Chapter 3

Reliable Broadcast in the Presence of Process Crashes and Unreliable Channels

The previous chapter presented several constructions for the uniform reliable broadcast (URB) abstraction. These constructions considered the asynchronous underlying system model $CAMP[\emptyset]$ in which processes may crash and channels are reliable. These constructions differ in the quality of service they provide to the application processes, this quality being defined with respect to the order in which the messages are delivered (namely, FIFO or CO order). This order restricts message asynchrony.

This chapter introduces constructions of URB-broadcast suited to asynchronous systems prone to process crashes and unreliable channels, i.e., asynchronous system models weaker than $CAMP_{n,t}[\emptyset]$.

Keywords Asynchronous system, Communication abstraction, Distributed algorithm, Fair channel, Fair lossy channel, Failure detector, Heartbeat failure detector, Impossibility result, Process crash failure, Quiescence property, Reliable broadcast, Uniform reliable broadcast, Theta failure detector, Unreliable channel.

3.1 A System Model with Unreliable Channels

3.1.1 Fairness Notions for Channels

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

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

Fairness with respect to μ-messages Considering a uni-directional channel that allows a process p_i to send messages to a process p_j, let us observe that, at the network level, process p_i can send the same message several times to p_j (for example, message re-transmission is needed to overcome message losses). This channel is *fair with respect to the message type* μ if it satisfies the three following

M. Raynal, *Fault-Tolerant Message-Passing Distributed Systems*,
https://doi.org/10.1007/978-3-319-94141-7_3

properties (all the messages that appear in these properties are messages carried by the channel from p_i to p_j):

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

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

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

3.1.2 Fair Channel (FC) and Fair Lossy Channel

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

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

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

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

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

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

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

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

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

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

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

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

3.1.3 Reliable Channel in the Presence of Process Crashes

An abstraction for the application layer A *reliable* channel is a communication abstraction that neither creates, nor duplicates, nor loses messages. Its definition is at the same abstraction level as

the definition of URB-broadcast. It is an abstraction offered to the application layer, and consequently considers application messages, each of them being unique.

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

- RC-validity. If p_j receives a message m from p_i, then m was previously sent by p_i to p_j.
- RC-integrity. Process p_j receives a message m at most once.
- RC-termination. If p_i completes the sending of k messages to p_j, then, if p_j is non-faulty and executes k times "receive () from p_i", p_j receives k messages from p_i.

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

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

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

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

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

3.1.4 System Model

In the rest of this chapter we consider an asynchronous system made up of n processes prone to process crashes and where each pair of processes is connected by two unreliable but fair channels (one in each direction). This system model is denoted $CAMP_{n,t}$[- FC], namely it is $CAMP_{n,t}[\emptyset]$, weakened by - FC (the fair channel assumption).

3.2 URB-broadcast in $CAMP_{n,t}$[- FC]

This section first presents an URB-broadcast construction suited to the system model $CAMP_{n,t}$[- FC] constrained by the condition $t < n/2$, i.e., any execution of an algorithm in this model assumes that there is a majority of processes – not known in advance – which never crash. This constrained model is consequently denoted $CAMP_{n,t}$[- FC, $t < n/2$]. It is then shown that this additional model assumption is a necessary requirement for the construction when processes are not provided with information on the actual failure pattern.

3.2.1 URB-broadcast in $CAMP_{n,t}[\text{- FC}, t < n/2]$

Principle Designing an algorithm that implements URB-broadcast in $CAMP_{n,t}[\text{- FC}, t < n/2]$ is pretty simple. The construction relies on two simple basic techniques:

- First, use the classical re-transmission technique in order to build a reliable channel on top of a fair channel.
- Second, locally urb-deliver an application message m to the upper application layer only when this message has been received by at least one non-faulty process. As there are at least $(n - t)$ non-faulty processes and $n - t > t$ (model assumption), this means that, without risking remaining blocked forever, a process p_i may urb-deliver m as soon as it knows that at least $(t + 1)$ processes have received a copy of m.

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

The construction The construction is described in Figure 3.1. When a process p_i wants to urb-broadcast a message m, it sends the protocol message MSG (m) to itself (to simplify and without loss of generality we assume there is reliable channel from a process to itself).

