Tight Failure Detection Bounds on Atomic Object Implementations[1]

CAROLE DELPORTE-GALLET
ESIEE-IGM Marne-La-Vallee, France
and
HUGUES FAUCONNIER
LIAFA Univ Paris VII, France
and
RACHID GUERRAOUI
EPFL Lausanne, Switzerland

This paper determines the weakest failure detectors to implement shared atomic objects in a distributed system with crash-prone processes. We first determine the weakest failure detector for the basic register object. Then we use that to determine the weakest failure detector for all popular atomic objects including test-and-set, fetch-and-add, queue, consensus and compare-and-swap, which we show is the same.

Categories and Subject Descriptors: C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design—*distributed networks*; C.2.4 [**Computer-Communication Networks**]: Distributed Systems; D.4.1 [**Operating Systems**]: Process Management—*concurrency, multiprocessing/multiprogramming, synchronization*; F.1.1 [**Computation by Abstract Devises**]: Models of Computation—*relations among models*

General Terms: Algorithms, Theory, Reliability

Additional Key Words and Phrases: atomic objects, failure detection

1. INTRODUCTION

1.1 Motivation

A shared atomic object is a data structure exporting a set of operations that can be invoked by concurrent processes. *Atomicity* means that any object operation appears to execute at some individual instant between its invocation and reply time events [Lamport 1986; Herlihy and Wing 1990]. Thanks to atomicity, the type of an object can solely be defined according to its *sequential specification*: the set of all possible sequential executions of the object operations [Herlihy and Wing 1990].

Many distributed algorithms are designed assuming, as underlying synchronization primitives, atomic objects, sometimes provided as hardware devices of a multiprocessor. These include objects of types register, test-and-set, fetch-and-add, queue, and compare-and-swap.

This paper studies necessary and sufficient conditions for implementing atomic object types in a distributed system where crash-prone processes communicate by exchanging messages: no physical shared memory is assumed. Through such im-

[1]Results of this paper appeared for the first time in the following papers by the same authors: *Shared Memory vs Message Passing* (EPFL Tech Report 03) and *(Almost) All Objects are Universal in a Message Passing System* (DISC'05).

plementations, algorithms based on shared atomic objects provided in hardware, can be automatically emulated in a message passing system.

We study *robust* [Attiya et al. 1995]implementations where any process that invokes an object operation and does not crash eventually gets a reply.

1.2 The synchronisation power of atomic objects

Some atomic object types have been shown to have more *synchronization power* than others in the sense that they can solve the seminal *consensus* problem among more processes [Fischer et al. 1985; Herlihy 1991]. What is meant here by a *type solving consensus* is that instances of that type can be used in a distributed algorithm to have processes agree on one among a set of proposed values. The *consensus* problem can be casted as an atomic object type also called consensus. We talk about a type solving the consensus problem or a type emulating the consensus object.

The ability for a type to emulate consensus among a certain number k of processes is important as it implies the ability to emulate any other type in a system of k processes, irrespective of how many of these processes may crash [Herlihy 1991].

The type register is in this sense weak as it can only emulate consensus for exactly 1 process [Loui and Abu-Amara 1987]. Test-and-set, fetch-and-add, or queue can emulate consensus among exactly 2 processes [Herlihy 1991]. Interestingly, for any number k, there is a type that can emulate consensus among exactly k processes [Herlihy 1991]. This leads to a hierarchy of types, called the *consensus hierarchy*, classifying types according to their *consensus number (k)*. Types such as compare-and-swap can emulate consensus among any number of processes. These types are said to be *universal* [Herlihy 1991]: their consensus number is ∞ and they are at the top of the hierarchy.

1.3 Atomic object implementations

We consider in this paper a message passing system where no physical shared memory is available. The processes are assumed to communicate through reliable channels but can fail by crashing. Assumptions about crashes are encapsulated within the notion of environment. In particular, the *majority* environment captures the assumption that a majority of the processes is correct.

If the message passing system is asynchronous (no assumption is made on process relative speeds and communication delays), then two fundamental results are known about atomic object implementations. (1) The type register can be implemented iff a majority of the processes do not crash [Attiya et al. 1995], and (2) we cannot implement any of the types with consensus number $k > 1$ (e.g., test-and-set, fetch-and-add, queue, compare-and-swap, and consensus) even if only one process may crash.

In most distributed systems however, certain synchrony assumptions [Dwork et al. 1988] can be made. Furthermore, we can be precisely express such assumptions through axiomatic properties of a *failure detector* abstraction [Chandra and Toueg 1996]: a distributed oracle that provides processes with hints about crashes, and which can itself be implemented based on synchrony assumptions, e.g., timeouts. Failure detectors can be precisely compared and a failure detector $\mathcal{D}$ is said to be *stronger* than a failure detector $\mathcal{D}'$ if there is a distributed algorithm that uses $\mathcal{D}$ to emulate $\mathcal{D}'$ ($\mathcal{D}'$ is said to be *weaker* than $\mathcal{D}$). Two failure detectors are said to be *equivalent* if each is stronger than the other.

In particular, the *weakest* failure detector to implement consensus in the majority environment has been shown to be an oracle which outputs, at any time and at every process, a single leader process such that, eventually, this leader is correct and is permanently the same at all processes [Chandra et al. 1996]. Interestingly, this oracle can be implemented using very weak synchrony assumptions [Aguilera et al. 2003; 2004; Mostéfaoui et al. 2003] that need only hold eventually.

This oracle, denoted by Ω, is the weakest to implement consensus (in the majority environment) [Chandra et al. 1996] in the sense that (a) there is a distributed algorithm that implements consensus using Ω (in the majority environment), and (b) for every failure detector $\mathcal{D}$ such that some algorithm implements consensus using $\mathcal{D}$, there is an algorithm that implements Ω using $\mathcal{D}$. Failure detector $\mathcal{D}$ encapsulates information about failures, and in some sense synchrony assumptions, that are at least as strong as those encapsulated by Ω.

The fact that Ω was established as the weakest failure detector to implement consensus in the majority environment directly implies that it is also the weakest to implement the compare-and-swap type in this environment. This follows from the very fact that, in any environment, the compare-and-swap type can directly emulate the consensus type, and the consensus type can emulate any atomic object type [Herlihy 1991].

—But what about other environments? As we pointed out, the basic type register cannot be implemented if the system is asynchronous and half of the processes might crash: it is thus natural to seek for the weakest failure detector to implement the basic register type in such environment? Similarly, in any environment where half of the processes might crash, Ω does clearly not implement consensus (or compare-and-swap). What is the weakest failure detector to implement types like consensus (or compare-and-swap) in such environment?

—What about all other types like queue, test-and-set, or fetch-and-add? Any of these can emulate consensus in a subsystem of two processes (unlike register) but cannot in any subsystem of more than two processes (unlike compare-and-swap) [Herlihy 1991]. At first glance, one would expect for instance that the weakest failure detector to implement a queue be stronger than the weakest to implement a register but weaker than the weakest to implement consensus.

1.4 Contributions

This paper addresses determines the weakest failure detectors to implement popular atomic object types like register, compare-and-swap, queue, test-and-set, and fetch-and-add in *any* environment. Furthermore, we do so among *any* subset of processes in the system. That is, we also consider the restrictions of the types to variants that are invokable only by a subset of the processes.

We proceed as follows:

(1) We first determine, for *any* environment, and for *any* subset S of processes in the system, (1) the weakest failure detector to implement a register shared by processes in S, and we derive from it (2) the weakest failure detector to implement a consensus shared by processes in S.

—The first failure detector, denoted by Σ_S, outputs, at any time and at every process of S, a set of processes such that (a) any two sets always intersect and (b) eventually every set contains only correct processes.
—The second failure detector, which we denote by $\Sigma_S * \Omega_S$, outputs, at any time and at every process of S, both outputs of failure detector Σ_S and a failure detector, which we introduce here and we denote by Ω_S. Failure detector Ω_S outputs, at any time and at every process of S, a single leader process, such that, eventually this leader is the same at all processes of S and is correct.

(2) We then show that, in any environment, and for any integer $1 < k \leq n$, the weakest failure detector to implement any type that emulates consensus among k processes is also $\Sigma * \Omega$. We derive from this, for *any* environment, and for *any* subset S of processes in the system, the weakest failure detector to implement types like test-and-set, fetch-and-add and queue, shared by the processes in S, is $\Sigma_S * \Omega_S$.

1.5 Scope and Interpretation

Our first result above on the weakest failure detectors to implement register and consensus generalizes two fundamental results in distributed computing.

—Given that a shared memory abstraction is usually assumed to be made of registers, the Σ_S failure detector encapsulates the exact information about failures needed to implement this abstraction over a subset S of processes communicating by message passing. This generalizes in a precise sense the result of [Attiya et al. 1995]. In particular, in a majority environment, Σ, the restriction of Σ_S to the case where S is the entire system, can be implemented with no synchrony assumption.
—Identifying failure detector $\Sigma_S * \Omega_S$ generalizes, for any subset of processes and any environment, the result of [Chandra et al. 1996]. Indeed, in a majority environment, $\Sigma * \Omega$, the restriction of $\Sigma_S * \Omega_S$ to the case where S is the entire system, is simply equivalent to Ω [Chandra et al. 1996].

Our second result above (Section 1.4) shows that the information about failures that is necessary and sufficient to implement a type like queue and test-and-set over message passing, is the same as the information that is necessary and sufficient to implement types like compare-and-swap and consensus. This conveys the interesting fact that the notion of consensus number [Herlihy 1991; Jayanti 1993] of a type has little impact on the information about failures needed to implement this type.

More generally, and given that most synchronization problems can be casted as atomic types, our result means that, as long as failure detection is concerned, adopting an ad hoc approach focusing on each problem individually is not better than a general approach where the failure detector $\Sigma * \Omega$ would be implemented as a common service underlying all problems, i.e., all type implementations.

1.6 Road-map

The rest of the paper is organized as follows. Section 2 defines our model. Section 3 introduces failure detectors Σ_S and Ω_S and establishes some preliminary results about the characteristics of these failure detectors. Section 4 determines the weakest failure detector to implement a register. Section 5 determines the weakest failure detector to implement atomic types with a consensus number $k > 1$. Section 6 concludes the paper by discussing our results in the shared memory model. In the appendix, we prove relations between the failure detectors introduced in this paper and those introduced in the literature.

2. SYSTEM MODEL

Stating and proving our result goes through defining a general model of distributed computation encompassing different kinds of abstractions: atomic objects, message passing and failure detectors. Our model, and in particular our notion of implementation, is a generalization of both the notions of shared memory object implementations of [Herlihy 1991] as well as failure detector reductions of [Chandra and Toueg 1996].

We consider a distributed system composed of a finite set of n processes $\Pi = \{p_1, p_2, \ldots, p_n\}$; $|\Pi| = n \geq 2$. (Sometimes, processes are denoted by p and q.) A discrete global clock is assumed, and Φ, the range of the clock's ticks, is the set of natural numbers. The global clock is not accessible to the processes.

2.1 Failure patterns and failure histories

Processes can fail by crashing. Failures are permanent, i.e., no process recovers after a crash. A process p is said to be crashed at time τ if p does not perform any action after time τ (the notion of action is defined below). Otherwise the process is said to be alive at time τ. A correct process is a process that does never crash (otherwise it is faulty).

A failure pattern is a function F from Φ to 2^{Π}, where $F(\tau)$ denotes the set of processes that have crashed by time τ. The set of correct processes in a failure pattern F is noted $correct(F)$. As in [Chandra and Toueg 1996], we assume that every failure pattern has at least one correct process. An environment is a set of failure patterns. Unless explicitly stated otherwise, our results are stated for all environments.

