# Shared Memory vs Message Passing*

Carole Delporte-Gallet Hugues Fauconnier
Rachid Guerraoui

Revised: 15 February 2004

**Abstract**

This paper determines the computational strength of the shared memory abstraction (a register) emulated over a message passing system, and compares it with fundamental message passing abstractions like consensus and various forms of reliable broadcast.

We introduce the notion of *Quorum* failure detectors and show that this notion captures the exact amount of information about failures needed to emulate a shared memory in a distributed message passing system where processes can fail by crashing. As a corollary of our result, we determine the weakest failure detector to implement consensus in all environments, including those where half of the processes can crash.

We also use our result to show that, in the environment where up to $n-1$ processes can crash (out of the total number of processes $n$), and assuming that failures cannot be predicted, register, consensus, reliable broadcast, as well as terminating reliable broadcast, are all, in a precise sense, equivalent.

# 1 Introduction

## 1.1 Emulating shared memory with message passing

Two communication models have mainly been considered in distributed computing: (1) the *message passing* model and (2) the *shared memory* model. In the first model, we typically assume that the processes are connected through *reliable communication channels*, which do not lose, create or alter messages. Processes communicate using *send* and *receive* primitives, which encapsulate TCP-like communication protocols provided in modern networks. The second model abstracts a hardware shared memory made of *registers*. The processes exchange information using *read* and *write* operations exported by the registers.

*Elements of this paper appeared in preliminary forms in two papers by the same authors: (1) *Tights Bounds on Failure Detection for Register and Consensus*, in the proceedings of the International Symposium on Distributed Computing (DISC'02), Springer Verlag (LNCS 2508); and (2) *Realistic Failure Detectors*, in the proceedings of the IEEE International Symposium on Dependable Systems and Networks (DSN'01).

A *write* returns a simple indication (*ok*) that the operation has been performed, whereas a *read* returns a value previously written in the register. Each operation appears to be executed at some individual instant between the time of its invocation and the time of its reply[1] and, unless it crashes, every process that invokes an operation eventually gets a reply.

It is trivial to build the abstraction of reliable communication channels out of a shared memory, i.e., out of registers. The converse consists in implementing a register abstraction (i.e., its *read* and *write* operations) in a message passing model, also called an *emulation of a distributed shared memory* [5], and is less trivial. Such emulation is very appealing because it is usually considered more convenient to write distributed programs using a shared memory model than using message passing, and many algorithms have been devised assuming a hardware shared memory [19].

Roughly speaking, the motivation of this work is to study the exact conditions under which the emulation is possible[2], and compare these conditions with those needed to build various message passing abstractions. We perform our study assuming a finite set of $n$ processes. Each process executes sequentially the deterministic algorithm assigned to them unless they crash, in which case they stop any computation. A *failure pattern* captures a crash scenario and is depicted by a function that associates, to each time $\tau$, the set of processes that have crashed by time $\tau$. A process that does not crash in a failure pattern is said to be *correct* in that failure pattern; otherwise the process is said to be *faulty.* If a correct process sends a message to a correct process, the message is eventually received.[3]

As in [6], we assume that at least one process is correct in every failure pattern. A set of failure patterns is called an *environment.* Considering for instance the environment where $t$ processes can crash, comes down to assuming the set of all failure patterns $F$ where strictly more than $n - t$ processes are correct in $F$.

## 1.2 On the weakest failure detector to emulate a shared memory

It is common knowledge that a necessary and sufficient condition for implementing a register in an *asynchronous* message passing system, with no bound on communication delays and process relative speeds, is the assumption of an environment with a majority of correct processes [5]. In particular, in such an asynchronous system, it is impossible to devise an algorithm that implements a register in any environment where half of the processes can crash. The impossibility stems from the absence of any synchrony assumption, and this absence

[1] We also talk about a *linearizable* register in the sense of [20] and an *atomic* register in the sense of [28].

[2] In some sense, we seek to precisely determine when results obtained in a shared memory model can be ported onto a message passing model.

[3] We will also consider the impact on our study of a stronger communication model where any message sent by a process to a correct process is eventually received.

makes any information about process failures possibly inaccurate, e.g., any accurate use of *timeouts* in an asynchronous system is simply impossible.

An elegant formalism to express information about failures (and hence encapsulate synchrony assumptions), and indirectly encapsulate synchrony assumptions, was introduced in [6] through the abstract concept of *failure detector*. This concept represents a distributed oracle that gives hints about the failures of processes and can be characterized through axiomatic properties describing the extent to which these hints reflect the actual failure pattern (i.e., actual crashes). A failure detector class gathers a set of failure detectors that ensures the same properties, and a class $A$ is said to implement some distributed abstraction $P$ (say a register) if there is an algorithm that implements $P$ with any failure detector of $A$. Failure detector classes can be classified within a hierarchy. Intuitively, a class that is at a high level of the hierarchy gathers failure detectors that provide more accurate information about crashes (and indirectly encapsulate stronger synchrony assumptions) than classes at a lower level of the hierarchy. More precisely, a failure detector class $A$ is said to be *stronger* (at a higher level of the hierarchy) than a failure detector class $B$, if there is an algorithm that, given any failure detector of class $A$, implements some failure detector of class $B$. We also say that $A$ *implements* $B$ and $B$ is *weaker* than $A$. If $A$ is stronger than $B$ and the converse is not true, we say that $A$ is *strictly stronger* than $B$ (and $B$ is *strictly weaker* than $A$).

In particular, three failure detector classes were identified in [6]: $\mathcal{P}$ (*Perfect*), $\mathcal{S}$ (*Strong*), and $\Diamond\mathcal{S}$ (*Eventually Strong*). All failure detectors of these classes output, at any time $\tau$ and any process $p_i$, a set of processes that are said to be *suspected* by $p_i$ at time $\tau$. Those of $\mathcal{P}$ ensure a *strong completeness* property, which states that eventually all crashed processes will be permanently suspected, and *strong accuracy*, which states that no process is falsely suspected; failure detectors of $\mathcal{S}$ ensure *strong completeness* and *weak accuracy*, which states that some correct process is never suspected; those of $\Diamond\mathcal{S}$ ensure, besides *strong completeness*, *eventual weak accuracy*, which states that, eventually, some correct process is never suspected. A fourth interesting class, denoted $\Omega$, was introduced in [7]. Failure detectors of this class output, at any time $\tau$ and any process $p_i$, a single process that is said to be *trusted* by $p_i$ at time $\tau$. The property ensured here is the *leader* property which guarantees that eventually, one correct process is permanently trusted by all correct processes.

Among these classes, $\mathcal{P}$ is the strongest whereas $\Diamond\mathcal{S}$ and $\Omega$ are equivalent and are the weakest. It is easy to see that failure detectors of class $\mathcal{P}$ can be implemented in a *synchronous* system with a bound on communication delays and process relative speeds, and failure detectors of classes $\Diamond\mathcal{S}$ and $\Omega$ can be implemented in an *eventually synchronous* system where these bounds are guaranteed to hold only eventually [6].

Not surprisingly, one can devise an algorithm that implements a register with failure detector class $\mathcal{P}$, in any environment. But can we do with a *strictly weaker* class? We address in this paper the question of the *weakest* failure detector class to implement a register, in any environment, including those where half of the processes may crash. More precisely, we seek to identify a failure detector class $\mathcal{X}$ such that: (1) $\mathcal{X}$ implements a register in every environment, and (2) every failure detector class $\mathcal{D}$ that implements a register is stronger than $\mathcal{X}$, i.e., $\mathcal{D}$ provides at least as much information about failures as $\mathcal{X}$.

Determining $\mathcal{X}$ means capturing the exact amount of information about failures and, in some sense, the minimal synchrony assumptions, needed to emulate shared memory in a message passing system. As we will discuss later, determining $\mathcal{X}$ will allow us to compare, in a precise sense, the strength of the shared memory abstraction (emulated in a message passing system) with other fundamental message passing abstractions. We will say that an abstraction $U$ is (strictly) *stronger* than an abstraction $V$ if the weakest failure detector class to implement $U$ is (strictly) stronger than the weakest failure detector class to implement $V$.

## 1.3 On the weakest failure detector to implement consensus

Interestingly, and as we discuss below, determining the weakest failure detector $\mathcal{X}$ to implement a register, in any environment, would make it trivial to determine the weakest failure detector class to implement the seminal *consensus* abstraction, in any environment [14].

Consensus is a distributed abstraction through which processes each propose an initial value and have to agree on a final decision value among one of these proposed values.[4] Implementing consensus is fundamental in distributed computing as it makes it possible to implement highly-available objects of any type [19].

The celebrated FLP impossibility result [14] states that, in an asynchronous distributed system, consensus cannot be implemented in any environment where at least one process can crash. This impossibility stems from the absence of any timing assumption on communication delays and process relative speeds, which might enable an algorithm to infer information about crashed processes. In [6], two consensus algorithms using failure detectors were presented. The first algorithm implements consensus with failure detector class $\mathcal{S}$ in any environment. The second algorithm implements consensus with failure detector class $\Omega$.[5] but assumes environments with a majority of correct processes. The very existence of this algorithm, together with the result of [7], implies that $\Omega$ is the weakest failure detector class to implement consensus in environments with a majority of correct processes.

What about more general environments? If more than a minority of the processes can crash, $\Omega$ cannot implement consensus [6]. What is the weakest failure detector class for consensus for this case? This question remained open for a decade now [7].

[4] We consider here the uniform variant of consensus where no disagreement is possible even with processes that decided and crashed [21]. The non-uniform variant will be discussed in the last section of the paper.

[5] The algorithm actually uses $\Diamond\mathcal{S}$ but can easily be modified to work with $\Omega$, as $\Diamond\mathcal{S}$ and $\Omega$ are equivalent

We explain in the following that determining the weakest failure detector class $\mathcal{X}$ to implement a register, trivially answers that question. In fact, we claim that failure detector class $\mathcal{X} \times \Omega$, which would provide both the information of $\mathcal{X}$ and the information of $\Omega$, is the weakest for consensus, in any environment. Our claim follows from the two observations below.

- In [30], it was shown that consensus can be implemented using registers and $\Omega$, in every environment. Hence, if $\mathcal{X}$ implements a register in any environment, then $\mathcal{X} \times \Omega$ implements consensus, in every environment.

- Let $\mathcal{D}$ be any failure detector class that implements consensus. We know from [7] that we can use $\mathcal{D}$ to implement $\Omega$ (i.e., $\mathcal{D}$ is stronger than $\Omega$). Given that we can trivially use consensus as a building block to implement a register in a message passing system, then $\mathcal{D}$ also implements a register. The very fact that $\mathcal{X}$ is the weakest to implement a register also means here that $\mathcal{D}$ is stronger than $\mathcal{X}$. Hence, $\mathcal{D}$ is stronger than $\mathcal{X} \times \Omega$.

## 1.4 Contributions

This paper introduces the class of *Quorum* failure detectors, which we denote by $\Sigma$, and which we show is the weakest to implement a register, in any environment, i.e., $\mathcal{X} = \Sigma$. Failure detectors of class $\Sigma$ output, for any failure pattern, any time $\tau$, and any process $p_i$, a set of processes that are said to be *trusted* by $p_i$ at time $\tau$, such that the two following properties are satisfied:

- Every two sets of trusted processes intersect;
- Eventually every trusted process is correct.

It is important to notice that, with a *Quorum* failure detector, the processes might keep on permanently changing their mind about which processes they trust. Furthermore, there is no obligation that the processes eventually agree on the same set of trusted processes.

We first show that any failure detector class $\mathcal{D}$ that implements a register in some environment, implements a failure detector of class $\Sigma$ in that environment (necessary condition), and then we give an algorithm that uses $\Sigma$ to implement a register in every environment (sufficient condition). We prove the first step of our result (i.e., necessary condition) in a strong message passing system model where any message sent to a correct process is eventually received, i.e., even if the sender crashes right after sending its message; our proof then directly applies to a weaker model where any message sent by a correct process to a correct process is eventually received. We show the sufficient condition in the stronger model; our proof then directly applies to a model where any message sent to a correct process is eventually received.

We use our result to show that $\Sigma \times \Omega$ is the weakest failure detector class for consensus in any environment, and consequently revisit the relationship between consensus and register abstractions in a message passing model.

- By observing that $\Sigma$ and $\Omega$ are actually equivalent in a system of 2 processes, we point out the interesting fact that, in a message passing system of 2 processes with a failure detector, trying to implement a register among two processes is as hard as trying to solve consensus. This is interesting to contrast with a shared memory model where a register cannot be used to solve consensus among two processes (one of which can crash) [14, 32]. In a sense, we show that a register given in hardware is significantly weaker than a register implemented in software (among processes that communicate by message passing).

- In the general case ($n \geq 3$), the classes $\Sigma$ and $\Omega$ are not equivalent and their complementarity helps better understand the information structure of consensus algorithms [8, 6, 29, 17]: $\Sigma$ encapsulates the information about failures needed to ensure the safety part of consensus (i.e., the quorum, as pointed in [33], that will lock the consensus value and prevent disagreement), whereas $\Omega$ encapsulates the information about failures needed to ensure the liveness part of consensus (i.e., to ensure that some correct process will eventually succeed in locking a decision value within a quorum).

We also use our result to compare the register abstraction with another fundamental abstraction: *(uniform) reliable broadcast abstraction* [22]. It is a known fact that both abstractions need a majority of correct processes to be implemented in an asynchronous message passing model. We show that in any environment where half of the processes can crash (the correct majority assumption does not hold), register is strictly stronger than reliable broadcast. We show furthermore that by restricting the universe of failure detectors to those that cannot provide information about future failures, and which we call *realistic* failure detectors, and we consider the *wait-free* environment [19] where $n - 1$ processes can crash, the weakest failure detector classes to implement a register, consensus, as well as reliable broadcast and other abstractions like *terminating reliable broadcast* [22] (a variant of Byzantine Agreement in a model with crash failures) are all the very same one. All these problems are in this case, and in a precise sense, equivalent.

To summarize, the contributions of this paper are the following:

- We determine the weakest failure detector to implement a register, in any environment.

- We determine the weakest failure detector to implement consensus, in any environment, and we observe that in a system of 2 processes, consensus and register are equivalent.

- We show that a register is strictly stronger than a reliable broadcast, in any environment where half of the processes can crash (otherwise they are equivalent).

- We show that, with respect to realistic failure detectors, register, consensus and terminating reliable broadcast are equivalent in the wait-free environment where $n-1$ processes can crash.

## 1.5 Related work

Several authors considered augmenting an asynchronous shared memory model with failure detectors (e.g., [30, 35]), i.e., augmenting register abstractions with failure detectors. To our knowledge however, the question of the weakest failure detector class to implement the register itself in a message passing system was never addressed. As we pointed out, in an asynchronous message passing system, a register can be implemented in any environment with a majority of correct processes, i.e., without any external failure detector [5]. To implement a register when more than half of the processes can crash, we need some synchrony assumptions to infer information about process crashes: this is precisely what $\Sigma$ encapsulates. Not surprisingly, it is easy to implement a failure detector of class $\Sigma$ in a message passing system if a majority of the processes is correct. Our result is in this sense a generalization of [5]. With a correct majority, our weakest failure detector result for consensus simply boils down to the result of [7]: $\Sigma \times \Omega$ is $\Omega$, which was indeed shown to be the weakest for consensus with a majority of correct processes. So far, the "weakest" failure detector class that was known to implement consensus in any environment was $\mathcal{S}$ [6]. We show in this paper that failure detector class $\Sigma \times \Omega$ is strictly weaker than failure detector class $\mathcal{S}$. In other words, a first glance intuition that $\mathcal{S}$ could be the weakest for consensus would have been wrong.

From a more general perspective, our approach shares some similarities with the approach, initiated in [19], and which aims at comparing the strength of shared memory abstractions using the very notion of *consensus number*. This notion measures the maximum number of processes that can solve consensus using a given abstraction. We also seek here to compare the strength of distributed programming abstractions, but we do so considering a message passing model and using, as a comparison metric, the notion of failure detector. This leads to some fundamental, and sometimes surprising differences.

