed iteration. The first iteration determines a first sequence of messages that the processes append at the tail of their variables $TO\_deliverable_i$. The second iteration determines a second sequence of messages that the processes append at the tail of their variables $TO\_deliverable_i$, etc. Hence, all the processes p_i append the same sequence of messages at the tail their variables $TO\_deliverable_i$, and, consequently, they all will be able to TO-deliver the messages in the same order. A consensus object is associated with each loop iteration in order the processes agree on the same sequence of messages to add to their variables $TO\_deliverable_i$.

From an operational point of view, a process p_i first waits for messages that have been URB-delivered but not yet added to the sequence $TO\_deliverable_i$. Then p_i orders these messages (sequence seq_i) that it proposes to the next consensus object, namely, $CS[sn_i]$. The way messages are ordered in seq_i is arbitrary, the important point is here that seq_i is a sequence. Finally, let res_i be the sequence of messages decided by this consensus object, i.e., the value returned by $CS[sn_i]$.propose (seq_i). The sequence res_i (that has been proposed by some process) is the sequence of messages that the processes have agreed upon during their sn-th iteration, and each process p_i appends it to its variable $TO\_deliverable_i$.

The loop is asynchronous, which means that some seq_i proposed by p_i can contain few messages, while other can contain many messages. Moreover, several consensus instances can be concurrent, but distinct consensus instances are totally independent. An important point is here that a non-faulty process never stops executing the task T.

Remark: propose messages or propose message identities to a consensus object? The previous algorithm considers that a consensus proposal is a sequence of messages. It could instead be the sequence of their identities, and the size of proposals would be consequently shorter. The algorithm can easily be modified to take into account this improvement. Then, $TO\_deliverable_i$ would be a sequence of message identities, and the full messages (content plus identity) would be kept in $URB\_delivered_i$. If we adopt this improvement, it becomes possible that a message identity belongs to $TO\_deliverable_i$ while the corresponding message has not yet been URB-delivered (and consequently is not presently in $URB\_delivered_i$). The TO-delivery of a message is now constrained by an additional **wait** statement. More precisely, when m has to be TO-delivered (its identity is the identity of the next message to be TO-delivered), while its content has not yet been URB-delivered, the process has to wait for the URB-delivery of that message in order to TO-deliver it.

Let us observe that (as done in Figure 5.3) when sequences of messages are proposed, it is possible that res_i contains a message m not yet URB-delivered p_i. But the previous problem cannot occur because, in that case, res_i contains the full message m and not only its identity.

Remark: on the number of consensus instances It is easy to see that, if processes URB-broadcast a finite number (k) of messages, $k' \leq k$ consensus instances will be used. This means that this construction is "quiescent with respect to consensus instances".

Theorem 5.1 *The algorithm described in Figure 5.3 constructs a TO-URB abstraction in* $\mathcal{AS}_{n,t}[CONS]$.

Proof Notations. For any i and any $sn \geq 1$, let $seq_i[sn]$, $res_i[sn]$, and $TO\_deliverable_i[sn]$ denote the values of seq_i, res_i, and $TO\_deliverable_i$, respectively, in the last two lines of the sn-th iteration of the task T executed by p_i. Let also $res[sn]$ denote the sequence of messages decided by the consensus object $CS[sn]$ (due to the consensus agreement property, $res[sn]$ is unique). Finally, let $TO\_deliverable_i[0]$ denote the initial value of $TO\_deliverable_i$ (i.e., the empty sequence).

Claim C1. For any two processes p_i and p_j such that p_j is correct, and any $sn \geq 1$: (1) if p_i invokes $CS[sn]$.propose () then p_j invokes $CS[sn]$.propose (), (2) if p_i returns from its invocation, we have $TO\_deliverable_i[sn] = TO\_deliverable_j[sn] = res[1] \oplus \cdots \oplus res[sn]$.
Proof of the claim C1. The proof is by simultaneous induction on (1) and (2).

- Base case: $sn = 1$. If p_i invokes $CS[1]$.propose (), then $URB\_delivered_i$ contains at least one message m. Due to the termination property of the underlying URB abstraction and the fact that p_j is non-faulty, eventually $m \in URB\_delivered_j$. Hence, there is a time after which the predicate $(URB\_delivered_j \setminus TO\_deliverable_j[0]) \neq \emptyset$ is true. When this occurs, p_j invokes $CS[1]$.propose ().

  Due to the termination property of the underlying consensus object $CS[1]$ and the fact that p_j is non-faulty, it returns from its invocation. Assuming that p_i also returns from its invocation, it follows from the agreement property of $CS[1]$ that $res_i[1] = res_j[1] = res[1]$, and as $TO\_deliverable_j[0]$ is the empty sequence, we have $TO\_deliverable_i[1] = TO\_deliverable_j[1] = res[1]$.

- Let us assume that the claim holds for all sn such that $1 \leq sn < k$. Let us first show that, if p_i (that is faulty or non-faulty) invokes $CS[k]$.propose () then the non-faulty process p_j invokes $CS[k]$.propose (). As p_i invokes $CS[k]$.propose (), $URB\_delivered_i$ must contain a message m such that $m \in URB\_delivered_i \setminus TO\_deliverable_i[k-1]$. As $TO\_deliverable_i[k-1] = TO\_deliverable_j[k-1] = res[1] \oplus \cdots \oplus res[k-1]$ (induction assumption), it follows that $m \notin TO\_deliverable_j[k-1]$. Moreover, due to the termination property of the URB abstraction and the fact that p_j is non-faulty, m eventually belongs to $URB\_delivered_j$. When this occurs, if not yet done due to another message m', p_j invokes $CS[k]$.propose (), which proves the first item of the claim.

  The proof of the second item is the same as in the base case (after having replaced the consensus object $CS[1]$ by $CS[k]$), and we then have $TO\_deliverable_i[k] = TO\_deliverable_j[k] =$

$TO\_deliverable_j[k-1] \oplus res[k] = res[1] \oplus \cdots \oplus res[k]$.

End of the proof of the claim C1.

Proof of the validity property. This property follows from a simple examination of the text of the algorithm that shows that the algorithm does not create messages.

Proof of the total order message delivery property. This property follows from the Claim C1. Any two non-faulty processes p_i and p_j execute the same sequence of iterations (Item 1), and for each iteration sn, we have $TO\_deliverable_i[sn] = TO\_deliverable_j[sn] = res[1] \oplus \cdots \oplus res[sn]$ (Item 2).

Let us now consider a faulty process p_k. That process executes a finite number sn_k of iterations. During these iterations, it obtains from the consensus objects $CS[1]$, ..., $CS[sn_k]$, the same outputs $res[1]$, ..., $res[sn_k]$ as the non-faulty processes. Hence, $TO\_deliverable_k[sn\_k] = res[1] \oplus \cdots \oplus res[sn\_k]$, and, consequently, p_k TO-delivers a prefix of the sequence $res[1] \oplus \cdots \oplus res[sn\_k] \oplus \cdots$ of the messages TO-delivered by the non-faulty processes.

Proof of the termination property. Let us first consider the case of a non-faulty process that TO-broadcasts a message m. Suppose by contradiction that m is never TO-delivered by a non-faulty process. Eventually, (due to the termination property of URB) all the non-faulty processes URB-deliver m. Moreover, there is a time after which all the faulty processes have crashed and there are only non-faulty processes in the system. It follows that, there is an iteration k such that each process p_i proposes a sequence $seq_i[k]$ such that $m \in seq_i[k]$. Whatever, the sequence of messages $res[k]$ decided by the consensus object $CS[k]$, we necessarily have $m \in res[k]$. Hence, m is added to $TO\_deliverable_i$ and it is eventually TO-delivered, contradicting the initial assumption.

Let us now consider the case of a process p_x that TO-delivers a message m. In that case, there is an iteration k such that the consensus object $CS[k]$ returns $res[k]$ to p_x with $m \in res[k]$ (which has entailed the addition of m to $TO\_deliverable_x[k]$). It follows from the first item of the claim C1, that all non-faulty processes invoke $CS[k]$.propose(). Hence, each non-faulty process p_i decides $res[k]$ from that consensus object, and consequently adds m to $TO\_deliverable_i$ which concludes the proof of the termination property.

Proof of the integrity property. We have to prove here that no message is TO-delivered twice. This property follows directly from the text of the algorithm. □*Theorem* 5.1

Remark The reader can check that the construction described in Figure 5.3 is incorrect if, in addition to the validity, integrity and termination properties stated in Section 5.1.2, the underlying consensus objects satisfy only the following weaker agreement property: no two non-faulty processes decide different values. As we can see, this agreement property is only on the non-faulty processes

and consequently is not *uniform* (uniformity states that a property is on all the processes, not only the non-faulty ones).

The algorithm is no longer correct because nothing prevent a faulty process to decide, before crashing, a value different from the value decided by the non-faulty processes. The construction of the TO-URB abstraction on top of non-uniform base consensus objects is possible, but it requires to appropriately modify the construction of Figure 5.3.

5.2.3 CONSENSUS AND TO-URB ARE EQUIVALENT

The previous section has shown that a TO-URB construction suited to the system model $\mathcal{AS}_{n,t}[CONS]$ is possible. This construction requires an unbounded number of consensus objects. (This number is actually upper bounded by the total number of messages that are TO-broadcast.) This is in agreement with the fact that consensus is a one-shot problem while TO-URB is not.

In a very interesting way, it is possible to construct a consensus object in any system that provides processes with a built-in TO-URB abstraction (e.g., in hardware). This consensus object construction, that is independent of the number of process crashes, is described in Figure 5.4.

Let CS be the consensus object that is being built. When a process p_i invokes CS.propose (v_i), where v_i is the value it proposes, it first TO-broadcasts a message containing that value. Then, it returns the value carried by the first message that it TO-delivers.

```
operation CS.propose (v_i):
    TO_broadcast (v_i);
    wait (the first value v that is TO-delivered);
    return (v).
```

Figure 5.4: Building a consensus object from the TO-URB abstraction (code for p_i)

Theorem 5.2 *The algorithm described in Figure 5.4 constructs a consensus object in any system that provides processes with a built-in TO-URB abstraction.*

Proof The validity, integrity, and termination follow directly from their TO-URB counterparts. The consensus agreement property follows from the following simple observation: there is a single first message (value) received by a process, and due to the total order message delivery property, this message is the same for all the processes. □*Theorem* 5.2

The next theorem follows directly from the previous theorems 5.1 and 5.2.

Theorem 5.3 *Consensus and total order uniform reliable broadcast are equivalent in* $\mathcal{AS}_{n,t}[\emptyset]$.

5.3 THE CASE OF ASYNCHRONOUS SYSTEMS WITH FAIR CHANNELS ($\mathcal{AS}\_\mathcal{F}_{n,t}[\emptyset]$)

5.3.1 ASYNCHRONOUS SYSTEM WITH FAIR CHANNELS

The previous section has considered message passing systems prone to process crashes, where every channel is asynchronous but reliable. This section considers the case where the channels are not only asynchronous but also unreliable; more precisely, they are only fair. Such a system has been introduced in Chapter 4, where it is denoted $\mathcal{AS}\_\mathcal{F}_{n,t}[\emptyset]$.

Reminder #1: fair channel A fair channel, connecting p_i to p_j, is defined by the operations "send m to p_j" and "receive () from p_i" that satisfy 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.

Reminder #2: Uniform reliable broadcast and failure detectors As we have seen, URB assumes that all the messages that are URB-broadcast are different. It is TO-URB without the "total order message delivery" property. The following important results, that are on URB and failure detectors, have been presented in Chapter 4.

- URB can be implemented in $\mathcal{AS}\_\mathcal{F}_{n,t}[t < n/2]$ and cannot in $\mathcal{AS}\_\mathcal{F}_{n,t}[t \geq n/2]$.
- When we consider the failure detector-based approach, URB can be implemented in $\mathcal{AS}\_\mathcal{F}_{n,t}[\Theta]$. Θ is the weakest class of failure detectors that allows URB to be solved despite fair channels, whatever the value of t.

It follows from these results that Theorem 5.3 can be extended as follows.

Theorem 5.4 *Consensus and TO-URB are equivalent in both $\mathcal{AS}\_\mathcal{F}_{n,t}[t < n/2]$ and $\mathcal{AS}\_\mathcal{F}_{n,t}[\Theta]$.*

The rest of section shows that TO-URB can be built in $\mathcal{AS}\_\mathcal{F}_{n,t}[\emptyset]$ as soon as this system is enriched with consensus objects. This means that, in these systems, it is not necessary to build an intermediate URB abstraction to obtains TO-URB. TO-URB can be obtained directly from consensus objects. In the following, both the case of binary consensus and the case of multivalued consensus are considered.

5.3.2 URB AND TO-URB IN $\mathcal{AS}\_\mathcal{F}_{n,t}[BIN\_CONS]$

We show here that the URB problem can be solved in the system model $\mathcal{AS}\_\mathcal{F}_{n,t}[\emptyset]$ enriched with binary consensus objects. This system model is denoted $\mathcal{AS}\_\mathcal{F}_{n,t}[BIN\_CONS]$. Said another way, this construction shows that binary consensus is powerful enough to cope with asynchrony, fair channels, and half or more process crashes when one has to construct the URB abstraction. (The interest of this construction is more theoretical than practical.)

The construction is described in Figure 5.5. It assumes that each message m that is URB-broadcast can be encoded in an integer such that no two messages have the same encoding (which is always possible with a perfect hash function).

Local variables Each process p_i manages the following variables.

- $received_i$ is a set (initialized to $\emptyset$) containing the messages (integers) that p_i has received.
- $URB\_delivered_i$ is a set (initialized to $\emptyset$) containing the messages (integers) that p_i has URB-delivered.
- $level_i$ is an integer (initialized to 0) that is at the core of the construction. It is used to repeatedly scan the integers in such a way that every integer (message) that has been received (i.e., belongs to $received_i$) is eventually URB-delivered.

Global variables The processes communicate by exchanging messages through fair channels and cooperate through binary consensus objects. These objects are denoted $BC[a, b]$ where $0 \leq a < +\infty$ and $0 \leq b \leq a$. A process accesses an object $BC[a, b]$ by invoking (at most once) $BC[a, b]$.bin_propose $(prop)$ where $prop$ is the value it proposes to that binary consensus instance.

Process behavior When a process p_i wants to URB-broadcast a message or receives a message, it deposits it in $received_i$. The rest of its behavior is made up of two tasks.

The first task is to overcome the fact that channels are not reliable. To that end, p_i regularly sends to each other process (through the fair channels connecting it to these processes) each message (integer) it has received but not yet URB-delivered. This repetitive sending, plus the termination property of a fair channel (namely, a message sent infinitely often is received infinitely often), guarantees that any message in $received_i \setminus URB\_delivered_i$ is eventually received by every non-faulty process.