Roughly speaking, a failure detector $\mathcal{D}$ is a distributed oracle which gives hints about failure patterns of a given environment $\mathcal{E}$. Each process p has a local failure detector module of $\mathcal{D}$, denoted by $\mathcal{D}_p$. 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 $p \in \Pi$, for every time $\tau \in \Phi$, $H(p, \tau)$ denotes the value of the failure detector module of process p at time τ, i.e., $H(p, \tau)$ denotes the value output by $\mathcal{D}_p$ at time τ.

A failure detector $\mathcal{D}$ is then defined as a function that maps each failure pattern F of $\mathcal{E}$ 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. Let $\mathcal{D}$ and $\mathcal{D}'$ be any two failure detectors, $\mathcal{D} * \mathcal{D}'$ denotes the failure detector, with range $R_{\mathcal{D}} * R_{\mathcal{D}'}$, which associates to every failure pattern F, the set of histories $\mathcal{D} * \mathcal{D}'(F) = \{(H, H') \mid H \in \mathcal{D}(F), H' \in \mathcal{D}'(F)\}$. This notation is naturally extended to a finite set of failure detectors K: $*\{\mathcal{D} \mid \mathcal{D} \in K\}$.

2.2 Actions, runs and schedules

To access its local state or shared services, a process p executes (deterministic) actions from a (possibly infinite) alphabet $\mathcal{A}_p$. Each action is associated with exactly one process and the set of all actions $\mathcal{A}$ is a disjoint union of sets of alphabets, each associated to a given process $\mathcal{A}_{p_i}$ $(1 \leq i \leq n)$. The state of a process after it executes action a in state s, is denoted $a(s)$. A configuration C is a function mapping each process to its local state. When applied to a configuration C, action a of $\mathcal{A}_{p_i}$ gives a new unique configuration denoted $a(C)$: for all $j \neq i$ $(a(C))(p_j) = C(p_j)$ and $(a(C))(p_i) = a(C(p_i))$.

An infinite sequence of actions is called a schedule. In the following, $Sc[i]$ denotes the i-th action of schedule Sc. Given $seq = a_1 \dots a_i a_{i+1}$ a prefix of a schedule and C a configuration, the new configuration $seq(C)$ resulting from the execution seq on some C is defined by induction as $a_{i+1}((a_1 \dots a_i)(C))$. To each schedule $Sc = a_1 \dots a_i a_{i+1} \dots$ and configuration C_0 correspond a unique sequence of configurations $C_0 C_1 \dots C_i C_{i+1} \dots$ such that $C_{i+1} = a_{i+1}(C_i)$.

A run is a tuple $\alpha =< F, C, Sc, T >$, where F is a failure pattern, C a configuration, Sc a schedule, and T a time assignment represented by an infinite sequence of increasing values such that: (1) for all k, if $Sc[k]$ is an action of process p then p is alive at time $T[k]$ $(p \notin F(T[k]))$ and (2) if p is correct then p executes an infinite number of actions. An event e is the occurrence of an action in Sc, and if e is the k-th action in Sc, then $T[k]$ is the time at which event e is executed.

Consider an alphabet of actions $\mathcal{A}$ and any subset $\mathcal{B}$ of $\mathcal{A}$. Let $Sc|\mathcal{B}$ be the subsequence of Sc consisting only of the actions of $\mathcal{B}$, and $T|\mathcal{B}$ be the subsequence of T corresponding to actions of $\mathcal{B}$ in $\alpha =< F, C, Sc, T >$. We call $< F, C, Sc|\mathcal{B}, T|\mathcal{B} >$ the history corresponding to $\mathcal{B}$, and we simply denote it by $\alpha|\mathcal{B}$. In particular, when $\mathcal{B} = \mathcal{A}_{p_i}$, $Sc|\mathcal{A}_{p_i}$, $T|\mathcal{A}_{p_i}$, $\alpha|\mathcal{A}_{p_i}$ are the restrictions to the process p_i; $\alpha|\mathcal{A}_{p_i}$ is called the history of process p_i in α.

In the following, we simply denote the restrictions to the process p_i by $Sc|p_i$, $T|p_i$, and $\alpha|p_i$.

2.3 Services

A service is defined by a pair $(Prim, Spec)$. Each element of $Prim$, denoted by $prim$, is a tuple $< s, p, arg, ret >$ representing an action of process p identified by a sort s, an input argument arg from some (possibly infinite) range In and an output argument (or return value) ret from some (possibly infinite) range Out. An empty argument is denoted by λ. The specification $Spec$ of a service X is defined by a set of runs. In this paper, we consider three kinds of services: *message passing* channels, *atomic objects* and *failure detectors*.

2.3.1 *Message passing.* The classical notion of point-to-point message passing channel, represented here by a service and denoted MP, is defined through primitive $send(m)$ *to* q of process p and primitive $receive()$ *from* q of process p. More formally, these primitives are respectively a tuple $< send\_to\_q, p, m, \lambda >$ with $m \in M$ where M is a set of messages and a tuple $< receive\_from\_q, p, \lambda, x >$ with $x \in M \cup \{\lambda\}$.

Primitive $receive()$ *from* q returns either some message m or the null message λ; in the first case we say that p received m. Each non null message is uniquely identified and has a unique sender as well as a unique potential receiver. The specification *Spec* of MP stipulates that: (1) the receiver of m receives it at most once and only if the sender of m has sent m; (2) if process p is correct and if process q executes an infinite number of *receive() from* p primitives, then all messages sent by p to q are received by q.

2.3.2 *Failure detector.* The only primitive defined for a failure detector service is a query without argument that returns one value in the failure detector range.

A run $\alpha =< F, C, Sc, T >$ satisfies the specification of a failure detector $\mathcal{D}$ if there is a failure detector history $H \in \mathcal{D}(F)$ such that for all k, if $Sc[k]$ is a query of $\mathcal{D}$ by process p that outputs v, then $H(p, T[k]) = v$. Any such history is said to be associated with run α.

2.3.3 *Atomic object.* Atomic objects are specific kinds of services defined by a sequential specification. Such a specification stipulates the values to be returned by the object's operations when invoked by non-concurrent processes. Each occurrence of an operation is realized through two actions: an invocation (i.e. a tuple $< op_{invoke}, p, arg, \lambda >$ where op is the operation and arg the argument of the operation op) and a reply (i.e. a tuple $< op_{reply}, p, \lambda, ret >$ where ret is the value return by the operation op). The sequential specification of an atomic object is defined by its type as well as an initial state of the object.

A type $\mathcal{T}$ is a tuple $< Q, Inv, Rep, L >$: where Q is the set of states of the type, Inv is a set of invocations, Rep is a set of replies, and L is a relation that carries each state st of the object, $st \in Q$ and invocation of Inv to a set of state and reply pairs, which are said to be legal, and denoted by $L(st, op)$. When L is a function, the type is said to be deterministic. An invocation inv and a reply rep are said to be matching if they are actions of the same process p and if there exist states st and st' such that (st', rep) belongs to $L(st, inv)$. A (finite or infinite) sequence $\sigma = (o_0 r_0)(o_1 r_1) \ldots (o_j r_j) \ldots$ where, for all l, o_l and r_l are respectively invocations and replies, is legal from state s if there is a corresponding sequence of states $s = s_0, s_1, \ldots s_j, \ldots$ such that, for each l $(s_{l+1}, r_{l+1}) \in L(s_l, o_l)$. Such a sequence is called a sequential history of object O from initial state s.

Only well-formed schedules are considered. Consider a schedule Sc, and its restriction to a process $Sc|p$. We say that some occurrence of invocation is pending if there is no matching reply. We say that a schedule Sc is well-formed if (1) no prefix of $Sc|p$ has more than one occurrence of a pending invocation and (2) $(Sc|p)|Prim$ begins with an invocation and has alternating matching invocations and replies. By extension, a run $\alpha =< F, C, Sc, T >$ is well-formed if its schedule Sc is well-formed and there is no pending invocation for correct processes in F. When reasoning about the atomicity of an object, we consider only operations that terminate, i.e., both invocation inv and a matching reply have taken place. If a process p performs an invocation inv and then p crashes before getting any reply, we assume that either the state of the object appears as if inv has not taken place, or inv has indeed terminated. An operation is said to precede another if the first terminates before the second starts. Two operations are concurrent if none precedes the other.

Let $\alpha =< F, C, Sc, T >$ be any well-formed run of an algorithm. Remember that C is an initial configuration, and configurations represent the state of the system, including the states of its objects. Let $\alpha|Prim$ be the history corresponding to object $O =< Prim, Spec >$ of type $\mathcal{T}$, a linearization of $\alpha|Prim$ with respect to $\mathcal{T}$ and state s is a pair (H, T') such that: (1) H is a sequential history of O from state s; (2) H includes all non pending invocations of operation in Sc; (3) If some invocation inv is pending in Sc, then either H does not include this pending invocation or includes a matching reply; (4) H includes no action other than the ones mentioned in (2) and (3); (5) T' is an infinite sequence such that if $Sc[k]$ is an invocation and $Sc[k']$ the matching reply, corresponding respectively to $H[l]$ and $H[l+1]$ then $T'[l] = T'[l+1]$ belongs to the interval $(T[k], T[k'])$. A run α is linearizable for type $\mathcal{T}$ and state s if α has a linearization with respect to $\mathcal{T}$ and state s. The specification $Spec$ associated to an object O of type $\mathcal{T}$ and initial state s is the set of runs well-formed for O that are linearizable with respect to $\mathcal{T}$ and state s [Herlihy and Wing 1990].

2.4 Algorithms and implementations

An algorithm $A =< A_1, \cdots, A_n, Serv >$, using a set of services $Serv$, is a collection of n deterministic automata A_i (one per process p_i) with transitions labeled by actions in $\mathcal{A}_i$ such that all operations defined for services in $Serv$ are included in $\mathcal{A}$. Every transition of A_i is a tuple (s, a, s') where s and s' are local states of p_i and a is an action of p_i such that $a(s) = s'$. Computation proceeds in steps of the algorithm: in each step of an algorithm A, a process p atomically executes an action in $\mathcal{A}$. If a is an action of p_i and C is a configuration, a is said to be applicable to C if there is a transition (s, a, s') in A_i such that $s = C(p_i)$. By extension, a schedule $Sc = Sc[1]Sc[2] \ldots Sc[k] \ldots$ is applicable to a configuration C if for each $k > 1$, $Sc[k]$ is applicable to configuration $(Sc[1] \ldots Sc[k-1])(C)$. A run of algorithm A is a run $\alpha =< F, C, Sc, T >$ such that Sc is a schedule applicable to configuration C, such that α satisfies the specifications of services in $Serv$.

Roughly speaking, implementing a service X using a set of services $Serv$ means providing the code of a set of subtasks associated with every process: one subtask for each primitive sort of X as well as a set of additional subtasks. The subtasks associated to the primitives are assumed to be sequential in the following sense: if a process p executes a primitive $prim$ (of the service to be implemented), the process launches the associated subtask and waits for it to terminate and return a reply before executing another primitive. All subtasks use services in $Serv$ to implement service X, in the sense that the only primitives used in these subtasks are primitives defined in $Serv$. More precisely, an implementation of a service $X =< Prim, Spec >$ with primitives of sorts $ps_1, \ldots, ps_m$, using a set of services $Serv$, among n processes, is defined by $I(X, n, Serv) =< (X_1, (ps_1^1, \ldots, ps_m^1)), \ldots, (X_n, (ps_1^1, \ldots, ps_m^n)) >$ where, for each i, X_i is the implementation subtask of p_i and ps_j^i is the primitive implementation subtask associated to process p_i and the primitive of sort ps_j of X such that the only primitives occurring in these subtasks are primitives defined in $Serv$.

An implementation $I(X, n, Serv)$ for environment $\mathcal{E}$ ensures that: for each algorithm $A =< A_1, \cdots, A_n, Serv' \cup \{X\} >$, the corresponding algorithm $A' =< A'_1, \cdots, A'_n, Serv \cup Serv' >$ in which X is implemented by $I(X, n, Serv)$ where, for each i, A'_i is the automaton corresponding to the subtasks A_i, X_i, $ps_1^i, \ldots, ps_j^i$ is such that all runs α of A', restricted to actions of $A_1, \cdots, A_n$, are runs of A.