- Our approach is more general because we do not restrict our space of abstractions to shared objects with a sequential specification [19], e.g, there is no to describe a reliable broadcast abstraction as a shared memory object and study its strength following the the approach of [19].

- Whereas it is known that a register has consensus number 1 (i.e., cannot be used to solve consensus among 2 processes), we show in this paper that the failure detector that is needed to emulate a register in a message passing system of 2 processes, can be used to solve consensus among 2 processes.

- In the general case ($n > 2$), consensus is the strongest abstraction in the sense of [19]. In our approach, this is only true if we restrict the space of failure detectors to realistic onces and consider environment where $n-1$ processes can crash. In this case, all interesting abstractions we know of (register, reliable broadcast, etc.) have the same strength.

### 1.6 Roadmap

The rest of the paper is organized as follows. Section 2 gives our system model. Section 3 defines failure detector class $\Sigma$. Section 4 recalls the register abstraction. Section 5 shows (necessary condition) that any failure detector class that implements a register implements $\Sigma$. Section 6 shows (sufficient condition) that $\Sigma$ implements a register in every environment. Section 7 derives the weakest failure detector class to implement consensus in any environment. Section 8 uses our result to compare a register with a reliable broadcast abstraction. Section 9 restricts our universe of failure detectors and environments and derives interesting equivalence relations between various distributed computing abstractions. Section 10 discusses the impact of non-uniform specifications.

Many proofs of results stated in the paper are gathered in the appendix. Most of these proofs have to do with failure detector comparisons. We conduct them using simple algorithm reductions. (like in [7] but unlike in [23]). Hence, by determining for instance the weakest failure detector class to implement some abstraction (e.g., a register or consensus), we determine what exact information about failures processes need to know and effectively compute to implement that abstraction.

## 2 System model

Our model of asynchronous computation with failure detection is the FLP model [14] augmented with the failure detector abstraction [6, 7]. A discrete global clock is assumed, and $\Phi$, the range of the clock's ticks, is the set of natural numbers. The global clock is used for presentation simplicity and is not accessible to the processes. We sketch here the basic elements of the model, focusing on those that are needed to prove our results. The reader interested in specific details about the model should consult [7, 9].

### 2.1 Failure patterns and environments

We consider a distributed system composed of a finite set of $n$ processes $\Pi = \{p_1, p_2, \ldots, p_n\}$. The processes are sometimes simply denoted by $1, 2, .., n$ where there is no ambiguity. Unless explicitly stated otherwise, we will assume that $|\Pi| = n \geq 3$. Processes can fail by crashing. A process $p_i$ is said to crash at time $\tau$ if $p_i$ does not perform any action after time $\tau$ (the notion of action is recalled below). Otherwise, the process is said to be alive at time $\tau$. Failures are permanent, i.e., no process recovers after a crash. A correct process is a process that does not crash. A failure pattern is a function $F$ from $\Phi$ to $2^{\Pi}$, where $F(\tau)$ denotes the set of processes that have crashed by time $\tau$. The set of correct processes in a failure pattern $F$ is noted *correct*$(F)$. As in [6], we assume that every failure pattern has at least one correct process. An environment is a set of failure patterns. Environments describe the crashes that can occur in a system. When we talk about the environment where $t$ processes can crash ($t < n$), denoted by $\mathcal{E}_t$, we implicitly assume the set of failure patterns where at most $t$ processes crash (in every failure pattern). Environment $\mathcal{E}_{n-1}$ will be called the wait-free environment.

## 2.2 Failure detectors

Roughly speaking, a failure detector $\mathcal{D}$ is a distributed oracle which gives hints about failure patterns. Each process $p_i$ has a local failure detector module of $\mathcal{D}$, denoted by $\mathcal{D}_i$. Associated with each failure detector $\mathcal{D}$ is a range $R_{\mathcal{D}}$ (when the context is clear we omit the subscript) of values output by the failure detector. A failure detector history $H$ with range $R$ is a function $H$ from $\Pi \times \Phi$ to $R$. For every process $p_i \in \Pi$, for every time $\tau \in \Phi$, $H(i, \tau)$ denotes the value of the failure detector module of process $p_i$ at time $\tau$, i.e., $H(i, \tau)$ denotes the value output by $\mathcal{D}_i$ at time $\tau$. A failure detector $\mathcal{D}$ is defined as a function that maps each failure pattern $F$ to a set of failure detector histories with range $R_{\mathcal{D}}$. $\mathcal{D}(F)$ denotes the set of all possible failure detector histories permitted for the failure pattern $F$, i.e., each history represents a possible behaviour of $\mathcal{D}$ for the failure pattern $F$.

Four classes of failure detectors introduced in [6, 7], and recalled in the introduction of this paper, are of interest in this paper. All have range $R = 2^{\Pi}$. For any failure detector $\mathcal{D}$ in any of the first three classes, any failure pattern $F$, and any history $H$ in $\mathcal{D}(F)$, $H(i, \tau)$ is the set of processes that are said to be *suspected* by process $p_i$ at time $\tau$. (1) The class of *Perfect* failure detectors ($\mathcal{P}$) gathers all those that ensure *strong completeness*, i.e., eventually every process that crashes is permanently suspected by every correct process, and *strong accuracy*, i.e., no process is suspected before it crashes. (2) The class of *Strong* failure detectors ($\mathcal{S}$) gathers those that ensure *strong completeness* and *weak accuracy*, i.e., some correct process is never suspected. (3) The class of *Eventually Strong* failure detectors ($\Diamond\mathcal{S}$) gathers those that ensure *strong completeness* and *eventual weak accuracy*, i.e., eventually, some correct process is never suspected.

The fourth class is the class $\Omega$: for any failure detector $\mathcal{D}$ in this class, any failure pattern $F$, and any history $H$ in $\mathcal{D}(F)$, $H(i, \tau)$ is the process that is said to be *trusted* by process $p_i$ at time $\tau$. Every failure detector of $\Omega$ ensures the following *leader* property: eventually, a single correct process is permanently trusted by all correct processes.

## 2.3 Algorithms

An algorithm using a failure detector $\mathcal{D}$ is a collection $A$ of $n$ deterministic automata $A_i$ (one per process $p_i$). Computation proceeds in steps of the algorithm. In each step of an algorithm $A$, a process $p_i$ atomically performs the following three actions: (1) $p_i$ receives a message from some process $p_j$, or a "null" message $\lambda$; (2) $p_i$ queries and receives a value $d$ from its failure detector module $\mathcal{D}_i$ ($d \in R_{\mathcal{D}}$ is said to be seen by $p_i$); (3) $p_i$ changes its state and sends a message (possibly null) to some process. This third action is performed according to (a) the automaton $A_i$, (b) the state of $p_i$ at the beginning of the step, (c) the message received in action 1, and (d) the value $d$ seen by $p_i$ in action 2. Messages that are not null are uniquely identified and the message received by a process $p_i$ is chosen non-deterministically among the messages in the message buffer destined to $p_i$, and the null message $\lambda$. In a step the process performs one *event*.

A configuration is a pair $(I, M)$ where $I$ is a function mapping each process $p_i$ to its local state, and $M$ is a set of messages currently in the message buffer. A configuration $(I, M)$ is an initial configuration if $M = \emptyset$ (no message is initially in the buffer): in this case, the states to which $I$ maps the processes are called *initial states*. A *step* of an algorithm $A$ is a tuple $e = (i, m, d, A)$, uniquely defined by the algorithm $A$, the identity of the process $p_i$ that takes the step, the message $m$ received by $p_i$, and the failure detector value $d$ seen by $p_i$ during the step. A step $e = (i, m, d, A)$ is applicable to a configuration $(I, M)$ if and only if $m \in M \cup \{\lambda\}$. The unique configuration that results from applying $e$ to configuration $C = (I, M)$ is noted $e(C)$.

## 2.4 Schedules and runs

A schedule of an algorithm $A$ is a (possibly infinite) sequence $S = S[1]; S[2]; \ldots S[k]; \ldots$ of steps of $A$. A schedule $S$ is applicable to a configuration $C$ if (1) $S$ is the empty schedule, or (2) $S[1]$ is applicable to $C$, $S[2]$ is applicable to $S[1](C)$ (the configuration obtained from applying $S[1]$ to $C$), etc.

Let $A$ be any algorithm and $\mathcal{D}$ any failure detector. A run of $A$ using $\mathcal{D}$ is a tuple $R =< F, H, C, S, T >$ where $H$ is a failure detector history and $H \in \mathcal{D}(F)$, $C$ is an initial configuration of $A$, $S$ is an infinite schedule of $A$, $T$ is an infinite sequence of increasing time values, and

- (1) $S$ is applicable to $C$,
- (2) for all $k$ where $S[k] = (i, m, d, A)$, we have $p_i \notin F(T[k])$ and $d = H(i, T[k])$,
- (3) every correct process takes an infinite number of steps, and
- (4. weak channel assumption) every message sent by a correct process to a correct process $p_i$ is eventually received by $p_i$.
- (4'. strong channel assumption) in fact, we will also discuss the impact on our results of an alternative model with a stronger definition of the channels where we require that every message sent to a correct process $p_j$ is eventually received by $p_j$. With this weaker definition, a message sent by a process that crashed after the sending might never be received.

Let $o$ and $o'$ be two events of a run, by definition, we say that $o$ *precedes* $o'$ ($o \prec o'$) if $o$ occurs before $o'$. If neither $o \prec o'$ nor $o' \prec o$, then $o$ and $o'$ are said to be *concurrent*.

### 2.5 Implementability

Every abstraction $U$ (e.g., register, consensus, etc) is associated with exactly one set of runs, which we also denote by $U$, i.e., the runs that obey the specification of $U$. We say that an algorithm $A$ implements an abstraction $U$ using a failure detector $\mathcal{D}$ in some environment if every run of $A$ using $\mathcal{D}$ in that environment is in $U$. We say that $\mathcal{D}$ implements (or solves) $U$ in some environment if there is an algorithm that implements $U$ using $\mathcal{D}$ in that environment. We say that a failure detector class implements (or solves) an abstraction $U$ in some environment if there is an algorithm that implements $U$, in that environment, with every failure detector of that class.

We say that a failure detector $\mathcal{D}1$ is stronger than a failure detector $\mathcal{D}2$ in some environment (written sometimes $\mathcal{D}2 \preceq \mathcal{D}1$) if there is an algorithm (sometimes called a reduction algorithm) that implements $\mathcal{D}2$ using $\mathcal{D}1$ in that environment, i.e., that can emulate the output of $\mathcal{D}2$ using $\mathcal{D}1$ [6]. The algorithm does not need to emulate all histories of $\mathcal{D}2$. It is required however that, for every run $R =< F, H, C, S, T >$, where $H \in \mathcal{D}1(F)$ and $F$ is in the environment, the output of the algorithm with $R$ be a history of $\mathcal{D}2(F)$. We say that $\mathcal{D}1$ is strictly stronger than $\mathcal{D}2$ in some environment ($\mathcal{D}2 \prec \mathcal{D}1$) if $\mathcal{D}2 \preceq \mathcal{D}1$ and $\neg(\mathcal{D}1 \preceq \mathcal{D}2)$ in that environment. We say that $\mathcal{D}1$ is equivalent to $\mathcal{D}2$ in some environment ($\mathcal{D}1 \equiv \mathcal{D}2$), if $\mathcal{D}2 \preceq \mathcal{D}1$ and $\mathcal{D}1 \preceq \mathcal{D}2$ in that environment.

We say that a failure detector class $\mathcal{D}$ is the weakest to implement an abstraction $U$ in some environment if (a. sufficient condition) $\mathcal{D}$ solves $U$ in that environment and (b. necessary condition) any failure detector class that solves $U$ is stronger than $\mathcal{D}$ in that environment. Similarly, an abstraction $U$ is said to be stronger (resp. strictly stronger) than an abstraction $V$ in some environment, if the weakest failure detector for $U$ is stronger (resp. strictly stronger) than the weakest failure detector for $V$.

## 3 Quorum failure detectors

In this section, we define the class of *Quorum* failure detectors, denoted by $\Sigma$, and which we will show in subsequent sections, is the weakest to implement a register, in any environment.

### 3.1 Properties

Quorum failure detectors have range $2^{\Pi}$. That is, they output, at any time $\tau$, and any process $p_i$, a list of processes. We say that these processes are *trusted* by $p_i$ at time $t$. By convention, and to simplify the presentation, we assume that after a process crashes, the output of any quorum failure detector at this process is $\Pi$, i.e., a crashed process trusts all processes. More precisely, if $p_i$ has crashed by time $\tau$, we assume that the output of any quorum failure detector $\mathcal{D}$ is $H_{\mathcal{D}}(i, \tau') = \Pi$ for any $\tau' \geq \tau$.

The two following properties are ensured by any *Quorum* failure detector $\mathcal{D}$:

1. *Intersection.* Given any two lists of trusted processes, possibly at different times and by different processes, at least one process belongs to both lists. More precisely:

   - $\forall F \in \mathcal{E}, \forall p_i, p_j \in \Pi, \forall H_{\mathcal{D}} \in \mathcal{D}(F), \forall \tau, \tau' \in \Phi : H_{\mathcal{D}}(i, \tau) \cap H_{\mathcal{D}}(j, \tau') \neq \emptyset$.

2. *Completeness.* Eventually no faulty process is ever trusted by any correct process. More precisely:

   - $\forall F \in \mathcal{E}, \forall p_i \in \mathit{correct}(F), \forall H_{\mathcal{D}} \in \mathcal{D}(F), \exists \tau \in \Phi, \forall \tau' > \tau \in \Phi : H_{\mathcal{D}}(i, \tau') \subseteq \mathit{correct}(F)$.

The combination of the *intersection* and *completeness* properties implies the following *accuracy* property, ensured by any *Quorum* failure detector $\mathcal{D}$:

- *Accuracy.* Every list of trusted processes contains at least one correct process. More precisely:

  – $\forall F \in \mathcal{E}, \forall p_i \in \Pi, \forall H_{\mathcal{D}} \in \mathcal{D}(F), \forall \tau \in \Phi : H_{\mathcal{D}}(i, \tau) \cap \mathit{correct}(F) \neq \emptyset$.

To see why *intersection* and *completeness* properties imply *accuracy*, remember first that we consider by default environments where at least one process is correct. Indeed, let $p_i$ be any correct process in failure pattern $F$, and consider any failure detector history $H_{\mathcal{D}}$ in $\mathcal{D}(F)$. From *completeness*, there is a time $\tau_0$ such that $H_{\mathcal{D}}(i, \tau_0)$ contains only correct processes. Consider any process $p_j$ and any time $\tau$. By *intersection*, $H_{\mathcal{D}}(j, \tau) \cap H_{\mathcal{D}}(i, \tau_0)$ is not empty and thus $H_{\mathcal{D}}(j, \tau)$ contains at least one correct process. Note that the accuracy property of *Quorum* failure detectors does not mean that any process that belongs to two lists of trusted processes is correct.

## 3.2 Strength

We discuss here the "strength" of $\Sigma$ by comparing it to various failure detectors introduced in the literature [6]. We only give here some results that we believe are useful to understand the rest of the paper. The full picture, together with corresponding proofs, are given in the appendix.

First, it is easy to see how to implement a *Quorum* failure detector in any environment where a majority of the processes is correct. Indeed, consider a model of communication channels according to which any message sent from a correct process to a correct process is eventually received (this is the weakest of the two models of message passing communication we consider in this paper). Initially, every process trusts all processes. Periodically, every process $p_i$ sends a message to all processes and waits for corresponding acknowledgments. When $p_i$ receives acknowledgments from a majority of processes, $p_i$ outputs this majority as its list of trusted processes. The *intersection* property follows from the fact that any two majorities intersect. Consider the time after which all faulty processes have crashed and all their acknowledgments (sent prior to their crashing) have been received or will never be received. Such a time exists by our communications channels assumption. By this very same assumption, after this time, every correct process $p_i$ keeps on receiving acknowledgments only from correct processes, and will only trust correct processes. This ensures the *completeness* property.