The second task is the heart of the construction. Process p_i proceeds by levels. At level $level_i = a$, it scan all the integers x (possible messages) between 0 and a. If such a message x has already been delivered, it has nothing to do. Otherwise, p_i strives to URB-deliver message x if it does correspond to a message that it has received. To that end, p_i uses the underlying binary consensus object $BC[level_i, x]$ to eliminate the integer x if it does not correspond to a message that has been received. More precisely, if x is a message that has been received, p_i proposes 1 to the binary consensus instance; otherwise, it proposes 0. If the value returned by $BC[level_i, x]$ is 1, p_i URB-delivers x; otherwise, it does not. Finally, when all the integers x such that $0 \leq x \leq level_i = a$, have

```
init: received_i ← ∅; URB_delivered_i ← ∅; level_i ← 0.

operation URB_broadcast (v): received_i ← received_i ∪ {v}.

when MSG (v) is received: received_i ← received_i ∪ {v}.

repeat forever
    for each v ∈ received_i \ URB_delivered_i do
       for each j ≠ i do send MSG (v) to p_j end for
    end for
end repeat.

repeat forever
    for x from 0 to level_i do
       if (x ∉ URB_delivered_i) then
          if (x ∈ received_i) then prop ← 1 else prop ← 0 end if;
          r ← BC[level_i, x].bin_propose (prop);
          if (r = 1) then URB_delivered_i ← URB_delivered_i ∪ {x};
                          URB_deliver (x)
          end if
       end if
    end for;
    level_i ← level_i + 1
end repeat.
```

Figure 5.5: An algorithm that constructs the URB abstraction in $\mathcal{AS}\_\mathcal{F}_{n,t}[BIN\_CONS]$ (code for p_i)

been scanned, p_i progresses to level $level_i + 1$. In that way, whatever the time a message (integer) is URB-broadcast and its value, the termination property of URB can be ensured.

Theorem 5.5 *The algorithm described in Figure 5.5 constructs an URB abstraction in* $\mathcal{AS}\_\mathcal{F}_{n,t}[BIN\_CONS]$.

Proof Claim C. The non-faulty processes execute the same infinite sequence of binary consensus instances. A faulty process executes a prefix of it.

Proof of claim C. It follows from the text of the algorithm that the first consensus instance executed by the processes is $BC[0, 0]$. Moreover, we have the following: (a) at least non-faulty processes execute it and each of them decides a value (consensus termination property), (agreement property of $BC[0, 0]$) (b) a single value is decided(agreement property of $BC[0, 0]$), and (c) that value is 0 or 1 (consensus validity property). There are two cases according to which value is decided.

- 1 is decided by $BC[0, 0]$. When a process p_i returns from $BC[0, 0]$.bin_propose $(prop)$, it URB-delivers 0, adds it to $URB\_delivered_i$, and proceeds to level 1. Moreover, after a process p_i has returned from $BC[0, 0]$.bin_propose $(prop)$, the local predicate $0 \in URB\_delivered_i$ remains forever true. It follows that no process will ever invoke $BC[a, 0]$.bin_propose () with $a \geq 1$ (this is because the predicate $0 \in URB\_delivered_i$ is then used to "discard" these consensus objects that have become useless because 0 can no longer be URB-delivered again).

- 0 is decided by $BC[0, 0]$. In that case, 0 is not URB-delivered and the consensus objects $BC[a, 0]$ with $a \geq 1$ are not discarded.

It follows that the next consensus object accessed by processes will be $BC[1, 1]$ if 0 has been URB-delivered, and $BC[1, 0]$ if 0 has not been URB-delivered. The important point here is that the second binary consensus object will be the same for all processes.

Then, considering the second consensus object used by processes, the same reasoning shows that the processes that will invoke a third consensus object will invoke the very same consensus object. It follows by induction that the non-faulty processes execute the same infinite sequence of binary consensus instances, and a faulty process executes a prefix of it. End of proof of laim C.

Proof of validity property. We have to show that if a message x is URB-delivered by a process, it has been previously URB-broadcast. If x is URB-delivered, there is a binary consensus instance $BC[level, x]$ that decided 1. It then follows from the consensus validity property that some process p_i has invoked $BC[level, x]$.bin_propose (1). Hence, we have $x \in received_i$. It is easy to see from the text of the algorithm that only values v that are URB-broadcast can be added to a set $received_i$, which completes the proof of URB validity property.

Proof of integrity property. We have to show that a process URB-delivers a message x at most once. This property follows directly from the following observations. First, for a message x to be URB-delivered at p_i, the predicate ($x \notin URB\_delivered_i$) has to be satisfied. Second, when it URB-delivers a message, p_i adds it to the set $URB\_delivered_i$. Hence, after x has been URB-delivered, the predicate ($x \notin URB\_delivered_i$) remains false forever. Hence, x cannot be URB-delivered several times.

Proof of termination property. Let us first show that if a process p_i URB-delivers a message x, any non-faulty process URB-delivers x. If p_i URB-delivers x, there is a consensus instance $BC[y, x]$ that decides 1. Moreover, every non-faulty process executes that instance (claim C). It then follows that every non-faulty process URB-delivers x.

Let us now show that, if a non-faulty process p_i URB-broadcasts x, every non-faulty process URB-delivers x. The proof is by contradiction. Let us suppose that x is never URB-delivered by any process (otherwise, the previous case applies). As the channels from p_i to each other process are fair, there is a finite time τ' after which every non-faulty process p_j is such that $x \in received_j$. Moreover, there is a time τ'' after which all faulty processes have crashed. As x is never URB-delivered by a process, it follows from claim C that there is a time $\tau \geq \max(\tau', \tau'')$ after which there is a level y such that all the processes propose 1 to the consensus instance $BC[y, x]$. Due to the consensus agreement property, every non-faulty process decides 1 when it invokes that instance. It follows that each non-faulty process URB-delivers x, contradicting the initial assumption. $\square_{Theorem\ 5.5}$

The next theorem is an immediate consequence of claim C that appears in proof of Theorem 5.5. (This claim states that the processes invoke the underlying binary consensus in the same order.)

Theorem 5.6 *The algorithm described in Figure 5.5 constructs a TO-URB abstraction in* $\mathcal{AS}\_\mathcal{F}_{n,t}[BIN\_CONS]$.

This theorem demonstrates that the computational power of binary consensus is strong enough to cope with the net effect of asynchrony, any number of process crashes, and unreliable (but fair) channels.

5.3.3 TO-URB IN $\mathcal{AS}\_\mathcal{F}_{n,t}[CONS]$

As proved by the previous construction, binary consensus is sufficient to build TO-URB on top of fair channels. An interesting question is then the following: is there an advantage to use multivalued consensus instead of binary consensus? The answer is "yes", in the sense that we obtain a more efficient construction.

```
init: sn_i ← 0; TO_deliverable_i ← ϵ; received_i ← ∅.

operation TO_broadcast (v): received_i ← received_i ∪ {v}.

when (TO_deliverable_i contains messages not yet TO-delivered):
     Let m be the first message of TO_deliverable_i not yet TO-delivered;
     TO_deliver (m).

when MSG (v) is received: received_i ← received_i ∪ {v}.

repeat forever
     for each v ∈ (received_i \ TO_delivered_i) do
          for each j ≠ i do send MSG (v) to p_j end for
     end for
end repeat.

repeat forever
     let seq_i = (received_i \ TO_deliverable_i);
     order (arbitrarily) the messages in seq_i;
     sn_i ← sn_i + 1;
     res_i ← CS[sn_i].propose (seq_i);
     TO_deliverable_i ← TO_deliverable_i ⊕ res_i
end repeat.
```

Figure 5.6: Building a TO-URB abstraction in $\mathcal{AS}\_\mathcal{F}_{n,t}[CONS]$ (code for p_i)

The construction Such a construction is described in Figure 5.6. It is a "merge" of the construction of Figure 5.3 (that builds TO-broadcast in $\mathcal{AS}_{n,t}[CONS]$) and the construction of Figure 5.5 (that builds URB in $\mathcal{AS}\_\mathcal{F}_{n,t}[BIN\_CONS]$).

Processes directly use fair channels (as in Figure 5.5), and they invoke periodically consensus instances (as in both constructions). The local variables have the same meaning as their counterparts in Figures 5.5 and 5.6. The main difference with respect to the TO-URB construction designed for $\mathcal{AS}_{n,t}[CONS]$ lies in the fact that now a process invokes a new instance even if the set

$received_i \setminus TO\_deliverable_i$ is empty. It follows that, differently from the construction in Figure 5.5, this construction is not "quiescent with respect to the number of consensus instances".

The proof of this construction is easily obtained by merging the proofs of Theorem 5.1 and Theorem 5.6.

Why a process has to forever invoke consensus Let us consider a system with 4 processes p_1, p_2, p_3 and p_4. Process p_1 TO-broadcasts m. It sends m to each other process, but only p_2 receives it (the messages to p_3 and p_4 are lost). Processes p_1 and p_2, that have then $seq_1 = seq_2 = \langle m \rangle$, invoke $CS[1]$.propose $(\langle m \rangle)$. Then, both URB-deliver m and crash. Moreover, suppose that no more messages are ever TO-broadcast. It is easy to see that, if a process p_i participates in its next consensus instance only if $seq_i \neq \emptyset$, the construction does not work.

5.4 THE UNIVERSALITY OF CONSENSUS

5.4.1 THE STATE MACHINE APPROACH

Provide a service to clients Practical systems provide clients with services. A *service* is usually defined by a set of commands (or requests) that each client can invoke. It is assumed that a client invokes one command at a time (hence, a client is a sequential entity). The state of the service is encoded in internal variables that are hidden to the clients. From the clients point of view, the service is defined by its commands.

A command (request) may cause a modification of the state of the service. It may also produce outputs that are sent to the client that invoked the command. It is assumed that the outputs are completely determined by the initial state of the service and the sequence of commands that have already been processed.

Replicate to tolerate failures If the service is implemented on a single machine, the failure of that machine is fatal for the service. So, a natural idea consists in replicating the service on physically distinct machines. More generally, the *state machine replication* technique is a methodology for making fault-tolerant a service offered to clients. The state of the service is replicated on several machines that can communicate with one another through a network.

Ideally, the replication has to be transparent to the clients. Everything has to appear as if the service was implemented on a single machine. This is called the *one copy equivalence* consistency condition. To attain this goal, the machines have to coordinate themselves. The main issue consists in ensuring that all the machines execute the commands in the same order. In that way, as the commands are deterministic, the copies of the state of the service will not diverge despite the crash of some of the machines. It is easy to see that, once each command issued by a client is encapsulated in a message, ensuring the *one copy equivalence* consistency condition amounts to construct a TO-URB abstraction among the machines.

Of course, according to the type of service, it is possible to partially weaken the total order requirement (for example, for the commands that are commutative). Similarly, for some services, the

commands that do not modify the state of the service are not required to be always processed by all replica.

5.4.2 CONSENSUS IS UNIVERSAL FOR BUILDING OBJECTS WITH A SEQUENTIAL SPECIFICATION

Let us consider all concurrent objects that have a sequential specification. Let us remember that this means that the correct behaviors of such objects can be described by a (possibility infinite) set of traces on their operations. The types of services described in the previous section are examples of objects with a sequential specification (each command is actually an object operation). As we have already seen, classical examples of concurrent objects defined by a sequential specification are atomic registers, and concurrent stacks, trees, or queues objects.

Considering an asynchronous distributed system prone to process crashes, a simple way to make such an object tolerant to process (machine) crash consists in replicating the object on each machine and using the TO-URB abstraction to ensure that the alive machines apply the same sequence of operations to their copy of the object. This section develops this approach.

The nature of the object The object, the implementation of which we want to make fault-tolerant, is defined by an initial state s_0, a finite set of m operations and a sequential specification. We consider that the operations are *total,* which means that any operation can be invoked in any state of the object. As an example, let us consider an unbounded stack. It has two operations, push() and pop(). As the stack is unbounded, the push() operation can always be invoked, and is, consequently, total. It is easy to define a pop() operation that is total by defining a meaning for pop() when the stack is empty (for example, pop() returns a default value (e.g., $\top$) when the stack is empty). (For a reason that will become clear below, the only constraint on that default value is that it has to be different from the control value $\perp$ used in Figure 5.7.)

An operation has the form $\mathsf{op}_x(param_x, result_x)$, with $1 \leq x \leq m$; $param_x$ is the list (possibly empty) of the input parameters of $\mathsf{op}_x()$, while $result_x$ denotes the result it returns to the invoking process. Instead of defining the set of all traces that describe the correct behavior of the object, its sequential specification can be defined by associating a pre-assertion and a post-assertion with each operation $\mathsf{op}_x()$. Assuming that $\mathsf{op}_x()$ is executed in a concurrency-free context, the pre-assertion describes the state of the object before the execution of $\mathsf{op}_x()$, while the post-assertion describes both its state after $\mathsf{op}_x()$ has been executed and the corresponding value of $result_x$ returned to the invoking process.

A sequence of operations applied to the object can be encoded by the values of variables that define its current state. The semantics of an operation can consequently be described by a transition function $\delta()$. This means that, s being the current state of the object, $\delta(s, \mathsf{op}_x(param_x))$ returns a pair (s', res) from a non-empty set of pairs $\{(s1, res1), \ldots, (sx, resx)\}$. Each pair of this set defines a possible output where s' is the new state of the object and res is the output parameter value returned to the invoking process (i.e., the value assigned to $result_x$).

If, for each operation op_x and for any state s of the object, the set $\{(s1, res1), \ldots, (sx, resx)\}$ contains exactly one pair, the object is deterministic. Otherwise, it is non-deterministic.

The universal construction for a deterministic object A *universal construction* is an algorithm that, given the sequential specification of an object, builds a fault-tolerant implementation of it. Such a construction, described in Figure 5.7, relies heavily on the TO-URB communication abstraction.

Each process p_i plays two roles: a client role for the upper layer application process it is associated with and a server role associated with the local implementation of the object. To that end, p_i manages a copy of the object in its local variable $state_i$.

When the upper layer application process invokes $\mathsf{op}(param)$, p_i builds a message (denoted msg_sent) containing that operation and the identity i, and TO-broadcasts it. Given such a message m, $m.op$ denotes the operation it contains, while $m.proc$ is the identity of the process that issued that operation. Then, p_i waits until the result associated with the invocation has been computed. Finally, it returns this result to the upper layer application process.