We study in this paper robust [Attiya et al. 1995] implementations of services:

every correct process that executes a primitive of an implemented service eventually gets a reply from that invocation. We will sometimes focus on implementations of $S-$services: the primitives of such a service can only be invoked by processes of a subset S of the system. In such implementation, the only restriction is the fact that only the processes in S contain each one subtask per primitive sort of the S-service (but all processes contain implementation tasks). If we do not specify the subset S, we implicitly assume the set of all processes.

2.5 Weakest failure detector

The notion of failure detector $\mathcal{D}2$ being reducible to $\mathcal{D}1$ in a given environment $\mathcal{E}$ [Chandra and Toueg 1996], means in our context that there is an implementation of $\mathcal{D}2$ using $\mathcal{D}1$ and MP in $\mathcal{E}$. Failure detector $\mathcal{D}1$ is said to be stronger than $\mathcal{D}2$ in $\mathcal{E}$ and written $\mathcal{D}2 \preceq_{\mathcal{E}} \mathcal{D}1$. All implementation subtasks use only MP and $\mathcal{D}$. We say that $\mathcal{D}1$ is equivalent to $\mathcal{D}2$ in $\mathcal{E}$ ($\mathcal{D}1 \equiv_{\mathcal{E}} \mathcal{D}2$), if $\mathcal{D}2 \preceq_{\mathcal{E}} \mathcal{D}1$ and $\mathcal{D}1 \preceq_{\mathcal{E}} \mathcal{D}2$ in $\mathcal{E}$.

We say that a failure detector $\mathcal{D}_1$ implements a given service (in environment $\mathcal{E}$) if there is an algorithm that uses $\mathcal{D}_1$ to implement that service (in $\mathcal{E}$).

We say that a failure detector $\mathcal{D}_1$ is the weakest to implement a given service in environment $\mathcal{E}$ if and only if the two following conditions are satisfied: (1) there is an implementation of the service using $\mathcal{D}_1$ and MP in $\mathcal{E}$, and (2) if there is an algorithm that implements the service using some failure detector $\mathcal{D}_2$ in $\mathcal{E}$, then $\mathcal{D}_2$ is stronger than $\mathcal{D}_1$ in $\mathcal{E}$.

Implicit assumptions

As pointed out earlier, given that most of our results hold for all environments, unless explicitly stated otherwise, we will implicitly assume all environments when giving our results.

Similarly, as most of our implementations use MP, unless explicitly stated otherwise, we will implicitly assume MP in the services that are used by our implementations. (This will not be the case in particular when considering implementations of atomic objects out of other atomic objects in Section 5.2).

3. THE QUORUM AND LEADER FAILURE DETECTORS

We introduce here two failure detectors: the *Quorum* and the *Leader*. Both are defined relatively to a subset of processes S in the system. The first one is denoted by Σ_S. The second one, denoted by Ω_S, generalizes Ω [Chandra et al. 1996].

We prove some properties of the composition of these failure detectors, which will be useful in proving later results of this paper.

3.1 Failure detector Σ_S

Given any subset S of processes in Π, failure detector Σ_S outputs, at each process in S, and at any time, a list of processes, called *trusted* processes, such that every list intersects with every other list, and eventually, all lists contain only correct processes. It is important to notice that two lists do not need to intersect at a correct process.

More precisely, failure detector Σ_S outputs lists of processes that satisfy the two following properties:

—*Intersection.* Every two lists of trusted processes intersect: $\forall F \in \mathcal{E}, \forall H \in \Sigma_S(F), \forall p, q \in S, \forall \tau, \tau' \in \Phi : H(p, \tau) \cap H(q, \tau') \neq \emptyset$;

—*Completeness.* Eventually, every list of processes trusted by a correct process contains only correct processes: $\forall F \in \mathcal{E}, \forall H \in \Sigma_S(F), \forall p \in S \cap correct(F), \exists \tau \in \Phi, \forall \tau' > \tau \in \Phi : H(p, \tau') \subseteq correct(F)$.

To simplify the definition, we consider that, at any process of S that has crashed, the list that is output is simply Π. It is easy to see that Σ can easily be implemented in an asynchronous message passing system assuming the majority environment (for presentation completeness, we prove this more formally in the appendix).

The following proposition is a direct consequence of the definition of Σ_Π:

PROPOSITION 1. *Let S be any subset of Π and let $\mathcal{L}$ be any family of subsets of S such that, for all $p, q \in S$, there exists $L \in \mathcal{L}$ such that $p \in L$ and $q \in L$. We have: $\Sigma_S \equiv *\{\Sigma_X | X \in \mathcal{L}\}$.*

An interesting particular case is where subsets X are pairs, i.e., for any $S \subseteq \Pi$, $\Sigma_S \equiv *\{\Sigma_{\{p,q\}} | p, q \in S\}$. The composition of all Σ_S, over all subsets S of size 2, is in this case Σ:

COROLLARY 2. *For all $S \subseteq \Pi$, $\Sigma_S \equiv *\{\Sigma_{\{p,q\}} | p, q \in S\}$.*

3.2 Failure detector Ω_S

Given any subset S of processes in Π, failure detector Ω_S outputs at any time and at any process, one process called the *leader*, such that all processes inside S eventually get the same correct leader. More precisely, the following property is satisfied:

—*Unique eventual leader:* $\forall F \in \mathcal{E}, \forall H \in \Omega_S(F), \exists l \in correct(F), \exists \tau \in \Phi, \forall \tau' > \tau, \forall x \in correct(F) \cap S, H(x, \tau') = \{l\}$

Note that processes outside S might never get the same leader or might permanently get crashed leaders. Note also that the leader process that is output (in particular to processes in S) does not need to be in S: it can be any process in Π. Failure detector Ω from [Chandra et al. 1996] corresponds to Ω_Π.

We state and prove below a useful property of the composition of Ω_S failure detectors over several subsets S. This property will be key to show later that the weakest failure detector to implement all objects with consensus number $k > 1$ is the same.

PROPOSITION 3. *Let $\mathcal{L}$ be any family of subsets of Π such that, for all $p, q \in \Pi$, there exists some $L \in \mathcal{L}$ such that $p \in L$ and $q \in L$. We have: $\Omega \equiv *\{\Omega_L | L \in \mathcal{L}\}$.*

PROOF. As Ω is also Ω_L for every $L \subseteq \Pi$, we directly get: $*\{\Omega_L | L \in \mathcal{L}\} \preceq \Omega$.

Proving that $\Omega \preceq *\{\Omega_L | L \in \mathcal{L}\}$ is more involved. The idea is for the processes to collectively use the outputs of their Ω_L failure detectors in order to construct a directed graph (digraph), and then use this graph to eventually extract the same correct process.

Consider any failure pattern F. Consider the digraph $G =< V, E >$ for which $V = correct(F)$, and $(p, q) \in E$ if and only if q is eventually permanently leader for p for some Ω_L such that $p \in L$. For all correct processes p, q, by definition of

$\mathcal{L}$, there is at least one L, say L_{pq}, within $\mathcal{L}$ such that p and q both belong to L_{pq}. Hence, there is a correct process x (the leader of p and q for this $\Omega_{L_{pq}}$) such that both (p, x) and (q, x) belong to E.

—We denote by $G* =< V', E' >$ the digraph of the strongly connected component of G: V' is the set of strongly connected components of G and $(C, C') \in E'$ if and only if there is at least one process $p \in C$ and one process $q \in C'$ such that $(p, q) \in E$.

—We say that $C \in V'$ is a *sink* of G if there is no edge going out of C. Note that this means that if $p \in C$ and $(p, q) \in E$ then $q \in C$.

—For correct process p in V, we denote by $G|p =< V', E' >$ the restriction of G to p: V' is the set of all $x \in V$ such that there is a path in G from p to x and (x, y) is in E' if (x, y) is in E. As for G, we define the sinks of $G|p$ by locating its strongly connected components without outgoing edges.

It is easy to see that G has exactly one sink S, which is also the unique sink of every restriction $G|p$ to any correct process p. We describe below an algorithm where every process p uses $*\{\Omega_L | L \in \mathcal{L}\}$ to eventually construct graph $G|p$ above and output in a variable $Trust_p$ a correct process from its sink S which will be the same for all correct processes (emulating the output of Ω).

Every process p periodically performs the following:

(1) p consults its Ω_L failure detectors (p has at least one such failure detector), gathers all outputs of those failure detectors in a variable $Leader_p^p$, and broadcasts the value of this variable to all processes. Basically, $Leader_p^p$ is the set of processes output as leaders from Ω_L for all L in $\mathcal{L}$ such that $p \in L$.

(2) Upon receiving a set of leaders from some process q, process p updates a variable $Leader_p^q$ gathering all leaders of q (as known to p). If p subsequently receives from q a new set X of leaders, then p replaces $Leader_p^q$ by X.

(3) Using variables $Leader_p^q$, process p maintains a directed graph (digraph) $G_p =< V_p, E_p >$ for which $V_p = \Pi$, and $(r, q) \in E_p$ if $q \in Leader_p^r$. We use the same notation $((G_p|p)$,$(G_p|p)*$, sink) for G_p as for G.

(4) Whenever it updates its graph $G_p =< V_p, E_p >$, p extracts two other digraphs: $G_p|p$ and $(G_p|p)*$

(5) Whenever it builds the digraph $(G_p|p)*$, p locates all sinks of $G_p|p$. If p locates exactly one sink, then p outputs in variable $Trust_p$ the process with the lowest id in that sink; else p outputs itself in variable $Trust_p$.

Let τ_0 be a time after which no process crashes and the output of failure detectors Ω_L, $L \in \mathcal{L}$, does not change. As any change in $Trust_p$ comes from changes in the output of Ω_L's, and no message is lost, then there is a time $\tau_1 \geq \tau_0$ after which $Trust_p$ does not change.

We consider the digraph G_p obtained by a correct process p after time τ_1. G_p has all the processes as vertexes, including faulty ones. If an edge has a crashed process as source, this edge is constructed from the output of Ω_L before time τ_1. When we consider $G_p|p$, we remove crashed processes from the set of vertexes, but we do not obtain G because we might have also removed some correct processes. We show in the following that $G|p$=$G_p|p$.

LEMMA 4. *If x is a correct process then (x, y) is an edge of G_p if and only if (x, y) is an edge of G.*

PROOF. If x is a correct process, then after time τ_1, $Leader_p^x$ is the set of leaders of x for some Ω_L such that $x \in L$. □

LEMMA 5. *The set of vertexes of $G_p|p$ is a subset of $correct(F)$*

PROOF. Let y be any vertex of $G_p|p$. There is a path from p to y: $s_0 = p$,s_1,..,$s_m = y$ such that (1) s_i is a vertex of G_p and (2) (s_i, s_{i+1}) is an edge of G_p. As a consequence, s_i is a vertex of $G_p|p$ and (2) (s_i, s_{i+1}) is an edge of $G_p|p$.

As p is correct and s_1 is the leader by Ω_L for some L that contains p, then s_1 is a correct process. By an easy induction, each s_i is a correct process.

□

LEMMA 6. $G|p$=$G_p|p$

PROOF. By Lemma 4 and Lemma 5, $G_p|p$ is a subgraph of $G|p$. We now prove that $G|p$ is also a subgraph of $G_p|p$.

Let v be any vertex of $G|p$; by construction of $G|p$ (1) v is correct, and (2) there is a path from p to v. By Lemma 4, this path is also a path in G_p, which implies that v is a vertex of $G_p|p$.