It is also easy to see that failure detector class $\mathcal{S}$ (and hence $\mathcal{P}$) is stronger than $\Sigma$ in any environment. Indeed, using any failure detector $\mathcal{D}$ of class $\mathcal{S}$, one can implement a failure detector of class $\Sigma$ by emulating its output through some distributed variable $Trust$ as follows. Every process $p_i$ periodically (a) consults the list of processes that are suspected to have crashed, i.e., the output of the local failure detector module $\mathcal{D}_i$, and (b) outputs in $Trust_i$, the local value of variable $Trust$ at $p_i$, the exact complement of that set, i.e., $p_i$ trusts all processes that are not suspected. The *completeness* property of $Trust$ follows from that of $\mathcal{S}$, whereas the *intersection* property of $Trust$ follows from the *accuracy* property of $\mathcal{S}$: at least one process will never be suspected and this process will belong to all sets of trusted processes. As we will show in Section 7, the converse (emulating $\mathcal{S}$ with $\Sigma$) is generally not possible. More precisely, we show in the appendix that failure detector class $\mathcal{S}$ is strictly stronger than $\Sigma$ in any environment $\mathcal{E}_t$ with $t > 0$. Furthermore, we show that failure detector classes $\Omega$ (and hence $\Diamond\mathcal{S}$) and $\Sigma$ are incomparable in any environment $\mathcal{E}_t$ with $t \geq n/2$. Remember that we consider a system with at least three processes ($n > 2$). We show in the appendix that in a system of 2 processes, $\Sigma$ and $\Omega$ are equivalent. This has some interesting consequences that we discuss later in the paper.

# 4 The atomic register abstraction

Before proving that $\Sigma$ is the weakest failure detector class to implement a register, we first recall in this section the definition of the register abstraction and we recall some related results that are used to prove that $\Sigma$ is the weakest failure detector class to implement a register.

## 4.1 Background

A register is a shared object accessed through two operations: *read* and *write*. The write operation takes as an input parameter a specific value to be stored in the register and returns a simple indication *ok* that the operation has been executed. The read operation is supposed to return a value written in the register; we assume that, initially, a specific write execution has initialized the value of the register. The registers we consider are *atomic* [28] (*linearisable* [19]) and *fault-tolerant*: they ensure that, despite concurrent invocations and possible crashes of the processes, every correct process that invokes an operation eventually gets a reply (a value for the read and an *ok* indication for the write), and any operation appears to be executed instantaneously between its invocation and reply time events. (A precise definition is given in [19, 3].)

So far, we have been implicitly assuming that any process can invoke any operation on a register. Such a register (i.e., that can be read and written by any process) is sometimes called a MWMR (*multi-writer/multi-reader*) register [28]: we simply call it a register here. A register that can be read by all processes and written by exactly one process (called the writer) is called a SWMR (*single-writer/multi-reader*) register [28]; and a register that can be read by exactly one process (called the reader) and written by exactly one process (called the writer) is called a SWSR (*single-writer/single-reader*) register [28]. Interestingly, it was shown that, in any environment, one can implement a (MWMR) register out of SWSR registers [24, 38].

We will exploit this relation between different kinds of registers to simplify our proofs. More precisely, we will show that a (MWMR) register can be implemented with $\Sigma$ in any environment, by simply showing that a SWSR register can be implemented with $\Sigma$ in that environment. Then we will assume that some failure detector class $\mathcal{D}$ implements a SWMR register in some environment, and show that $\mathcal{D}$ implements $\Sigma$ in that environment.

## 4.2 Properties

In the following, we precisely define a SWMR register abstraction through three properties (along the lines of [3]). These properties will be used later in our proofs.

Consider any run of an algorithm implementing a register abstraction. Let $o$ be any read or write operation execution; when there is no ambiguity, we say operation for operation execution. We assume that different write operations store different values in the register; this can simply be achieved by appending to every input value of a write the identity of the writer process together with some local timestamp.

We denote by $o_b$ and $o_e$, respectively, the events corresponding to the invocation of $o$ by some process (beginning of $o$) and the return of $o$'s reply by that process (end of $o$). We say that the operation *$o$ terminates* when its reply event occurs, and we say that a value has been written (resp. read) if the corresponding write (resp. read) was terminated.

We assume that each process (reader or writer) invokes the operations of a register in a sequential manner: the next (read or write) operation begins after the previous one has terminated. Furthermore, we assume that the register initially contains a specific value $\perp$. For uniformity of presentation, we assume that this value was initially written by the writer.

An algorithm is said to implement a SWMR register if it implements its read and write operations in such a way that the following three properties are satisfied.

- *Liveness:* If a correct $p_i$ invokes an operation, the operation eventually terminates.
- *Validity (safety 1):* Every read operation returns either the value written by the last write that precedes it, or a value written concurrently with this read.
- *Ordering (safety 2):* If a read operation $r$ precedes a read operation $r'$ then $r'$ cannot return a value written before the value returned by $r$.

# 5 The necessary condition

We show in this section that any failure detector class that implements a register in a given environment, is stronger than $\Sigma$ in that environment.

## 5.1 Overview

We describe in the following an algorithm, denoted by $R$, that uses any algorithm that implements a register in some environment and with some failure detector $\mathcal{D}$, to emulate the output of a failure detector of class $\Sigma$. The underlying algorithm is supposed to operate in a strong message passing system model where any message sent to a correct process $p$ is eventually received by $p$.

The emulation is achieved within a distributed variable, denoted by $Trust$: the local value of this variable at process $p_i$ is denoted by $Trust_i$. Algorithm $R$ ensures that variable $Trust$ ensures the *completeness* and *intersection* properties of $\Sigma$.

Our algorithm $R$ makes use of the very fact that some algorithm uses $\mathcal{D}$ to implement a register, i.e., to implement its write and read operations. More precisely, we assume here $n$ SWMR registers. Every process $p_i$ is associated with exactly one register $Reg_i$: $p_i$ is the only writer of $Reg_i$ and all processes read in $Reg_i$.

There are three key ideas underlying our algorithm $R$:

1. Every process $p_i$ periodically writes in $Reg_i$ a timestamp $k$, together with a specific value that we will discuss below (in the third item): the timestamp $k$ is incremented for every new write performed by $p_i$. Process $p_i$ determines the processes that *participate* in every *write(k,*)* (i.e., in the exchange of messages underlying *write(k,*)*) and this set is denoted by $P_i(k)$. Roughly speaking, this set is determined by having every process $p_j$ that receives some message $m$ in the context of the $k-$th write from $p_i$, tag every message that causally follows $m$, with $k$, $p_j$, as well as the list of processes from which messages have been received with those tags.

When $p_i$ terminates *write(k,*)*, $p_i$ gathers in $P_i(k)$ the set of all processes that *participated* in *write(k,*)*.

An important property of every set $P_i(k)$ is that it must contain at least one correct process.

2. Every process $p_i$ maintains a list of process sets, $E_i$, where each set within $E_i$ gathers the processes that participated in some previous write performed by $p_i$. Basically, before *write(k,*)* in $Reg_i$, $E_i := \{P_i(0), P_i(1), P_i(2), ..P_i(k-1)\}$. Initially, $E_i$ contains exactly one set: the set of all processes $\Pi$, i.e., we assume that $P_i(0) = \Pi$. Then, whenever $p_i$ terminates some *write*, $p_i$ updates the set $E_i$.

   An important property of the set $E_i$ is that, eventually, all new sets $(P_i(k))$ that will be added to $E_i$ will contain only correct processes. This is because after all faulty processes have crashed, the processes that will participate in new write operations will necessarily be correct.

3. The value that process $p_i$ writes in $Reg_i$, together with $k$ (i.e., its $k-$th *write*), is the value of $E_i$ after the $(k-1)-$th *write*. After a process $p_i$ writes $E_i$ in $Reg_i$, $p_i$ reads every register $Reg_j$ written by every process $p_j$. Process $p_i$ selects at least one process $p_x$ from every set it reads (in some register $Reg_j$) by sending a message to all processes in this set and waiting for at least one reply. The value of the variable $Trust_i$ is the value of $P_i(k-1)$ augmented with every process $p_x$ that $p_i$ selected.

   An important property of variable $Trust_i$ is that it is permanently updated if process $p_i$ is correct. This is because $p_i$ only waits for a message from one process in every set that it reads in a register, and every such set contains at least one process.

In short, the *completeness* property of $\Sigma$ is ensured because every correct process $p_i$ will permanently keep on updating variable $Trust_i$ and this variable will eventually contain only correct processes. The *intersection* property of $\Sigma$ is ensured because every process $p_i$ writes in its register before reading all other registers and updating $Trust_i$.

## 5.2 The algorithm

To describe our algorithm $R$ more precisely, we divide it in two parts. We describe in Figure 1 how to emulate variable $Trust$ using a specific SWMR register customized to our needs. Then we show in Figure 2 how to implement that specific register with any algorithm that implements a traditional SWMR register in a message passing system with some failure detector.

The specific SWMR register we consider has a traditional read and a non-traditional write. The write has, besides any possible input parameter that the writer might be using to store some value in the register, a specific input parameter: an integer that the writer uses to indicates the number of times the write operation has been invoked. Furthermore, the write returns an output, which is the list of processes that *participated* in the *write* (i.e., in the underlying message passing exchange).

We first define here more precisely what *participate* means. Let $w$ be some write operation invoked by some writer process $p_i$ in the specific SWMR register we consider; $w_b$, respectively $w_e$, denotes the beginning event, respectively the termination event of the write operation $w$. Let $\preceq$ be the causality relation of [27], the set of *participants* in $w$, is the set of processes:

$$\{p_j \in \Pi | \exists e \text{ event of } p_j : w_b \preceq e \preceq w_e\}$$

The algorithm of Figure 2 describes how to track and return the set $P_i(k)$ of participants during every *write(k,*)* operation. For this, let $p_i$ be the writer of a SWMR atomic register $Reg_i$ and consider an algorithm implementing this register with some failure detector. We tag every message causally after the beginning of the $k$–th write of $Reg_i$ and causally before the beginning of the $k+1$–th write with a pair $(k, L)$, where $L$ is the list of participants to the $k$–th write.

### 5.3 Correctness

The following lemma states that the set of processes returned by the algorithm of Figure 2 is indeed the set of processes that participate in the write.

**Lemma 5.1** *In the algorithm of Figure 2, the set of participants of the $k$–th terminated write is $P_i(k)$.*

**Proof.** Let $w$ be the $k$–th terminated write of $p_i$. We denote the list of participants in $w$ as $\mathcal{P}(w)$.

1. We prove first that $\mathcal{P}(w) \subseteq P_i(k)$. Let $x$ be any process of $\mathcal{P}(w)$. There exists an event $e$ of $x$ such that $w_b \preceq e \preceq w_e$. Let $M_1$ be the causal chain of messages from $w_b$ to $e$ and $M_2$ be the causal chain from $e$ to $w_e$. Every message in $M_1$ or $M_2$ can only be tagged by $(j, *)$ with $j \geq k$. As $p_i$ does not begin the $j$–th write, with $j > k$, before the end of the $k$–th write, hence all messages of $M_2$ are tagged by $(k, *)$. Moreover, an easy induction proves that every message in $M_2$ has tag $(k, K)$ such that $x$ is in $K$. As the tags of these messages are in $P_i(k)$ we have $x \in P_i(k)$.

2. Now we prove that $P_i(k) \subseteq \mathcal{P}(w)$. Let $x$ be any process of $P_i(k)$.

   As only $x$ itself can add its identity to the list $L$ of the tag $(k, L)$ of a message, any $p_u$ can only receive a message with tag $(k, L)$ such that $x \in L$ only causally after that $x$ sends some message with tag $(k, M)$ with $x \in M$. Let $e_0$ be the event corresponding to the first time $x$ sends message with tag $(k, L)$ and let $e_1$ be the event corresponding to the first

```
Code for every process p_i
Initialization:
   P_i(0) := Π
   E_i := {P_i(0)} /* E_i is the set of subsets of processes that participate in write on Reg_i */
   k:=0 /* k represents the number of times a write on was invoked by p_i */
   F_i := ∅ /* F_i is a temporary value of trusted processes */
   Trust_i := Π /* Initially, all processes are trusted */
start task 1 and task 2

task 1:
loop forever
   k := k + 1
   P_i(k) := Reg_i.write(k, E_i)
   E_i := E_i ∪ {P_i(k)}
   F_i := P_i(k − 1)
   forall p_j ∈ Π do
      L_j := Reg_j.read()
      forall X ∈ L_j do
         send(k, ?) to all processes in X
         wait until receive(k, ok) from at least one process p_t ∈ X
         F_i := F_i ∪ {p_t}
   Trust_i := F_i

task 2:
   upon receive(l, ?) from p_j send(l, ok) to p_j
```

Figure 1: Emulating a Quorum failure detector

time $p_i$ receives a message with tag $(k, L)$ with $x \in L$ we have: $e_0 \preceq e_1$. Moreover, as $x \in P_i(k)$, the algorithm ensures that $e_1 \preceq w_e$ and then (1) $e_0 \preceq w_e$.

As only $p_i$ increments the value of $j$ in tag $(j, *)$, and the value of $Current$ for $x$ can only be set to $k$ when $x$ receives a message with tag $(k, *)$. Given that in $e_0$ the value of $Current$ for $x$ is $k$, we have: (2) $w_b \preceq e_0$.

We can deduce from (1) and (2) that $e_0$ is an event of $x$ such that $w_b \preceq e_0 \preceq w_e$. Hence $x \in \mathcal{P}(w)$

■

The following lemma states that any set of processes participating to some write contains at least one correct process.

**Lemma 5.2** *Let $w$ be the $k-th$ terminated write in some failure pattern $F$:* $P_i(k) \cap correct(F) \neq \emptyset$

/* Every process $p_i$ tags every message it sends with $Tag$ */
/* $Tag$, the tag for register $Reg_i$, is a pair $(k, L)$*/

*Initialization:*
  $Current := 0$
  $Tag := (0, \emptyset)$

Code for every process $p_j$ (including the writer $p_i$ of $Reg_i$):
  When $p_j$ receives a message tagged with $(k, L)$ for $Reg_i$
    **case** $Current > k$ **: skip**
    **case** $Current = k$ **:** $Tag := (k, L \cup Tag.L \cup \{p_j\})$
    **case** $Current < k$ **:** $Tag := (k, L \cup \{p_j\})$
        $Current := k$;

Code for the writer $p_i$ of $Reg_i$:
  When $p_i$ begins the $k$-th *write* on register $Reg_i$
    $Current := k$
    $Tag := (k, \{p_i\})$
  When $p_i$ ends the $k$-th write operation on register $Reg_i$
    **Return** $L$ **: such that** $Tag = (k, L)$
    /* $L$ is the set of *participants* of the $k$–th write */

Figure 2: Tagging for register $Reg_i$

Remember first that two consecutive write operations always write two different values.

**Proof.** Remember that $correct(F) \neq \emptyset$. In order to obtain a contradiction, assume that for some terminating write $w$ of $p_i$ in register $Reg_i$ and run $\alpha =< F, H, C, S, T >$, we have $P_i(k) \cap correct(F) = \emptyset$.

In the following, we exhibit several runs; all these runs have $F$ as failure pattern and $H$ as failure detector history. They may differ from run $\alpha$ above by the time at which processes take steps: we strongly use the fact that the system is asynchronous.

- We construct first run $\alpha_0$, identical to $\alpha$ up to $w_e$, and for which the writer $p_i$ does not invoke any write after $w_e$, clearly, the set of participants in the k-th write w, $P_i(k)$, is the same in $\alpha$ and $\alpha_0$. Let $v$ be the value of register $Reg_i$ before the terminating write $w$ and $v'$ the value after (recall we assume that $v \neq v'$). Let $\tau_e$ be the time of the event $w_e$ in $\alpha$, and consider time $\tau \geq \tau_e$ after which no more processes crash. Remark that, by hypothesis, at time $\tau$ all participants in $w$ have crashed. For any process $x$ in $P_i(k)$, let $b_x$ be the first event of $x$ such that $w_b \preceq b_x \preceq w_e$, and $e_x$ be the last event of $x$ such that $w_b \preceq e_x \preceq w_e$. In the following, $b_x^{-1}$ denotes the last event of $x$ before $b_x$.