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

```
operation URB_broadcast (m) is send MSG (m) to p_i.

when MSG (m) is received from p_k do
(1)   if (first reception of m)
(2)      then allocate rec_by_i[m]; rec_by_i[m] ← {i, k};
(3)           activate task Diffuse_i(m)
(4)      else rec_by_i[m] ← rec_by_i[m] ∪ {k}
(5)   end if.

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

task Diffuse_i(m) is
(7)   repeat forever
(8)      for each j ∈ {1, ..., n} do send MSG (m) to p_j end for
(9)   end repeat.
```

Figure 3.1: Uniform reliable broadcast in $CAMP_{n,t}[\text{- FC}, t < n/2]$ (code for p_i)

When it receives MSG (m) for the first time (line 1), 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) (line 2). Then p_i activates a task, denoted $Diffuse_i(m)$ (line 3). If it is not the first time that MSG (m) has been received, p_i only adds k to $rec\_by_i[m]$ (line 4). $Diffuse_i(m)$ is the local task that is in charge of re-transmitting the protocol message MSG (m) to the other processes in order to ensure the eventual URB-delivery of m, namely p_i repeatedly forwards the protocol message MSG (m) to each other process p_j.

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 (line 6).

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, which is on any protocol message reception (i.e., whatever its MSG or ACK type) is valid throughout this chapter (ACK protocol messages will be used in Section 3.5). However, several local tasks $Diffuse_i(m1)$, $Diffuse_i(m2)$, etc., are allowed to run concurrently.

Remark acknowledgment messages It is important to note that the task $Diffuse_i(m)$ forever sends protocol messages (and consequently never terminates). The use of acknowledgments (which would be used to fill in the set $rec\_by_i[m]$ to prevent useless re-transmissions) cannot prevent this infinite sending of protocol messages, as shown by the following scenario. Let p_j be a process that has crashed before a process p_i issues URB_broadcast (m). In this case p_j will never acknowledge MSG (m), and consequently p_i will forever execute MSG (m) to p_j. To prevent these infinite re-transmissions, processes must be provided with appropriate information on failures. This is the topic addressed in Section 3.5 of this chapter.

Theorem 8. *The algorithm described in Fig. 3.1 implements the* URB-broadcast *abstraction in the system model* $CAMP_{n,t}$[- FC, $t < n/2$].

Proof (The proof of this construction is a simplified version of the proof of the more general construction given in Section 3.5.) The validity property (neither creation nor 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-broadcast abstraction. There are two cases:

- Let us first consider a non-faulty process p_i that urb-broadcasts a message m. We have to show that each non-faulty process urb-delivers m. As p_i is non-faulty, it activates the task $Diffuse_i(m)$ and forever sends MSG (m) to every other process p_j. As the channels are fair, it follows that each non-faulty process p_x eventually receives MSG (m). The first time this occurs, p_x activates the task $Diffuse_x(m)$. Hence, each non-faulty process infinitely often sends MSG (m) to every process. Due to termination property of the fair channels, and the assumption that there is a majority of non-faulty processes, it follows that the set $rec\_by_i[m]$ eventually contains $(t+1)$ process identities (lines 2 and 4). Hence, the URB-delivery condition of m eventually becomes true at every non-faulty process, which proves the theorem for the case of a non-faulty process that urb-broadcasts an application message.
- We have now to prove the second case of the URB-broadcast 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 received the protocol message MSG (m). When this non-faulty process p_i received MSG (m) for the first time, it activated the task $Diffuse_i(m)$. The rest of the proof is then the same as the previous case.

$\square_{Theorem\ 8}$

3.2.2 An Impossibility Result

This section shows that the assumption $t < n/2$ is a necessary requirement on the maximal number of process crashes when one wants to construct URB-broadcast in the system model $CAMP_{n,t}$[- FC]. The proof of this impossibility is based on an "indistinguishability" argument.

Theorem 9. *There is no algorithm implementing* URB-broadcast *in* $CAMP_{n,t}$[- FC, $t \geq n/2$].

Proof The proof is by contradiction. Let us assume that there is an algorithm A that constructs the URB-broadcast abstraction in $CAMP_{n,t}$[- FC, $t \geq n/2$]. 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 this execution, the processes of $P2$ crash initially, and 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 this 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 they urb-deliver m, each process of $P1$ crashes, and all the protocol messages ever sent by a process of $P1$ are lost (and consequently are never received by the processes 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 must be non-faulty in order 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.)

  As, in the execution E_2, no process of $P2$ ever receives a message from a process of $P1$, none of these processes can urb-deliver m, which completes the proof of the theorem.

$\square_{Theorem\ 9}$

Impossibility vs uniformity requirement 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 to "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 (non-uniform) reliable broadcast, and the impossibility result no longer holds. When we look at the construction in Fig. 3.1, 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.