Let (x, y) be any edge of $G|p$; by construction of $G|p$, there is a path in G from p to x, and an edge from x to y. By Lemma 4, this path and this edge are also in G_p. Let P be the set of vertexes in this path. From all the vertexes z in $P \cup \{y\}$, there is a path from p to z in G_p, which means that $P \cup \{y\}$ is a subset of the vertexes of $G_p|p$. By construction (x, y) is an edge of $G_p|p$.
□

Hence, the (unique) sink of $G_p|p$ is also the unique sink S of $G|p$. As the unique sink of $G|p$ is also the unique sink of G, all correct processes extract S. As S is a non-empty subset of correct processes, all correct processes eventually output the same correct process.

In particular, for the family of subsets of two elements:

COROLLARY 7. $\Omega \equiv *\{\Omega_{\{p,q\}}|p, q \in \Pi\}$.

Remarks. An attentive reader would wonder why the formulation of Proposition 1 and Proposition 3 are different. This is inherent to the definitions of the two failure detectors Ω_S and Σ_S as we show for completeness in the optional appendix (Proposition 30).

What is important however for our incoming weakest failure detector results (next two sections) is the case where S is the overall set of processes Π and the subsets are simply pairs, namely Corollary 7 and Corollary 2.

4. THE WEAKEST FAILURE DETECTOR TO IMPLEMENT A REGISTER

In this section we focus on the basic register type. This object has two operations, *read()* and *write()*, and its sequential specification simply stipulates that a read() returns the last value written.

Consider any subset S of processes in the system Π. A S-register is one where any process in S can read or write: the processes outside S cannot. When S is the overall set Π of processes, such a register (i.e., that can be read and written by any process) is also called a *multi-writer/multi-reader* register [Lamport 1986], or simply a register.

We show in this section that Σ_S is the weakest failure detector to implement a S-register. As a direct corollary, Σ is the weakest failure detector to implement a register.

PROPOSITION 8. *Σ_S is the weakest failure detector to implement a $S-$register.*

PROOF. *Overview.* Our proof is based on the existence of two algorithms.

(1) (Necessary condition) Our first algorithm emulates the output of failure detector Σ_S using any algorithm that implements a S-register. In a sense, this algorithm extracts Σ_S from any (failure detector-based) S-register algorithm. In short, processes keep using instances of this algorithm to track faulty processes and extract quorums.

(2) (Sufficient condition) Our second algorithm uses Σ_S to implement a S-register. The algorithm is an adaptation of a classical implementation of a register in a message passing system with a majority of correct processes [Attiya et al. 1995]. Instead of the assumption of a majority of correct processes, we use Σ_S.

Preliminaries. A single-writer/multi-reader S-register. Before exhibiting the algorithms underlying our proof, we introduce below a particular S-register. We consider a S-register that can be read by all processes in S and written by exactly one process p in S (the writer), and which we call a (p, S)-register. When S is Π, such a register corresponds to a *single-writer/multi-reader* register in the sense of [Lamport 1986].

We assume that different write operations store different values in the register: this can simply be achieved by appending to every value the identity of the writer process together with some local timestamp. We say that a value has been written (resp. read) if the corresponding write (resp. read) has returned a reply (i.e. was terminated). 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.

Along the lines of [Attiya et al. 1995; Attiya and Welch 1998; Herlihy and Wing 1990], the correctness of an implementation of an atomic (p, S)-register can be conveniently expressed through three properties. These properties need to be satisfied by the subtasks implementing primitives invoke and reply corresponding to read and write operations:

—(1) Termination (liveness). If a correct process of S invokes an operation, the operation eventually terminates.

—(2) 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.

—(3) 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.

Overview of the necessary condition proof (extracting Σ_S from any S-register algorithm). We describe in the following an algorithm, denoted by R, that uses any algorithm that implements a S-register (possibly using some failure detector $\mathcal{D}$), to implement failure detector Σ_S, i.e., to emulate the output of failure detector Σ_S. The emulation is achieved within a distributed variable, denoted by $Trust$ (the local value of $Trust$ at process p is denoted by $Trust_p$). When query() is invoked by a process p (to access the value of failure detector Σ_S that is emulated), it returns the value of $Trust_p$.

Algorithm R is such that variable $Trust$ ensures the *completeness* and *intersection* properties of Σ_S.

In the following, we first give an idea of the algorithm R, then we describe the algorithm in more details and finally we prove its correctness.

Our algorithm R relies on the very fact that some algorithm implements the write and read operations of a set of (p, S)-registers: every process p of S is associated with exactly one (p, S)-register, denoted by Reg_p: p is the only writer of Reg_p and all processes of S read in Reg_p.

There are three key ideas underlying our algorithm R:

(1) Every process p periodically writes in Reg_p 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. Process p determines the processes that *participate* in every *write(k,*)* and this set is denoted by $P_p(k)$. This set is determined by having every process that receives some message m in the context of the $k-$th write from p, tags every message that causally [Lamport 1978] follows m, with k, p, as well as with the list of processes from which messages have been received with those tags. Note that p belongs to $P_p(k)$. When p terminates *write(k,*)*, p gathers in $P_p(k)$ the set of all processes that participated in *write(k,*)*.

An important property of every set $P_p(k)$ is that, if there is at least one correct reader, then $P_p(k)$ contains at least one correct process. Thus, if p is correct or at least one reader is correct, then $P_p(k)$ contains at least one correct process.

(2) Every process p maintains a list of process sets, we denote by E_p, where each set within E_p gathers the processes that participated in some previous write performed by p. Basically, before *write(k,*)* is performed on Reg_p, $E_p := \{P_p(0), P_p(1),\ P_p(2),\ ..., P_p(k-1)\}$. Initially, E_p contains exactly one set: the set of all processes Π, i.e., we assume that $P_p(0) = \Pi$. Hence, whenever p terminates some write, p updates the set E_p.

An important property of every set E_p is that, eventually, all new sets that are added to it contain only correct processes. This is because, after all faulty processes have crashed, the processes that participate in new write operations are necessarily correct.

(3) The value that process p writes in Reg_p, together with k (i.e., its $k-$th write),

is the value of E_p after the $(k-1)-$th write. After a process p writes E_p in Reg_p, p reads every register Reg_q written by every process q in S. For every set $P_q(l)$ in E_q, process p selects at least one process q by sending a message to all processes in $P_q(l)$ and waiting for at least one reply. The value of the variable $Trust_p$ is the value of $P_p(k-1)$ augmented with every process x that p selected during the read phase.

Every correct process p keeps on permanently updating variable $Trust_p$. In fact, if p is correct, then there is at least one correct reader of all registers from which p reads and, as previously pointed out, every $P_q(k)$, for some correct q, contains at least one correct process. So, p only waits for a message from one process in every set X in E_q, and every such set contains at least one correct process. In short:

—The *completeness* property of Σ_S is ensured because every correct process p keeps permanently updating variable $Trust_p$ and this variable eventually contains only correct processes.

—The *intersection* property of Σ_S is ensured because every process p writes into its register first, and only then reads all other registers before finally updating $Trust_p$.

```
Code for every process p
/* each Σ_S failure detector consultation returns the value of variable Trust */

Initialization:
   P(0) := Π
   E := {P(0)}        /* E is the set of subsets of participants in some write on Reg_p */
   k:=0               /* k represents the number of times a write was invoked by p */
   F := ∅             /* F_i is a temporary value of trusted processes */
   Trust := Π         /* Initially, all processes are trusted */
start task 1 and task 2

task 1:
loop forever
   k := k + 1
   P(k) := Reg_p.write(k, E)
   E := E ∪ {P(k)}
   F := P(k − 1)
   forall q ∈ S do
      L := Reg_q.read()
      forall X ∈ L do
         send(k, ?) to all processes in X
         wait until receive(k, ok) from at least one process q ∈ X
         F := F ∪ {q}
   Trust := F

task 2:
   upon receive(k, ?) from q send(k, ok) to q
```

Fig. 1. Implementing Σ_S using a specific S-register

```
/* Every process p tags every message it sends with Tag */
/* Tag, the tag for register Reg_p, is a pair (k, LL)*/

Initialization:
   Current := 0
   Tag := (0, ∅)

Code for every process q (including the writer p of Reg_p):
   When q receives a message tagged with (k, LL) for Reg_p
      case Current > k : skip
      case Current = k : Let X such that Tag = (k, X)
                         Tag := (k, X ∪ LL ∪ {q})
      case Current < k : Tag := (k, LL ∪ {q})
                         Current := k;

Code for the writer p of Reg_p:
   When p begins the k-th write on register Reg_p
      Current := k
      Tag := (k, {p})
   When p ends the k-th write operation on register Reg_p
      return LL : such that Tag = (k, LL)
```

Fig. 2. Tagging for (p, S)-register Reg_p

Proof of the necessary condition (extracting Σ_S from any S-register algorithm). We describe here in details the emulation of Σ_S from a S-register algorithm. For modularity purposes, we present our algorithm R in two parts. We describe in Figure 1 how to emulate variable $Trust$ using a specific S-register with a specification customized to our needs. Then we show in Figure 2 how to implement this specific register with any algorithm that implements a S-register in a message passing system.

The specific S-register has a traditional read but a non-traditional write. The register has one writer so this non-traditional write can be performed by only one process. 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 indicate 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 in the specific (p, S)-register we consider; w_b, respectively w_e, denote the beginning event, respectively the termination event of the write operation w. Let $\preceq$ be the causality relation of [Lamport 1978]. The set of *participants* in w, $\mathcal{P}(w)$, is the following set of processes:

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

The algorithm of Figure 2 tracks and returns the set of participants in every *write(k,*)* operation. Let p be the writer of a (p, S)-register Reg_p and consider an algorithm implementing this register (possibly using some failure detector). We tag every message causally after the beginning of the k–th write of Reg_p and causally before the beginning of the $k+1$–th write with a pair (k, LL) where LL is the list of participants to the k–th write.

The following lemma states that the set of processes returned by the algorithm of Figure 2 at the end of the k-th write by the process p is indeed the set of processes that participate in the write. More precisely, let $P_p(k)$ be the value returned by the algorithm of Figure 2 for the k–th write, we have:

LEMMA 9. *In the algorithm of Figure 2, the set of participants of the k–th terminated write of Reg_p is the value returned for $P_p(k)$.*

PROOF. Let w the k–th terminated write of p.

(1) We first show that $\mathcal{P}(w) \subseteq P_p(k)$. 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. All messages in M_1 or M_2 can only be tagged by $(j, *)$ with $j \geq k$. As p does not begin the j–th write, with $j > k$, before the end of the k–th write, 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 $P_p(k)$, we have $x \in P_p(k)$.

(2) Now we prove that $P_p(k) \subseteq \mathcal{P}(w)$. Let x be any process of $P_p(k)$.
As only x can add its identity to the list LL of the tag (k, LL) of a message, any p_u can only receive a message with tag (k, LL) such that $x \in LL$ 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 a message with tag (k, LL) and let e_1 be the event corresponding to the first time p receives a message with tag (k, M) for some M with $x \in M$, we have: $e_0 \preceq e_1$. Moreover, as $x \in P_p(k)$, the algorithm ensures that $e_1 \preceq w_e$ and then (1) $e_0 \preceq w_e$.
As only p 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 $(k, *)$. As in e_0 the value of $Current$ for x is k, we have: (2) $w_b \preceq e_0$.
From (1) and (2), 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 10. *Let w be the $k-th$ terminated write of p in some failure pattern F, if $S \cap correct(F) \neq \emptyset$, then $P_p(k) \cap correct(F) \neq \emptyset$*

PROOF. Remember that we assume any two different write operations to write two different values. Notice first that $p \in P_p(k)$ for all k, hence:

—if p is correct, then the lemma is trivial.

—if p is faulty and all the readers are faulty, then the lemma is also trivial.

In order to obtain a contradiction, assume that, for some terminating write w of p in register Reg_p, some run $\alpha =< F, C, Sc, T >$ and some associated failure detector history H, we have a correct reader, say q, and $P_p(k) \cap correct(F) = \emptyset$.