- We now construct run $\beta$. For any process $x$ of $P_i(k)$, $\beta$ is identical to $\alpha_0$ up to $b_x^{-1}$, but after $b_x^{-1}$, $x$ does not take any step until time $\tau$. As after time $\tau$, $x$ has crashed, $x$ does not take any step after $b_x^{-1}$. The processes of $\Pi - P_i(k)$ take steps exactly as in $\alpha$ up to time $\tau$ (at the same time, but perhaps the step is not the same). At time $\tau$, a correct process, say $p_j$, reads the register $Reg_i$ and $p_j$ ends the read at time $\tau'$. As $w$ is not a write operation in run $\beta$, $p_j$ reads the value $v$ of the register.

- We now construct run $\gamma$. For any process $x$ of $P_i(k)$, $\gamma$ is identical to $\alpha$ up to $e_x$. After $e_x$, $x$ does not take any step until time $\tau$. If $x$ sends after $b_x$ a message to a process of $\Pi - P_i(k)$, the reception of this message is delayed until after time $\tau'$. Processes of $\Pi - P_i(k)$ take steps exactly as in $\beta$ up to time $\tau'$ (now the steps that these processes take in $\gamma$ and $\beta$, are the same steps up to time $\tau'$). After time $\tau'$, the correct processes may receive the pending messages and runs $\beta$ and $\gamma$ may differ.

  In $\gamma$, the writer has completed its write operation $w$. The reader $p_j$ begins the read after the end of the write: by the properties of a SWMR register $p_j$ reads $v'$.

For $p_j$, $\gamma$ and $\beta$ are indistinguishable. As $p_j$ reads $v$ in $\beta$, $p_j$ reads $v' \neq v$ in $\gamma$ — a contradiction. ■

**Proposition 5.3** *The algorithms of Figure 1 and Figure 2 emulate in variable* $Trust$ *a failure detector of class* $\Sigma$.

**Proof.** Variable $Trust$ outputs a list of processes. We show that this list ensures the *completeness* and *intersection* properties of $\Sigma$. Remember that the copy of the variable *Trust* at process $p_i$ is denoted by $Trust_i$.

In the following, we denote by $T_i = Trust_i^1, \ldots, Trust_i^m \ldots$ the (finite or infinite) sequence of values written by $p_i$ in its variable $Trust_i$ (Line 20 of the algorithm of Figure 1), and we denote by $\tau_i^1, \ldots, \tau_i^m, \ldots$ the corresponding sequence of times at which $p_i$ updates variable $Trust_i$: more precisely, $p_i$ writes $Trust_i$ for the $k$-th time at time $\tau_i^k$ and the value written is $Trust_i^k$. By definition, the sequence $T_i$ is also the sequence of outputs of the emulated failure detector at process $p_i$.

1. We first prove the *completeness* property of *Trust*. We need to show that for every correct process $p_i$, there is an integer $m$ such that for every $m' > m$, $Trust_i^m$ contains only correct processes.

   At least one of the processes in every set $P_j(k)$ at any process is correct by Lemma 5.2. Therefore, at least one of the processes in a set $X$ answers to the message $(k, ?)$ from process $p_i$. Therefore, process $p_i$ cannot block on Line 18 and if $p_i$ is a correct process, $p_i$ updates infinitely often variable $Trust_i$ and the sequences $Trust_i^1, \ldots Trust_i^m \ldots$ as well as $\tau_i^1, \ldots, \tau_i^m, \ldots$ are infinite sequences.

   By Lemma 5.1 and the very fact that eventually, all processes that participate in the write operations are correct, there is a time $\tau$ after which all faulty processes have crashed. As $p_i$ is correct, consider some $m$ such that $\tau_i^m > \tau$. Let $p_l$ be any process that belongs to $Trust_i^{m'}$ with $m' \geq m + 2$:

- either $p_l$ belongs to $P_i(m'-1)$, meaning that $p_l$ is a correct process.
- or $p_l$ answered some message $(m', ?)$ from $p_i$, ensuring that $p_l$ was not crashed at least until time $\tau_i^m$.

In both cases, $p_l$ is correct, proving the *completeness* property.

2. We now prove that *Trust* ensures the *intersection* property. More precisely, we prove that, given any two processes $p_i$ and $p_j$, for all $k, l$ such that $Trust_i^k$ and $Trust_j^l$ are both defined, we have $Trust_i^k \cap Trust_j^l \neq \emptyset$.

   Remark that if $l = 0$ or $k = 0$ then either $Trust_i^l$ or $Trust_j^k$ is the set of all processes $\Pi$, as clearly $Trust$ is never empty, in this case we have $Trust_i^l \cap Trust_j^k \neq \emptyset$. Therefore assume that $l > 0$ and $k > 0$.

   Notice the following facts:

   - If process $p_i$ writes $E_i$ in its register $Reg_i$ (Line 11) during the $k$-th iteration, then, for all $k' < k$, $P_i(k') \in E_i$. By construction, the value of $E_i$ for the $k$-th *write* of register $p_i$ is the set of all sets $P_i(k')$ for $k' < k$.
   - It is clear from Line 19 that $P_i(l-1) \subseteq Trust_i^l$.

   As each process $p_i$ writes its own register $Reg_i$ and then reads every register of all other processes, due to the atomicity of registers, either the $k$–write of register $Reg_j$ by $p_j$ occurs before the $l$–th read of this register by process $p_i$, or the $l$-write of register $Reg_i$ is before the $k$–th read of this register by process $p_j$.

   Therefore, assume without loss of generality that $p_i$ does its $l$–th read of the register $Reg_j$ after the $k$-th write of $Reg_j$ by $p_j$. From the algorithm, at least one $s \in Trust_i^l$ (1) comes from the set of sets $L_j$ read by $p_i$ in the $l$–th read of $Reg_j$ and (2) is such that $p_i$ has received an $(l, OK)$ answer from $s$. As we assume that the $l$–th read is after the end of the $k$–th write of $Reg_j$ by $p_j$, we deduce that at least one $s \in Trust_i^l$ belongs to $P_j(k-1)$, as $P_j(k-1) \subseteq Trust_j^k$, $s$ belongs to $Trust_j^k$, proving the *intersection* property.

■

# 6 The sufficient condition

The aim of this section is to show that failure detector class $\Sigma$ implements a register abstraction in any environment. More precisely, we describe an algorithm that implements a SWSR register in any environment and with any failure detector of $\Sigma$. We give our algorithm assuming a weak communication model where any message sent by a process $p_i$ to a process $p_j$ is guaranteed to be received by $p_j$ provided that both processes are correct. The writer of our register is denoted by $p_w$ whereas the reader is denoted by $p_r$.

Our algorithm, described in Figure 3 is an adaptation of [5] and works as follows: Roughly speaking, where the algorithm of [5] uses the assumption of a majority of correct processes to implement the read and write operations, we use the *Quorum* failure detector. Basically, every process (including the writer $p_w$ and the reader $p_r$) maintains the current value of the register.

- The writer process $p_w$ tags each write invocation with a unique sequence number, incremented for every new write invocation. The writer then sends the value to be written with the associated sequence number to all processes. Each process $p_i$ is supposed to store this value with its sequence number and sends back an acknowledgment to the writer, unless $p_i$ had already stored a value with a higher sequence number. The writer $p_w$ waits until receiving acknowledgments from every process trusted by $p_w$, i.e., from every process output by its failure detector module, before terminating the *write*.

- For any read operation, the reader $p_r$ sends a request to read to all. Every process is supposed to return a message containing the last value written and the corresponding sequence number. The reader then selects the value with the largest sequence number among those received from the trusted processes and the one previously hold by the reader. Finally, the reader updates its own value and timestamp with the selected value, and then returns this value.

It is important to notice that the set of processes trusted by the reader (resp. the writer) process might change several times between the time the process sends its message and the time it receives acknowledgments. We implicitly assume here that the reader (resp. the writer) process keeps periodically consulting the list of processes that are output by its failure detector module, and stops waiting when the process has received acknowledgments from all processes in the list.

Roughly speaking, the *completeness* property of the failure detector ensures that, unless it crashes, the reader (resp. the writer) does not block waiting forever for acknowledgments. The *intersection* property of the failure detector (together with the use of timestamps) ensures that a reader would not miss a value that was written.

## 6.1 Correctness

**Proposition 6.1** *The algorithm of Figure 3 implements a SWSR register with $\Sigma$ in any environment.*

**Proof.** Remember that we assume a single writer, denoted by $p_w$, and a single reader, denoted by $p_r$. Remark first that:

```
1  Every process p_i (including p_w and p_r) executes the following code:
2     Initialization:
3        current := ⊥
4        last_write := −1
5        upon receive (WRITE, y, s) from the writer
6        if s > last_write then
7           current := y
8           last_write := s
9           send(ACK_WRITE, s) to the writer
10       upon receive (READ, s) from the reader
11          send(ACK_READ, last_write, current, s) to the reader
12 Code for p_w the (unique) writer:
13    Initialization:
14       seq := 0 /* sequence number */
15    procedure write(x)
16       send(WRITE, x, seq) to all
17       wait until received (ACK_WRITE, seq)
             from all processes trusted by the local failure detector module
18       seq := seq + 1
19    end write
20 Code for p_r the (unique) reader:
21    Initialization:
22       rc := 0 /* reading counter */
23    function read()
24       rc := rc + 1
25       send(READ, rc) to all
26       wait until received (ACK_READ, ∗, ∗, rc)
             from all processes trusted by the local failure detector module
27       a := max{v | (ACK_READ, v, ∗, rc) is a received message}
28       if a > last_write then
29          current := v such that (ACK_READ, a, v, rc) is a received message
30          last_write := a
31       return(current)
32    end read
```

Figure 3: Implementation of a SWSR register.

(A) if $p_w$ has not terminated its $k$–th write (after line 17) then, at all processes, the value of variable *last_write* is less or equal to $k$.

Indeed, the *last_write* is updated according to the value obtained from some write operation: remember that we assume that no message can be received unless it was sent. By the assumption that processes are sequential, the writer does not begin its $(k+1)$-th write operation before ending its $k$–th one.

Assume that the $k$–th write by $p_w$ is for value $v$. We have:

(B) At the time when a process stores $k$ in its variable *last_write* (line 8), the value of its variable *current* is $v$.

From this we deduce the following:

(C) If any process sends an $(ACK\_READ,\ s, v, *)$ message, then $v$ is the value of the $s$-th write operation.

In particular (C) implies that for all $(ACK\_READ,\ s, v, *)$, $(ACK\_READ,\ s', v', *)$ messages: $s = s' \Rightarrow v = v'$.

Now we proceed to prove the properties of the SWSR register using (A), (B) and (C) above:

*Liveness:* Assume by contradiction that $p_w$ is correct and $p_w$ invokes but does not terminate its $k$–th write operation.

This is only possible if $p_w$ waits forever in line 17. From the *completeness* property of the failure detector, there is a time $\tau$ after which the list $L$ of processes trusted by $p_w$ contains only correct processes. By the properties of the communication channels, every correct process $p_i$ eventually receives the $(WRITE, *, k)$ message from $p_w$. From (A), $p_i$ replies with a $(ACK\_WRITE, k)$ message and $p_w$ eventually receives $(ACK\_WRITE, k)$ messages from all processes within $L$ –a contradiction.

A similar argument proves that, unless the reader crashes, every read operation invoked by the reader always terminates.

*Validity:* Let $R$ be the $j$–th read operation invoked by the reader, let $W$ be the last write operation terminated before the beginning of $R$, and assume that $W$ is the $k$–th write of the writer. Such a write exists because we assume that even initially, the default value of the register was written by $p_w$.

The writer $p_w$ terminates this write operation (after line 17) after having received $(ACK\_WRITE, k)$ messages from a set $L_w$ of trusted processes.

When the reader $p_r$ terminates its read operation $R$, $p_r$ has received $(ACK\_READ, *, *, j)$ messages from a list $L_r$ of trusted processes. By the *intersection* property of the failure detector, at least one process $p_i$ belong to both $L_w$ and $L_r$.

As $p_i$ sends an $(ACK\_READ,\ s, *, j)$ message to $p_r$ after having sent an $(ACK\_WRITE, k)$ message to $p_w$, then $s \geq k$. Hence, $a$, the maximum of $v$ in the $(ACK\_READ, v, *, j)$ messages received by the reader $p_r$ for operation $R$ is such that $a \geq s \geq k$ and then: (D) $a \geq k$.

From (A), the $a$–th write has begun before read $R$ has terminated, and by (C) the value returned by $R$ is the value of the $a$–th write.

Consider the two following cases:

- The read operation $R$ is not concurrent with any write operation. Then, from (A), $(ACK\_READ, x, *, j)$ messages received by $p_r$ for $R$ are such that $x \leq k$. From (D), we can deduce that $a = k$ and the value returned by $R$ is the value of write $W$.

- The read is concurrent with some write. In this case, $a \geq k$. The value returned is either the value of write $W$ or the value of some concurrent write.

*Ordering:* We need to show here that if the reader reads $x$, then reads $y$, then the writer could not have written $x$ after $y$.

Assume that the values of *last_write* are respectively $rx$ and $ry$, at the reader, when it returns respectively $x$ and $y$. From the algorithm, $x$ is the written value by the $rx$-th write and $y$ is the written value by the $ry$-th write. As *last_write* is always non decreasing, we have $ry \geq rx$, hence $p_w$ wrote $y$ after $x$. ■

The following corollary follows from Proposition 5.3 and Proposition 6.1.

**Corollary 6.2** *In any environment, $\Sigma$ is the weakest failure detector class to implement a register.*

# 7 Consensus

We define below the class of failure detectors $\Sigma \times \Omega$, which we show is the weakest to solve consensus in every environment. Failure detectors of this class have range $2^{\Pi} \times \Pi$. For every failure pattern, they output pairs of process sets, at any time $\tau$, and any process $p_i$, such that the first set satisfies the *completeness* and *intersection* properties of $\Sigma$ and the second set satisfies the *leader* property of $\Omega$. (Clearly, this class is stronger than each of the classes $\Sigma$ and $\Omega$.)

## 7.1 The weakest failure detector class

The following corollary states that consensus can be implemented using failure detector class $\Sigma \times \Omega$ (in every environment). Such implementation can be achieved by first implementing registers out of $\Sigma$, and then consensus out of registers and $\Omega$ [30][6]:

**Corollary 7.1** $\Sigma \times \Omega$ *implements consensus in every environment.*

The following corollary follows from Corollary 6.2 and the fact that consensus can be used to implement registers in every environment:

**Corollary 7.2** *Any failure detector class $\mathcal{D}$ that solves consensus in some environment, is stronger than $\Sigma \times \Omega$ in that environment.*

From the above two corollaries, we deduce the following one:

**Corollary 7.3** $\Sigma \times \Omega$ *is the weakest failure detector class to implement consensus in every environment.*

[6]The resulting consensus is obviously not efficient as it first emulates a register and then builds consensus on top it. More efficient algorithms (bypassing the register emulation) can be obtained following the approach of [33].

## 7.2 An interesting particular case

We show in the appendix that, in our system model with at least 3 processes, $\Sigma \times \Omega$ is strictly stronger than $\Sigma$ in every environment $\mathcal{E}_t$ with $t > 0$, and strictly stronger than $\Omega$ in every environment $\mathcal{E}_t$ with $t \geq n/2$. In such environment, the two classes $\Sigma$ and $\Omega$ are indeed complementary. We also show that $\Sigma \times \Omega \preceq \mathcal{S}$. So far, $\mathcal{S}$ was the weakest failure dectector among known failure dectectors that implement consensus in any environment.

Interestingly, if we consider a system of $n = 2$ processes, failure detector classes $\Sigma$ and $\Omega$ are equivalent (we give the proof in the appendix). As a consequence, $\Sigma$ is in this case the weakest failure detector class for consensus. In other words, among two processes communicating by message passing, solving consensus and implementing a register require the same information about failures.