The server role of p_i consists in implementing a local copy of the object (kept in $state_i$). This is realized by a background task T that is an infinite loop. During each iteration, p_i first TO-delivers a message msg_rec (let us observe that this can entail T to wait if presently there is no message to be TO-delivered). Then, p_i invokes the transition function $\delta(state_i, msg\_rec.op)$ that computes the new local state of the object and the value returned to the invocation of the operation $msg\_rec.op$ that has been issued by the process whose identity is $msg\_rec.proc$. If this process is p_i, T deposits the result in $result_i$. In all cases, the task starts another iteration.

The **wait until** statement and the invocation of TO_deliver() can entail p_i to wait. It is assumed that the application process associated with p_i is sequential, i.e., after it has invoked an operation, it waits for the result of that operation before invoking another one.

```
when the operation op (param) is locally invoked by the application process:
    result_i ← ⊥;
    let msg_sent = ⟨op (param), i⟩;
    TO_broadcast (msg_sent);
    wait until (result ≠ ⊥);
    return (result_i).

background task T:
    repeat forever
        msg_rec ←TO_deliver();
        (state_i, res) ← δ(state_i, msg_rec.op);
        if (msg_rec.proc = i) then result_i ← res end if
    end repeat.
```

Figure 5.7: A TO-URB-based universal construction (code for p_i)

Due to the properties of the underlying TO-broadcast abstraction, it is easy to see that (1) the non-faulty processes apply the same sequence of operations to their local copy of the object, (2) any

faulty process applies a prefix of this sequence to its local copy, and (3) this sequence includes all the operations issued by the non-faulty processes and the operations issued by each faulty process until it crashes (the last operation issued by a faulty process can belong or not to this sequence; it depends on the run).

The case of a non-deterministic object There are two ways to deal with non-deterministic objects. The first is to ignore non-determinism. This can easily be done by using a non-genuine construction, i.e., a construction that implements any deterministic reduction of the object. This can be very easily done as follows. For each transition such that $\delta(s, \mathsf{op}_x(param_x))$ returns any pair from a non-empty set $\{(s1, res1), \ldots, (sx, resx)\}$, the set $\{(s1, res1), \ldots, (sx, resx)\}$ is arbitrarily reduced to a single of its pairs.

Differently, a genuine construction keeps the non-determinism of the object specification. Such a construction can easily be obtained by replacing the deterministic line

$$(state_i, res) \leftarrow \delta(state_i, msg\_rec.op)$$

by the following line:

$$pair_i \leftarrow \delta(state_i, msg\_rec.op); sn_i \leftarrow sn_i + 1; (state_i, res) \leftarrow CS.[sn_i].\mathsf{propose}\ (pair_i),$$

where the unique value of the pair $(state_i, res)$ is determined with the help of a consensus object $CS.[sn_i]$. The local variable sn_i (initialized to 0) is used to identify the consecutive consensus objects $CS.[1]$, $CS.[2]$, ... For the sn_i-th pair it has TO-delivered and deposited in msg_rec, each process p_i first computes, with the help of the transition function, a proposal (denoted $pair_i$) for the pair $(state_i, res)$. Each process p_i then proposes $pair_i$ to the consensus object $C[sn_i]$. The single value decided from that consensus object is then deposited by p_i in $(state_i, res)$. It follows from the properties of the consensus object that all the processes associate the same pair $(state, res)$ with the sn_i-th TO-delivered operation.

Universality of consensus Figure 5.7 has described a universal construction that makes an object fault-tolerant. The name "universal" comes from the fact that the construction works for any object that has total operations and is defined by a sequential specification.

It is because there is a construction based on the TO-broadcast abstraction, and such an abstraction can be built in $\mathcal{AS}_{n,t}[CONS]$, that the consensus object is called *universal*.

Said differently, let μ be any object type, with total operations, that is defined by a sequential specification. Consensus objects allow the construction of a concurrent object of type μ (i.e., an object that can be accessed concurrently by processes) in an asynchronous system prone to any number of process crashes.

5.5 CONSENSUS VS NON-BLOCKING ATOMIC COMMIT

This section shows how consensus can help solve the non-blocking atomic commit problem in asynchronous systems enriched with an appropriate failure detector. (It is important to notice that this

does not mean that consensus and non-blocking atomic commit are equivalent in a pure asynchronous system $\mathcal{AS}_{n,t}[\emptyset]$. See Section 5.5.3.)

5.5.1 THE NON-BLOCKING ATOMIC COMMIT PROBLEM

This problem (in short, NBAC) is a particular agreement problem in which each process votes (*yes* or *no*) and the processes have to validate their previous local computations (COMMIT) or back track to a predetermined previous state (ABORT). The validation/backtracking alternative is governed by the votes issued by the processes and the failure pattern. A vote *yes* means that the issuing process proposes to validate the local computations because everything went well on its side, while a vote *no* means that the issuing process cannot validate its local computation (because something went wrong on its side).

Definition More precisely, the NBAC problem is a one-shot problem (each process invokes at most once the operation) defined by the following properties. It is assumed that each alive process invokes *NBAC*.propose(v) where *NBAC* is the object associated with the considered NBAC problem instance, and v is the value (*yes* or *no*) of its vote. If p_i does not crash, its invocation *NBAC*.propose(v) returns it a value.

- Validity. A decided value is COMMIT or ABORT. Moreover,
  - Justification. If a process decides COMMIT, all the processes have voted *yes*.
  - Obligation. If all the processes vote *yes* and there is no crash, no process decides ABORT.
- Integrity. A process decides at most once.
- Agreement. No two processes decide different values.
- Termination. Each non-faulty process decides.

This problem has the same agreement, integrity and termination properties as the consensus problem. It differs from it in the validity property, namely, a decided value is not a proposed value (vote) but a value that depends on both the proposed values *and* the failure pattern. Differently, the properties defining the consensus problem do not refer directly to the failure pattern.

A remark on non-determinism Let us remark that if the obligation property was missing, nothing would prevent the processes to always decide ABORT. Hence, the idea of the definition is that, if everything went well (i.e., all the processes voted *yes*, and there is no crash) then the processes have no choice: they have to commit their local computations.

Differently, if a process voted *no* (whether it crashes or not) the decision can be only ABORT. The reader can check that if a process crashes before voting, the only possible decision is also ABORT. In the cases where all the processes have voted *yes* and at least one of them crashes, the decision can be COMMIT or ABORT. Hence, in some cases, the decision is fixed, while in other cases the decision depends on the actual run. In that precise sense, and similarly to consensus, NBAC allows non-determinism to be solved: the decided value cannot always be computed from a deterministic

function on the proposed values, it depends on the asynchrony and failure pattern that occur during the considered run.

Sequential vs non-sequential specification The specification of a concurrent object such as a register, a stack or a queue is sequential in the sense that all the correct behaviors of the object can be described by the sequences on their operations that are allowed. As an example, a stack, a tree, a set, a register, or a queue can be defined by a sequential specification.

It is important to notice that the NBAC problem has no sequential specification. There is no sequence of $NBAC$.propose() invocations that defines the correct behavior of the NBAC object. The specification is concurrent in the sense that the value output by an NBAC object involves "some input" from each process (either its vote, or an information on the fact that it has crashed).

5.5.2 CONSENSUS-BASED NBAC

Anonymously perfect failure detector Let $?P$ be the class of failure detectors that provide each process p_i with a variable $failure_i$, whose range is $\{red, green\}$ and that p_i can only read (red means "a failure has occurred", while "green" means "no failure has yet occurred"). The set of variables $\{failure_i\}_{1 \leq i \leq n}$ satisfies the following properties. Let us remember that, given a failure pattern F, $Faulty(F)$ and $Correct(F)$ denote the set of processes that are faulty and non-faulty in F, respectively. Moreover, $failure_i^\tau$ is the value of $failure_i$ at time τ.

- Anonymous completeness. $(Faulty(F) \neq \emptyset) \Rightarrow (\forall i \in Correct(F) : \exists \tau : \forall \tau' \geq \tau : failure_i^{\tau'} = red)$.
- Anonymous accuracy. $\forall \tau : \big(F(\tau) = \emptyset\big) \Rightarrow \big(\forall i : failure_i^\tau = green\big)$.

The completeness property states that if a process crashes, the non-faulty processes will be informed of a crash, while the accuracy property states that no process is informed of a crash before a crash does occur. The anonymity attribute is due to fact that, if crashes occur, no process ever knows the identity of the crashed processes.

Notation Let $\mathcal{AS}_{n,t}[?P, CONS]$ be an asynchronous system where any number t of processes can crash, and where the processes can use underlying consensus objects and a failure detector of the class $?P$. This means that, in this type of system, the consensus objects and $?P$ are given for free.

Solving NBAC in $\mathcal{AS}_{n,t}[?P, CONS]$ The algorithm described in Figure 5.8 constructs an NBAC object in asynchronous systems prone to any number of process crashes, enriched with a failure detector of the class $?P$ and consensus objects (one consensus object is actually sufficient).

The NBAC object that is built is denoted $NBAC$. The underlying consensus object is denoted CS. It is assumed that each non-faulty process invokes $NBAC$.propose (v) where v is its vote. When it invokes it, a process p_i first sends its vote to each process, including itself (the broadcast MY_VOTE (v_i) operation is a best effort broadcast: if p_i crashes during the broadcast, an arbitrary subset of the processes receive the message).

```
operation NBAC.propose (v_i): % the vote v_i is yes or no %
    broadcast MY_VOTE (v_i);
    wait until ( (MY_VOTE (−) received from each process) ∨ (failure_i = red));
    if ( a vote yes has been received from every process)
        then output_i ← CS.propose (COMMIT)
        else output_i ← CS.propose (ABORT)
    end if;
    return (output_i).
```

Figure 5.8: An algorithm that constructs an NBAC object in $\mathcal{AS}_{n,t}[?P, CONS]$ (code for p_i)

Then, p_i waits until either it has received a vote from each process, or it is informed of a crash. Then, if it has received a *yes* vote from all, it proposes COMMIT to the underlying consensus object; otherwise, it proposes ABORT to that object. Finally, p_i returns the value output by the consensus object.

Theorem 5.7 *The algorithm described in Figure 5.8 constructs an NBAC object in* $\mathcal{AS}_{n,t}[?P, CONS]$.

Proof As a process returns at most once, the NBAC integrity property is guaranteed. Moreover, as a process returns the value obtained from the underlying consensus object, the NBAC agreement property is inherited from consensus agreement.

Let us now consider the NBAC termination property. If no process is faulty, they all broadcast their vote, and whatever the value of the votes, their **wait until** statements terminate. If a process is faulty, the local variables $failure_i$ of the non-faulty processes are eventually assigned the value red, and their **wait until** statements terminate. Hence, whatever the case, all non-faulty processes eventually invoke the consensus object and, due to its termination property, do terminate their invocation, which proves the NBAC termination property.

As far as the validity property is concerned, it follows from the consensus validity property that only the value COMMIT or the value ABORT can be decided. As far as the justification property is concerned, we have the following. If a process decides COMMIT, it follows from the consensus validity property that a process p_i has proposed COMMIT to the consensus object. We conclude then, from the text of the algorithm, that p_i has received a vote *yes* from each process, which concludes the proof of the justification property.

Let us finally consider the obligation property. If there is no crash and all processes vote *yes*, they all propose COMMIT to the consensus object. As a single value is proposed to the consensus object C, only that value can be decided from CS, and consequently all processes eventually return COMMIT, which concludes the proof of the NBAC obligation property. □*Theorem* 5.7

5.5.3 CONSENSUS VS NBAC

The previous section has shown how the NBAC problem can be solved in $\mathcal{AS}_{n,t}[\emptyset]$ enriched with both a failure detector of the class $?P$ and a consensus object. The algorithm described in Figure 5.8 is a reduction of NBAC to consensus in the model $\mathcal{AS}_{n,t}[?P]$.

This does not mean that the consensus problem and the NBAC problem can be compared in $\mathcal{AS}_{n,t}[\emptyset]$. Actually, they are incomparable in $\mathcal{AS}_{n,t}[\emptyset]$. More precisely, there is a failure detector class *FD1* such that the NBAC problem can be solved in $\mathcal{AS}_{n,t}[FD1]$ while consensus cannot, and there is failure detector class *FD2* such that the consensus can be solved in $\mathcal{AS}_{n,t}[FD2]$ while NBAC cannot. These classes of failure detectors are not addressed in this book.

Let us finally remember that the consensus problem has a sequential specification while the NBAC problem has not.

5.6 CONSENSUS IMPOSSIBILITY IN $\mathcal{AS}_{n,t}[\emptyset]$

This section shows that the consensus problem cannot be solved in $\mathcal{AS}_{n,t}[\emptyset]$. Solving it requires a distributed system model "stronger" than $\mathcal{AS}_{n,t}[\emptyset]$. (This is the famous FLP impossibility result.)

5.6.1 THE INTUITION THAT UNDERLIES THE IMPOSSIBILITY

To stop waiting or not to stop waiting, that is the question The impossibility of solving some distributed computing problems comes from the uncertainty created by the net effect of asynchrony and failures. This uncertainty makes it impossible to distinguish a crashed process from a process that is slow or a process with which communication is slow.

Let us consider a process p that is waiting for a message from another process q. In the system model $\mathcal{AS}_{n,t}[\emptyset]$, the main issue the process p has to solve is to stop waiting for the message from q or to continue waiting. Basically, allowing p to stop waiting can entail a violation of the safety property of the problem if q is currently alive, while forcing p to wait the message from q can prevent the liveness property of the problem to be satisfied (if q has crashed before sending the required message).

Synchrony rules out uncertainty Let us consider a synchronous system involving two processes p_i and p_j. Synchronous means that the transfer delays are upper bounded (let Δ be the corresponding bound) and there is a lower bound and an upper bound on the speed of the processes. In order to simplify (and without loss of generality), we consider here that processing time are negligible with respect to message transit times and are consequently equal to 0.

In such a synchronous context, let us consider a problem P where each process has an initial value (v_i and v_j, respectively), and they have to compute a result that depends on these values as follows. If no process crashes, the result is $f(v_i, v_j)$. If p_j (resp., p_i) crashes, the result is $f(v_i, v_j)$ or $f(v_i, \perp)$ (resp., $f(\perp, v_j)$).

Each process sends its value and waits for the value of the other process. When it receives the other value, a process sends its value if not yet done. In order not to wait forever, the value of

the other process (say p_j), the process p_i uses a timer as follows. It sets the timer to 2Δ when it sends its value. If it has not received the value of p_j when the timer expires, it concludes that p_j has crashed before sending its value and returns $f(v_i, \perp)$. In the other cases, it has received v_j and returns $f(v_i, v_j)$. These two cases are described in Figure 5.9 (in the figure at the right, the cross on p_j's axis indicates its crash).