3.3 Failure Detectors: an Approach to Circumvent Impossibilities

3.3.1 The Concept of a Failure Detector

The concept of a *failure detector* is one of the main approaches that have been proposed to circumvent impossibility results in fault-tolerant asynchronous distributed computing models. It is due to T. Chandra and S. Toueg (1996). From an operational point of view, a failure detector can be seen as an oracle made up of several modules, each associated with a process. The module attached to process p_i provides it with hints concerning which processes have failed. Failure detectors are divided into classes based on the particular type of information they provide on failures. Different problems may require different classes of failure detectors in order to be solved in an otherwise fault-prone asynchronous distributed system model.

There are two main characteristics of the failure detector approach, one associated with its software engineering feature, and the other associated with its computability dimension.

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

system (in the following we sometimes say "failure detector FD" for "any failure detector of the class FD"). This *software engineering dimension* of the failure detector approach favors algorithm design, algorithm proof, modularity, and portability.

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

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

Let A be an abstraction (object, problem) that can be solved in a system model enriched with a failure detector FD. The failure detector concept favors separation of concerns as follows:

- Design and prove correct a distributed algorithm that implements (solves) A in a system model enriched with FD.
- Independently from the previous item, investigate the system behavioral properties that have to be satisfied for FD to be implementable, and provide an implementation of FD for these systems.

The computability dimension of failure detectors Given a problem Pb that cannot be solved in an asynchronous system prone to failures (e.g., build URB-broadcast in $CAMP_{n,t}[\text{- FC}, \ t \geq n/2]$), the failure detector approach allows us to investigate and state the minimal assumptions on failures the processes have to be provided with, in order for the problem Pb to be solved. This is the *computability dimension* of the failure detector approach.

An interesting side of this computability dimension lies in the ranking of problems according to the weakest failure detectors that these problems require to be solved. (The notion of "weakest" failure detector for the register problem will be discussed later in the book, e.g., in Chap. 7 and Chap. 17.) This provides us with a failure detector-based method to establish a hierarchy among distributed computing problems.

3.3.2 Formal Definitions

Failure pattern A failure pattern defines a possible set of failures, along with their occurrence times, that can occur during an execution. Formally, a failure pattern is a function $F : \mathbb{N} \to 2^{\Pi}$, where $\mathbb{N}$ is the set of natural numbers (time domain), and 2^{Π} is the power-set of Π (the set of all sets of process identities). The time domain has to be understood as the time of an external observer, which is inaccessible to the processes.

Considering the models with process crash failures (e.g., $CAMP_{n,t}[\emptyset]$), $F(\tau)$ denotes the set of processes that have crashed up to time τ. As a crashed process does not recover, we have $F(\tau) \subseteq F(\tau + 1)$. Let $Faulty(F)$ be a set of processes that crash in an execution with failure pattern F. Let τ_{max} denote the end of that execution. We then have $Faulty(F) = F(\tau_{max})$. As τ_{max} is not known and depends on the execution, and we want to be as general as possible (and not tied to a time-specific class of executions), we (conceptually) consider that an execution never ends, i.e., we consider that $\tau_{max} = +\infty$. We have accordingly $Faulty(F) = \cup_{1 \leq \tau < +\infty} F(\tau) = \lim_{\tau \to +\infty} F(\tau)$. Let $Non\text{-}faulty(F) = \Pi - Faulty(F)$ (the set of processes that do not crash in F). $Correct(F)$ is used as a synonym of $Non\text{-}faulty(F)$.

It is important to notice that the notions of faulty process and correct process are defined with respect to a failure pattern, i.e., to the failure pattern that occurs in a given execution. Different executions might have different failure patterns.

Failure detector history with range $\mathcal{R}$ A *failure detector history with range $\mathcal{R}$* describes the behavior of a failure detector during an execution. $\mathcal{R}$ defines the type of information on failures provided to the processes. Here we consider failure detectors whose range is the set of process identities, or arrays of natural integers, whose dimension n is the number of processes.

A *failure detector history* is a function $H : \Pi \times \mathbb{N} \to \mathcal{R}$, where $H(p_i, \tau)$ is the value of the failure detector module of process p_i at time τ. This means that each process p_i is provided with a read-only local variable that contains the current value of $H(p_i, \tau)$.

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

Environment It is important to notice that the output of a failure detector does not depend on the computation produced by an algorithm; it depends only on the actual failure pattern, and is a feature of what is called the *environment*. More generally, the notion of an environment captures everything that is not under the control of the algorithm (failures, speed of processes, message transit times, non-determinism, etc.).

Moreover, a given failure detector might associate several histories with each failure pattern. Each history represents a possible sequence of outputs for the same failure pattern; this feature captures the inherent non-determinism of a failure detector.