# 8 Reliable broadcast

*(Uniform) reliable broadcast* [22] (we omit the term uniform in the rest of the paper) is a distributed computing abstraction that shares some similarities with the register abstraction. In an asynchronous system with reliable message passing, both can be implemented only in environments with a majority of correct processes. In such environments, they are in a precise sense equivalent [7].

In the following, we use our previous result on the weakest failure detector class to implement a register, to show that a register is strictly stronger than reliable broadcast in environments where at least half of the processes can crash. We first recall the definition of reliable broadcast and we show to implement it using registers. Then we show that the weakest failure detector class to implement reliable broadcast [1] is strictly weaker than $\Sigma$ in every environment where at least half of the processes can crash.

## 8.1 Reliable broadcast with registers

The reliable broadcast abstraction is defined through two primitives, *broadcast* and *deliver*, that the processes use to exchange messages. When a process invokes the primitive *broadcast* with a message $m$ as a parameter, we say that the process broadcasts m; when a process returns from the invocation of *deliver* with $m$ as a parameter, we say that the process delivers $m$. Messages are supposed to be uniquely identified. The *broadcast* and *deliver* primitives ensure the following properties.

- *Validity:* If a correct process broadcasts a message $m$, then it eventually delivers $m$.

[7] Remember that we consider by default the weak channel assumption. With a strong channel assumption, reliable broadcast is trivial and can be implemented in every environment.

- *(Uniform) Agreement:* If a process delivers a message $m$, then all correct processes eventually deliver $m$.
- *(Uniform) Integrity:* For every message $m$, every process delivers $m$ at most once, and only if $m$ was previously broadcast by *sender(m)*.

**Lemma 8.1** *Register implements reliable broadcast in every environment.*

**Proof.** We give a simple algorithm that use registers to implement the primitives *broadcast* and *deliver* with the above properties of a reliable broadcast. The idea of the algorithm is the following. The processes make use of $n$ registers: one per process. More precisely, every process $p_i$ in $\Pi$ is associated with one register, denoted by $Reg_i$. Process $p_i$ is the unique writer of register $Reg_i$ but all processes can read in all registers. In addition, every process $p_i$ maintains the list of messages it has broadcast so far, denoted by $bList_i$, as well as the list of messages it has delivered so far, denoted by $dList_i$.

- For a process $p_i$ to *broadcast* a message $m$, $p_i$ adds $m$ to $bList_i$, and then writes $bList_i$ in $Reg_i$.
- Periodically, every process $p_i$ reads every register $Reg_j$ and selects every message $m$ in $bList_j$ that $p_i$ did not deliver so far, i.e., every message $m$ that was not yet in $dList_i$. Process $p_i$ then adds every selected message $m$ to $dList_i$ and *delivers* $m$.

We show below that this algorithm ensures the properties associated with a reliable broadcast and recalled above.

1. Consider *validity* and assume that some correct process $p_i$ broadcasts a message $m$. By the algorithm, and given that $p_i$ is correct, $p_i$ eventually adds $m$ to its list $bList_i$ and writes that list in $Reg_i$. From there on, any value written in $Reg_i$ contains message $m$; this is because $p_i$ will keep on adding the messages to be broadcast to $bList_i$ and writing the new lists in $Reg_i$. Given that $p_i$ is correct, $p_i$ eventually reads $Reg_i$ and delivers $m$.
2. Consider now *agreement*. Let $p_i$ be any process that delivers a message $m$ and let $p_j$ be any correct process. For $p_i$ to deliver $m$, $p_i$ must have read $m$ from some register $Reg_k$. After this read, and by the atomicity property of registers, any process that reads $Reg_k$ reads a value $bList_k$ that contains $m$. Process $p_j$ is correct and then eventually reads $m$ and delivers $m$.
3. Finally, consider *integrity*. Let $p_i$ be any process that delivers some message $m$. Process $p_i$ can only do so if it did not deliver $m$ before; remember that $p_i$ maintains the list of delivered messages in $dList_i$. Process $p_i$ must have read $m$ from the list $bList_k$ of some register $R_k$, written by $p_k$. By the atomicity property of the registers, process $p_k$ must has written $bList_k$ in $Reg_k$ with message $m$ in $bList_k$. By the algorithm, $p_k$ must have broadcast $m$.

■

### 8.2 A register is stronger than reliable broadcast

It was shown in [1] that the weakest failure detector class to implement reliable broadcast is the class $\Theta$. Failure detectors of this class output a list of processes that are trusted to be up (just like our class $\Sigma$). For presentation simplicity, we assume that after a process $p$ crashes, its failure detector of $\Theta$ permanently outputs all processes. Every failure detector of $\Theta$ ensures the two following properties, for any failure pattern and any associated failure detector history $H$:

- *(Completeness):* there is a time $\tau$ such that for every time $\tau' > \tau$, for every correct process $p$: $H(p, \tau')$ contains only correct processes,
- *(Accuracy):* for every process $p$, for every time $\tau$, $H(p, \tau)$ contains at least one correct process.

**Lemma 8.2** *In any environment $\mathcal{E}_t$ with $t \geq n/2$, $\Sigma$ is strictly stronger than $\Theta$.*

**Proof.** It is obvious from the definitions of $\Theta$ and $\Sigma$ that any failure detector of $\Sigma$ is also a failure detector of $\Theta$: therefore $\Sigma$ is stronger than $\Theta$. We show in the following that the converse is not true in any environment where at least half of the processes can crash. The fundamental characteristic of such environment is that we can partition the system into two subsets $I$ and $J$ such that all processes in any of these subsets can crash in the same failure pattern. Then we assume by contradiction the existence of an algorithm $R$ that emulates, within some distributed variable $Trust$, a failure detector of class $\Sigma$, using any failure detector of class $\Theta$.

Consider a failure detector $\mathcal{D}$ that, in any failure pattern where at least one process from $I$ (resp. $J$) is correct, output at all processes of $I$ (resp. $J$), the same correct process also from $I$ (resp. $J$). In failure patterns where all processes from $I$ (resp. $J$) are faulty, $\mathcal{D}$ outputs at all processes, the same correct process from $J$ (resp. $I$). Clearly, $\mathcal{D}$ is of class $\Theta$: no faulty process is ever output and at least one correct process is always output at every correct process. Consider the variable $Trust$ emulated by algorithm $R$ and consider a failure pattern $F$ where some process of $I$ is correct, say $p_i$, and this process is the one output by $\mathcal{D}$ at $p_i$, and similarly, some process of $J$ is correct, say $p_j$, and this process is the one output by $\mathcal{D}$ at $p_j$. Process $p_i$ (resp. $p_j$) cannot distinguish this failure pattern from the failure pattern where all processes in $J$ (resp. $I$) are faulty. Hence, to guarantee the *completeness* property of $\Sigma$, there is a time at which $\mathcal{D}$ does not output any process from $J$ (resp. $I$) at $p_i$ (resp. $p_j$), violating the *intersection* property of $Trust$. ■

## 9 Realistic failure detectors

So far we considered the original model of [6] according to which a failure detector is any function of the failure pattern: this enables for instance to make of our weakest failure detector result for consensus a strict generalization of [7]. The original definition of [6] includes failure detectors (i.e., failure pattern functions) that provide information about *future* failures.

In this section, we restrict our space to failure detectors as functions of the "past" failure pattern. We first define the class $\mathcal{R}$ of *realistic* failure detectors (those that cannot guess the future), which includes among others, failure detectors of all classes we discussed so far, i.e., $\mathcal{P}$, $\mathcal{S}$, $\Diamond\mathcal{S}$, $\Omega$, and $\Sigma$. We illustrate this notion through two simple examples. Then we consider the *wait-free* environment where $n-1$ processes can crash and we show that many distributed programming abstractions, including register, consensus, reliable broadcast, as well as terminating reliable broadcast [22] in a crash failure model which we recall below) are equivalent (for any $n$).

## 9.1 Definition

Roughly speaking, we say that a failure detector is *realistic* if it cannot guess the future. In other words, there is no time $\tau$ and no failure pattern $F$ at which the failure detector can provide information about crashes that will hold after $\tau$ in $F$. More precisely, we define the class of realistic failure detector $\mathcal{R}$, as the set of failure detectors $\mathcal{D}$ that satisfy the following property for every environment $\mathcal{E}$:

- $\forall (F, F') \in \mathcal{E}\ \forall \tau \in \Phi$ s.t. $\forall \tau_1 \leq \tau, F(\tau_1) = F'(\tau_1)$, we have:
  - $\forall H \in \mathcal{D}(F), \exists H' \in \mathcal{D}(F')$ s.t.: $\forall \tau_1 \leq \tau, \forall p_i \in \Pi : H(p_i, \tau_1) = H'(p_i, \tau_1)$.

Basically, a failure detector $\mathcal{D}$ is *realistic* if for any pair of failure patterns $F$ and $F'$ that are similar up to a given time $\tau$, whenever $\mathcal{D}$ outputs some information at some time $\tau - k$ in $F$, $\mathcal{D}$ could output the very same information at the same time in $F'$. In other words, a realistic failure detector cannot distinguish two failure patterns according to what will happen in the future.

## 9.2 Examples

We illustrate below our notion through two examples: the first is a failure detector that is realistic, and actually constitutes the *strongest* (class) among these, whereas the second one is a non-realistic failure detector.

### 9.2.1 The Scribe.

We describe here the *Scribe* failure detector $\mathcal{C}$, which we show constitutes (as a singleton) the strongest class among all classes of realistic failure detectors. In short, failure detector $\mathcal{C}$ sees what happens at all processes at real time and takes notes of what it sees. More precisely, in any failure pattern $F$, failure detector $\mathcal{C}$ outputs, at any time $\tau$, the list of values of $F$ up to time $\tau$: we denote this list by $F[\tau]$. More precisely, for each failure pattern $F$, $\mathcal{C}(F)$ is the singleton that contains the failure detector history $H$ such that:

- $\forall \tau \in \Phi, \forall p_i \in \Pi,\ H(p_i, \tau) = F[\tau]$.

It is obvious to see that failure detector $\mathcal{C}$ is realistic. In fact, the singleton $\{\mathcal{C}\}$ is the strongest among classes of realistic failure detectors. In other words, given any realistic failure detector $\mathcal{D}$, there is an algorithm $A$ that transforms $\mathcal{C}$ into $\mathcal{D}$. Given that $\mathcal{D}$ is a realistic failure detector, for any failure pattern $F$, any process $p_i \in \Pi$ and any time $t \in \Phi$, the output of $\mathcal{D}$ is a function $\mathcal{D}(F[\tau])$ of $F[\tau]$. For any failure pattern $F$, any process $p_i \in \Pi$ and any time $\tau \in \Phi$, the transformation algorithm $A$ simply takes $F[\tau]$ (i.e., the output of $\mathcal{C}$ at $p_i$ and $\tau$), and transforms it into $\mathcal{D}(F[\tau])$.

### 9.2.2 The Marabout.

Consider failure detector $\mathcal{M}$ (*Marabout*), defined in [18]. This failure detector outputs a list of processes. For any failure pattern $F$ and at any process $p_i$, the output of the failure detector $\mathcal{M}$ is constant: it is the list of faulty processes in $F$, i.e., $\mathcal{M}$ outputs the list of processes that *have* crashed or *will* crash in $F$. Clearly, $\mathcal{M}$ is not realistic: it does not belong to the set $\mathcal{R}$. To see why, consider failure patterns $F$ and $F'$ such that:

1. In $F_1$, all processes are correct, except $p_1$ which crashes at time 10.
2. In $F_2$, all processes are correct.

Consider $H_2$, any history in $\mathcal{M}(F_2)$. By the definition of $\mathcal{M}$, the output at any process and any time of $H_2$ is $\emptyset$. Consider time $T = 9$. Up to this time, $F_1$ and $F_2$ are the same. If $\mathcal{M}$ was realistic, $\mathcal{M}$ would have had a failure detector history $H_1$ in $\mathcal{M}(F_1)$ such that $H_2$ and $H_1$ are the same (at any process) up to time 9. This is clearly impossible since for any history $H_1 \in \mathcal{M}(F_1)$, for any process $p_i$, and any time $\tau \in \Phi$, $H_1(p_i, \tau) = \{p_1\}$. As observed in [18], the class $\mathcal{M}$ and the class $\mathcal{P}$ are incomparable. In short, $\mathcal{M}$ is accurate about the future whereas $\mathcal{P}$ is accurate about the past.

## 9.3 k-Perfect failure detectors

We define here the generic class $\mathcal{P}^k$, of *k-Perfect* failure detectors, where $0 \leq k \leq n$, which we show is the weakest to implement a register in environment $\mathcal{E}_k$ . This class is generic in the sense that its semantics depend on the value of the integer $k$. Failure detectors of class $\mathcal{P}^k$ output, at each process $p$ and each time $\tau$, a list of suspected processes $\mathcal{P}^k(p, \tau)$ (i.e., the range of $\mathcal{P}^k$ is $2^{\Pi}$). These failure detectors ensure *strong completeness* as well as the following *k-accuracy* property: at any time $\tau$, no process suspects more than $n-k-1$ processes that are alive at time $\tau$. More precisely:

- *k*- **Accuracy:** $\forall p \in \Pi,\ \forall \tau \in \Phi\ |\mathcal{P}^k(p, \tau) \setminus F(\tau)| \leq max(n-k-1, 0)$.

For $k \geq n-1$, processes do not make false suspicions and $\mathcal{P}^k$ is in $\mathcal{P}$. For $k < n-1$, processes can make false suspicions and can even permanently disagree on the processes they falsely suspect. To better illustrate the behaviour of a *k-Perfect* failure detector, consider a system of 5 processes $\{p_1, p_2, p_3, p_4, p_5\}$ and the case $k = 2$. The failure detector should eventually suspect permanently all crashed processes and should not falsely suspect more that 2 processes at every process. Consider a failure pattern where $p_1$ and $p_2$ crash. It can be the case that after some time $\tau$, $p_3$ permanently suspects $\{p_1, p_2, p_4, p_5\}$, $p_4$ permanently suspects $\{p_1, p_2, p_3, p_5\}$, and $p_5$ permanently suspects $\{p_1, p_2, p_3, p_4\}$. It can also be the case that after some time $\tau$, $p_5$ forever alternately suspects $\{p_1, p_2, p_3\}$ and $\{p_1, p_2, p_4\}$.

Given a failure detector class $\mathcal{X}$, we denote by $\mathcal{X}^r = \mathcal{X} \cap \mathcal{R}$ the realistic part of $\mathcal{X}$. Clearly, $\mathcal{P}^r$, $\mathcal{S}^r$, $\Diamond\mathcal{S}^r$, $\Sigma^r$ and $\Theta^r$ are all non-empty. Given any failure detector classes $\mathcal{X}$, $\mathcal{Y}$, such that $\mathcal{X}^r$ and $\mathcal{Y}^r$ are not empty if $\neg(\mathcal{X}^r \preceq \mathcal{Y}^r)$ then $\neg(\mathcal{X} \preceq \mathcal{Y})$. Moreover, if $\mathcal{X} \preceq \mathcal{Y}$ then $\mathcal{X}^r \preceq \mathcal{Y}^r$.

Interestingly, in the realistic case, we end up with another characterization of the weakest failure detector class to implement a register [9]:

**Proposition 9.1** *For any $t < n$, $\mathcal{P}^t$ is the weakest realistic failure detector class to implement a register in environment $\mathcal{E}_t$*

In other words, for every $t$, failure $\mathcal{P}^t$ is equivalent to $\Sigma^r$ in environment $\mathcal{E}_t$. Furthermore, in any environment $\Sigma \preceq \mathcal{P}^t$ (We prove these in the appendix). Althought, one might notice here that failure detector class $\mathcal{P}^t$ does not satisfy directly the *intersection* property of $\Sigma$.