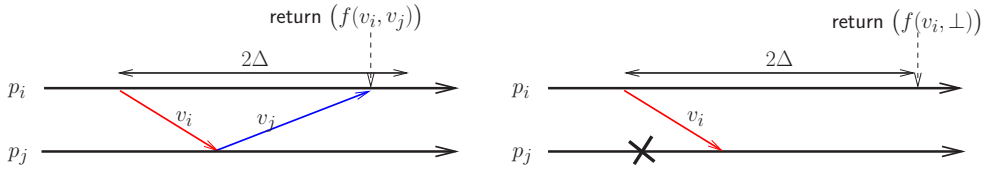

Figure 5.9: Synchrony rules out uncertainty

The execution on the left is failure-free, and p_j sends its value at the latest, i.e., when it receives the value v_i from p_i. In that case, p_i returns $f(v_i, v_j)$ (and the timer is useless). Differently, in the execution on the right, p_j has crashed before sending its value, and, consequently, p_i returns $f(v_i, \perp)$ when the timer expires. (If p_j had sent its value before crashing, p_i would have received it and would have returned $f(v_i, v_j)$ when receiving v_j). The uncertainty on the state of p_j is controlled by the timeout value. The timer is conservatively set in both cases, as p_i does not know in advance if p_j has crashed or not.

... While asynchrony cannot Let us now consider that, while processing times remain equal to 0, message transfer delays are finite but arbitrary. So, the system is asynchronous as far as messages are concerned.

A process can use a local clock and an "estimate" of the round-trip delay, but, unfortunately, there is no guarantee that (whatever its value) this estimate be an upper bound on the round trip delay in the current execution (otherwise, the system would be synchronous).

Using such an "estimate", several cases can occur. It is possible that, in the current execution, the estimate is actually a correct estimate. In that case, the synchrony assumption used by the processes is correct, and we are luckily in the case of the previous synchronous system described in Figure 5.9.

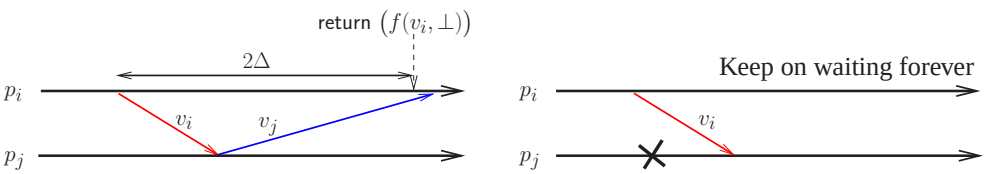

Figure 5.10: To wait or not to wait in presence of asynchrony and failures?

Unfortunately, as already said, there is no guarantee that (whatever its value) the estimate value used is a correct estimate. This is described on the left side of Figure 5.10, where p_i returns

$f(v_i, \perp)$ when the timer expires, while it should return $f(v_i, v_j)$. In that case, the incorrectness of the estimate value entails the violation of the safety property of the problem. So, timers cannot be used safely. But if p_i does not use a timer while p_j has crashed before sending its value (right side of Figure 5.10), it will wait forever, violating the liveness property of the problem.

This simple example shows that it can be impossible to guarantee both the safety property and the liveness property of a problem P, when one has to solve P in an asynchronous system.

5.6.2 REFINING THE DEFINITION OF $\mathcal{AS}_{n,1}[\emptyset]$

Before proving the impossibility result in the next sections, this section refines the definition of the underlying asynchronous model $\mathcal{AS}_{n,1}[\emptyset]$. Let us remark that we now assume $t = 1$.

Modeling the communication system The system consists of a set of n processes that communicate by sending and receiving messages with the operations "send m to $proc$" and "receive ()". Each message m is assumed to contain the identity of its sender ($m.sender$) and the identity of its destination process ($m.dest$). Moreover, without loss of generality, all messages are assumed to be different (this can easily be done by adding sequence numbers).

When a process sends a message to a process p_i, that message is deposited in a set denoted *buffer*. When a process p_i invokes the receive operation, it obtains either a message m such that $m.dest = i$ that has been deposited into *buffer*, or the default value $\perp$ that indicates "no message". If the message value that is returned is not $\perp$, the corresponding message is withdrawn from *buffer*. It is possible that *buffer* contains messages m such that $m.dest = i$, while p_i obtains $\perp$. The fact that a message can remain an arbitrary time in *buffer* is used to model communication asynchrony (but, while arbitrary, this time duration is finite, see below).

The network is reliable in the sense that there is neither message creation, nor message duplication. Moreover, the "no loss" property of the communication system is modeled by the following *fairness* assumption: given any process p_i and any message m that has been deposited into *buffer* and is such that $m.dest = i$, if p_i executes receive() infinitely often, it eventually obtains m.

Modeling the processes The behavior of a process is defined by an automaton that proceeds by executing steps. A step is a represented by a pair (i, m) where i is a process name and m a message or the default value $\perp$. When it executes the step (i, m), process p_i performs atomically the following:

- Either it receives a message m previously sent to it (in that case $m \in$ *buffer*, $m.dest = i$ and m is then withdrawn from *buffer*), or it "receives" the value $m = \perp$ (meaning that there is no message to be received yet).
- Then according to the value received (a message value or $\perp$), it sends a finite number of messages to the processes (those are deposited in *buffer*), and it changes its local state.

Hence, (until it possibly crashes) each process executes a sequence of steps (as defined by its automaton), and each step takes one global state to another. Let σ_i be the current local state of p_i. The execution of its next step by p_i entails its progress from σ_i to a new local state σ_i'. Let us observe

that the behavior of a process is deterministic. The next state of p_i and the message it sends (if any) are entirely determined by its initial state and the sequence of messages and $\perp$ values that it has received so far.

Input vector Given a consensus instance, let v_i be the value proposed to that consensus instance by process p_i. That value is part of its initial local state. The corresponding input vector, denoted $Input[1..n]$, is the vector such that $Input[i] = v_i$, $1 \leq i \leq n$. When considering binary consensus, the set of all possible input vectors is the set $\{0, 1\}^n$.

System global state A *global state* Σ (also called *configuration*) is a vector of n local states $[\sigma_1, \ldots, \sigma_n]$ (one per process p_i), plus a set of messages that represents the current value of *buffer* (those messages are the messages that are in transit with respect to the corresponding global state). A *non-faulty* global sate is a global state in which no process has crashed.

An initial global state Σ_0 is such that each σ_i, $1 \leq i \leq n$, is an initial local state of p_i, and *buffer* is the empty set.

A step $s = (i, \perp)$ can be applied to any global state Σ. A step $s = (i, m)$ where $m \neq \perp$ can be applied to a global state Σ only if *buffer* contains m. If an applicable step s is applied to global state Σ, the resulting global state is denoted $\Sigma' = s(\Sigma)$.

Schedule, reachability and accessibility A *schedule* is a (finite or infinite) sequence of steps $s_1, s_2, \ldots$ issued by the processes. A schedule σ is *applicable* to a global state Σ, if for all $i \geq 1$ (and $i \leq |\sigma|$ if σ is finite), s_i is applicable to Σ_{i-1} where $\Sigma_0 = \Sigma$ and $\Sigma_i = \sigma_i(\Sigma_{i-1})$.

A global state Σ' is *reachable* from Σ if there is a finite schedule σ such that $\Sigma' = \sigma(\Sigma)$. A global state is *accessible* if it is reachable from an initial global state.

Runs of an algorithm The algorithm A that solves consensus despite asynchrony and one process crash is encoded in a set of n automata, one per process (as defined previously). The local state of each process p_i contains a local variable $decided_i$. That variable, initialized to $\perp$, is a one-write variable that is assigned by p_i to the value it decides upon.

It is assumed that the algorithm executed by a process is such that, after it has decided (if it ever decides) a non-faulty process keeps on executing steps forever. Hence, a correct process executes an infinite number of steps. Given an initial global state, a *run* is an infinite schedule that starts from this global state.

In the context of the impossibility proof, a run is *admissible* if at most one process crashes and all messages that have been sent to the non-faulty processes are eventually received.

A tree of admissible runs Given an initial state Σ_0 and a consensus algorithm A, all its possible runs define a tree, denoted $\mathcal{T}(A, \Sigma_0)$, where each node represents a global state of A, and each edge represents a step by a process.

5.6.3 NOTION OF VALENCE OF A GLOBAL STATE

The impossibility proof considers the binary consensus problem, i.e., the case where only two values (0 and 1) can be proposed. It is a proof by contradiction: it assumes that there is an algorithm A that solves binary consensus in $\mathcal{AS}_{n,1}[\emptyset]$, and exhibits a contradiction. Trivially, as binary consensus cannot be solved when one process can crash, it cannot be solved either when $t \geq 1$ processes can crash, and multivalued consensus cannot be solved either for $t \geq 1$.

Given an initial global state Σ_0, let us consider the tree $\mathcal{T}(A, \Sigma_0)$. It is possible to associate a *valence* notion with each state Σ of this tree, defined as follows. The valence of a node (global state) $\Sigma \in \mathcal{T}(A, \Sigma_0)$ is the set of values that can be decided upon in a global state reachable from Σ. Let us observe that, due to the termination property of the consensus algorithm A, the set $valence(\Sigma)$ is not empty. As the consensus is binary, it is equal to one of the following sets: $\{0\}$, $\{1\}$ or $\{0, 1\}$. More explicitly:

- Σ is *bivalent* if the eventual decision value of the consensus is not yet fixed in Σ. This means that, when considering $\mathcal{T}(A, \Sigma_0)$, the bivalent global state Σ is the root of a subtree including both global states where processes decide value 1, and global states where processes decide value 0. Said differently, an external observer (who would have an instantaneous view of the process states and the channel states) cannot determine the value that will be decided from Σ.

- Σ is *univalent* if the eventual decision value is fixed in Σ: all runs starting from Σ decide the same value. If that value is 0, Σ is *0-valent*; otherwise, it is *1-valent*. Hence, if Σ is x-valent ($x \in \{0, 1\}$), all nodes of the subtree of $\mathcal{T}(A, \Sigma_0)$ that is rooted at Σ are x-valent. This means that, given Σ, an external observer could determine the single value that can be decided from that global state. Let us observe that it is possible that no local state σ_i of Σ allows the corresponding process p_i to know that Σ is univalent.

A part of a tree $\mathcal{T}(A, \Sigma_0)$ for a system of two processes is described in Figure 5.11. (As there are only two processes, each global state has at most two successors.) Each node (global sate) is labeled with the set of values that can be decided from it. If this set contains a single value, the corresponding global state is univalent (and then all its successors have the same valence). Otherwise, it is bivalent.

The notion of valence captures a notion of non-determinism. Said differently, if state Σ is univalent, "the dice are cast": the decision value (that is perhaps not yet explicitly known by processes) is determined. If Σ is bivalent, "the dice are not yet cast": the decision value is not yet determined (it still depends on the run that will occur from Σ, which in turn depends on asynchrony and the failure pattern).

5.6.4 CONSENSUS IS IMPOSSIBLE IN $\mathcal{AS}_{n,1}[\emptyset]$

As already indicated, the proof is by contradiction: assuming that there is an algorithm A that solves binary consensus in $\mathcal{AS}_{n,1}[\emptyset]$, it exhibits a contradiction. More precisely, the proof shows that there

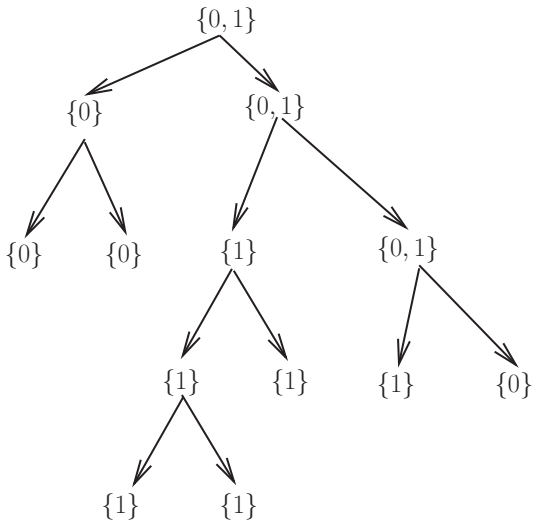

Figure 5.11: Bivalent vs univalent global states

is an initial global state Σ_0 such that $\mathcal{T}(A, \Sigma_0)$ has an infinite run of A whose all global states are bivalent. Said another way, assuming that A always preserves the safety property (at most one value is decided) implies that it has executions that never decide.

Bivalent initial state The next lemma shows that, whatever the consensus algorithm A, there is at least one input vector $Input[1.n]$ such that the corresponding initial global state is bivalent. This means that, while the value decided by A is determined by the input vector only, when the corresponding initial global state Σ_0 is univalent, this cannot be true for all the input vectors.

Lemma 5.8 *Every consensus algorithm A has a bivalent initial global state.*

Proof Let C_0 be the initial global state in which all processes propose 0 (so its input vector is $[0, \ldots, 0]$), and C_i, $1 \leq i \leq n$, the initial global state in which the processes from p_1 to p_i propose the value 1, while all the other processes propose 0. So, the input vector of C_n is $[1, \ldots, 1]$ (all processes propose 1).

These initial global states constitute a sequence in which any two adjacent global states C_{i-1} and C_i, $1 \leq i \leq n$, differ only in the value proposed by the process p_i: it proposes the value 0 in C_{i-1} and the value 1 in C_i. Moreover, it follows from the validity property of the consensus algorithm A, that C_0 is 0-valent, while C_n is 1-valent.

Let us assume that all the previous configurations are univalent. It follows that, in the previous sequence, there is (at least) one pair of consecutive configurations, say C_{i-1} and C_i, such that C_{i-1} is

0-valent and C_i is 1-valent. Assuming that there is a consensus algorithm A in $\mathcal{AS}_{n,1}[\emptyset]$, we exhibit a contradiction.

Assuming that no process crashes, let us consider a run of A that starts from global state C_{i-1}, in which process p_i executes no step for an arbitrarily long period (the end of that period is defined below). Let us observe that, as the algorithm A can cope with one process crash, no process executing A (but p_i) is able to distinguish the case where p_i is slow and the case where it has crashed.

As (by assumption) the algorithm satisfies the consensus termination property despite up to one crash, all the processes (but p_i) decide after a finite number of steps. The sequence of steps that starts at the very beginning of the run and ends when all the processes have decided (but p_i, which has not yet executed a step), defines a schedule σ. (See the upper part of Figure 5.12 where, within the input vector C_{i-1}, the value proposed by p_i is inside a box.) As C_{i-1} is 0-valent, the global state $\sigma(C_{i-1})$ is also 0-valent (let us recall that $\sigma(C_{i-1})$ is the global state attained by executing the sequence σ from C_{i-1}). Finally, after all the steps of σ have been executed, p_i starts executing and decides. As $R(C_{i-1})$ is 0-valent, p_i decides 0.