In the following, we exhibit several runs; all have F as failure pattern and H as associated failure detector history. They may differ from α by the time at which processes take steps (we use the fact that the system is asynchronous).

—Run α_0. This run is identical to α up to w_e, and the writer p does not invoke any write after w_e. The set of participants in w, $P_p(k)$, is the same in α and α_0, but in α_0, p does not invoke any write after w. Let v be the value of register Reg_p before the terminating write w, and v' the value after w (recall that we assume $v \neq v'$).
Let τ_e be the time of the event w_e of α, and consider time $\tau \geq \tau_e$ after which no more processes crash. Note that, by hypothesis, at time τ all participants of w have crashed. For any process x in $P_p(k)$, 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.

—Run β. For any process x of $P_p(k)$, β is identical to α_0 up to b_x^{-1}, but after b_x^{-1}, x does not take any step until time τ. As after time τ, x has crashed, x does not take any step after b_x^{-1}. The processes of $\Pi - P_p(k)$, in particular q, take steps exactly as in α_0 up to time τ (at the same time, but perhaps the step is not the same). At time τ, q reads the register Reg_p and q ends the read at time τ'. As w is not in run β, then q reads v in the register.

—Run γ. For any process x of $P_p(k)$, γ is identical to α_0 up to e_x. After e_x, x does not take any step until time τ. If after b_x, x sends a message to a process of $\Pi - P_p(k)$, the reception of this message is delayed until after time τ'. Processes of $\Pi - P_p(k)$ take steps exactly as in β up to time τ' (the steps that these processes take in γ and β are the same steps up to time τ'). After time τ', the correct processes may receive the pending messages: runs β and γ may then differ.
In γ, the writer has completed its write operation w. The reader q begins the read after the end of the write, by the atomicity of the S-register, q reads v'.

For q, γ and β are indistinguishable. Process q reads v in β, q reads v' in γ — a contradiction. □

Lemma 11. *The algorithms of Figure 1 and Figure 2 implement failure detector Σ_S (in variable $Trust$).*

Proof. Let p be any process in S and recall that v_p denotes the local variable of any distributed variable v of p (e.g., $Trust$). Local variable $Trust_p$ contains a list of processes. We show that this list ensures the *completeness* and *intersection* properties of Σ_S.

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

If p is correct, then there is at least one correct process in S. By Lemma 10, for every process q and every integer k, $P_q(k)$ contains at least one correct process.

Line 15, reading Reg_q, p sets L_p with $P_q(k)$ sets. Therefore, at least one of the processes in these $P_q(k)$ sets answers to the message $(*, ?)$ from process p and process p cannot block on Line 18. Hence p updates infinitely often variable $Trust_p$ and the sequences $Trust_p^1, \dots Trust_p^m \dots$ as well as $\tau_p^1, \dots, \tau_p^m, \dots$ are infinite sequences.

Notice that if p is faulty, it can block on Line 18 until it crashes.

(1) We first prove the *completeness* property of *Trust*. We need to show that for every correct process p, there is an integer m such that, for every $m' > m$, $Trust_p^{m'}$ contains only correct processes.
Consider the time τ after which all faulty processes have crashed. As p is correct, then there is some m such that $\tau_p^m > \tau$. Let x be any process that belongs to $Trust_p^{m'}$ with $m' \geq m + 2$. Then:
—either x belongs to $P_p(m' - 1)$, meaning that x is a correct process by the very fact that all processes that participate in the write operations beginning after τ are correct.
—or x answered to some message $(m', ?)$ from p, ensuring that x was not crashed at least until time τ_p^m.
In both cases, x is correct.

(2) We now show that *Trust* ensures the *intersection* property. More precisely, we prove that, for any two processes p and q, for all k, l such that $Trust_p^k$ and $Trust_q^l$ are both defined, we have: $Trust_p^k \cap Trust_q^l \neq \emptyset$.
Remark first that, if $l = 0$ or $k = 0$, then either $Trust_p^l$ or $Trust_q^k$ is the set of all processes Π, as for every process r, $Trust_r$ is never empty ($Trust_r$ contains at least r), in this case we have $Trust_p^l \cap Trust_q^k \neq \emptyset$. Therefore, assume that $l > 0$ and $k > 0$.
Notice the following facts:
—If process p writes E_p in its register Reg_p (Line 11) during the k-th iteration, then, for all $k' < k$, $P_p(k') \in E_p$. By construction, the value of E_p(Line 11) for the k-th write of register p is the set of all sets $P_p(k')$ for $k' < k$.
—It is clear from Lines 13, 19 and 20 that for every process r every integer m $P_r(m - 1) \subseteq Trust_r^m$.
As each process p writes in its own register Reg_p, and then reads every register of all other processes, due to the atomicity of registers, either the k-th write of the register Reg_q by q is before the l–th read of this register by process p, or the l-write of register Reg_p is before the k–th read of this register by process q. Assume without loss of generality that p performs its l–th read of register Reg_q after the k-th write of Reg_q by q. From the algorithm, at least one $s \in Trust_p^l$ (1) comes from each set of the set of sets L_q read by p in the l–th read of Reg_q, and (2) is such that p 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_q by q, we deduce that at least one $s \in Trust_p^l$ belongs to $P_q(k - 1)$ and, as $P_q(k - 1) \subseteq Trust_q^k$, s belongs to $Trust_q^k$, proving the *intersection* property.

□

Finally, we get:

LEMMA 12. *If failure detector $\mathcal{D}$ implements a S-register, then $\Sigma_S \preceq \mathcal{D}$.*

Sufficient condition (implementing a S-register with Σ_S). A special case of a (p, S)-register is a register that can be read by exactly one process q (the reader) and written by exactly one process p (called the writer). We call it a (p, q)-register: when S is Π, the register corresponds to a *single-writer/single-reader* register in the sense of [Lamport 1986].

In the following, we give an algorithm that, for any $p, q \in S$, implements a (p, q)-register using Σ_S. Using the register transformations of [Israeli and Li 1993; Vitanyi and Awerbuch 1986], we derive the fact that a S-register can be implemented, using (p, q)-registers for all $p, q \in S$.

Basically, each process maintains the current value of the register. The writer process tags each write invocation with a unique sequence number, incremented for every new write invocation. In order to perform its read (resp. write) operation, the reader p_r (resp. the writer p_w) sends a message to all processes and waits until it receives acknowledgments from every process trusted by p_r (resp p_w), i.e., output by its failure detector module. It is important to notice that the set of processes trusted by the reader (resp. the writer) might change between the time the reader (resp. the writer) sends its message and the time it receives acknowledgments. We implicitly assume here that the reader (resp. the writer) keeps periodically consulting the list of processes that are output by its failure detector module (i.e., the trusted processes) and stops waiting when the reader (resp. the writer) has received acknowledgments from all processes in the list.

For every write operation, the writer sends the value to be written with the associated sequence number to all processes. Each process p stores this value with its sequence number and sends back an acknowledgment to the writer, unless p has crashed or has already stored a value with a higher sequence number.

For every read operation, the reader sends a request to read to all. Every process that does not crash returns an acknowledgment 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 returns it.

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

LEMMA 13. *The algorithm of Figure 3 implements a (p_w, p_r)-register using Σ_S such that $p_w, p_r \in S$.*

PROOF. We consider one writer, denoted by p_w, and one reader, denoted by p_r. In the pseudo-code of Figure 3, we assume that the **if...then...** statement is atomic.

Remark first that:

—(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. As read/write invocations are sequential on each process, the writer does not begin its $(k + 1)$-th write operation before ending its k–th one.

```
1  Every process p (including p_w and p_r) executes the following subtask:
2    Initialization:
3      current := ⊥
4      last_write := 0
5      upon receive (WRITE, y, s) from p_w
6      if s > last_write then
7        current := y
8        last_write := s
9        send(ACK_WRITE, s) to p_w
10     upon receive (READ, s) from p_r
11       send(ACK_READ, last_write, current, s) to p_r
12 subtask for p_w the (unique) writer:
13   Initialization:
14     seq := 0 /* sequence number */
15   function write(x)
16     seq := seq + 1
       send(WRITE, x, seq) to all
17     wait until received (ACK_WRITE, seq)
           from all processes output by Σ_S
18     return(OK)
19   end write
20 subtask 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 output by Σ_S
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
```

Fig. 3. Implementation of a (p_w, p_r)-register using Σ_S with $p_w, p_r \in S$.

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

—(B) When a process writes k in its *last_write* variable (Line 8) the value of its *current* variable 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 (p_w, p_r)-register:

Termination: Assume by contradiction that p_w is correct and that p_w invokes but does not terminate its k–th write operation. This would be possible only if p_w waited forever in Line 17. From the *completeness* property of the failure detector, there is a time τ after which the list M of processes trusted by p_w contains only correct processes. By the properties of the message passing service, every correct process p eventually receives the $(WRITE, *, k)$ message from p_w. From (A), p replies with a (ACK_WRITE, k) message and p_w eventually receives (ACK_WRITE, k) messages from all processes within M —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. 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 belong to both L_w and L_r.

As p sends an $(ACK\_READ, s, *, j)$ message to p_r only after having sent an (ACK_WRITE, k) message to p_w, then $s \geq k$. Hence, let a be the maximum of v in the $(ACK\_READ, v, *, j)$ messages received by the reader p_r for operation R, we have $a \geq s \geq k$ 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. Hence, 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.

In the same way, this proof applies if there is no write before the j-th read operation.

Ordering: Assume that the reader reads x then y and let rx, respectively ry, be the corresponding values of *last_write* for the reader. 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 non decreasing, we have $ry \geq rx$, hence p_w wrote y after x. □

Using the register transformations of [Israeli and Li 1993; Vitanyi and Awerbuch 1986], we can now derive the fact that S-registers can be implemented out of (p, q)-registers for all $p, q \in S$. We finally get:

LEMMA 14. *There is an algorithm that implements a S-register using Σ_S for any subset S.*

With Lemma 12, we get our complete proof.

The following is then a simple corollary of Proposition 8:

COROLLARY 15. Σ *is the weakest failure detector to implement a* register.

5. THE WEAKEST FAILURE DETECTOR TO IMPLEMENT CONSENSUS

We determine here the weakest failure detector to implement the seminal consensus object type. Our result applies to all environments, including those where more than half of the processes might crash, as well as to any consensus object shared by a subset of processes in the system. We later derive the weakest failure detector to implement any object type with consensus number $k > 1$.

5.1 Implementing consensus

The sequential specification of the consensus object stipulates that all *propose()* operations return one of the values proposed. For any subset S of processes in the system, the $S-$consensus object is accessible only to the processes of S: the *propose()* operation of S-consensus can only be invoked by the processes of S.

In this section, we prove that $\Sigma_S * \Omega_S$ is the weakest failure detector to implement S-consensus. As a direct corollary, $\Sigma * \Omega$ is the weakest failure detector to implement consensus (shared by all the processes in the system Π).

PROPOSITION 16. $\Sigma_S * \Omega_S$ *is the weakest failure detector to implement $S-$consensus.*

PROOF. *Overview.* In order to prove that $\Sigma_S * \Omega_S$ is the weakest failure detector to implement S-consensus, we first prove (necessary condition) that, from any implementation of S-consensus, we can extract Σ_S and Ω_S and then we exhibit an algorithm that implements S-consensus using $\Sigma_S * \Omega_S$.