## 9.4 The wait-free environment

Interestingly, in the realistic case, and considering the wait-free environment, most of the failure detector classes we have considered are equivalent:

**Proposition 9.2** *In the wait-free environment, $\Theta^r$, $\Sigma^r$, $\mathcal{S}^r$, and $\mathcal{P}^r$ are all equivalent.*[8]

As a direct consequence, in the realistic case and assuming a wait-free environment, register, consensus, reliable broadcast and terminating reliable broadcast are all equivalent. We show below that these are also equivalent in this case to another distributed programming abstraction: *terminating reliable broadcast*[22].

In this problem, a specific process (called the initiator) is supposed to broadcast a message. The processes are supposed to deliver that message but can

[8] The situation is different concerning $\Diamond\mathcal{S}$. In the wait free-environment, $\Diamond\mathcal{S}$ is not sufficient to implement consensus. If the environment is not wait free $\Theta^r \times \Diamond\mathcal{S}$ is not sufficient to implement consensus. In such environment, $\mathcal{P}^t \times \Diamond\mathcal{S}$ is strictly weaker than $\mathcal{S}^r$. To summarize: In $\mathcal{E}_t$, with $n/2 < t < n-1$, we have: $\Theta^r \times \Diamond\mathcal{S} \prec \Sigma^r \times \Diamond\mathcal{S} \prec S^r$ From this, we deduce that if the environment is not wait-free, the previous equivalences do not hold and inequalities are strict: In $\mathcal{E}_t$ with $n/2 < t < n-1$ we have: $\Theta^r \prec \Sigma^r \prec \mathcal{S}^r \prec \mathcal{P}$.

deliver a specific value *nil* if the sender process has crashed [22]. We actually consider a general variant of the problem where every process is a potential initiator of the broadcast. We denote by $(i, k)$ the $k$'th instance of the problem where the initiator of the broadcast is $p_i$. Instance $(i, *)$ is defined with the following properties: (1) *validity* if a correct process $p_i$ broadcasts a message $m$, then $p_i$ eventually delivers $m$, (2) *agreement* if a process delivers a message $m$, then every correct process delivers $m$; and (3) *integrity* if a process delivers a message $m$ and $p_i$ is correct, then $sender(m) = p_i$. We state and show here that if we do not restrict the number of faulty processes and consider realistic failure detectors, the *weakest* failure detector class to solve terminating reliable broadcast is $\mathcal{P}$.

**Proposition 9.3** *In the wait-free environment, the weakest realistic failure detector class for terminating reliable broadcast is* $\mathcal{P}^r$.

Proof (sketch): (*1. Sufficient condition.*) It is easy to see that any *Perfect* failure detector, solves the terminating reliable broadcast problem. When executing instance $(k, k')$ of the problem, every process that suspects $p_k$ delivers *nil*. Otherwise, $p_i$ waits for $p_k$'s message. (*2. Necessary condition.*) Let $A$ be any terminating reliable broadcast algorithm using $\mathcal{D}$. It is easy to see how we can emulate out of $\mathcal{D}$ a failure detector of class $\mathcal{P}^r$ in a distributed variable *output*$(\mathcal{P})$. Whenever a process $p_j$ delivers *nil* for an instance $(i, *)$ of the problem, $p_j$ adds $p_i$ to *output*$(\mathcal{P})_j$. Any process that crashes will eventually be permanently added to *output*$(\mathcal{P})$ at every correct process: *strong completeness* will hence be ensured. Let $p_i$ be any process that is added to *output*$(\mathcal{P})_j$ at some time $t$. This can only be possible if $p_i$ is faulty. Since we assume here that $\mathcal{D}$ is realistic, then $p_i$ must have crashed by time $t$, ensuring also *strong validity.*
□

# 10 The impact of uniformity

We considered throughout the paper the *uniform* variant of consensus: no two processes should disagree on the decision, even if any of them has crashed. An alternative variant is the *correct-restricted* one [36], where two processes can decide differently, as long as one of them is faulty. Although this might not make any sense in practice (before crashing, a process can give an inconsistent output to the application), studying this variant is theoretically interesting. In fact, $\Sigma \times \Diamond\mathcal{S}$ is not the weakest for correct-restricted consensus. Indeed, consider the class of *Partially Perfect* failure detectors, denoted by $\mathcal{P}_<$, and introduced in [15]. Failure detectors of this class output, at any time $\tau$ and at any process $p$, a list of processes suspected by $p$ at time $\tau$, and satisfy the following properties: (1) the *strong accuracy* property of $\mathcal{P}$ and (2) the following *partial completeness* property: if a process $p_i$ crashes, then eventually every correct process $p_j$ such that $j > i$ permanently suspects $p_i$.[9] There is an algorithm given in [15] that implements correct-restricted consensus with $\mathcal{P}_<$ in any environment. Furthermore, in any environment where half of the processes can crash, $\mathcal{P}_<$ is clearly not stronger than $\Sigma$. As a consenquence, $\Sigma \times \Diamond\mathcal{S}$ is thus not the weakest for correct-restricted consensus. This somehow conveys a fundamental difference between correct-restricted consensus and uniform consensus: the former is strictly weaker than the latter.

In fact, it is interesting to notice that correct-restricted consensus cannot even implement a register in a message passing system.


## Acknowledgments

Comments from Partha Dutta, Petr Kouznetsov, and Bastian Pochon helped improve the quality of the presentation of this paper. In particular, we thank Johny Eisler, Michel Raynal and Sam Toueg for hepful comments on a previous version of this paper.


## References


[1] M. Aguilera, S. Toueg, and B. Deianov, *Revisitin the Weakest Failure Detector for Uniform Reliable Broadcast.* Proceedings of the 13th International Conference on Distributed Computing, DISC 1999. Springer Verlag (LNCS), 1999.

[2] M. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. *Thrifty Generic Broadcast.* Proceedings of the 14th International Conference on Distributed Computing, DISC 2000. Springer Verlag (LNCS 1914), 2000.

[3] H. Attiya and J. Welch. *Distributed Computing. Fundamentals, Simulations, and Advanced Topics.* McGraw-Hill, 1998.

[4] K. Birman and R. van Renessee. *Reliable Distributed Computing with the Isis Toolkit.* IEEE Computer Society Press, 1993.

[5] H. Attiya, A. Bar-Noy, and D. Dolev. *Sharing Memory Robustly in Message Passing Systems.* Journal of the ACM, 42(1), January 1995.

[6] T. Chandra and S. Toueg. *Unreliable Failure Detectors for Reliable Distributed Systems.* Journal of the ACM, 43(2), March 1996.

[7] T. Chandra, V. Hadzilacos and S. Toueg. *The Weakest Failure Detector for Solving Consensus.* Journal of the ACM, 43(4), July 1996. (Revised and extended version of a PODC92 paper)


[9] Similar to the failure detector class $\Omega_i$, introduced in [35], $\mathcal{P}_<$ has a restricted completeness property: some faulty process might never be suspected. However, the two classes are different: with $\mathcal{P}_<$, completeness is restricted *a priori*: the only faulty process that might never be suspected (by anyone) is $p_n$. With $\Omega_i$, except when $i = 1$, any faulty process might never be suspected (by anyone).

[8] C. Dwork, N. Lynch and L. Stockmeyer. *Consensus in the presence of partial synchrony.* Journal of the ACM, 35(2), 1988.

[9] C. Delporte-Gallet, H. Fauconnier and R. Guerraoui. *A Realistic Look at Failure Detectors.* Proceedings of the IEEE International Conference on Dependable Systems and Networks, Washington DC, June 2002.

[10] C. Delporte-Gallet, H. Fauconnier and R. Guerraoui. *Failure Detection Lower Bounds on Registers and Consensus.* Technical Report LIAFA, (Paris 2002) and EPFL, IC/2002/030 (Lausanne 2002).

[11] C. Delporte-Gallet, H. Fauconnier and R. Guerraoui. *Failure Detection Lower Bounds on Registers and Consensus.* Proceedings of the 15th International Conference DISC 2002, (Toulouse, Fance) Springer Verlag (LNCS), 2002.

[12] P. Dutta and R. Guerraoui. *Fast indulgent consensus with zero degradation.* Proceedings of the European Dependable Computing Conference (EDCC'02), (Toulouse, Fance) Springer Verlag (LNCS), 2002.

[13] C. Fetzer. *Enforcing Perfect Failure Detection.* Proceedings of the IEEE International conference on Distributed Computing Systems, Phoenix, April 2001.

[14] M. Fischer, N. Lynch and M. Paterson. *Impossibility of Distributed Consensus with One Faulty Process.* Journal of the ACM, 32(2), 1985.

[15] R. Guerraoui. *Revisiting the Relationship Between the Atomic Commitment and Consensus Problems.* Proceedings of the International Workshop on Distributed Algorithms, Springer Verlag (LNCS 972), 1995.

[16] R. Guerraoui and A. Schiper. $\Gamma$-*Accurate Failure Detectors.* Proceedings of the International Workshop on Distributed Algorithms, Springer Verlag (LNCS 1151), 1996.

[17] R. Guerraoui and M. Raynal. *The Information Structure of Indulgent Consensus.* IEEE Transactions on Computers, 53 (4), April 2004.

[18] R. Guerraoui. *On the Hardness of Failure Sensitive Agreement Problems.* Information Processing Letters, 79, 2001.

[19] M. Herlihy. *Wait-free synchronization.* ACM Transactions on Programming Languages and Systems, 13(1), 123–149, 1991.

[20] M. Herlihy and J. Wing *Linearizability: a correctness condition for linearizable objects.* ACM Transactions on Programming Languages and Systems, 12(3), 463-492, 1990.

[21] V. Hadzilacos. *On the Relationship Between the Atomic Commitment and Consensus Problems.* Proceedings of the International Workshop on Fault-Tolerant Distributed Computing, Springer Verlag (LNCS 448), 1986.

[22] V. Hadzilacos, and S. Toueg. *A modular approach to fault-tolerant broadcasts and related problems*. Technical Report TR94-1425, Cornell University, May 1994.

[23] J. Halpern and A. Ricciardi. *A Knowledge-Theoretic Analysis of Uniform Distributed Coordination and Failure Detectors*. Proceedings of the ACM Symposium on Principles of Distributed Computing, 1999.

[24] A. Israeli and M. Li. *Bounded time-stamps*. Distributed Computing, 6 (4), July 1993.

[25] P. Jayanti. *Wait-free Computing*. Proceedings of the International Workshop on Distributed Algorithms, Springer Verlag (LNCS 972), 1995.

[26] L. Lamport. *Concurrent Reading and Writing*. Communications of the ACM, 20(11), 1977.

[27] L. Lamport. *Time, clocks and the ordering of events in a distributed system*. Communications of the ACM, 21(7), 1979.

[28] L. Lamport. *On Interprocess Communication (parts I and II)*. Distributed Computing, 1, 1986.

[29] L. Lamport. *The Part Time Parliament*. ACM Transactions on Computer Systems, 16 (2), 1998.

[30] W-K. Lo and V. Hadzilacos. *Using Failure Detectors to Solve Consensus in Asynchronous Shared-Memory Systems*. Proceedings of the International Workshop on Distributed Algorithms, Springer Verlag (LNCS 857), 1994.

[31] N. Lynch. *Distributed Algorithms.* Morgan Kaufmann, 1996.

[32] M. C. Loui and H. Abu-Amara. *Memory requirements for agreement among unreliable asynchronous processes.* Advances in Computing Research, 1987.

[33] A. Mostéfaoui and M. Raynal. *Solving Consensus Using Chandra-Toueg's Unreliable Failure detectors: A General Quorum Based Approach*. Proceedings of the 13th International Conference, DISC 1999. Springer Verlag (LNCS 1693), 1999.

[34] A. Mostéfaoui and M. Raynal. *k-Set Agreement with Limited Accuracy Failure Detectors*. Proceedings of the ACM Symposium on Principles of Distributed Computing, 2000.

[35] G. Neiger. *Failure Detectors and the Wait-Free Hierarchy*. Proceedings of the ACM Symposium on Principles of Distributed Computing, 1995.

[36] G. Neiger and S. Toueg. *Simulating Synchronized Clocks and Common Knowledge in Distributed Systems.* Journal of the ACM, 40(2), April 1993.

[37] F. Schneider. *Replication Management using the State Machine Approach.* Chapter in Distributed Systems, Addison-Wesley, 1993.

[38] P. Vitanyi and B. Awerbuch. *Atomic shared register access by asynchronous hardware.* Proceedings of the IEEE Symposium om Foundations of Computer Science, 1986.

# 11 Appendix

## 11.1 Comparison between failure detectors classes

In this section we prove some reductions between failure detector classes. These reductions are summarized in Figure 5.

Remember first that if $\mathcal{X}$ and $\mathcal{Y}$ are failure detectors, $\mathcal{X} \preceq \mathcal{Y}$ means that there is a reduction algorithm from $\mathcal{Y}$ to $\mathcal{X}$. If $\mathcal{C}_1$ and $\mathcal{C}_2$ are failure detector classes then $\mathcal{C}_1 \preceq \mathcal{C}_2$ means that for every $\mathcal{X} \in \mathcal{C}_1$, there exists $\mathcal{Y} \in \mathcal{C}_2$ such that $\mathcal{X} \preceq \mathcal{Y}$.

**Proposition 11.1** *$\Sigma$ can be implemented in any environment with a majority of correct processes.*

**Proof.** The algorithm of Figure 4 emulates, within variable *Output* the behavior of a failure detector of $\Sigma$.

By an easy induction, as the number of faulty processes is strictly less than $n/2$, no correct process waits forever in line 6 and therefore $r_p$ is unbounded for any such process. Consider failure pattern $F$, we prove:

1. **Completeness:** In $F$, let $p$ be a correct process and $q$ be a faulty process. As $q$ is faulty, there exists a time after which $q$ is crashed. Therefore $q$ sends a finite number of $(I\_AM\_ALIVE, *)$ messages. Let $x$ be such that $(I\_AM\_ALIVE, x)$ is the last message $(I\_AM\_ALIVE, *)$ sent by $q$. Let $\tau$ be the time for which variable $r_p$ is greater than $x$. After time $\tau$, $q$ does not belong to $Output_p$.

2. **Intersection:** As $t$ is less than $n/2$, for all $p$, $Output_p$ contains at least $\lceil (n+1)/2 \rceil$ processes. As a consequence, there is always at least one process that is both in $Output_p$ and in $Output_q$.

■

**Proposition 11.2** *For $\mathcal{E}_{n-1}$ and $n = 2$, $\Sigma$ is equivalent to $\mathcal{S}$.*

**Proof.** Denote by $p_1$ and $p_2$ the two processes in the system. Consider any run $R$ of this system (equipped with a failure detector of $\Sigma$). If no process crashes in $R$, then by the *intersection* property of $\Sigma$ one correct process is trusted forever by $p_1$ and $p_2$. If some process, say $p_1$ crashes, then by the *completeness* property of $\Sigma$, after some time $\tau$, $p_2$ is the only process trusted by $p_2$. By *intersection*

```
Every process p executes the following code:
  Initialization:
    r:=0
  Task 1:
    repeat forever
      send(ARE_YOU_ALIVE, r) to all
      wait until receive (I_AM_ALIVE, r) from n − t processes
      Output_p:={q | a message (I_AM_ALIVE, r) from q received by p }
      /* Output_p is the output for p of the failure detector A */
      r:=r+1
  Task 2:
    upon receive (ARE_YOU_ALIVE, x) from q
    send(I_AM_ALIVE, x) to q
```

Figure 4: Implementation of $\mathcal{A}$ in environment $\mathcal{E} \subseteq \mathcal{E}_t$

property of $\Sigma$ $p_2$ has been trusted forever by $p_1$. Therefore, in all cases, at least one correct is never suspected. This proves the *accuracy* property of $\mathcal{S}$.

■

Figure 5 summarizes our reductibility results.