Remark The failure detector classes presented in this book do not appear in their historical order (the order in which they have been chronologically introduced in research articles). They are introduced according to the order in which this book presents the problems that they allow us to solve.

3.4 URB-broadcast in $CAMP_{n,t}$[- FC] Enriched with a Failure Detector

The previous impossibility result (Theorem 9) states that there is no algorithm implementing the URB-broadcast abstraction in $CAMP_{n,t}$[- FC, $t \geq n/2$]. Whereas if we know in advance that there is a predefined process p_x that never crashes, URB-broadcast can be solved (the other processes can use it as centralized server). Hence the following natural question: 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 first presents the failure detector class, denoted Θ (the weakest failure detector class that answers the previous question), and then an algorithm building URB-broadcast in the system model $CAMP_{n,t}$[- FC, Θ].

3.4.1 Definition of the Failure Detector Class Θ

The failure detector class Θ was introduced by M. Aguilera, S. Toueg, and B. Deianov (1999). A failure detector of this class provides each process p_i with a read-only local variable, a set denoted $trusted_i$. Let $trusted_i^\tau$ denote the value of $trusted_i$ at time τ. 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 formal notation introduced in Section 3.3.2, 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 stating 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 always be 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.

3.4.2 Solving URB-broadcast in $CAMP_{n,t}$[- FC, Θ]

Constructing an URB abstraction in the system model $CAMP_{n,t}$[- FC] enriched with a failure detector of the class Θ is particularly easy. The only modification of the construction described in Fig. 3.1 consists in replacing the urb-delivery predicate (just before line 6), namely, replacing

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

with

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

The accuracy property of Θ guarantees that, when p_i urb-delivers an application message m, at least one non-faulty process has a copy of m. As we have seen in the construction of Fig. 3.1, 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 (let us observe that, if a faulty process could remain forever in $trusted_i$, it could prevent the predicate $trusted_i \subseteq rec\_by_i[m]$) from becoming true).

3.4.3 Building a Failure Detector Θ in $CAMP_{n,t}$[- FC, $t < n/2$]

As urb-broadcast can be implemented in $CAMP_{n,t}$[- FC, $t < n/2$], and in the more general system model $CAMP_{n,t}$[- FC, Θ] (i.e., whatever the value of t), it follows that Θ can be implemented in $CAMP_{n,t}$[- FC, $t < n/2$].

The corresponding construction is described in Fig. 3.2. Each process p_i manages a queue $queue_i$, which initially contains all the processes in any order. Process p_i repeatedly broadcasts the message ALIVE(), and, when it receives a message ALIVE() from p_k, it moves p_k at the head of the queue, and sets $trusted_i$ to the $\lceil \frac{n+1}{2} \rceil$ processes at the head of the queue.

initialization: $trusted_i \leftarrow$ any set of $\lceil \frac{n+1}{2} \rceil$ processes.

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

when ALIVE () **is received from** p_k **do**
(1) suppress p_k from $queue_i$; add p_k at the head of $queue_i$;
(2) $trusted_i \leftarrow$ the $\lceil \frac{n+1}{2} \rceil$ processes at the head of $queue_i$.

Figure 3.2: Building Θ in $CAMP_{n,t}$[- FC, $t < n/2$] (code for p_i)

Theorem 10. *The algorithm described in Fig. 3.2 implements a failure detector Θ in the system model $CAMP_{n,t}$[- FC, $t < n/2$].*

Proof The accuracy property follows from the fact that $trusted_i$ always contains a majority of processes, and, as $t < n/2$, there is always a correct process in the first $\lceil \frac{n+1}{2} \rceil$ processes at the head of any queue $queue_i$.

The liveness property follows from the following observation. After some time the faulty processes no longer send messages ALIVE(), while, as the channels are fair, each correct process receives an infinite number of messages from each correct process. It follows that, after some finite time, each correct process repeatedly appears at the head of any queue, and faulty processes are shifted to the

end of the queue. As there is a majority of correct processes, there is a finite time after which the first $\lceil \frac{n+1}{2} \rceil$ processes at the head of the queue $queue_i$ of any correct process p_i are correct processes. $\square_{Theorem\ 10}$

3.4.4 The Fundamental Added Value Supplied by a Failure Detector

When considering a failure detector, here Θ, the fundamental added value with respect to the assumption $t < n/2$ lies in the fact that a failure detector allows us to *know* which is the weakest information on failures the processes have to be provided with for a problem to be solved. The condition $t < n/2$ is a model assumption, it is not the weakest information on failures that allows the construction of URB-broadcast in an asynchronous system whose communication channels are fair. Even when $t \geq n/2$, the "oracle" Θ allows URB-broadcast to be built.