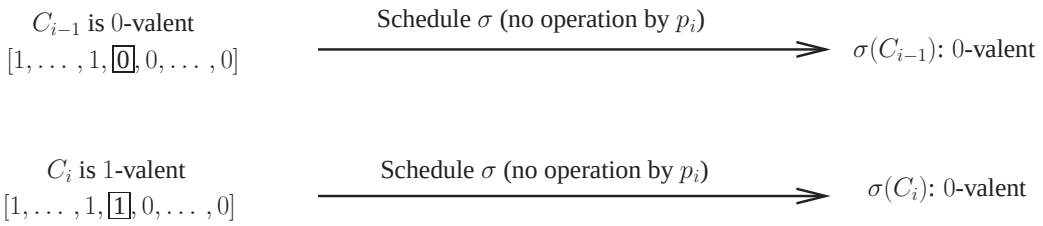

Figure 5.12: There is a bivalent initial configuration

Let us observe (lower part of Figure 5.12) that the same schedule σ can be produced by the algorithm A from the global state C_i. This is because (1) as the global states C_{i-1} and C_i differ only in the value proposed by p_i, and (2) p_i executes no step in σ, the decided value cannot depend on the value proposed by p_i. It follows that, as $\sigma(C_{i-1})$ is 0-valent, the global state $\sigma(C_i)$ is also 0-valent. But as the global state C_i is 1-valent, we conclude that $\sigma(C_i)$ is necessarily 1-valent, which contradicts the initial assumption and concludes the proof of the lemma. $\square_{Lemma\ 5.8}$

A remark on the validity property: strengthening the lemma In addition to the fact that at most one process can crash, the previous lemma is based on the validity property satisfied by the algorithm A that states that the decided value is one of the proposed values (from which we have concluded that C_0 and C_n are 0-valent and 1-valent, respectively).

The reader can check that the lemma remains valid if the validity property is weakened as follows: "there are runs in which the value 0 is decided, and there are runs in which the value 1 is decided". This validity property is weaker because it does not prevent a consensus algorithm from deciding 0 when all the processes propose 1. Despite its weakness, this property is sufficient to

conclude that, if we assume that all the initial global states are univalent, then there is at least one initial global state that is 0-valent and at least one initial global state that is 1-valent. The proof of the lemma follows easily from this simple observation.

The lemma is then stronger (i.e., it applies to more runs) because it requires weaker assumptions (validity property satisfied by the consensus algorithm A).

Remark: crash vs asynchrony The previous proof is based on the assumption that, despite asynchrony and the possibility for one process to crash, the algorithm A allows all the correct processes to decide and terminate. This allows the proof to play with process speed and consider a schedule σ during which a process p_i executes no step. We could have instead considered that p_i has initially crashed (i.e., p_i crashes before executing any step). During the schedule σ, the consensus algorithm A has no way to know in which case the system really is (has p_i initially crashed or is it only very slow?). This shows that, for some problems, asynchrony and process crashes are two facets of the same "uncertainty" algorithms have to cope with.

Lemma 5.9 *Let Σ be a non-faulty bivalent global state, and let $s = (i, m)$ be a step that is applicable to Σ. Then, there is a finite schedule σ (not including s) such that $s(\sigma(\Sigma))$ is bivalent.*

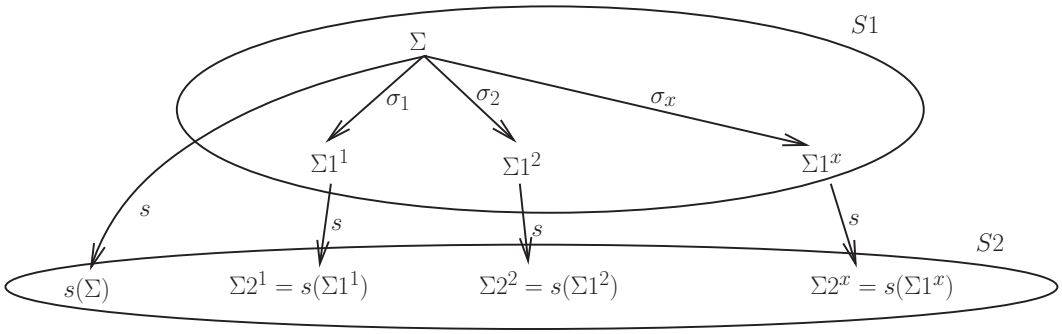

Figure 5.13: Illustrating the definitions used in Lemma 5.9

Proof Let $S1$ be the set of global states reachable from Σ with a finite schedule not including s, and $S2 = s(S1) = \{s(\Sigma 1) \mid \Sigma 1 \in S1\}$ (see Figure 5.13.) We have to show that $S2$ contains a non-faulty bivalent global state.

Let us first notice that, as s is applicable to Σ, it follows from the definition of $S1$ and the fact that messages can be delayed for arbitrarily long periods, that s is applicable to every global state $\Sigma' \in S1$. The proof is by contradiction. Let us assume that every global state $\Sigma 2 \in S2$ is univalent.

Claim C1. $S2$ contains both 0-valent and 1-valent global states.
Proof of the claim. Since Σ is bivalent, for each $v \in \{0, 1\}$, there is a finite schedule σ_v that is

applicable to Σ and such that the global state $C_v = \sigma_v(\Sigma)$ is v-valent. We consider two cases according to the fact that σ_v contains s or not.

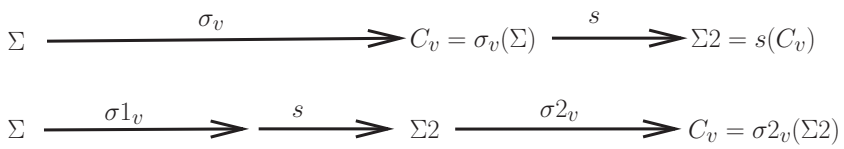

Figure 5.14: $\Sigma 2$ contains 0-valent and 1-valent global states

- Case 1: σ_v does not contain s (top part of Figure 5.14). In that case, taking $\Sigma 2 = s(C_v)$, we trivially have $\Sigma 2 \in S2$. As C_v is v-valent, it follows that $\Sigma 2$ also is v-valent.
- Case 2: σ_v contains s (bottom part of Figure 5.14). Then, there are two schedules $\sigma 1_v$ and $\sigma 2_v$ such that $\sigma_v = \sigma 1_v \; s \; \sigma 2_v$. In that case, taking $\Sigma 2 = s(\sigma 1_v(\Sigma))$, we trivially have $\Sigma 2 \in S2$. As all global states in $S2$ are univalent, $\Sigma 2$ is univalent. Finally, as $C_v = \sigma 2_v(\Sigma 2)$ is v-valent, it follows that $\Sigma 2$ also is v-valent. End of the proof of the claim C1.

Claim C2. Let two global states be *neighbors* if one is reachable from the other in a single step. There exist two neighbors $\Sigma 1', \Sigma 1'' \in S1$ such that $\Sigma 2' = s(\Sigma 1')$ is 0-valent and $\Sigma 2'' = s(\Sigma 1'')$ is 1-valent.

Proof of the claim. Considering the global states in $S1$ as the nodes of a graph G in which any two adjacent nodes are connected by an edge, let us label a node X in G with $v \in \{0, 1\}$ if and only if $s(X) \in S2$ is v-valent. As by assumption any global state in $S2$ is univalent, every node of G has a well-defined label. It follows from Claim C1 that there are nodes labeled 0 and nodes labeled 1. Moreover, as Σ belongs to $S1$ (this is because the empty schedule is a finite schedule), it also belongs to G and has, consequently, a label. Finally, as (a) there a path between any two nodes of G (through the node associated with Σ), and (b) all nodes of G are labeled 0 or 1, there are necessarily two adjacent nodes that have distinct labels. End of the proof of the claim.

Let two neighbors $\Sigma 1', \Sigma 1'' \in S1$ such that $\Sigma 2' = s(\Sigma 1')$ is 0-valent and $\Sigma 2'' = s(\Sigma 1'')$ is 1-valent (due to Claim C2, they exist). Moreover, let $s' = (i', m')$ be the step such that $\Sigma 1'' = s'(\Sigma 1')$ (Figure 5.15). We consider two cases.

- Case $i \neq i'$. As the steps s and s' are independent (s is not the reception of a message sent by s' and s' is not the reception of a message sent by s), it follows that $\Sigma 2'' = s'(s(\Sigma 1')) = s(s'(\Sigma 1'))$, which means that $\Sigma 2''$ has to be bivalent. This contradicts the fact that $\Sigma 2''$is 1-valent, and proves the lemma for that case.
- Case $i = i'$. Let us consider Figure 5.16 where, according to the previous notations, the global state $\Sigma 2'$ is 0-valent, while $\Sigma 2''$ is 1-valent. Let us consider a schedule that starts from $\Sigma 1'$ in which p_i takes no steps and all other processes decide. Such a schedule exists because the

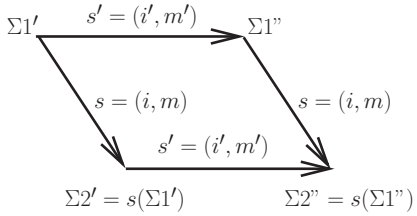

Figure 5.15: Valence contradiction when $i \neq i'$

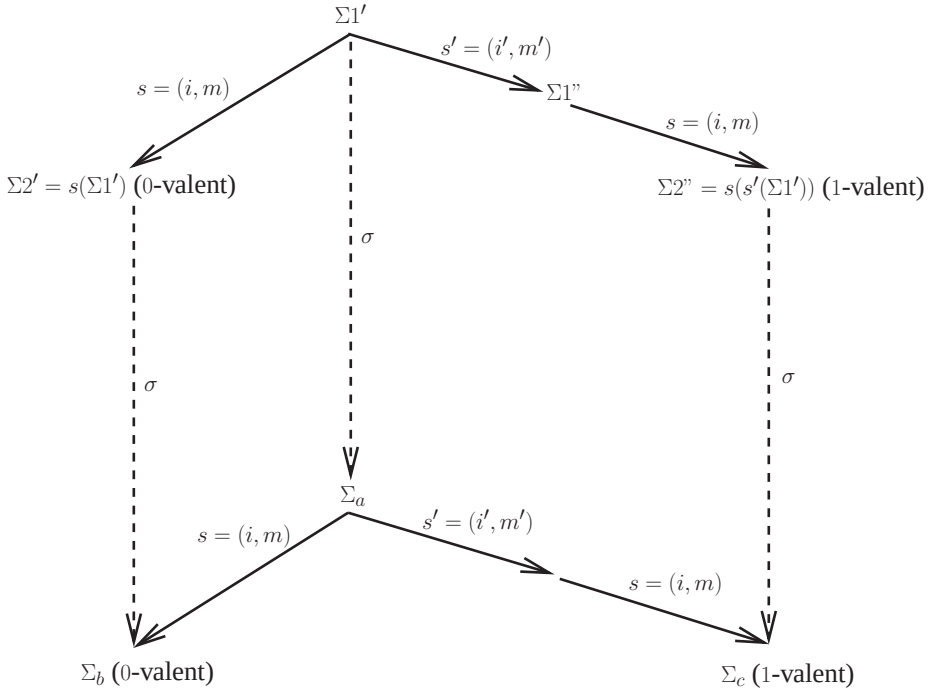

Figure 5.16: Valence contradiction when $i = i'$

algorithm A is correct, and it copes with one process crash. In that schedule, everything appears as if p_i has crashed in $\Sigma 1'$. It follows that $\Sigma_a = \sigma(\Sigma 1')$ is univalent.

As σ includes no step by p_i, the very same schedule σ can be applied to both $\Sigma 2'$ and $\Sigma 2''$ and we obtain the following.

- $\Sigma_b = \sigma(s(\Sigma 1')) = s(\sigma(\Sigma 1')) = s(\Sigma_a)$. This is because, as σ and s are independent, when the schedule $s\ \sigma$ and the schedule $\sigma\ s$ are applied to the same global state $(\Sigma 1')$, they necessarily produce the same global state (Σ_b).
- $\Sigma_c = \sigma(s(s'(\Sigma 1'))) = s(s'(\sigma(\Sigma 1'))) = s(s'(\Sigma_a))$. As before, this is because, as the schedules σ and $s'\ s$ are independent, when the schedule $s'\ s\ \sigma$ and the schedule $\sigma\ s'\ s$ are applied to the same global state $(\Sigma 1')$, they necessarily produce the same global state (Σ_c).

It follows that we have $\Sigma_b = s(\Sigma_a)$ and $\Sigma_c = s(s'(\Sigma_a))$.

As $\Sigma 2'$ is 0-valent, so is Σ_b. Similarly, as $\Sigma 2''$ is 1-valent, so is Σ_c. It then follows from $\Sigma_b = s(\Sigma_a)$ and $\Sigma_c = s(s'(\Sigma_a))$ that Σ_a is bivalent, contradicting the fact that is univalent and concludes the proof of the lemma.

□*Lemma 5.9*

Theorem 5.10 *There is no consensus algorithm A in* $\mathcal{AS}_{n,1}[\emptyset]$.

Proof The proof consists in building an infinite run in which no process decides. The idea is the following. The algorithm A is started in a bivalent global state (that exists due to Lemma 5.8), and then the steps executed by the processes are selected in such a way that the processes proceed from a bivalent global state to a new bivalent state (that exists due to Lemma 5.9). This run has to be admissible (there is at most one process crash, and any message send by a correct process is be eventually received).

The admissible run that is built is actually a failure-free run (each process takes infinitely many steps). The processes are initially placed in queue (in arbitrary order).

1. The initial global state Σ is any bivalent global state. The run R is initialized to the empty sequence. Then, repeatedly, the following sequence is executed.
2. Let p_i be the process at the head of the queue. If the input buffer of p_i contains a message m such that (i, m) is applicable to Σ, then let $s = (i, m)$ be the oldest such step, else let $s = (i, \perp)$.
3. Let us select σ be a schedule such that $s(\sigma(\Sigma))$ is bivalent (such a global state exists due to Lemma 5.9).
4. Assign $s(\sigma(\Sigma))$ to Σ, update R to the sequence $R\ \sigma\ s$, move p_i at the end of the queue, and go to item 2.

It is easy to see that the run R is admissible (no process crash, and any message is delivered and processed). Moreover, the run is infinite and no process ever decides, which concludes the proof of the impossibility result.

□*Theorem 5.10*

Strengthening the impossibility In order to obtain a stronger impossibility result, it is possible to consider a weaker version of the problem. The reader can check that the consensus impossibility result is still valid when the consensus termination property is weakened into "some process eventually decides" (instead of "all non-faulty processes decide").