**Proposition 11.3** *In every environment:*

*(1)* $\mathcal{S} \preceq \mathcal{P}$

*(2)* $\Sigma \times \Omega \preceq \mathcal{S}$

*(3)* $\Sigma \preceq \Sigma \times \Omega$

*(4)* $\Omega \preceq \Sigma \times \Omega$

*(5)* $\Theta \preceq \Sigma$

*In environments* $\mathcal{E}_t$ *such that* $0 < t < n/2$*:*

*(6)* $\Sigma \preceq \Omega$

*(7)* $\Sigma \preceq \Theta$

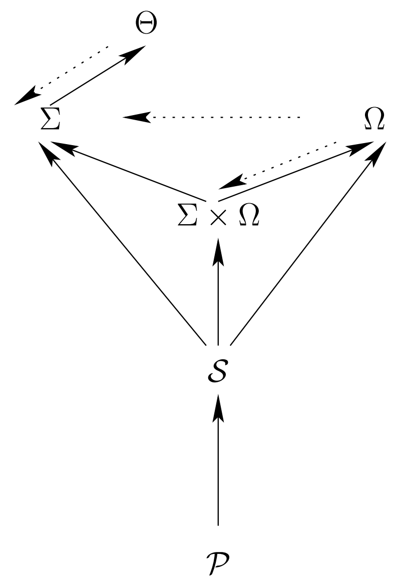
*(8)* $\Sigma \times \Omega \preceq \Omega$

*For* $n = 2$ *and* $0 < t < 2$*:*

*(9)* $\mathcal{S} \preceq \Sigma$ *and* $\Sigma \preceq \mathcal{S}$

**Proof.**

(1) follows directly from the definitions




is stronger in environments such that $t > n/2$

is stronger in all environments

Figure 5: Reductions between classes

(2) follows from the fact that $\Sigma \times \Omega$ is the weakest failure detector to implement consensus in every environment and $\mathcal{S}$ implements consensus in every environment

(3) and (4) are trivial

(5) follows from the *completeness* property of $\Sigma$; after some time, the output of a failure detector in $\Sigma$ contains only correct processes, by the *intersection* property, every output of $\Sigma$ contains at least one correct process.

(6), (7) and (8) follow directly from Proposition 11.1

(9) follows directly from Proposition 11.2

■

## 11.2 Comparison between realistic failure detectors

Remark first that, clearly, the behaviour of an algorithm is in some sense realistic. More precisely, let $\mathcal{A}$ be an algorithm using a realistic failure detector

$\mathcal{R}$. Consider $F$ and $F'$ two failure patterns such that $F$ and $F'$ are identical up to time $\tau$, and a run $R = \langle F, H, I, S, T\rangle$ of $\mathcal{A}$ for $F$ with $H \in \mathcal{R}(F)$. As $\mathcal{R}$ is realistic, there is some failure detector history $H' \in \mathcal{R}(F')$ identical to $H$ up to time $\tau$. Hence, there is a schedule $S'$ identical to $S$ up to time $\tau$ such that $R' = \langle F', H', I, S', T\rangle$ is a run of $\mathcal{A}$. Therefore, there is a run $R'$ identical to $R$ up to time $\tau$ for the failure pattern $F'$. Applying this to reduction algorithms we get:

**Proposition 11.4** *Let $\mathcal{X}$ be a realistic failure detector and $\mathcal{A}$ a reduction algorithm, then the image of $\mathcal{X}$ is realistic.*

Then:

**Corollary 11.5** *Let $\mathcal{A}$ and $\mathcal{B}$ be any two classes of failure detectors, if $\mathcal{A} \preceq \mathcal{B}$ then $\mathcal{A}^r \preceq \mathcal{B}^r$.*

Therefore all the relations between failure detectors of Figure 5 hold also for the realistic part of these classes.

Figure 6 summarizes our reducibility results in the realistic case.

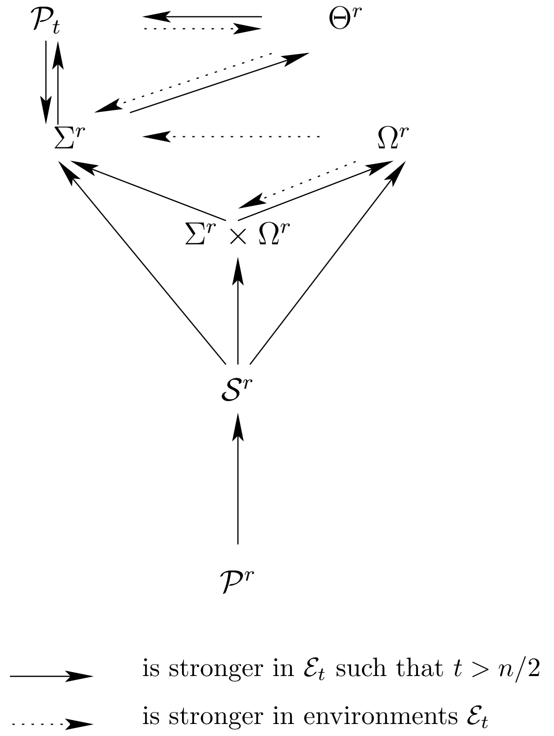



Figure 6: Reductions between classes: the realistic case

We prove here that in every environment $\mathcal{E}_t$, $\mathcal{P}_t$ can be reduced to $\Sigma$.[10]

---

1 task 1:
2 **for** $r = 1, \ldots$ **do**
3 $X1 = \Pi - P^t$
4 **send** $(PING, r, p)$ **to all**
5 **wait until** receiving $(PONG, r, q)$ **from** $n - t$ processes
6 $X2 := \{q \in \Pi | (PONG, r, q)$ received by $p\}$
7 $Output = X_1 \cup X_2$ /* *failure detector output* */

8 task 2:
9 **upon receive**$(PING, r, q)$**send** $(PONG, r, p)$ **to** $q$

task 1 || task 2

---

Figure 7: From $\mathcal{P}^t$ to $\Sigma$

**Proof.**

Recall the definition of $\mathcal{P}^t$: $\mathcal{P}^t$ gives a list of suspected processes that (1) ensures the completeness and (2) at each time $\tau$, $|\mathcal{P}^t(p, \tau) \setminus F(\tau)| \leq \max(n - t - 1, 0)$.

As, there is at most $t$ dead processes, line 5 does not block, and eventually $X2$ contains only correct processes. From this and the completeness of $\mathcal{P}^t$ we deduce the completeness of the emulated failure detector.

Consider now the intersection property. Let $p_1 \in \Pi$ and $p_2 \in \Pi$ (possibly $p_1 = p_2$) and let $r_1$-th (resp. the $r_2$-th) be the iteration of task 1 of $p_1$ (resp. $p_2$). In the following we only consider the $r_1$-th iteration for $p_1$ and the $r_2$-th iteration of $r_2$, hence $V_{p_i}$ where $i = 1$ or $i = 2$ will denote the value of variable $V$ of $p_i$ at the end of the $r_i$-th iteration. For example, $Output_{p_2}$ is the output of the emulated failure detector for $p_2$ at the end of the $r_2$-th iteration of task 1.

Let $\tau_1$ (resp. $\tau_2$) the time at which $p_1$ (resp. $p_2$) sets variable $X1$ in the $r_1$-th (resp. $r_2$-th) iteration with the output of its failure detector $\mathcal{P}^t$. Assume without loss of generality that $\tau_1 \leq \tau_2$.

In the following $+$ denotes the disjoint union.

Given any time $\tau$, let $A(\tau)$ denote the set of process alive at time $\tau$: $A(\tau) = \Pi - F(\tau)$.

As $\tau_1 \leq \tau_2$ we have:

$$A(\tau_2) \subseteq A(\tau_1) \tag{1}$$

As $X2_{p_2}$ contains only identities of processes that have answered to the $PING$ message from $p_2$ in the $r_2$-th iteration, we have: $X2_{p_2} \subseteq A(\tau_2)$. From (1), we get:

$$X2_{p_2} \subseteq A(\tau_2) \subseteq A(\tau_1) \tag{2}$$

[10] This was shown indirectly in [10], where we proved that $\mathcal{P}_t$ is the weakest failure detector class to implement a register in the realistic case.

Consider $\mathcal{P}^t(p_1, \tau_1)$ and let $P_A$ be $\mathcal{P}^t(p_1, \tau_1) \cap A(\tau_1)$. By definition of $\mathcal{P}^t$, $|P_A| \leq \max(n-t-1, 0)$. Then every alive process $p \in A(\tau_1)$ belongs to $P_A$ or to $\Pi - \mathcal{P}^t(p_1, \tau_1)$. Moreover, $P_A \cap (\Pi - \mathcal{P}^t(p_1, \tau_1)) = \emptyset$. As $X1_{p_1} = \Pi - \mathcal{P}^t(p_1, \tau_1)$ we get:

$$A(\tau_1) \subseteq X1_{p_1} + P_A$$

From (2):

$$X2_{p_2} \subseteq A(\tau_2) \subseteq A(\tau_1) \subseteq X1_{p_1} + P_A$$

As $|X2_{p_2}| = n-t$ and $|P_A| < \max(n-t, 1)$, there is at least one $q \in X2_{p_2} \cap X1_{p_1}$. This proves $Output_{p_2} \cap Output_{p_1} \neq \emptyset$. ■

Reciprocally:

**Proposition 11.6** *For any environment $\mathcal{E}_t$ $(0 < t < n)$: $\mathcal{P}_t \preceq \Sigma^r$*

**Proof.** In fact, we prove that any realistic failure detector in $\Sigma$ is in $\mathcal{P}_t$.

The *completeness* property is ensured for all failure detectors in $\Sigma$.

Let $\mathcal{X}$ be a realistic failure detector in $\Sigma$. Assume by contradiction that the properties of $\mathcal{P}_t$ are not ensured by $\mathcal{X}$. Let $F$ be a failure pattern in $\mathcal{E}_t$ and $H$ a failure detector history $H \in \mathcal{X}(F)$, such that at some time $\tau$, strictly more than $n - t - 1$ processes in the set of alive processes $V$ at this time $\tau$ are suspected. This means that a set $E$ of at least $n-t$ alive processes are suspected. Consider $F'$ identical to $F$, up to time $\tau$, and then all processes but processes in $E$ crash. As $\mathcal{X}$ is realistic, there is a failure detector history for $F'$ that is identical up to time $\tau$ to $F$. From the *completeness* property of $\Sigma$, after some time $\tau'$, only processes in $E$ are not suspected for $F'$. But $V \cap E = \emptyset$, contradicting the *intersection* property. ■

Then we get:

**Proposition 11.7** *For any environment $\mathcal{E}_t$, $\mathcal{P}_t$ and $\Sigma^r$ are equivalent.*

## 11.3 Impossibility results

In this section we give some impossibility results about failure detector reductions. These results prove that some classes are incomparable or that some classes are strictly stronger than others.

Before proving the impossibility results, we first give a technical lemma about the limit of runs. Given two runs $R = \langle F, H, C, S, T \rangle$ and $R' = \langle F', H', C, S', T' \rangle$, we say that $R$ and $R'$ are identical up to $m$ if for all $m' \leq m$ : $H(m') = H'(m')$, $S(m') = S'(m')$ and $T(m') = T'(m')$. By extension, we say that $R$ and $R'$ are identical up to time $\tau$, if $R$ and $R'$ are identical up to $n$ for some $m$ such that $T(m) \geq \tau$. Note that if $R$ and $R'$ are identical up to time $\tau$, then up to time $\tau$, the two runs are indistinguishable for all processes that are alive.

**Lemma 11.8** *Let $\mathcal{A}$ be any algorithm using some failure detector $\mathcal{X}$. Let $R = \langle F, H, C, S, T \rangle$ be any run such that:*

1. *for all $m \in \mathcal{N}$ there is $m' > m$ and a run $R_{m'}$ of algorithm $\mathcal{A}$ for $\mathcal{X}$ identical to $R$ up to $m'$*

2. *in $S$ every correct process takes an infinity of steps*

3. *every message sent in $R$ is received in $R$*

4. $H \in \mathcal{X}(F)$

*Then $R$ is a run of algorithm $\mathcal{A}$ for the failure detector $\mathcal{X}$.*

**Proof.** As $R$ is identical, up to $m$, to some run $R_{m'}$ of algorithm $\mathcal{A}$, for all $m'' < m$, $S(m''+1)$ is applicable to $S[m'']C$, and $S[m'']$ is a step of a process $p \notin F(T[m])$. Then, by an easy induction, $S$ is applicable to $C$ and for all $m$, $S(m)$ is a step of a process $p \notin F(T[m])$. Then condition (2), (3) and (4) of the lemma ensure that $R$ is a run of algorithm $\mathcal{A}$. ■

Concerning the realistic part of $\Sigma$. In the following proof we consider failure detector $\mathcal{Z}$ defined as follows: $\mathcal{Z}$ outputs either a set of $n-1$ processes or the set of alive processes; moreover $\mathcal{Z}$ ensures the *completeness* property of $\Sigma$. More formally, $\mathcal{Z}$ is the failure detector such that for every failure pattern $F$, $H \in \mathcal{Z}(F)$ if and only (i) for all $p$ at each time $\tau$ $H(p,\tau)$ is either a subset of $n-1$ processes or the set of alive processes at time $\tau$ and (ii) there is a time $\tau'$ such that after time $\tau'$, $H(p,\tau')$ is included in the set of processes that are alive at time $\tau'$.

**Lemma 11.9** *If $t < n-1$ and $n > 2$ then $\mathcal{Z} \in \Sigma^r$*

**Proof.** $\mathcal{Z}$ is clearly realistic. As $t < n-1$, in every subset of $n-1$ processes, there is at least one correct process. Let $F$ be any failure pattern such that $H \in \mathcal{Z}(F)$. Consider $H(p,\tau)$ and $H(p',\tau')$, we distinguish the following cases:

- $H(p,\tau)$ and $H(p',\tau')$ are both sets of alive processes, assume without loss of generality that $\tau < \tau'$ then $H(p,\tau) \subseteq H(p',\tau')$.

- $H(p,\tau)$ and $H(p',\tau')$ are subsets of $n-1$ processes, then clearly $H(p,\tau) \cap H(p',\tau') \neq \emptyset$.

- One of them, say $H(p,\tau)$, is a subset of $n-1$ processes and the other one, $H(p',\tau')$ is the set of alive processes at time $\tau'$. As $t < n-1$, then there is at least one correct process in $H(p,\tau)$ and this process is in $H(p',\tau')$, proving that $H(p,\tau) \cap H(p',\tau') \neq \emptyset$.

Hence the *intersection* property is ensured. The *completeness* property is ensured by the definition of $\mathcal{Z}$ itself. ■

**Proposition 11.10** *In every environment $\mathcal{E}_t$, there is no reduction from $\Sigma$ to $\Omega$. Moreover in the realistic case, if $t < n-1$, then there is no reduction in $\mathcal{E}_t$ from $\Sigma^r$ to $\Omega$.*

**Proof.** In the following failure pattern $\emptyset$ denotes the failure free pattern where no process crashes.

Recall that $n > 2$ and that we consider only environments $\mathcal{E}_t$ with $0 < t < n$.

With $\Omega$, at each time $\tau$, each process $p$ trusts only one process that is called the *leader* for $p$ at time $\tau$. The properties of $\Omega$ ensure that for each run there is a time after which every correct process has the same leader and this leader is a correct process.

By contradiction assume that there exists a reduction algorithm $\mathcal{A}$ from $\Sigma$ to $\Omega$.

Let $R^p_\tau = \langle \emptyset, H, C, S, T\rangle$ be a run of $\mathcal{A}$ in which $H$ is a failure detector history of $\Sigma$ such that, after time $\tau$, all processes but $p$ are trusted forever by all other processes, then we have:

**Lemma 11.11** *For every such run $R^p_\tau = \langle \emptyset, H, C, S, T\rangle$ of $\mathcal{A}$, there exists a run $R'_\tau = \langle \emptyset, H, C, S', T\rangle$ of $\mathcal{A}$ and a time $\tau' > \tau$ such that $p$ is not leader for some correct process at time $\tau'$.*

This lemma means that, in some way, the leader must be chosen among the trusted processes.