3.5 Quiescent Uniform Reliable Broadcast

After introducing the quiescence property, this section introduces three failure detector classes that can be used to obtain quiescent URB-broadcast algorithms. The first one is the class of *perfect* failure detectors (denoted P), the second one the class of *eventually perfect* failure detectors (denoted $\diamond P$), and the third one 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. Finally, the class HB allows quiescent uniform reliable broadcast to be implemented. The failure detectors of this class have unbounded outputs, but can be implemented in pure asynchronous systems (their implementations are not quiescent).

3.5.1 The Quiescence Property

Prevent an infinity of protocol messages In the previous URB-broadcast constructions, a correct process is required to send protocol messages forever. This is highly undesirable. The use of acknowledgment messages can easily solve this problem 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 3.2.1), this classic "re-transmission + 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 is known as *quiescence* problem, and solving it requires appropriate failure detectors.

Quiescence property: definition 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 processes 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 (the algorithm that implements it). Hence, among all the constructions that correctly implement a communication abstraction, some are quiescent while others are not.

3.5.2 Quiescent URB-broadcast 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, introduced by T. Chandra and S. Toueg (1996), provides each process p_i with a local variable $suspected_i$, which is a set 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 to have crashed.

More formally (as defined in Section 3.3.2), 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. Observe that $Correct(F)$ 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 Chap. 18).

P with respect to Θ A failure detector of the class Θ can easily be built in $CAMP_{n,t}[P]$ (system model $CAMP_{n,t}[\emptyset]$ enriched with a perfect failure detector P). This can be done by defining $trusted_i$ as being always equal to the current value of $\{1, \ldots, n\} \setminus suspected_i$.

Whereas a failure detector of the class P cannot be built in $CAMP_{n,t}[\Theta]$, from which it follows that P is a failure detector class strictly stronger than Θ. This means that $CAMP_{n,t}[\Theta, P]$ is not computationally stronger than $CAMP_{n,t}[P]$. Nevertheless, even if Θ can be built in $CAMP_{n,t}[P]$ we still use the model notation $CAMP_{n,t}[\Theta, P]$ which provides us with Θ for free. This favors an incremental design (on top of the algorithm described in Fig. 3.1), whose modularity (separation of concerns) facilitates the understanding and the proof.

A quiescent URB construction in $CAMP_{n,t}[\Theta, P]$ In this model, each process p_i has read-only access to both the failure detector-provided local variables: $trusted_i$ and $suspected_i$.

- As we have already seen, Θ is used to ensure the second part of the termination property, namely, if a process urb-delivers an application 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.

```
operation URB_broadcast (m) is send MSG (m) to p_i.

when MSG (m) is received from p_k do
(1)  if (first reception of m)
(2)      then allocate rec_by_i[m]; rec_by_i[m] ← {i, k};
(3)           activate task Diffuse_i(m)
(4)      else rec_by_i[m] ← rec_by_i[m] ∪ {k}
(5)  end if;
(6)  send ACK (m) to p_k.

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

when (trusted_i ⊆ rec_by_i[m]) ∧ (p_i has not yet urb-delivered m) do
(8)  URB_deliver (m).

task Diffuse_i(m) is
(9)  repeat
(10)   for each j ∈ {1, ..., n} \ rec_by_i[m] do
(11)     if (j ∉ suspected_i) then send MSG (m) to p_j end if
(12)   end for
(13) until (rec_by_i[m] ∪ suspected_i) = {1, ..., n} end repeat.
```

Figure 3.3: Quiescent uniform reliable broadcast in $CAMP_{n,t}[\text{- FC}, \Theta, P]$ (code for p_i)

The quiescent URB construction for $CAMP_{n,t}[\Theta, P]$ is described in Fig. 3.3. It is the same as the one described in Fig. 3.1 (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 to its sender an acknowledgment message denoted ACK (m) (line 6). 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 $rec\_by_i[m]$ (line 7). (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 forever sending protocol messages to a crashed process p_j, the task $Diffuse_i(m)$ is appropriately modified. A process p_i repeatedly sends the protocol message MSG (m) to a process p_j only if $j \notin (rec\_by_i[m] \cup suspected_i)$ (lines 10-11). Due to the completeness property of the failure detector class P, p_j will eventually appear in $suspected_i$ if it crashes. Moreover, due to the strong accuracy property of the failure detector class P, 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 shown in Fig. 3.4 for the system model $CAMP_{n,t}[\text{- FC}, \Theta, HB]$. It is consequently left to the reader.