(1) Proving the necessary condition goes in two steps. We show first how to extract Σ_S (necessary condition (a)) and then how to extract Ω_S (necessary condition (b)).
  —Proving the first step of the necessary condition (a) goes essentially through the same steps as the necessary part of the proof of the weakest failure detector for consensus [Chandra et al. 1996] (i.e., Π-consensus). Interestingly the fact that S can be a subset of processes does not fundamentally change the proof. We will mainly recall the main steps of the proof of [Chandra et al. 1996] and point out some special cases that are sensitive to our generalization.
  —To prove the second step of the necessary condition (b), use the traditional fault-tolerant state machine replication approach [Lamport 1998; Schneider 1993], transforming consensus into the total order broadcast communication abstraction and implementing any object type, including the type register.
(2) To prove the sufficient condition, we give an algorithm implementing S-consensus using $\Sigma_S * \Omega_S$. The algorithm can be viewed as a deconstruction of the rotating coordinator algorithm of [Chandra and Toueg 1996]. As in [Chandra and Toueg 1996], processes are considered coordinators in a round-robin way and they each try to impose a decision. Eventually, one of the correct processes remains coordinator (the one output by Ω_S) and succeeds in imposing a decision. Agreement is ensured because no process decides until it consults a quorum of processes: this is where Σ_S is used.

Preliminaries. Before diving into our algorithm, we first precise the meaning of implementing consensus. A correct implementation of a S-consensus object can be defined through three properties, along the lines of [Fischer et al. 1985].

Every process $p \in S$ can propose a value to S-consensus and, unless it crashes, is supposed to decide a value (i.e. return a value from that invocation) such that:

—*Termination (liveness):* every correct process in S eventually decides;

—*Agreement (safety 1):* for any two processes p and p' in S, if p decides v and p' decides v' then $v = v'$;

—*Validity (safety 2):* if any process in S decides a value v, then v is the proposed value of some process in S.

In the following, we assume without loss of generality that if a correct process invokes some S-consensus object, then all correct processes participate to the implementation of that object. More precisely, we assume that all processes obey the following procedure. Let $\mathcal{A}$ be any algorithm that implements S-consensus. When a process p invokes *propose()* on the object, and p is not already running $\mathcal{A}$, then p sends a message containing its proposed value to all other processes in S and p starts running $\mathcal{A}$. When a process q receives such a message from p, if q is not already running $\mathcal{A}$, then q adopts the value proposal of p as its initial proposal and sends it to all other processes in S, before q itself runs $\mathcal{A}$. When a decision is made in $\mathcal{A}$ at some process p, either p has invoked *propose()* and the decided value by $\mathcal{A}$ is the value return by *propose()*, or the decided value is stored in case p invokes *propose()* on the S-consensus object in which case the decision value is immediately returned.

Implementing a S-consensus object is in this sense equivalent to solving the S-consensus problem (as defined here).

*(Necessary condition (a)) Extracting Ω_S from S-*consensus.

LEMMA 17. *If there is an implementation of S-*consensus *using $\mathcal{D}$, then $\Omega_S \preceq \mathcal{D}$.*

PROOF. (Sketch) As we pointed out, we mainly go through the main steps of the proof of [Chandra et al. 1996] and discuss how it generalizes to subsets S.

In [Chandra et al. 1996], all correct processes need to eventually output a correct process p^*. In our case, all processes in S have to output the same correct process p^*.

The way processes in [Chandra et al. 1996] eventually locate the same correct process is by executing an extraction algorithm, which is composed of two parts: the *communication* component and the *computation* component. As we will recall below, the goal of the communication component is to exchange values of the underlying failure detector $\mathcal{D}$ and build a directed acyclic graph (DAG) of such values, whereas the goal of the computation component is to use the DAG and simulate runs of S-consensus that will help extract the correct process p^*. In our case, processes that are not in S are involved only in the communication component.

Consider a set S of processes. Let $\mathcal{D}$ be any failure detector that implements S-consensus in $\mathcal{E}$ through some algorithm we denote by $Consensus_S^{\mathcal{D}}$.

Consider any arbitrary run of $Consensus_S^{\mathcal{D}}$ using $\mathcal{D}$ with failure pattern $F \in \mathcal{E}$ and any history $H_{\mathcal{D}}$ in $\mathcal{D}(\mathcal{F})$.

The processes periodically query their failure detector and exchange information about the values of $H_{\mathcal{D}}$ they see in the run.

Using this information, all processes construct a directed acyclic graph (DAG) that represents a sampling of failure detector values in $H_{\mathcal{D}}$ and some temporal relationships between the sampled values. By periodically sending the current state of its DAG to all processes, and by incorporating information from all others processes into its own DAG, every correct process constructs ever increasing approximations of one (infinite) limit DAG G.

In the computation component, the DAG G is used to simulate runs of $Consensus_S^{\mathcal{D}}$ for failure pattern F and failure history H: all these runs could have occurred for F and $H_{\mathcal{D}}$. These are used to eventually locate the same correct process p^*.

Consider any initial configuration I of $Consensus_S^{\mathcal{D}}$. The set of simulated schedules of $Consensus_S^{\mathcal{D}}$ that are compatible with some path of G and are applicable to I are organized as a tree. Given $S = \{q_1, \ldots, q_k\}$, consider the $k+1$ initial configurations, I_j for $0 \leq j \leq k$ such that in I_j the initial value of q_m is 0 for all $k \geq m > j$ and is 1 for all $j \geq m \geq 1$. Let $\Gamma_G^{I_i}$ be the tree of simulated schedules with initial configuration I_i. And let Γ_G be the forest of all these trees.

The main point is that, from Γ_G, it is possible for the correct processes of S to extract the identity of a correct process, say p^*. To do so, each vertex of every tree of Γ_G is tagged with 0 and 1: A vertex V is tagged with k if and only if it has a descendant V' such that some process in S has decided k in V'. As all processes (correct or faulty) that decide, decide on the same value, we can take the decision value of any process in S. Γ^i denotes the tagged tree $\Gamma_G^{I_i}$. Remark that a vertex is either tagged with $\{1\}$ or $\{0\}$ or $\{1,0\}$. In the first case, the vertex is said to be 1-valent, in the second 0-valent and in the third case bivalent.

By the validity property of S-consensus, the root of Γ^0 is 0-valent and the root of Γ^k is 1-valent. By an easy induction, there exists an index i such that either the root of Γ^i is bivalent, or the root of Γ^{i-1} is 0-valent and the root of Γ^i is 1-valent. In the second case, it is proved [Chandra et al. 1996] that q_i is the correct process p^*.

In the first case, locating p^* is more complicated. In [Chandra et al. 1996], it is shown that Γ^i contains a special subtree, named a decision gadget. Intuitively, in a decision gadget, the step of a particular process is crucial. The first process to be involved in such a gadget is p^*.

From a bivalent vertex, one of its step, directly or indirectly, leads to a 0-valent vertex and another step to a 1-valent vertex. This process is necessarily a correct process. Notice that the decision gadget is in a finite subgraph of Γ_G.

Each process in S tries to extract p^*. But since the limit of its forest over time is Γ_G, and the information necessary to select p^* is in a finite subgraph of Γ_G, eventually the process will keep forever selecting the correct process p^*. □

*(Necessary condition (b)) Extracting Σ_S from S-*consensus.

LEMMA 18. *If there is an implementation of S-*consensus *using $\mathcal{D}$, then $\Sigma_S \preceq \mathcal{D}$.*

PROOF. Let X be any algorithm implementing S-consensus, using failure detector $\mathcal{D}$: as $\mathcal{D}$ depends only on the failure pattern (and not on the state of any process), we can use X with $\mathcal{D}$ at any time. With X, we can implement [Hadzilacos and Toueg 1993] a total order broadcast abstraction [Aguilera et al. 2000], that is restricted to S. We first recall the specification of this abstraction in terms of the primitives S-*ABroadcast* and S-*ADeliver*:

—*Validity*: If a correct process in S S-*ABroadcasts* a message m, then it eventually S-*Adelivers* m.

—*Uniform Agreement*: If a process in S S-*ADelivers* a message m, then all correct processes in S eventually S-*Adeliver* m.

—*Uniform Integrity*: For every message m, every process in S S-*ADelivers* m at most once, and only if m was previously S-*ABroadcast* by some process in S.

—*Uniform Total Order*: If some process in S-*ADelivers* a message m before a message m' then no process in S S-*ADelivers* m' before it S-*ADelivered* m.

With this abstraction, the algorithm of Figure 4 implements a S-Register. The proof is straightforward. Hence, by Lemma 12, Σ_S can be implemented and we get $\Sigma_S \preceq \mathcal{D}$. □

```
Every process p in S executes the following code:
  Initialization:
    current := ⊥
  function write(x)
    seqw := false
    S-ABroadcast(WRITE, x, p)
    wait until seqw
    return(OK)
  end write

  function read()
    seqr := false
    S-ABroadcast(READ, p)
    wait until seqr
    return(val)
  end read

  forever
    waitS-ADeliver(m)
    if m = (WRITE, x, q) for some x and some q then
      current := x
      if p = q then seqw := true
    if m = (READ, p) then val := current; seqr := true
```

Fig. 4. Implementation of a S-Register from total order broadcast.

*(Sufficient condition) Implementing S-*consensus *with $\Sigma_S * \Omega_S$.* The algorithm of Figure 5 implements S-consensus using failure detector $\Sigma_S * \Omega_S$. The idea of the algorithm is the following. Processes are promoted coordinators in a round-robin way and they each try to impose a decision. These coordinators do not need to be in S.

The key to ensuring agreement is for the coordinator processes to always propose for decision a value adopted by a quorum of processes output by Σ_S, and only impose the decision if a quorum of processes output by Σ_S adopts that value (not necessarily the same quorum). Note that the processes in a quorum do not need to be in S.

Eventually, one of the processes remains coordinator (the one output by Ω_S) and succeeds in imposing a decision. Agreement is ensured because no process decides until it consults a quorum of processes: this is where Σ_S is used.

When proving the correctness of our algorithm, and for convenience purposes, for any process p, we will say that p suspects q by Ω_S if the output of Ω_S is not q.

```
Code for p:
  Initialization:
    if p ∈ S then v = v_p                          /* v_p is the proposed value of p */
    r := 0                                         /* round number */

    if p ∈ S then
      start Task 0 and Task 1 and Task 2
    else start Task 0

Task 0:
    upon receive(COORD, ∗, k) for the first time
        let (COORD, w, k) be this message
        send(ONE, w, k) to all processes in S

    upon receive(STORE, ∗, k) for the first time
        let (STORE, w, k) be this message
        send(TWO, w, k) to all processes in S

Task 1:
  loop forever
    C := 1 + r mod n                               /* coordinator */
    send(COORD, v, r) to p_C
    wait until receive (ONE, ∗, r) from p_C or p suspects p_C by Ω_S
        if (ONE, w, r) is received then
          estfromC := w
        else
          estFromC := ⊥

    send(STORE, estFromC, r) to all
    wait until receive(TWO, ∗, r) from all processes output by Σ_S
        let L = {w | (TWO, w, r) is received }
        if L = {rec} for some rec ≠ ⊥ then
          send (DECIDE, rec) to all
          decide(rec)
          halt
        else
          if L = {rec, ⊥} for some rec ≠ ⊥ then
            v := rec
    r := r + 1

Task 2:
    upon received(DECIDE, k) from q
          send(DECIDE, k) to all
          decide(k)
          halt
```

Fig. 5. Round based S-consensus algorithm.

Lemma 19. *The algorithm of Figure 5 implements S-consensus using $\Sigma_S * \Omega_S$ for every subset S of Π.*

Proof. To prove this lemma, we go through intermediate lemmas:

Lemma 20.

(1) *If p and q execute Line 15 to 20 of a round r, then:*
   (1.1) if $estFromC_p = x$ for some $x \neq \bot$ then $estFromC_q \in \{\bot, x\}$,

(2) *If p and q end Line 23 of a round r, then:*
   (2.1) either $L_p = \{\bot\}$ or $L_p = \{x\}$ or $L_p = \{x, \bot\}$ for some $x \neq \bot$;
   (2.2) if $L_p = \{x\}$ for some $x \neq \bot$ then $L_q = \{x\}$ or $L_q = \{x, \bot\}$,
   (2.3) if $L_p = \{\bot, x\}$ for some $x \neq \bot$ then $L_q = \{x\}$ or $L_q = \{x, \bot\}$ or $L_q = \{\bot\}$.