5.7 BIBLIOGRAPHIC NOTES

- The first explicit formulation of the consensus problem, has appeared in the context of synchronous systems under the name *Byzantine generals problem*. It is due to Lamport, Shostack and Pease [111].
- A strong connection relating distributed agreement problems and error-correcting codes is established in [70]. An informal introduction to agreement problems is presented in [150].
- The state machine approach has been first proposed and developed by Lamport [107, 108]. A survey appears in [154]. The total order broadcast abstraction (TO-URB) has been formalized in [93].
- The construction of the TO-URB abstraction in $\mathcal{AS}_{n,t}[CONS]$ is due to Chandra and Toueg [34], who have also shown that the TO-URB problem and the consensus problem are equivalent in $\mathcal{AS}_{n,t}[\emptyset]$.

  Consensus-based total order multicast has been studied in [69]. The use of total order broadcast for quorum-based data replication is investigated in [152]. Consensus-based total order broadcast algorithms can save consensus executions in some execution patterns. Such an approach is presented in [135].
- The universality of consensus objects to build any concurrent object, defined by a sequential specification, in asynchronous systems prone to process crashes is due to Herlihy [96].
- The non-blocking atomic commit problem originated in databases [23, 81, 156]. The first investigations of its relation with the consensus problem are due to Hadzilacos [92] and Guerraoui [82].

  The class $?P$ of anonymous failure detectors is due to Guerraoui [84]. The weakest class of failure detectors to solve the non-blocking atomic commit problem is described in [48] where it is shown that none of these problems can help solving the other in $\mathcal{AS}_{n,t}[\emptyset]$.
- The construction of uniform reliable broadcast from binary consensus objects and the construction of total order uniform reliable broadcast from multivalued consensus objects in asynchronous systems with fair channels are due to Zhang and Chen [162].
- The impossibility of solving consensus in asynchronous message-passing systems prone to even a single process crash failure is due to Fischer, Lynch and Paterson [68]. This fundamental result is known in the literature under the acronym FLP. It is one of the most celebrated results of fault-tolerant distributed computing.

- The first proof of the impossibility of consensus in asynchronous read/write shared memory systems prone to even a single process crash appeared in [116]. Another proof is given in [96]. See also [97].

CHAPTER 6

Consensus Algorithms for Asynchronous Systems Enriched with Various Failure Detectors

This chapter is devoted to consensus algorithms in asynchronous message-passing systems prone to process crashes, enriched with various types of failure detectors (oracles that provide processes with information on failures). All these algorithms assume that each non-faulty process p_i proposes a value to the consensus instance, i.e., invokes the operation "propose (v_i)" where v_i is the value it proposes to that instance.

Many algorithms are presented, each designed for an appropriately enriched version of the base system model $\mathcal{AS}_{n,t}[\emptyset]$. Before entering into the technical developments, this list of items describes the structure of the chapter.

- First, Section 6.1 considers $\mathcal{AS}_{n,t}[\emptyset]$ enriched with a perfect failure detector. It presents two consensus algorithms for that model, that works for any value of t. In the first algorithm the processes decide in $t+1$ asynchronous rounds (communication steps), The second algorithm allows for early decision, namely, a process decides (and stops) in at most $\min(f+2, t+1)$ rounds (where f is the actual number of process crashes).
- Section 6.2 introduces the weakest failure detector class (Ω) that allows consensus to be solved in $\mathcal{AS}_{n,t}[t < n/2]$. Then, Section 6.3 presents a consensus algorithm suited to $\mathcal{AS}_{n,t}[t < n/2, \Omega]$ and studies its properties.
- Section 6.4 considers then the case where, instead of a failure detector, the underlying system $\mathcal{AS}_{n,t}[t < n/2]$ is enriched with a random number generator; hence the system becomes a randomized system. Several binary consensus algorithms for different types of randomized systems are presented.

  The notion of an hybrid system is also presented. Such a system is enriched with both a failure detector of the class Ω and a random number generator. A corresponding algorithm is described and analyzed.
- Assuming that the underlying system $\mathcal{AS}_{n,t}[\emptyset]$ is enriched with a binary consensus algorithm, Section 6.5 presents two algorithms that solve multivalued consensus on top of such an enriched system.
- Section 6.6 investigates the cases where consensus can be solved in one communication step.

- Finally, Section 6.7 addresses anonymous asynchronous systems. It presents failure detector classes suited to this type of systems and associated consensus algorithms.

As we have seen in previous chapters, the operation ' 'broadcast m" that is used is not a reliable broadcast abstraction. It is only a shortcut for "**for each** $j \in \{1, \ldots, n\}$ **do** send m to p_j **end for**". (The sending by a process of a message to itself can be trivially eliminated. We nevertheless keep this sending in order to have a simpler statement of the algorithms that are presented.)

6.1 ENRICHING THE ASYNCHRONOUS SYSTEM WITH A PERFECT FAILURE DETECTOR

This section presents a few simple algorithms that solve the consensus problem in $\mathcal{AS}_{n,t}[P]$ where P is the class of perfect failure detectors.

Let us remember that any failure detector of that class P provides each process p_i with a set $suspected_i$ that (1) eventually contains all crashed processes (completeness), and (2) never contains a process before it crashes (strong accuracy).

6.1.1 A SIMPLE ALGORITHM BASED ON A ROTATING COORDINATOR

Principle and description of the algorithm An algorithm that constructs a consensus object in $\mathcal{AS}_{n,t}[P]$ is described in Figure 6.1, where v_i is the value proposed by process p_i. This algorithm is coordinator-based: the processes proceed in consecutive asynchronous rounds, and each round is coordinated by a process. More precisely, we have the following:

- Each process p_i executes a sequence of $(t + 1)$ asynchronous rounds at the end of which it decides (if it has not crashed before). As rounds are asynchronous (they are not given for free by an external device) each p_i has to manage a local variable r_i that contains its current round number. Let us observe that, due to asynchrony, nothing prevent two processes to be at different rounds at the same time.
- Each process p_i manages a local variable est_i that contains its current estimate of the decision value (so, est_i is initialized to the value it proposes, namely v_i). Each round is statically assigned a coordinator: round r is coordinated by process p_r. This means that, during that round, p_r tries to impose its current estimate as the decision value. To that end, p_r broadcasts the message EST(est_r).

  If a process receives EST (est) from p_r during round r, it updates its estimate of the decision value est_i to the value of the estimate est it has received, and then proceeds to the next round. If it suspects p_r, it proceeds directly to the next round.

  Let us notice that a message does not carry its sending round number.

Theorem 6.1 *The algorithm described in Figure 6.1 solves the consensus problem in* $\mathcal{AS}_{n,t}[P]$.

```
operation propose (v_i):
  est_i ← v_i; r_i ← 1;
  while r_i ≤ t + 1 do
      begin asynchronous round
      if (r_i = i) then broadcast EST (est_i) end if;
      wait until ((EST (est) received from p_{r_i}) ∨ (r_i ∈ suspected_i));
      if (EST (est) received from p_{r_i}) then est_i ← est end if;
      r_i ← r_i + 1
      end asynchronous round
  end while;
  return (est_i).
```

Figure 6.1: A coordinator-based consensus algorithm for $\mathcal{AS}_{n,t}[P]$ (code for p_i)

Proof The proof of the integrity property (a process decides at most once) is trivial: if it does not crash before the end of the last round, a process decides when it executes the return () statement that terminates its participation to the algorithm. The proof of the validity property (a decided value is a proposed value) follows from the observation that any EST() message carries the current value of an est_i local variable, and initially these variables contain only proposed values.

Proof of the termination property. The proof consists in showing that no non-faulty process blocks forever in the **wait until** statement executed during a round. Let us consider the first round. If p_1 is non-faulty it invokes propose (v) and, consequently, sends the message EST (v_1) to each process that (as the channels are reliable) eventually receives it. If p_1 crashes, we eventually have $1 \in suspected_i$ (let us remember 1 is p_1's identity). It follows that no process p_i can block forever during the first round and consequently each non-faulty process enters the second round. Applying inductively the same reasoning to rounds 2, 3, etc., until $t + 1$, allows us to conclude that each non-faulty process returns (decides) a value.

Proof of the agreement property. As t is an upper bound on the number of faulty processes, it follows that at least one among the $(t + 1)$ processes $p_1, \ldots, p_{t+1}$, is non-faulty. Let p_x be the first of these non-faulty processes. Due to the termination property, p_x executes the round $r = x$. As it is the coordinator of this round, it sends its current estimate $est_x = v$ to every process. As it is non-faulty, no process p_i suspects it, which implies that the predicate $x \in suspected_i$ remains forever false. Consequently, each process p_i receives EST (v) and executes $est_i \leftarrow v$. It follows that all processes that terminate round x have the same estimate value v. The agreement property follows from the observation that no value different from v can thereafter be sent in a later round. □*Theorem* 6.1

Cost It is easy to see that the algorithm requires $(t + 1)$ (asynchronous) rounds. Moreover, in each round, at most one process broadcasts a message whose size is independent of the algorithm. So, in

the worst case (no crash), $n(t+1)$ messages are sent (considering that a process does send message to itself). Let $|v|$ be the bit size of a proposed value. The bit communication complexity of the P-based consensus algorithm is consequently $n(t+1)|v|$.

P is not the weakest failure detector class to solve consensus Let us consider the failure detector class denoted S defined by the following properties:

- Completeness. Each set $suspected_i$ eventually contains all faulty processes. (Same as for P.)
- Weak accuracy. Some correct processes is never suspected.

On the one hand, both P and S have the same completeness property (this property is used to prevent the processes that use them to block forever waiting for messages from crashed processes). On the other hand, the accuracy property of S is strictly weaker than the one of P. A failure detector of the class P never suspects a process before it crashes, while a failure detector of the class S can erroneously suspect not only faulty processes before they crash, but also (intermittently or forever) all but one non-faulty processes. Hence, it is not possible to build a failure detector of the class P in $\mathcal{AS}_{n,t}[S]$. The class P is strictly stronger than the class S.

The reader can check that the algorithm described in Figure 6.1 solves the consensus problem in $\mathcal{AS}_{n,t}[S]$, when each process is required to execute n rounds (instead of $t+1$).

The proof is the same as for this algorithm. The important point is that, due to the accuracy property of S and the fact that $t = n-1$, one of the $t+1 = n$ coordinators is necessarily a non-faulty process that is never suspected.

The class S is strictly weaker than the class P. This follows from the observation that S does not prevent a correct process from being always suspected. Hence, P cannot be the weakest class of failure detectors that allows solving consensus in an asynchronous system prone to process crashes.

Fairness with respect to the processes When we consider the algorithm described in Figure 6.1, only the processes $p_1, \ldots, p_{t+1}$ are round coordinators and, consequently, the value decided can only be one of the values they propose. No value proposed by processes p_{t+2} until p_n can ever be decided (except the ones of them that, incidentally, are proposed by a coordinator). In that sense, the construction is *unfair* with respect to the set of processes. Hence, the definition of the following fairness property.

A consensus algorithm is *fair with respect to the processes* if the value that is decided is not predetermined by process identities. If the value v proposed by process p_i is decided, the same value v proposed by the same process could have been decided if, instead of having the identities $1, 2, \ldots, n$ the processes had the identities $\pi(1), \pi(2), \ldots, \pi(n)$ where $\pi()$ is any permutation of the set $\{1, 2, \ldots, n\}$. Fairness guarantees independence between the value that is decided and the identity of the process that proposes it. While this property does not belong to the consensus definition, some applications can benefit from it.

Shuffling the inputs in order to ensure fairness A simple way to ensure the fairness property consists in adding a preliminary communication round before the **while** loop. During this shuffling round,

each process sends its value to the $(t+1)$ statically defined coordinators, and a coordinator adopts the first value it receives (from any process). The code of this preliminary round is the following for p_i.

```
broadcast SHUFFLE (est_i):
if i ∈{1, ..., t + 1} then
    wait (SHUFFLE (est) from any process p_j );
    est_i ← est
end if.
```

It is easy to see that this shuffling of the proposed values can assign arbitrarily any proposed value with any coordinator. From an operational point of view, the result of this re-assignment depends on the failure pattern and asynchrony pattern. The P-based algorithm obtained that way is fair (and simple), but requires $(t+2)$ rounds.

6.1.2 A FAIR ALGORITHM IN $(t+1)$ ROUNDS

It is actually possible to design a fair P-based algorithm that requires only $t+1$ rounds. Such an algorithm is described below in Figure 6.3. Let us remember that $t+1$ rounds is a lower bound for consensus in synchronous systems prone to up to t process crashes (see bibliographic notes). Hence, the algorithm that is presented for $\mathcal{AS}_{n,t}[P]$ shows that, as far as consensus is concerned, the same efficiency can be attained in $\mathcal{AS}_{n,t}[P]$.

Synchronous system In a round-based synchronous system the round notion is given for free. Processes progress in a lock-step manner: they all execute the same round at the same time. During a round r a process executes the following steps.

- First p_i sends a round r message to the other processes. If it crashes while executing this step, an arbitrary subset of processes receive its round r message.
- Then, p_i waits for round r messages from other processes.
- And finally p_i executes local computation.

The progress from a round r to round $r+1$ is not managed by each process separately (as done in an asynchronous system), but governed by the system that directs the processes to proceed from r to $r+1$. The most important feature of a synchronous system is that a message sent in a round is received in the very same round. It follows that if a process p_i does not receive a round r message from a process p_j, it can safely conclude that p_j has crashed; the absence of message indicates a process crash.

Strengthening the definition of P In order to point out differences between a synchronous system and an asynchronous system enriched with P, let us consider Figure 6.2 where the round boundaries are indicated by bold lines. There are two processes p_i and p_j. During round $(r-1)$, each receives

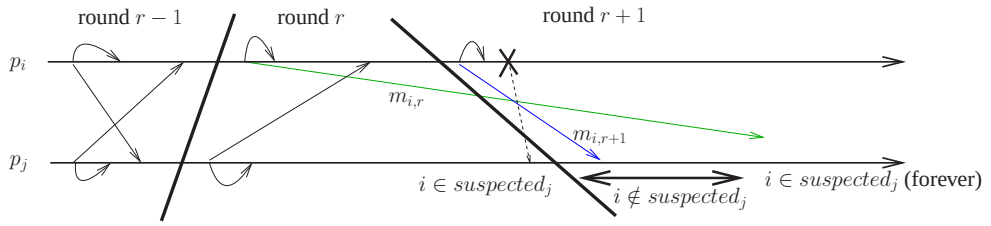

Figure 6.2: Rounds, asynchrony and perfect failure detector

two messages and proceeds to round r. During round r, p_i sends the message $m_{i,r}$, receives two messages, and proceeds to round $(r+1)$. Moreover, in round $(r+1)$, p_i sends the message $m_{i,r+1}$ and crashes. On its side, during round r, after having sent a message to p_i, the process p_j waits for a message from p_i. But, in the scenario described on the figure, the corresponding message $m_{i,r}$ is very slow (asynchrony), and p_j is informed of the crash of p_i before receiving $m_{i,r}$. As P provides it with safe suspicions, p_j stops waiting and proceeds to round $(r+1)$.