Terminating construction Let us observe that the construction in Fig. 3.3 is not only quiescent but also *terminating*. Termination is a stronger property than quiescence.

More precisely, for each application message m, the task $Diffuse_i(m)$ not only stops sending messages, but eventually terminates. This means that there is a finite time after which the predicate $(rec\_by_i[m] \cup suspected_i) = \{1, \ldots, n\}$, which controls the exit of the repeat loop, becomes satisfied. When this occurs, the task $Diffuse_i(m)$ no longer has 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 $rec\_by_i[m] \cup suspected_i = \{1, \ldots, n\}$ eventually becomes true, and (2) when the

set $suspected_i$ becomes true it 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 Like the class P, the class of eventually perfect failure detectors, denoted $\diamond P$, was introduced by T. Chandra and S. Toueg (1996). It provides each process p_i with a set $suspected_i$ that satisfies the following property: the sets $suspected_i$ can arbitrarily output 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 no correct process is suspected. Hence, the set $suspected_i$ of a 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 (see Exercise 3 in Section 3.8).

$\diamond P$-based quiescent (but not terminating) URB A quiescent URB construction that works in the model $CAMP_{n,t}[\text{- FC},\ \Theta, \diamond P]$ is obtained by replacing the predicate that controls the termination of the task $Diffuse_i(m)$ (line 13 in Fig. 3.3), 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 permanently guarantees that all the processes it contains have crashed. As previously mentioned, during a 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 $CAMP_{n,t}[\text{- FC},\ \Theta, \diamond P]$ is quiescent but not necessarily terminating (according to the failure pattern, it is possible that the termination predicate $rec\_by_i[m] = \{1, \ldots, n\}$ is never satisfied).

3.5.3 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 where the channels are unreliable but fair. Unfortunately, it is impossible to implement a failure detector of the class $\diamond P$ in $CAMP_{n,t}[\emptyset]$ and consequently it is also impossible in $CAMP_{n,t}[\text{- FC}]$ (such an implementation would need additional synchrony assumptions).

How can uniformity and quiescence be obtained These properties can be obtained in $CAMP_{n,t}[\emptyset]$ as soon as this system is enriched with:

1. Uniformity. This part of the termination property states that if a message is urb-delivered by a (correct or faulty) process, it will be urb-delivered by any correct process. This can be obtained thanks to assumption $t < n/2$ or a failure detector of the class Θ.
2. Quiescence. This property can be obtained by the use of a failure detector of the class denoted HB (defined below), which has a simple implementation with unbounded outputs.

The class HB of heartbeat failure detectors This class of failure detectors was introduced by M. Aguilera, W. Chen, and S. Toueg (1999). Formally, a failure detector of the class HB provides each process with a read-only array $HB_i[1..n]$ (heartbeat), whose entries contain natural integers, defined by the following two properties (where $HB_i^\tau[j]$ is the value of $HB_i[j]$ at time τ):

- Completeness. $\forall\, i \in Correct(F), \forall l\ j \in Faulty(F)$: $\exists K$: $\forall\, \tau \in \mathbb{N}$: $HB_i^\tau[j] < K$.
- Liveness.
  1. $\forall\, i, j \in \Pi$: $\forall\, \tau \in \mathbb{N}$: $HB_i^\tau[j] \leq HB_i^{\tau+1}[j]$, and
  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. Unlike 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 the 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 the system $CAMP_{n,t}[\text{- FC}]$. 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 correct processes to sends 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 $CAMP_{n,t}[\text{- FC},\ \Theta, HB]$. In fact, a failure detector of a class such as P, $\diamond P$, or Θ provides a system with additional computational power. Whereas a failure detector of a class HB constitutes an abstraction that "hides" implementation details (all of the non-quiescent part is pieced together in a separate module, namely, the heartbeat failure detector).

A remark on oracles The notion of an *oracle* was first introduced as a language whose words could 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 produced by the algorithm. They depend on the environment. According to the previous terminology, we can say that classes such as P, $\diamond P$, or Θ, are classes of "guessing" failure detectors, while HB is a class of "hiding" failure detectors.