**Proof.** (of lemma) Consider a run $R = \langle F, H, C, S, T\rangle$ of $\mathcal{A}$ that is identical to $R^p_\tau$, up to time $\tau$, such that in $F$, all processes but $p$ are correct and $p$ crashes at time $\mu > \tau$. As $R$ is a run of $\mathcal{A}$, there is some time $\tau' > \mu$ such that $p$ is not the leader for some correct process $q$. Let $R' = \langle \emptyset, H, C, S', T\rangle$ be a run identical to $R$ up to time $\mu$ (in $R''$ all messages sent by $p$ are delayed until after time $\tau'$). For $q$, $R$ and $R'$ and indistinguishable until time $\tau'$, and hence at time $\tau'$, $p$ is not leader for at least one correct process. ■

Now consider the sets $E_i = \Pi - \{p_i\}$. All failure detector histories we consider here will output at each time, and for each process, one of the sets $E_i$. Clearly, as $n > 2$, these failure detector histories ensure the *intersection* property of $\Sigma$.

We construct by induction a strictly increasing infinite sequence of times $(\tau_i)_{i \in \mathcal{N}}$ and an infinite sequence of runs $(R_i = \langle \emptyset, H_i, S_i, I, T\rangle)_{i \in \mathcal{N}}$ of $\mathcal{A}$ such that for all $i$:

(Q1) for any $p$, for any time $\tau$: $H_i(p, \tau) = E_j$ for some $j$

(Q2) $p_{1+i \mod n}$ is not leader for some correct process at time $\tau_i \in R_i$

(Q3) if $i > 0$ then $R_i$ and $R_{i-1}$ are identical up to time $\tau_{i-1}$

(Q4) if $i > 0$ then every message sent by time $\tau_{i-1}$ in $R_i$ is received by time $\tau_i$ and, between time $\tau_{i-1}$ and time $\tau_i$, every process makes at least one step

- (basis: $i = 0$) Let $R'_0 = \langle \emptyset, H_0, C, S'_0, T\rangle$ be any run of $\mathcal{A}$ such that for all $\tau$ and all $p$ $H_0(p, \tau) = E_0$.

  By the previous lemma, there is a run $R_0 = \langle \emptyset, H, C, S_0, T\rangle$ of $\mathcal{A}$ such that at some time $\tau_0$, $p_1$ is not leader for some correct process.

  (Q1) and (Q2) are ensured by construction, (Q2) and (Q4) are trivially verified.

- (induction: $i > 0$) Assume we have $R_i$ and $\tau_i$ verifying (Q1), (Q2), (Q3) and (Q4).

  Consider $R_i = \langle \emptyset, H_i, C, S_i, T\rangle$, there is a time $\tau' > \tau_i$ such that by time $\tau'$ every message sent before $\tau_i$ is received and each process takes at least one step between $\tau_i$ and $\tau'$.

  Let $R' = \langle \emptyset, H_{i+1}, C, S', T\rangle$ be a run of $\mathcal{A}$ identical to $R_i$ up to time $\tau'$, but after time $\tau'$, $H_{i+1}$ outputs always $E_{1+(i+1) \mod n}$. By the previous Lemma, there is a run $R_{i+1} = \langle, \emptyset, H_{i+1}, C, S_{i+1}, T\rangle$ identical to $R'$ up to time $\tau'$ such that, at time $\tau_{i+1} > \tau'$, $p_{i+1 \mod n}$ is suspected by some correct process.

  clearly (Q1), (Q2), (Q3) and (Q4) hold for $i$.

Let $R_{lim} = \langle \emptyset, H_{lim}, C, S_{lim}, T\rangle$ be the limit of the runs $R_i$ that is for all $i$, $R_{lim}$ and $R_i$ are identical up to time $\tau_i$.

As by (Q1) the output of failure detectors in $R_{lim}$ is always one of the $E_i$, $H_{lim} \in \Sigma(\emptyset)$. Then from (Q3), (Q4) and Lemma 11.8, $R_{lim}$ is a run of $\mathcal{A}$.

By (Q2) every process in $\Pi$ is infinitely often not a leader for at least one correct process, contradicting the property of $\Omega$.

Consider the realistic case: every failure detector history considered so far in the proof are in $\mathcal{Z}$. Hence, from Lemma 11.9, if $t < n-1$, there is no reduction from $\Sigma^r$ to $\Omega$. ■

**Proposition 11.12** *In every environment $\mathcal{E}_t$ ($0 < t < n$ and $n > 2$) $\Sigma \times \Omega$ is strictly weaker than $\mathcal{S}$. Moreover, if $t < n-1$, $\Sigma^r \times \Omega^r$ is strictly weaker than $\mathcal{S}^r$.*

**Proof.** As in any environment, $\Sigma \times \Omega$ is the weaker failure detector class to solve consensus and, in any environment, consensus can be solved with $\mathcal{S}$, then $\Sigma \times \Omega \preceq \mathcal{S}$ and by Corollary 11.5 $\Sigma^r \times \Omega^r \preceq \mathcal{S}$.

Now, we prove that there is no reduction from $\Sigma \times \Omega$ to $\mathcal{S}$. For this, consider any environment $\mathcal{E}_t$ with $0 < t < n$ and $n > 2$. In such an environment, for each run at least one process is correct and we have at least three processes.

By contradiction, assume that there exists a reduction algorithm $\mathcal{A}$ such that for each failure pattern $F \in \mathcal{E}_t$ and failure detector history $H \in \Sigma \times \Omega(F)$, outputs a failure detector history $H_{\mathcal{S}} \in \mathcal{S}(F)$.

In the following, if $R = \langle F, (H_1, H_2), C, S, T\rangle$ is a run of a reduction algorithm $\mathcal{A}$, the failure detector history in $\mathcal{S}(F)$ output by this run is denoted by $\mathcal{A}(R)$

Consider the following finite sequence of runs $R_i$ ($0 \le i \le n$) and sequence of times $\tau_i$ ($0 \le i \le n$) such that for all $i < n$:

$$\text{if } i > 0 \text{ then } R_i \text{ identical up to time } \tau_{i-1} \text{ to } R_{i-1} \tag{3}$$

$$p_i \text{ is suspected by all alive processes at time } \tau_i \text{ in } \mathcal{A}(R_i) \tag{4}$$

- (basis) In failure pattern $F_0$, $p_0$ crashes at time 0 and no other process crashes; in run $R_0 = \langle F_0, (H_0^0, H^1), C, S_0, T\rangle$, for $H_0^0$ every alive process trusts all processes in $\Pi$ but $p_0$.

  By the *completeness* property of $\mathcal{S}$, there is a time $\tau_0$ after which, in $\mathcal{A}(R_0)$, $p_0$ is suspected by all processes.

  (3) is trivially verified and (4) is true by construction.

- (induction step) $(0 \leq i < n)$

  Assume that we have built $R_i = \langle F_i, (H_i^0, H_i^1), C, S_i, T\rangle$ and $\tau_i$:

  Then $R_{i+1} = \langle F_{i+1}, C, (H_{i+1}^0, H_{i+1}^1), S_{i+1}, T\rangle$ a run of $\mathcal{A}$ such (1) that $R_i$ and $R_{i+1}$ are identical up to time $\tau_i$ and (2) at time $\tau' > \tau_i$ $p_{i+1}$ crashes and no other process crashes and (3) after time $\tau'$ in $H_{i+1}^0$ every process trusts forever every process in $\Pi - \{p_{i+1}\}$ and (4) $H_{i+1}^1 \in \Omega(F_{i+1})$.

  Remark that in $R_{i+1}$ only process $p_{i+1}$ is not correct.

  By an easy induction, we deduce that at each time $\tau$ for $H_{i+1}^0$ every process $p$ trusts $H_{i+1}^0(p,\tau)$ a subset of $n-1$ processes. As $n > 2$, then for all $\tau, \tau'$ and all $p, p'$ $H_{i+1}^0(p,\tau) \cap H_{i+1}^0(p',\tau') \neq \emptyset$ proving that $H_{i+1}^0 \in \Sigma \times \Omega$ and hence $R_{i+1}$ is a run of $\mathcal{A}$.

  By construction (3) and (4) are true.

Up to time $\tau_i$, $R_{i+1}$ and $R_i$ are indistinguishable for all processes but $p_{i+1}$. Then, $p_i$ being suspected by every other process at time $\tau_i$ in $\mathcal{A}(R_i)$ is also suspected at time $\tau_i$ in $\mathcal{A}(R_{i+1})$. This proves (4). By the *completeness* of $\mathcal{S}$ there is a time $\tau_{i+1} > \tau'$ after which $p_{i+1}$ is suspected by all alive processes.

Consider $R_n = \langle F_n, (H_n^0, H_n^1), C, S_n, T\rangle$, $R_n$ is a run of $\mathcal{A}$. By an easy induction and (3), for all $0 \leq i < n$, $R_{n-1}$ and $R_i$ are identical up to time $\tau_i$. Hence from (4) in $\mathcal{A}(R_{n-1})$, each process $p_i$ in $\Pi$ is suspected at time $\tau_i$, contradicting the *accuracy* property of $\mathcal{S}$.

Concerning the realistic case, remark that all failure detectors histories $H_i^0$ are in $\mathcal{Z}$, then there is no reduction from $\mathcal{Z} \times \Omega$ and therefore no reduction from the $\Sigma^r \times \Omega^r$ to $\mathcal{S}$. ∎

**Proposition 11.13** *There is no reduction:*

*(1) From $\mathcal{S}$ to $\mathcal{P}$ in any environment $\mathcal{E}_t$ such that $0 < t < n$*

*(2) From $\Omega$ to $\mathcal{S}$ in any environment $\mathcal{E}_t$ such that $0 < t < n$*

*(3) From $\Sigma$ to $\Omega$ in any environment $\mathcal{E}_t$ such that $0 < t < n$ and $n > 2$*

*(4) From $\Sigma$ to $\mathcal{S}$ in any environment $\mathcal{E}_t$ such that $0 < t < n$ and $n > 2$*

*(5) From $\Omega$ to $\Sigma$ in any environment $\mathcal{E}_t$ such that $n/2 \leq t < n$*

*(6) From $\Sigma \times \Omega$ to $\mathcal{S}$ in any environment $\mathcal{E}_t$ such that $0 < t < n$ and $n > 2$*

*(7) From $\Omega$ to $\Sigma \times \Omega$ in any environment $\mathcal{E}_t$ such that $n/2 \leq t < n$*

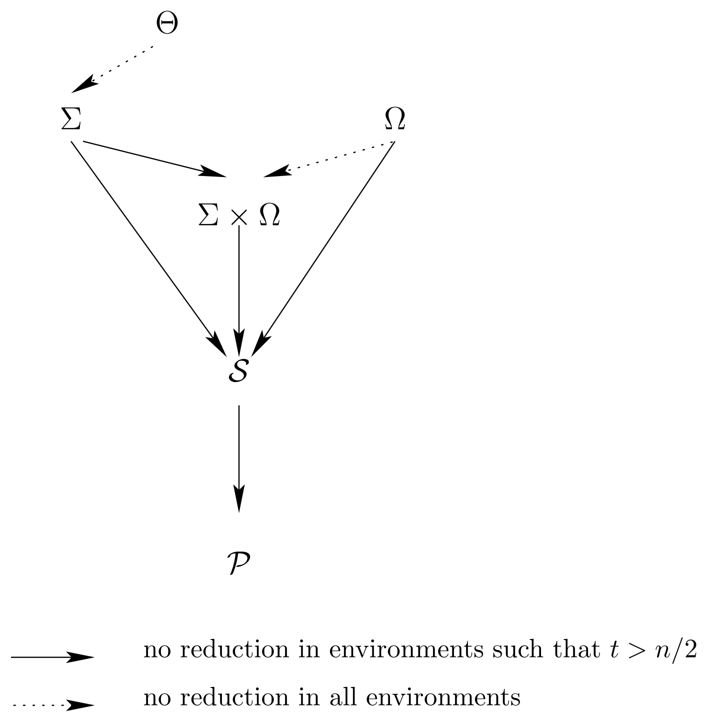



Figure 8: No reduction between classes

*(8) From* $\Sigma$ *to* $\Sigma \times \Omega$ *in any environment* $\mathcal{E}_t$ *for* $0 < t < n$ *and* $n > 2$

*(9) From* $\Theta$ *to* $\Sigma$ *in any environment* $\mathcal{E}_t$ *such that* $n/2 \leq t < n$

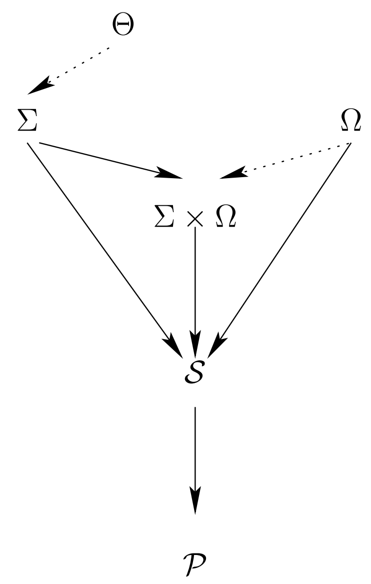
**Proof.**

- (1) and (2) from [6] and [7]
- (3) from Proposition 11.10
- (4) As $\Omega \preceq \mathcal{S}$, if $\mathcal{S} \preceq \Sigma$, we would have $\Omega \preceq \Sigma$ contradicting (3)
- (5) Recall that in environment $\mathcal{E}_t$ with $n/2 \leq t < n$, $\Omega$ is not sufficient to implement Consensus [6].

  If $\Omega$ were reducible to $\Sigma$ in an environment $\mathcal{E}_t$ ($n/2 \leq t < n$) then $\Omega$ could be reducible to $\Sigma \times \Omega$, and then Consensus could be solvable with $\Omega$ in such an environment –a contradiction.
- (6) From Proposition 11.12
- (7) If there were a reduction from $\Omega$ to $\Sigma \times \Omega$ in environments $\mathcal{E}_t$ such that $n/2 \leq t < n$, we would get a reduction from $\Omega$ to $\Sigma$ contradicting (5)

- (8) If there were a reduction from $\Sigma$ to $\Sigma \times \Omega$ in environments $\mathcal{E}_t$ for $0 < t < n$, then we would have a reduction from $\Sigma$ to $\Omega$ in such environments contradicting (6)
- (9) Lemma 8.2

■




$\mathcal{A} \longrightarrow \mathcal{B}$
no reduction from $\mathcal{A}$ to $\mathcal{B}$ in environments such that $t < n - 1$

$\mathcal{A} \dashrightarrow \mathcal{B}$
no reduction from $\mathcal{A}$ to $\mathcal{B}$ in envrironemts such that $t < n/2$

Figure 9: No reduction between classes: the realistic case

### 11.4 The wait free case

**Proposition 11.14** *In any environment $\mathcal{E}_t$ with $t = n - 1$: $\Theta^r$, $\Sigma^r$, $\mathcal{S}^r$ and $\mathcal{P}^r$ are all equivalent.*

**Proof.** As by Corollary 11.5 and Proposition 11.3 $\Theta^r \preceq \Sigma^r \preceq \mathcal{S}^r \preceq \mathcal{P}^r$, we only prove that: $\mathcal{P}^r \preceq \Theta^r$.

For this, in fact, we prove that, in $\mathcal{E}_{n-1}$, every failure detector $\mathcal{X}$ in $\Theta^r$ is in $\mathcal{P}^r$:

- The *completeness* property is ensured by definition,

- Assume by contradiction that the *accuracy* property is not ensured, then there is a failure pattern $F$ and a failure detector history $H$ of $\mathcal{X}(F)$ such that at some time $\tau$, a process that is alive, say $p$ is suspected by some process $q$. Then consider the failure pattern $F'$ identical to $F$ up to time $\tau$, but after time $\tau$ all processes but $p$ crash. As $\mathcal{X}$ is realistic, there is a failure detector history $H'$ of $\mathcal{X}(F')$ such that $H$ and $H'$ are identical up to time $\tau$. But, then at time $\tau$, $p$, the only correct process of $F'$ is suspected by $q$ –a contradiction.

$\blacksquare$