Proof. (1.1): Notice first that for any process q, v_q is always a value proposed by some process and obviously $v_q \neq \bot$.

If $estFromC_p = x$ for some $x \neq \bot$ then p has received one message (ONE, x, r) from the coordinator $p_{1+r \bmod n}$. By the algorithm, the coordinator $p_{1+r \bmod n}$ sends only one message $(ONE, *, r)$ per round to all processes in S. Either q suspects the coordinator by Ω_S and then $estFromC_q = \bot$, or q does not suspect the coordinator by Ω_S and waits for message ONE, and then $estFromC_q = x$.

(2.1): the algorithm ensures that all values in L_p comes from $estFromC_q$ values, hence (2.1) is a direct consequence of (1.1).

(2.2) and (2.3): If $L_p = \{\bot, x\}$ or $L_p = \{x\}$ then at least one process of S, say u, ends the first part (Lines 15 to 20) of round r, and $estFromC_u = x$. By (1.1), at most two values, $\bot$ and x, are sent by processes of S to all processes in Line 21. A process sends (TWO, a, r) to all processes in S if it has received a message $(STORE, a, r)$ from some process of S. Then $a = x$ or $a = \bot$. And hence for any process q that ends round r either $L_q = \{x\}$ or $L_q = \{\bot, x\}$ or $L_q = \{\bot\}$. This concludes the proof of (2.3). For (2.2), it remains to show that $L_q \neq \{\bot\}$.

By the *intersection* property of Σ_S, there is at least one process s output by Σ_p and Σ_q. The algorithm ensures that s sends at most one message $(TWO, *, *)$ per round. Then s sends message (TWO, y, r) with either $y = \bot$ or $y = x$. As we assume $L_p = \{x\}$, then y equals to x, proving that x belongs to L_q; by (2.1) this proves (2.3). □

Lemma 21. *If all processes p of S, that start the round r, start round r with the same value d for v_p, then every process p of S ending r either decides d or has $v_p = d$ at the end of this round.*

Proof. Let r be such a round, then all $(COORD, x, r)$ messages are such that $x = d$. Hence, every (ONE, x, r) message is such that $x = d$. From the previous lemma, every process p ending round r ends this round either with $\{\bot\}$ and does not change its v_p or with $L_p = \{d\}$ and decides d or with $L_p = \{d, \bot\}$ and sets v_p to d. □

Lemma 22. *The algorithm of Figure 5 ensures agreement.*

Proof. Consider the first time a process, say p, of S sends a $(DECIDE, *)$ message and let $(DECIDE, d)$ be this message. By an easy induction, this send occurs in Task 1, say in round r. In this round, after Line 23, L_p is $\{d\}$. Let q be any other process ending round r. By Lemma 20, in this round L_q is either $\{d\}$ and q decides in round r, or $\{d, \perp\}$ and q ends the round r with $v = d$.

By Lemma 21 and an easy induction, in every round $r' \geq r$, every process in S either decides d or ends the round with $v = d$. Hence, all processes which decide in Task 1, decide d. If a process decides in Task 2 then, by an easy induction, this decision is issued from a process which has decided in Task 1. This proves agreement. □

Lemma 23. *The algorithm of Figure 5 ensures validity.*

Proof. By the algorithm, the processes of $\Pi - S$ send the values they have just received and they never insert in the algorithm a value of their own. □

Lemma 24. *The algorithm of Figure 5 ensures termination.*

Proof. If all processes of S are faulty, then termination is trivial. Consider now any correct process in S. By contradiction assume that no process decides. If a correct process has decided by Task 2 then all correct processes decide. If no correct process decides by Task 2, the *completeness* of $\Sigma_S * \Omega_S$ ensures that no process waits forever in Lines 16 and 22; hence every correct process terminates round r for all r.

By the property of Ω_S, there is a time τ after which (1) all faulty processes have crashed and (2) the failure detectors of all correct processes of S output forever the same correct process, say p_l.

Consider the set of rounds R in which the correct processes of S are at this time τ and r the greatest elements of R Let r_0 be the first round number greater than r in which p_l is the coordinator ($p_l = 1 + r_0 \ mod \ n$). When the processes of S are in round r_0, they do not suspect coordinator p_l of the round r_0 by Ω_S. Then they adopt for $estFromC$ the value sent by p_l. And so their L is reduced to one element which is different from $\perp$ and they decide − a contradiction. □

This concludes the proof of Lemma 19.

By Lemma 19, $\Sigma_S * \Omega_S$ implements S-consensus. Consider any failure detector $\mathcal{D}$ that implements S-consensus. By Lemma 18, $\Sigma_S \preceq \mathcal{D}$. By Lemma 17, $\Omega_S \preceq \mathcal{D}$ and then $\Sigma_S * \Omega_S \preceq \mathcal{D}$. This concludes the proof of Proposition 16.

As a direct corollary, we get:

Corollary 25. $\Sigma * \Omega$ *is the weakest failure detector to implement* consensus.

We directly get from Corollaries 7 and 2 and Proposition 16 the following:

Corollary 26. *For every k such that $2 \leq k \leq n$, for any failure detector $\mathcal{D}$, $\mathcal{D}$ implements* consensus *if and only if $\mathcal{D}$ implements S-*consensus *for all S such that $|S| = k$.*

5.2 Implementing other object types

In the following, we say that types $T_1, \cdots, T_n$ emulate k-consensus if there is an implementation of S-consensus using only $T_1, \cdots, T_n$ for any subset S of k processes in Π.

Proposition 27. *If a type T emulates* 2-consensus*, then (1) the weakest failure detector to implement T is $\Sigma * \Omega$ and (2) any failure detector that implements T implements any type.*

Proof. Let T be any type emulating 2-consensus. This means that there is an algorithm using T that implements 2-consensus (i.e., consensus among any pair of processes). Clearly, this algorithm with any failure detector $\mathcal{D}$ implementing T implements 2-consensus too and by Corollary 26 it implements consensus. Then, by Proposition 16 we get: (a) $\Sigma * \Omega \preceq \mathcal{D}$.

Remark that $\Sigma * \Omega$ implements any number of instances of consensus. Hence, using the universality result of consensus [Herlihy 1991], we derive that $\Sigma * \Omega$ implements any type. Then by (a) any failure detector that implements T implements any type proving (2). Moreover, as $\Sigma * \Omega$ implements any type, it implements in particular T. Together with (a), this proves (1). □

An interesting application of Proposition 27 concerns the notion of consensus number, as we discuss below.

In fact, several definitions of the notion of consensus number of a type T (sometimes also called consensus power) have been be considered [Jayanti 1993]. All are based on the maximum number k of processes for which there is an algorithm that, using T, emulates k-consensus. The definitions differ on whether or not the implementation can use several instances of T, and whether the type register can also be used. Notation h_1 means one instance and no register, h_1^r means one instance and several registers, h_m means many instances, no register, and h_m^r means many instances and many registers.[2]

From Proposition 27, the weakest failure detector to implement type T such that $h_1(T) = 2$ or $h_m(T) = 2$ is $\Sigma * \Omega$. If T is deterministic, we can derive from [Bazzi et al. 1997] that $h_m(T) = h_m^r(T)$. Hence we get the following result:

Proposition 28. *For every k such that $2 \leq k \leq n$, $\Sigma * \Omega$ is the weakest failure detector to implement (1) any type T such that $k = h_1(T)$, (1') any type T such that $k = h_m(T)$, (2) any deterministic type T such that $k = h_1^r(T)$, and (2') any deterministic type T such that $k = h_m^r(T)$.*

6. CONCLUDING REMARKS

We show in this paper that the information about failures that is necessary and sufficient to implement types like queue and test-and-set, is the same as the information that is necessary and sufficient to implement types like compare-and-swap and consensus. All these types are in a precise sense equivalent, according to the information about failures needed to implement them.

For completeness, we consider in the optional appendix the case of a system of 2 processes and we prove that, in such a system, the necessary and sufficient information about failures to implement a register is the same as that necessary and

[2] We implicitly assume here *n-ported* types, i.e., every instance of a type has n ports in our system of n processes [Jayanti 1993].

sufficient to implement consensus.[3] That is, in a system of 2 processes, the weakest failure detector to implement a register is the same as the weakest to implement consensus. This contrasts with the fact that there is no asynchronous algorithm that implements consensus using registers even in a system of 2 processes [Loui and Abu-Amara 1987].

Acknowledgments

Comments from Partha Dutta, Petr Kouznetsov, Nancy Lynch, Bastian Pochon, and Michel Raynal helped improve the presentation of this paper.

REFERENCES

Aguilera, M., Delporte-Gallet, C., Fauconnier, H., and Toueg, S. 2000. Thrifty generic broadcast. In *Proceedings of the 14th International Symposium on Distributed Computing*, Springer-Verlag, Ed. LNCS 1914. 268–283.

Aguilera, M. K., Delporte-Gallet, C., Fauconnier, H., and Toueg, S. 2003. On implementing 0mega with weak reliability and synchrony assumptions. In *22th ACM Symposium on Principles of Distributed Computing*. 306–314.

Aguilera, M. K., Delporte-Gallet, C., Fauconnier, H., and Toueg, S. 2004. Communication-efficient leader election and consensus with limited link synchrony. In *23th ACM Symposium on Principles of Distributed Computing*. 262–269.

Attiya, H., Bar-Noy, A., and Dolev, D. 1995. Sharing memory robustly in message passing systems. *42,* 2 (Jan.), 124–142.

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

Bazzi, R. A., Neiger, G., and Peterson, G. L. 1997. On the use of registers in achieving wait-free consensus. *Distributed Computing 10,* 3, 117–127.

Chandra, T. D., Hadzilacos, V., and Toueg, S. 1996. The weakest failure detector for solving consensus. *43,* 4 (July), 685–722.

Chandra, T. D. and Toueg, S. 1996. Unreliable failure detectors for reliable distributed systems. *43,* 2 (Mar.), 225–267.

Dwork, C., Lynch, N. A., and Stockmeyer, L. 1988. Consensus in the presence of partial synchrony. *35,* 2 (Apr.), 288–323.

Fischer, M. J., Lynch, N. A., and Paterson, M. S. 1985. Impossibility of distributed consensus with one faulty process. *32,* 2 (Apr.), 374–382.

Guerraoui, R. and Kouznetsov, P. 2003. On failure detectors and type boosters. In *Proceedings of the 17th International Symposium on Distributed Computing*. LNCS 2848. Springer-Verlag, 292–305.

Hadzilacos, V. and Toueg, S. 1993. Fault-tolerant broadcasts and related problems. In *Distributed Systems*, S. J. Mullender, Ed. Addison-Wesley, Chapter 5, 97–145.

Herlihy, M. and Wing, J. M. 1990. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst. 12,* 3, 463–492.

Herlihy, M. P. 1991. Wait-free synchronization. *ACM Trans. Program. Lang. Syst. 13,* 1 (Jan.), 123–149.

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

Jayanti, P. 1993. On the robustness of herlihy's hierarchy. In *12th ACM Symposium on Principles of Distributed Computing*. 145–157.

Lamport, L. 1978. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21,* 7 (July), 558–565.

[3]More specifically, we prove in Proposition 33 and Proposition 35 that, in a system of more than 2 processes, Ω is strictly stronger than Σ in the majority environment, and that they are incomparable in other environments. In a system of 2 processes, Ω and Σ are equivalent.

Lamport, L. 1986. On interprocess communication; part I and II. *Distributed Computing 1,* 2, 77–101.

Lamport, L. 1998. The Part-Time parliament. *ACM Transactions on Computer Systems 16,* 2 (May), 133–169.

Lo, W.-K. and Hadzilacos, V. 1994. Using failure detectors to solve consensus in asynchronous shared memory systems. In *Proceedings of the 8th International Workshop on Distributed Algorithms*. LNCS 857. 280–295.

Loui, M. and Abu-Amara, H. 1987. Memory requirements for agreement among unreliable asynchronous processes. *Advances in Computing Research 4*, 163–183.