3.5.4 Quiescent URB-broadcast in $CAMP_{n,t}[\text{- FC}, \Theta, HB]$

A URB Construction in $CAMP_{n,t}[\text{- FC}, \Theta, HB]$ A quiescent algorithm implementing the URB-broadcast communication abstraction in $CAMP_{n,t}[\text{- FC}, \Theta, HB]$ is described in Fig. 3.4. Designed by M. Aguilera, W. Chen and S. Toueg (2000), it is similar to the one for $CAMP_{n,t}[\text{- FC}, \Theta, P]$ described in Fig. 3.3. It differs in the addition of two local variables per application message ($prev\_hb_i[m]$ and $cur\_hb_i[m]$ which contain previous and current heartbeat arrays, line 2), and in the task $Diffuse_i(m)$. 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 the application message m), and $HB_i[j]$ has increased since the last test (from p_i's point of view, p_j is alive, predicate of line 14). The local variables $prev\_hb_i[m][j]$ and $cur\_hb_i[m][j]$ are used to keep the two last values read from $HB_i[j]$.

```
operation URB_broadcast (m) is send MSG (m) to p_i.

when MSG (m) is received from p_k do
(1)  if (first reception of m)
(2)     then allocate rec_by_i[m], prev_hb_i[m], cur_hb_i[m];
(3)          rec_by_i[m] ← {i, k};
(4)          activate task Diffuse(m)
(5)     else rec_by_i[m] ← rec_by_i[m] ∪ {k}
(6)  end if;
(7)  send ACK (m) to p_k.

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

when (trusted_i ⊆ rec_by_i[m]) ∧ (p_i has not yet urb-delivered m) do
(9)  URB_deliver (m).

task Diffuse_i(m) is
(10) prev_hb_i[m] ← [−1, ..., −1];
(11) repeat
(12)   cur_hb_i[m] ← HB_i;
(13)   for each j ∈ {1, ..., n} \ rec_by_i[m] do
(14)    if (prev_hb_i[m][j] < cur_hb_i[m][j]) then send MSG (m) to p_j end if
(15)   end for;
(16)   prev_hb_i[m] ← cur_hb_i[m]
(17) until rec_by_i[m] = {1, ..., n} end repeat.
```

Figure 3.4: Quiescent uniform reliable broadcast in $CAMP_{n,t}[\text{- FC}, \Theta, HB]$ (code for p_i)

Theorem 11. *The algorithm described in Fig. 3.4 is a quiescent construction of the* URB-broadcast *communication abstraction in* $CAMP_{n,t}[\text{- FC}, \Theta, HB]$.

Proof The proof of the URB-validity property (no creation of application messages) and the 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 p_i activates $Diffuse_i(m)$, all the non-faulty processes p_j activate $Diffuse_j(m)$.
Proof of claim C1. Let us consider a non-faulty process p_i that activates $Diffuse_i(m)$. 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 received MSG (m). The first

time it received this protocol message, it activated $Diffuse_j(m)$, which proves the claim for this case.

- 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 it 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 was received, p_j activated the task $Diffuse(m)_j$, which concludes the proof of claim C1.

Claim C2. If all the non-faulty processes activate $Diffuse(m)$, they all eventually execute the operation URB_deliver (m).
Proof of claim C2. Let p_i and p_j be any pair of non-faulty processes. As p_i executes $Diffuse_i(m)$ 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.

Moreover, 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 after which the predicate $(trusted_i \subseteq rec\_by_i[m])$ becomes and remains true forever, and consequently p_i eventually urb-delivers m. End of the proof of claim C2.

Proof of the termination property. Let us first show that, if a non-faulty process p_i invokes the operation URB_broadcast (m), all the non-faulty processes urb-deliver the application message 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 $Diffuse_i(m)$. It follows from claim C1 that every non-faulty process p_j activates $Diffuse_j(m)$. We conclude then from claim C2 that each correct process urb-delivers 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, $trusted_i$ always contains 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 received such a message, p_j activated $Diffuse_j(m)$. It then follows from claim C1 that every non-faulty p_x process activates $Diffuse_x(m)$, and from claim C2 that all the non-faulty processes urb-deliver m.

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

Let us first observe that (a) the reception of a protocol message ACK () never entails the sending of protocol messages, and (b) a protocol message ACK (m) is only sent when a protocol message MSG (m) is received. So, the proof amounts to showing 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 messages MSG (m) sent by each non-faulty process p_i is finite. Such messages are sent only inside the task $Diffuse_i(m)$. Let p_j be a process to which the non-faulty process p_i sends MSG (m). If there is a time after 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 false forever. There are two cases:

- Case p_j is faulty. In this 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 after which the predicate

($prev\_hb_i[m][j] < cur\_hb_i[m][j]$) remains false forever. When this occurs, p_i stops sending MSG (m) to p_j, which proves the case.

- Case p_j is non-faulty. We show a contradiction. In this case, the predicate $prev\_hb_i[m][j] > cur\_hb_i[m][j]$ is true infinitely often. It follows that p_i sends MSG (m) to p_j infinitely often. 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.