Then during round $(r+1)$ the following can happen. Let us remember that any failure detector of the class P ensures that no process is suspected before it crashes (accuracy property), and if a process p_i crashes, there is a time after which i remains forever in $suspected_j$ (completeness property). This does not prevent the scenario described in the figure that depicts a finite period during which i does not belong to $suspected_j$. This period starts when i is added to $suspected_i$ for the first time, and it ends when it is added to $suspected_j$ for the last time. If this occurs, p_j does not suspect p_i when it receives the message $m_{i,r+1}$ (and consequently considers that message): p_j has then received from p_i a round $(r+1)$ message and no round r message.

This behavior can never occur in a synchronous system. (In a synchronous system, the round boundary lines are "orthogonal" with respect to the time axis.) The message $m_{i,r}$ would be received by p_j during round r, and the message $m_{i,r+1}$ during round $(r+1)$. In a synchronous system with the crash pattern presented in Figure 6.2, p_j would discover the crash of p_i during round $(r+2)$: as it will not receive a round $(r+2)$ message from p_i, p_j will conclude that p_i has crashed during round $(r+1)$ or $(r+2)$.

A fair consensus algorithm for $\mathcal{AS}_{n,t}[P]$ The idea is to simulate the behavior of a synchronous consensus algorithm in $\mathcal{AS}_{n,t}[P]$. In order to guarantee the fairness property, the algorithm is based on the following principle that consists in directing each process to broadcast its current estimate during each round (this is called a *flooding technique*). Moreover, a deterministic function is used to select the decided value (e.g., a process decides the smallest value it has even seen).

In order to prevent the bad phenomenon described in Figure 6.2, each process p_i manages a set $crashed_i$ (that is initially empty). Differently from the set $suspected_i$, the set $crashed_i$ is monotone, i.e., $\forall \tau, \tau' : (\tau \leq \tau') \Rightarrow (crashed_i^{\tau} \subseteq crashed_i^{\tau'})$ (where $crashed_i^x$ is the value of $crashed_i$ at time x). To implement $crashed_i$, p_i manages a thread that forever executes $crashed_i \leftarrow$

$crashed_i \cup suspected_i$. It is easy to see, that the sets $\{crashed_i\}_{1 \leq i \leq n}$ of the processes satisfy the completeness and accuracy properties of P plus the monotonicity property that rules out the bad phenomenon described in Figure 6.2. This thread is not described in Figure 6.3.

Each process p_i manages the following local variables.

- est_i: p_i's current estimate of the decision value. Its initial value is v_i (value proposed by p_i).
- r_i: p_i's current round number. Its initial value is 1.
- Let us recall that p_i can read $crashed_i$ that is a monotone version of $suspected_i$.

The scope of the other auxiliary local variables is one round. Their meaning follows clearly from their identifier.

```
operation propose (v_i):
(1)  est_i ← v_i; r_i ← 1;
(2)  while r_i ≤ t + 1 do
(3)      begin asynchronous round
(4)      broadcast EST (r_i, est_i);
(5)      wait until (∀ j ∉ crashed_i: (EST (r_i, −) received from p_j);
(6)      let rec_from_i = {1, ..., n} \ crashed_i;
(7)      let est_rec_i = {est received during r_i from the processes in rec_from_i};
(8)      est_i ← min(est_rec_i);
(9)      r_i ← r_i + 1
(10)     end asynchronous round
(11) end while;
(12) return (est_i).
```

Figure 6.3: A fair consensus algorithm for $\mathcal{AS}_{n,t}[P]$ (code for p_i)

As in a round-based synchronous system, the behavior of a process p_i during a round r is made up of two phases, a communication phase followed by a local computation phase.

- Communication phase. A process p_i first broadcasts EST (r_i, est_i) to inform the other processes on its current state. Then, it waits until it has received a message EST $(r_i, -)$ from each process p_j but the processes that, to its knowledge, have crashed. Let rec_from_i denote the set of processes from which p_i considers and processes the messages it has received.
- Computation phase. Then p_i computes its new estimate value, that is the smallest value carried by a message it has received from a process in rec_from_i.

Notation Let $min(r)$ denote the smallest value of the current estimates of the processes that terminate round r (i.e., after the updates of est_i during round r). Moreover, $min(0)$ denotes the smallest proposed value.

Theorem 6.2 *The algorithm described in Figure 6.3 solves consensus in* $\mathcal{AS}_{n,t}[P]$. *Moreover, the decided value is* $\min(t)$.

Proof

Proof of the validity property. This property follows trivially from the following observations: (a) a process decides the current value of its estimate, (b) the initial values of the estimates are the proposed values, and (c) when updated, a local estimate is set to the minimum of a set of current estimates it has received.

Proof of the termination property. Due to the completeness property of the failure detector, a process cannot be blocked forever in the **wait** statement of line 5. Consequently, all correct processes terminate round $t + 1$ and decide.

Proof of the agreement property. In order to establish this property, we first prove a claim that captures the synchronization provided by the net effect of the **wait** statement and the use of a class P failure detector.

Claim C1. For any r, $1 \leq r \leq t + 1$, when a process p_i terminates round r (this occurs when it sets r_i to $r + 1$), there is no alive process in a round $r' < r$. (At any time, any two alive processes are separated by at most one round.)

Proof of the claim C1. The claim follows from the accuracy property of P. If p_j is alive and executing round $r - 1$ or a smaller round (or is alive and has not yet started if $r = 0$), we have $j \notin suspected_i$, which implies $j \notin crashed_i$. It then follows from the predicate "$\forall j \notin crashed_i$: (EST $(r_i, -)$ received from p_j" (**wait** statement) that controls the progress of p_i from r to $r + 1$, that p_i cannot proceed to $r + 1$. End of proof of claim C1.

The proof of the agreement property is by contradiction. Let us suppose that some process decides a value different from $min(t)$. We show that there are then $(t + 1)$ crashes, which contradicts the model definition. To that end, let us consider the following definitions where r is any round number such that $1 \leq r \leq t + 1$.

- $Q(r) = \{\ p_x \mid est_x \leq min(t)$ when p_x starts round $r\ \}$.
- $R(r) = Q(r) \setminus (R(0) \cup R(1) \ldots R(r - 1))$ with $R(0) = \emptyset$. $R(r)$ is the set of processes p_x that start round r with $est_x \leq min(t)$ but did not start a round $r' < r$ with $est_x \leq min(t)$.

By a slight abuse of language, we say that, when a process decides, it is executing the communication-free round $t + 2$. Hence, a process that crashes after having decided crashes during round $t + 2$.

Claim C2. If agreement is violated at the end of round $(t + 1)$, $\forall r :\ 1 \leq r \leq t + 1$: (i) $|R(r)| \geq 1$, and (ii) all processes in $R(r)$ crash while executing round r or $(r + 1)$.

It follows from claim C2 that, for each round r, $1 \leq r \leq t + 1$, at least one process crashes during r or $r + 1$, from which we conclude that $(t + 1)$ processes crash. This contradicts the fact

that at most t processes may crash and proves, consequently, the agreement property.

Proof of claim C2. The proof is by induction on r. Let us first consider the base case $r = 1$.

- Proof of (i): $|R(1)| \geq 1$. As $min(t)$ is the smallest estimate value at the end of round t, and the algorithm does not create estimate values, there is at least one process whose proposed value is equal to $min(t)$.

- Proof of (ii): all processes in $R(1)$ crash while executing round 1 or round 2. Let us assume by contradiction that there is a process $p_i \in R(1)$ that does not crash while executing round 1 or round 2 (hence, p_i terminates round 2). Due to the claim C1, there is no process alive in round 1 when p_i terminates $r = 2$. It follows that all processes that have terminated the first round have received and processed the message EST $(1, est_i)$, with $est_i \leq min(t)$. Consequently, every process p_j that enters the second round has updated its estimate est_j to a value $\leq min(t)$ before entering the second round. Since estimates can only decrease during the execution, this implies that any process p_j that starts the last round has an estimate $est_j \leq min(t)$. By the very definition of $min(t)$ it means that such an estimate $est_j = min(t)$. Hence all processes that start the last round have the same estimate, $min(t)$ and then no other value can be decided. It contradicts the fact that (by assumption) some process decides a value different from $min(t)$. This contradiction concludes the proof of item (ii) for the base case.

Let us now consider the induction step. So, assuming that items (i) and (ii) are satisfied from the first round until round $r - 1$, let us show that they are satisfied for round $r \leq t + 1$.

- Proof of (i): $|R(r)| \geq 1$. Let p_i be a process that terminates round r and such that $est_i \leq min(t)$ at the end of round r. Due to the definition of $min(t)$ and the fact that there is no creation of value, such a process necessarily exists. There are two cases.
  Case 1: $p_i \in Q(r)$. As p_i terminates round r, and no process in $R(r-1), R(r-2), \ldots, R(1)$ terminates round r (induction assumption), it follows that $p_i \notin R(0) \cup R(1) \ldots R(r-1)$. By the definition of $R(r)$, it follows that $p_i \in R(r)$. Hence, $|R(r)| \geq 1$, which proves item (i) for the case.
  Case 2: $p_i \notin Q(r)$. As $p_i \notin Q(r)$, it follows that p_i has received a message EST (r, est_j) from a process p_j such that $est_j \leq min(t)$ at the beginning of r, i.e., $p_j \in Q(r)$.

  Claim C3 : $p_j \notin Q(r-1)$.
  Assuming C3, as $p_j \notin Q(r-1)$ and due to the fact that est_j can only decrease during an execution, $p_j \notin Q(r')$ for all $r' < r$. Since $p_j \in Q(r)$ we have $p_j \in R(r)$, and, consequently, $|R(r)| \geq 1$, which proves item (i) for Case 2.

  Proof of the claim C3. As p_i has received EST (r, est_j) with $est_j \leq min(t)$ from process p_j during round r, we conclude from the monotonicity property of the set $crashed_i$, that p_i has received a message EST $(r-1, -)$ from process p_j during round $r - 1$. Let us assume by contradiction that $p_j \in Q(r-1)$. It then follows that p_i has received EST $(r-1, est_j)$ from

p_j during round $r-1$ with $est_j \leq min(t)$, and, consequently, we have $est_i \leq min(t)$, at the end of round $r-1$, i.e., $p_i \in Q(r)$ which contradicts the assumption associated with Case 2. Hence, $p_j \notin Q(r-1)$. End of proof of claim C3.

- Proof of (ii) of claim C2: All processes in $R(r)$ crash while executing round r or round $r+1$. The proof of this item is verbatim the proof as for the base case after changing round numbers 1 and 2 by round numbers r and $r+1$, respectively. So, it is not repeated here. End of proof of claim C2.

$\square_{Theorem\ 6.2}$

6.1.3 EARLY DECISION

Motivation Failures do occur but are not frequent. Hence, it is interesting to have a consensus algorithm for the model $\mathcal{AS}_{n,t}[P]$, in which the processes decide and halt in very few rounds when there are few failures. This is the *early decision/halting* notion. Let f be the number of processes that crash in a run, $0 \leq f \leq t$. This section presents an algorithm that directs the processes to decide and stop in $\min(f+2, t+1)$ rounds when executed in the model $\mathcal{AS}_{n,t}[P]$. This means that a process decides and stops in two rounds in failure-free runs.

The lower bound on the number of rounds for early deciding/stopping consensus in synchronous systems $\min(f+2, t+1)$. The algorithm that follows shows that this bound can be attained in asynchronous systems such as $\mathcal{AS}_{n,t}[P]$.

An early deciding/stopping algorithm The principle that underlies the proposed early-deciding/stopping algorithm (described in Figure 6.4) is as follows. A process decides as soon as it knows that (1) its current estimate of the decision value is the smallest estimate value present in the system, and (2) at least one non-faulty process knows that value. This algorithm extends the previous algorithm as follows. A line tagged (x) or (x') corresponds to line (x) in the non early-deciding algorithm. The lines tagged (Ny) are new lines that ensure early decision/stopping. In addition to the previous local variables, the algorithm uses the following ones.

- i_know_i is a boolean, initialized to *false*. This boolean is set to *true* at the end of a round r, if p_i learned -during that round- that its current estimate est_i contains the smallest estimate value among the processes that start round r. This means that est_i contains the smallest value still present in the system. This boolean is stable (once true, it remains true forever).
- $they\_know_i$ is a set, initialized to $\emptyset$. This set contains the identities of the processes that, to p_i's knowledge, have the smallest estimate value present in the system.

Behavior of a process As in the previous algorithm, the behavior of a process p_i during a round r is made up of two phases, a communication phase followed by a local computation phase.

- Communication phase. A process p_i first broadcasts EST (r_i, est_i, i_know_i) to inform the other processes on its current state. Then, it waits until it has received a message EST $(r_i, -, -)$

```
operation propose (v_i):
(1')  est_i ← v_i; r_i ← 1; they_know_i ← ∅; i_know_i ← false;
(2)   while r_i ≤ t + 1 do
(3)       begin asynchronous round
(4')      broadcast EST (r_i, est_i, i_know_i);
(5')      wait until (∀ j ∉ ((crashed_i ∪ they_know_i) \ {i}): (EST (r_i, −, −) received from p_j));
(N1)      let crashed_or_knowing_i be the set (crashed_i ∪ they_known_i) when the wait terminates;
(6')      let rec_from_i = {1, ..., n} \ crashed_or_knowing_i;
(7)       let est_rec_i = {est received during r_i from the processes in rec_from_i};
(8)       est_i ← min(est_rec_i);
(N2)      they_know_i ← they_know_i ∪ {x | EST (r_i, −, true) rec. from p_x with x ∈ rec_from_i};
(N3)      if ((|crashed_i ∪ they_know_i| ≥ t + 1) ∧ i_know_i) then return (est_i) end if;
(N4)      let some_knows_i = (∃ EST (r_i, −, true) received from p_x with x ∈ rec_from_i);
(N5)      i_know_i ← (some_knows_i) ∨ (|rec_from_i| ≥ n − r_i + 1);
(9)       r_i ← r_i + 1
(10)      end asynchronous round
(11)  end while;
(12)  return (est_i).
```

Figure 6.4: Early deciding/stopping consensus in $\min(f+2, t+1)$ rounds in $\mathcal{AS}_{n,t}[P]$ (code for p_i)

from each process p_j but the processes that, to its knowledge, have crashed or know the smallest estimate value. Moreover, we assume that it always receives its own message (while not necessary, this assumption simplifies the proof). Now, rec_from_i denotes the set of processes from which p_i considers and processes the messages it has received (those are the messages from itself and the processes not in $crashed_i \cup they\_know_i$).

Let us observe that a process p_k that knows the smallest estimate value can have decided and stopped in a previous round, and consequently, p_i could block forever if it was directed to wait for a message from p_k.

- Computation phase. Then p_i computes its new estimate value, that is the smallest value carried by a message it has received from a process in rec_from_i. It also updates the set $they\_know_i$ according to the boolean values carried by the EST $(r_i, -, -)$ messages it has received from the processes in rec_from_i. Then p_i strives to early decide. To that end, it does the following.
  - If it knows the smallest among the estimate values of the processes that start round r (i.e., i_know_i is true), and knows also that value is know by at least one non-faulty process (i.e., $|crashed_i \cup they\_know_i| \geq t+1$), then p_i decides and stops its participation to the algorithm.
  - Otherwise, p_i updates its local state before starting a new round. To that end, it first computes the value of i_know_i. If p_i has received a message EST $(r_i, v, true)$, it has learned from another process that v is the smallest estimate value, and consequently sets i_know_i to $true$.

As we will see in the proof, if $|rec\_from_i| \geq n - r + 1$, then p_i discovers that the current value of its estimate is the smallest estimate value in the system. In that case, it sets i_know_i to the value *true*.

Theorem 6.3 *Let f be the number of processes that crash in a run ($0 \leq f \leq t$). The algorithm described in Figure 6.4 solves consensus in at most* $\min(f + 2, t + 1)$ *rounds in* $\mathcal{AS}_{n,t}[P]$.

Notation Let xxx_i be a local variable of process p_i. It is easy to see, that such a variable is updated at most once in a round r. The notation $xxx_i(r)$ is used to denote its value at the end of round r.

Proof The proof of the validity property is the same as in Theorem 6.2. Before proving the other properties, a few claims are proved.

Claim C1. $i\_know_i(r) \Rightarrow (est_i(r) = min(r - 1))$ (i.e., at the end of round r, p_i knows the smallest estimate value that was present in the system at the end of $r - 1$).
Proof of claim C1. The proof is by induction on the round number.

- If a process p_i updates its boolean i_know_i to true during round 1 (we have then $|rec\_from_i| = n$), it has received n messages containing the n initial estimates. Hence p_i knows the minimum value in the system amongst all the estimate values.
- Let p_i be a process that updates its boolean i_know_i to true during round r. There are two cases.
  - p_i has received a message EST $(r, -, true)$ from p_j. It means that p_j has updated its boolean i_know_j to true in a previous round, and then by induction, p_i knows the minimum value.
  - p_i has received $n - r + 1$ messages in round r and it has never received (in round r or in previous rounds) a message EST $(-, -, true)$. From p_i's point of view, the execution is similar to an execution of the algorithm described in Figure 6.3 in which we would have $t = r - 1$. This follows from the fact that (a) line N5 updates i_know_i only in round r, and (b) lines N1, N2, N3, and N4 do not modify local variables up to round r (they are consequently useless up to that round). Moreover, due to the current values of $i\_know_i = false$ and $they\_know_i = \emptyset$ before line N5 of round r, (1) the lines tagged (x') behave as the line tagged (x) in the algorithm of Figure 6.3, and (2) we have $|rec\_from_i(r)| \geq n - r + 1$. Consequently, at the end of round r, i_know_i is equal to *true*, and it follows from Theorem 6.2that we have then $est_i(r) = min(r - 1)$ (smallest estimate value of processes that terminate round $r - 1$). End of proof of claim C1.

Claim C2. No process blocks forever in a round.

Proof of claim C2. Let us assume by contradiction that a process blocks forever in a round. Let r be the first round during which a process p_i blocks forever. This happens in the **wait** statement where p_i waits for messages EST $(r, -, -)$ from processes that currently are not in $crashed_i \cup they\_know_i$. If a process p_j crashes, it eventually appears in the set $crashed_i$ (let us remember that this set can only increase). Hence, a process that crashes cannot prevent p_i from progressing. If p_j is non-faulty, there are two cases.

- Case 1: p_j has not decided during a round $r' \leq r$. In that case, as by assumption r is the first round during which processes block forever, it follows that p_j sends a message EST $(r, -, -)$. Consequently, such a process p_j cannot block forever p_i in round r.

- Case 2: p_j has decided during a round $r' < r$. Before deciding during r', p_j sent EST $(r', -, \mathit{true})$ to p_i, and (as p_j does not crash) p_i received and processed this message. Hence, $j \in they\_know_i(r')$. As the set $they\_know_i$ never decreases, and $r' < r$, it follows that $j \in they\_know_i(r-1)$ (that is the value of $they\_know_i$ when p_i waits during round r). Hence, in that case also, p_i has not to wait for a message from p_j. End of proof of claim C2.

Proof of agreement property. Let r be the first round during which a process (say p_i) decides. Due to Claim C1, we have $est_i(r) = min(r-1)$. If p_j decides at round r, we also have $est_j(r) = min(r-1)$, from which we conclude that the processes that decide at round r, decide the same value.

Let p_k be a process that proceeds to round $r+1$. Let us observe that, when p_i decides, we have $|crashed_i \cup they\_know_i| \geq t+1$, from which we conclude that there is a non-faulty process p_x that sent EST $(r', est_x(r'-1), i\_knows_x(r'-1))$ (with $i\_knows_x(r'-1) = \mathit{true}$) during round $r' < r$. As p_x is non-faulty, it follows that every non-crashed processes received this message during round r'. Due to Claim C1, we have $i\_knows_x(r'-1) \Rightarrow \big(est_i(r'-1) = min(r'-2)\big)$. (Notice that $r' > 1$ because any i_know_x is initially equal to *false*).

As p_i received $(r', est_x(r'-1), \mathit{true})$, it follows that $est_i(r) = est_x(r'-1)$. Consequently, any process p_k that proceeds to round $r+1$ is such that $est_k(r) = est_x(r'-1) = est_i(r)$, i.e., the estimate values of all the processes that proceed to round $r+1$ are equal to $est_i(r)$. It follows that no value different from $est_i(r)$ can be decided in a round $r'' \geq r$, which completes the proof of that agreement property.

Proof of termination property. Due to Claim C2, no process blocks forever in a round. It follows that if a process neither crashes, nor decides at a round $r < t+1$, it proceeds to round $t+1$ during which it decides (in the worst case at the last line of the algorithm).

Proof of early decision property. Let us first observe that a process executes at most $t+1$ rounds. Hence, considering that $f < t$ processes crash, we have to show that no process decides after round $(f+2)$. There are two cases.

- Case 1: a correct process p_c sends EST $(r, -, \mathit{true})$, during a round $r \leq f + 1$. As p_c is non-faulty, every process that executes round r receives this message during round r. It follows that we have $i\_know_i(r) = \mathit{true}$ at each process p_i that terminates round r. Consequently, every process p_j that executes round $r + 1$ $(\leq f + 2)$ sends EST $(r + 1, -, \mathit{true})$, and hence we have $|crashed_j(r + 1) \cup they\_know_j(r + 1)| = n$. It follows that the early decision predicate is satisfied, and every process that executes round $r + 1$ decides (and stops).

- Case 2: no correct process sends EST $(r, -, \mathit{true})$ during a round $r \leq f + 1$. In that case, no correct process appears in a set $they\_know_x(r)$ for $r \leq f + 1$. Moreover, as f processes crash, and no correct process has decided, at least $(n - f)$ processes send messages during each round. Hence, at any round, we have $|rec\_from_i| \geq n - f$.

  Any process p_i that, during round $f + 1$, evaluates the predicate $|rec\_from_i| \geq n - r + 1$ finds that it is satisfied (because we have then $|rec\_from_i| \geq n - f = n - r + 1$). Consequently, every process p_i that terminates round $f + 1$, sets i_know_i to true.

  Then, for any process p_i that does not crash during $r = f + 2$, and any process p_j, we have $j \in crashed_i$ or $j \in rec\_from_i$ when p_i terminates its **wait** statement. Moreover, if $j \in rec\_from_i$, p_i has received EST $(f + 2, -, \mathit{true})$ from p_j. It follows that $|crashed_i(f + 2) \cup they\_know_i(f + 2)| = n$. Consequently, during round $f + 2$, p_i executes return(), i.e., it decides and stops.

□*Theorem* 6.3

6.2 THE WEAKEST FAILURE DETECTOR CLASS TO SOLVE CONSENSUS

The weakest failure detector class to solve consensus As we have seen, the class P of perfect failure detectors is not the weakest class of failure detectors that permit to solve consensus. Neither is the class S previously described. This means that these failure detector classes provide the processes with more information on failures than what is sufficient to solve consensus in an asynchronous message-passing system prone to process crashes.

The weakest failure detector class to solve consensus in such a system is the combination of two failure detector classes Σ and Ω and is consequently denoted $\Sigma \times \Omega$. It provides each process with two outputs, one associated with Σ, the other one with Ω.

- The class Σ is the class of quorum failure detectors introduced in the second chapter, where it has been shown to be the weakest failure detector class for constructing a shared register in a crash-prone asynchronous message-passing system.

  A failure detector of the class Σ provides each process with a quorum local variable (set of process identities) such that each quorum eventually contains only non-faulty processes, and the values of any two quorums, each considered at any time, do intersect.

- The class Ω, formally defined below, is the class of *eventual leader* failure detectors. It provides each process with a local variable such that eventually all these local variables contain forever the identity of the same non-faulty process.

The class Ω of eventual leader failure detectors This class of failure detectors provides each process p_i with a local variable $leader_i$ such that the set of local variables $\{leader_i\}_{1 \leq i \leq n}$ collectively satisfy the following properties.

Let $leader_i^\tau$ be the value of $leader_i$ at time τ (considering the failure detector history H associated with Ω we have $H(i, \tau) \equiv leader_i^\tau$). Let us remember a few definitions stated in Chapter 2. F denotes a crash pattern ($F(\tau)$ is the set of processes crashed at time τ), $Faulty(F)$ the set of processes that crash in the failure pattern F, and $Correct(F)$ the set of processes that are non-faulty in the failure pattern F.

- Validity. $\forall i: \forall \tau: leader_i^\tau$ contains a process identity.
- Eventual leadership. $\exists \ell \in Correct(F), \exists \tau: \forall \tau' \geq \tau: \forall i \in Correct(F): leader_i^{\tau'} = \ell$.

These properties state that a unique leader is eventually elected; this leader is not a faulty process, but there is no knowledge of when this leader is elected. Several leaders can co-exist during an arbitrary long (but finite) period of time, and there is no way for a process to know when this anarchy period is over. During the anarchy period, it is possible that crashed processes are considered as leaders by non-faulty processes and different processes may have different leaders.

The proof that Ω is the weakest class of failure detectors that allows the consensus problem to be solved despite asynchrony, and at most, t process crashes (with $1 \leq t < n/2$) is pretty involved and very technical. It is not presented here.

The system model $\mathcal{AS}_{n,t}[t < n/2, \Omega]$ As we have seen in second chapter, the assumption of a majority of non-faulty processes allows implementing a failure detector of the class Σ. Hence, in the following, we consider only systems that satisfy the assumption $t < n/2$. The asynchronous model considered is consequently $\mathcal{AS}_{n,t}[t < n/2, \Omega]$, i.e., $\mathcal{AS}_{n,t}[t < n/2]$ enriched with any failure detector of the class Ω.

Let p_i and p_j be any pair of processes. An algorithm can easily benefit from the assumption $n > 2t$ to force p_i and p_j to receive at least one message broadcast by the same process. To that end, let us direct p_i and p_j to wait for messages broadcast by $n - t$ distinct processes, and let Q_i (resp., Q_j) be the set of processes from which p_i (resp., p_j) receives a message. We have $|Q_i| = |Q_j| = n - t > n/2 > t$ (each set Q_i and Q_j is a majority set). Hence, $Q_i \cap Q_j \neq \emptyset$, and there is at least one process that belongs to both Q_i and Q_j.

As a quorum failure detector provides the processes with the same intersection property (without assuming the requirement of a majority of correct processes), the algorithms described in this chapter remain correct when the non-empty intersection property provided by $t < n/2$ is obtained by a quorum failure detector. As it is very simple (and some examples have been given in Chapter 4 devoted to the construction of the URB-broadcast abstraction in presence of fair

channels), the replacement of the assumption $t < n/2$ by a failure detector of the class Σ in these algorithms is left to the reader.

6.3 A ZERO-DEGRADING CONSENSUS ALGORITHM FOR $\mathcal{AS}_{n,t}[t < n/2, \Omega]$

The Ω-based consensus algorithm presented in this section satisfies several noteworthy properties, including the zero-degradation property. That property will be defined in Section 6.3.4 (intuitively, this property says that failures that occurred before a consensus instance is started must not impact on the efficiency of that instance).

6.3.1 PRESENTATION OF THE ALGORITHM

Structure of the algorithm Each process p_i proceeds through consecutive asynchronous rounds. Each round is made up of two phases. During the first phase of a round, the processes strive to select the same value called estimate value. This is done with the help of the failure detector Ω. Then, they try to decide during the second phase. This occurs when they obtain the same value at the end of the first phase. Let us observe that, as the rounds are asynchronous, it is possible that not all the processes are at the same round at the same time.

Moreover, when a process is about to decide a value v, it first broadcasts a message DECIDE(v), and then decides by executing the statement return(v). When a process receives a message DECIDE(v), if forwards (i.e., broadcasts) it before deciding. Let us remember that when a processes executes return(v), it terminates participating in the algorithm. The reception of a message DECIDE () can occur at any time. As we will see later, this is to prevent permanent blocking that could otherwise occur. (When a process broadcasts a message DECIDE(v), it can save the sending to itself and to the process from which it receives that message, if it is the case.)

Local variables Each process p_i manages the following local variables.

- r_i: current round number.
- $est1_i$: local estimate of the decision value at the beginning of the first phase of a round.
- $est2_i$: local estimate of the decision value at the beginning of the second phase of a round.
- $my\_leaders_i$ and rec_i are auxiliary variables used by p_i in the first phase and the second phase of a round, perspectively.

The special value $\perp$ denotes a default value which cannot be proposed by a process.

The behavior of a process during the first phase of a round The algorithm executed by every process p_i is described in Figure 6.5. The aim of the first phase of a round r is to provide the processes with the same value v in their local estimate $est2_i$. When this occurs, a decision will be obtained during the second phase of the round r. As we are about to see, this aim is always attained when the eventual leader is elected.