$\square_{Theorem\ 11}$

Quiescence vs termination Unlike the quiescent URB construction for $CAMP_{n,t}[\text{- FC}, \Theta, P]$ (described in Fig. 3.3), but similar to the quiescent construction for $CAMP_{n,t}[\text{- FC}, \Theta, \diamond P]$, the construction described in Fig. 3.4 for $CAMP_{n,t}[\text{- FC}, \Theta, HB]$ is not terminating. It is easy to see that it is possible that the task $Diffuse_i(m)$ of a process p_i never terminates. In fact, while quiescence concerns only the activity of the underlying network (due to message transfers), termination is a more general property that concerns 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 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 (because 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.

3.6 Summary

This chapter addressed uniform reliable broadcast in the context of asynchronous systems where processes may crash, and communication channels are unreliable but fair, which intuitively means that, if a process repeatedly re-transmits the same message, the channel cannot lose all of the copies due to these re-transmissions.

It has been shown that, in the presence of asynchrony and fair channels, URB-broadcast can be implemented only if a majority of processes do not crash. This assumption has been captured at a more abstract level, namely with the concept of a failure detector. The chapter also introduced the notion of a quiescent implementation, where "quiescent" means that, at the implementation level, an application message cannot give rise to an infinite number of protocol messages. It has been shown that URB-broadcast quiescent algorithms require appropriate failure detectors.

3.7 Bibliographic Notes

- The concept of a failure detector was introduced by T. Chandra and S. Toueg in [102] where they 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 [121, 211].
- The oracle notion in sequential computing is presented in numerous textbooks. Among other books, the reader can consult [182, 222].

- The weakest failure detector class Θ that allows the construction of the URB-broadcast abstraction despite asynchrony, any number of process crashes, and fair channels, was proposed by M.K. Aguilera, S. Toueg, and B. Deianov [22].
- The notion of quiescent communication and the heartbeat failure detector class were introduced by M.K. Aguilera, W. Chen and S. Toueg in [10, 12]. These notions were investigated in [11] in the context of partitionable networks.

  The very weak communication model and the corresponding quiescent URB-broadcast construction presented in Exercise 4 (Section 3.8) was introduced in [12].
- 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 N. Lynch's book [271] presents an in-depth study of the power and limits of unreliable channels.
- The effects of fair lossy channels on problems in general, and in asynchronous systems that are not enriched with failure detectors, is addressed in [54].
- 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 [154]. 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 [412].

  However, 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 [5].

3.8 Exercises and Problems

1. Considering the algorithm in Fig. 3.1, let us replace line 8

   for each $j \in \{1, \ldots, n\}$ **do** send MSG (m) to p_j **end for**,

   with

   for each $j \in \{1, \ldots, n\} \setminus rec\_by_i[m]$ **do** send MSG (m) to p_j **end for**.

   Show that this modification can prevent a correct process p_i, which issues URB_broadcast (m), from urb-delivering the message m.
2. Show that no failure detector of the class P can be built in $CAMP_{n,t}[\Theta]$.
3. Show that failure detector classes $\Diamond P$ and Θ cannot be compared (hint: a set $trusted_i$ is never required to contain the identity of all correct processes).
4. A more difficult problem.
   The processes are asynchronous and may crash (as before). On the network side 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 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 Fig. 3.5. 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).

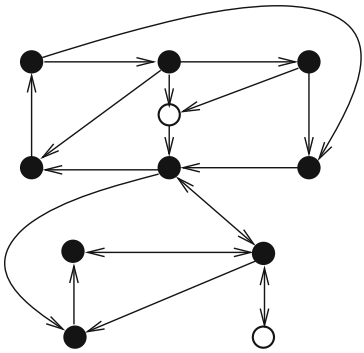

Figure 3.5: An example of a network with fair paths

Notion of fair path In order to be able to construct a communication abstraction that, in any run, allows any pair of non-faulty processes to communicate, basic assumptions on the connectivity of the non-faulty processes are required. These assumptions are based on the notion of a *fair path*. Hence, given an execution, it is assumed that every directed pair of non-faulty processes is connected by a directed path made up of non-faulty processes and fair channels, which is known as a *fair path*.

When considering Fig. 3.5, 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.

What has to be done Considering the previous system mode with very weak connectivity, design:

- an algorithm implementing a Heartbeat failure detector, and
- an algorithm building URB-broadcast with the help of a Heartbeat failure detector, and a failure detector of the class Θ.

Solution in [12] (original paper) and in Chapter 4 of the monograph [366].