Mostéfaoui, A., Mourgaya, E., and Raynal, M. 2003. Asynchronous implementation of failure detectors. In *International Conference on Dependable Systems and Networks (DSN)*. 351–360.

Neiger, G. 1995. Failure detectors and the wait-free hierarchy. In *14th ACM Symposium on Principles of Distributed Computing*.

Schneider, M. 1993. State machine replication. *ACM Computing Surveys 25,* 1.

Vitanyi, P. and Awerbuch, B. 1986. Atomic shared register access by asynchronous hardware. In *Proceedings of the IEEE Symposium om Foundations of Computer Science*. 233–243.

Appendix

We complement here the core of the paper by proving some relations between failure detectors. We first prove some results about the composition of failure detectors, then we prove relations between different failure detectors, including classical ones from the literature.

Failure detector reductions

We recall here the notion of reduction between failure detectors. Remember that we assume by default a message passing service. A failure detector $\mathcal{D}1$ is reducible to a failure detector $\mathcal{D}2$ in an environment $\mathcal{E}$ if there is an implementation of $\mathcal{D}2$ using $\mathcal{D}1$ in $\mathcal{E}$. The implementation uses $\mathcal{D}1$ to maintain a variable $Output_p$, at every process p, that emulates a possible output of $\mathcal{D}2$ at p in any failure pattern of that environment [Chandra et al. 1996]. When query() is invoked by a process p (to access the value of failure detector $\mathcal{D}2$ that is emulated), we assume that query() returns the value of $Output_p$. The implementation is such that the variable $Output_p$ ensures the properties of $\mathcal{D}2$. Variable $Output_p$ does not need to emulate all histories of $\mathcal{D}2$. It is required however that, for every run $\alpha =< F, C, Sc, T >$ with associated failure detector history H, where $H \in \mathcal{D}1(F)$ and F is in the environment, the history emulated by $Output_p$ in the run α be a history of $\mathcal{D}2(F)$.

Technical lemma

Before proving our results, we give a technical lemma about runs of algorithms using failure detectors. To state and prove this lemma, we introduce some notations and definitions. If $\alpha = \langle F, C, Sc, T\rangle$ denotes a run and τ a time, then $T^{-1}[\tau]$ denotes the first step (if any) corresponding to time τ: $T^{-1}[\tau] = \min\{n|T[n] \geq \tau\}$.

If H and H' are failure detector histories, H and H' are said to be identical up to time τ if for every process p and time $\tau' \leq \tau$, we have: $H(p, \tau') = H'(p, \tau')$. Given two runs $\alpha = \langle F, C, Sc, T\rangle$ with associated failure detector history H and $\alpha' = \langle F', C, Sc', T'\rangle$ with associated failure detector history H', we say that α and α' are identical up to m if H and H' are identical up to time $T[m]$ and for all $m' \leq m$ we have $Sc[m'] = Sc'[m']$ and $T[m'] = T'[m']$.

By extension, we say that α and α' are identical up to time τ, if α and α' are identical up to $T^{-1}[\tau]$. Note that if α and α' are identical up to time τ, then up to time τ, the two runs are indistinguishable for all processes that are alive.

Lemma 29. *Given a failure pattern F and a failure detector $\mathcal{F}$, let $\alpha_m = \langle F, C, Sc_m, T_m\rangle_{m\in\mathcal{N}}$, $(H_m)_{m\in\mathcal{N}}$, and $(\tau_m)_m \in \mathcal{N}$ an strictly increasing sequence of times such that for every $m \in \mathcal{N}$:*

(1) *H_m is a failure detector history in $\mathcal{F}(F)$ and α_m is a run of algorithm A for failure detector history H_m;*

(2) *α_m with associated failure detector history H_m and α_{m+1} with associated failure detector history H_{m+1} are identical up to time τ_m;*

(3) *every correct process takes at least one step between time τ_m and τ_{m+1};*

(4) *all messages sent by correct processes to correct processes by time τ_m in α_m are received in α_{m+1} by time τ_{m+1} ;*

(5) *failure detector history H identical to H_m up to time τ_m is a failure detector history of $\mathcal{F}(F)$;*

Then $\alpha = \langle F, C, Sc, T \rangle$, such that for, all m, α is identical to α_m up to time τ_m, is a run of A for failure detector $H \in \mathcal{F}(F)$.

PROOF. Remark first that as all α_m with failure detector history H_m and $\alpha_{m'}$ with failure detector history $H_{m'}$ are identical up to time $\min(\tau_m, \tau_{m'})$, α and H are well defined. Let l be $T^{-1}[\tau]$; as α is identical to α_m up to l, $Sc_m[l] = Sc[l]$ and $(Sc[1] \ldots Sc[l-1])(C) = (Sc_m[1] \ldots Sc_m[l-1])(C)$, then $Sc[l]$ is a step of a process $p \notin F(T[l])$ and is applicable to $(Sc[1] \ldots Sc[l-1])(C)$. Then condition (3), (4) of the lemma ensure that α is a run of algorithm A for failure detector history H. □

Compositions

We have shown in the paper (Proposition 1) that for any subset S of Π, for any family $\mathcal{L}$ of subsets of S such that, for all $p, q \in S$, there exists $L \in \mathcal{L}$ such that $p \in L$ and $q \in L$; We have: $\Sigma_S \equiv *\{\Sigma_X | X \in \mathcal{L}\}$.

We have also shown a slightly weaker composition property for Ω_S. Namely, we proved the following (Proposition 3). Let $\mathcal{L}$ be any family of subsets of Π such that, for all $p, q \in \Pi$, there exists some $L \in \mathcal{L}$ such that $p \in L$ and $q \in L$. We have: $\Omega \equiv *\{\Omega_L | L \in \mathcal{L}\}$.

We now complete the picture by proving the following property showing that it is not possible to restrict $\mathcal{L}$ to be a family of subsets of S:

PROPOSITION 30. *There exist a system of n processes, a non-empty subset of Π, S, an environment $\mathcal{E}$ and a set of failure detectors $\Omega_{\{p,q\}}$, for all p, q in S, such that $\Omega_S \not\equiv *\{\Omega_{\{p,q\}} \mid p, q \in S\}$ in $\mathcal{E}$.*

PROOF. Consider $n = 6$, $\Pi = \{a, b, c, X, Y, Z\}$, the environment of all failure patterns where 3 processes might fail, and $S = \{a, b, c\}$.

By contradiction, assume there exists an algorithm A that implements Ω_S from $*\{\Omega_{\{p,q\}} \mid p, q \in S\}$. Basically, algorithm A eventually outputs the same correct leader at all correct processes of S.

By assumption, for each run α of A, there is a correct process l, denoted $leader(\alpha)$, and a time after which for all correct processes of S A outputs l. Let τ_α be the first time such that for all time τ', $\tau' \geq \tau_\alpha$, A outputs $leader(\alpha)$ for all correct processes of S.

Consider the failure free failure pattern $F_\emptyset$, and a history H of $*\{\Omega_{\{p,q\}} \mid p, q \in S\}$ such that:

—$\Omega_{\{a,b\}}$ outputs Z for a and b,
—$\Omega_{\{a,c\}}$ outputs Y for a and c,
—$\Omega_{\{c,b\}}$ outputs X for c and b,
—and all these failure detectors output $\emptyset$ for all other processes.

In order to obtain a contradiction, we build a run of A with no leader: in this run the leader changes infinitely often.

In the following, all runs we consider are runs of A with failure detector history H.

LEMMA 31. *Let any* $\alpha =< F_\emptyset, C, Sc, T >$ *be a run of* A *associated with failure detector history* H*, and any time* τ*, there is a time* $s(\tau) > \tau$ *and a failure free run* $s(\alpha,\tau)$ *identical to* α *up to time* τ *such that (1)* A *outputs as leader for processes in* S *a process different from* $leader(\alpha)$ *between time* τ *and time* $s(\tau)$ *and (2) each process takes at least one step between time* τ *and time* $s(\tau)$ *(3) all messages sent to correct processes by time* τ *are received by time* $s(\tau)$*.*

PROOF. We consider two cases: either $leader(\alpha)$ is in S or $leader(\alpha)$ is not in S.

For the first case, assume without loss of generality that $leader(\alpha)$ is a. If $\tau < \tau_\alpha$ there is nothing to prove. Then assume that $\tau \geq \tau_\alpha$ and therefore we assume that for all time greater or equal to τ $leader(\alpha)$ is a. Consider the failure pattern F_0: in F_0 only a is faulty and crashes at some time $\tau_0 > \tau$. H is also a failure detector history of $*\{\Omega_{\{p,q\}} \mid p,q \in S\}(F_0)$.

Consider now a run $\beta_0 =< F_0, C, Sc_0, T >$ of A with associated failure detector history H. In β_0, the processes take the same steps at the same time as in α, up to time τ_0. Thus, for every process, β_0 and α are indistinguishable up to time τ_0. Hence A outputs a from time τ to time τ_0. As a consequence, there is time τ_1 after which A outputs forever $leader(\beta_0)$. As $leader(\beta_0)$ is a correct process, we have $leader(\beta_0) \neq a$.

Consider now a failure free run $\beta_1 =< F_\emptyset, C, Sc_0, T >$ of A with associated failure detector history H. In β_1, the processes take the same steps at the same time as in β_0, until time τ_1, and at time τ_1, A outputs $leader(\beta_0) \neq a$ for some process of S. Let $\tau_2 > \tau_1$ be a time for which all messages sent before τ are received and all processes of Π have taken at least one step since τ. Hence β_1 is a failure free run, identical to α, up to time τ. Let $s(\tau)$ be τ_2 and $s(\alpha,\tau)$ be run β_1, $s(\tau)$ and $s(\alpha,\tau)$ verify properties (1), (2) and (3) of the lemma.

For the second case, assume without loss of generality that $leader(\alpha)$ is X. As before we assume that $\tau \geq \tau_\alpha$. Consider the failure pattern F'_0: in F'_0 only b, c, X are faulty and crash at some time $\tau'_0 > \tau$ (note that only $\Omega_{\{b,c\}}$ outputs A for b and c). H is also a failure detector history of $*\{\Omega_{\{p,q\}} \mid p,q \in S\}(F'_0)$.

Consider now run $\beta'_0 =< F'_0, C, Sc'_0, T >$ of A with associated failure detector history H. In β'_0, the processes take the same steps at the same time as in α, up to time τ'_0. Hence, for each process β'_0 and α are indistinguishable up to time τ'_0. Therefore, A outputs A from time τ to time τ'_0, and there is time τ'_1 after which A outputs forever $leader(\beta'_0)$ for a. As $leader(\beta'_0)$ is a correct process, $leader(\beta'_0) \neq X$.

Consider now a failure free run $\beta'_1 =< F_\emptyset, C, Sc'_1, T >$ of A with associated failure detector history H. In β'_1, the processes take the same steps at the same time as in β'_0, until time τ'_1. At τ'_1, A outputs $leader(\beta'_0) \neq X$ for some process of S. Let time $\tau'_2 > \tau'_1$ be a time for which all messages sent before τ are received and all processes of Π have taken at least one step since τ. Hence β'_1 is a failure free run, identical to α up to time τ. Let $s(\tau)$ be τ'_2 and $s(\alpha,\tau)$ be run β'_1, $s(\tau)$ and $s(\alpha,\tau)$ verify properties (1) (2) and (3) of the lemma. $\square$

By induction, construct runs γ_m and time τ_m as follows:

—Run $\gamma_0 =< F_\emptyset, C, Sc, T >$ with associated failure detector history H is a failure free run of A and $\tau_0 = 0$.

—if $leader(\gamma_m)$ is in S, then γ_{m+1} is $s(\gamma_m, \tau_m)$ and $\tau_{m+1} = s(\tau_m)$;
if $leader(\gamma_m)$ is not in S, then γ_{m+1} is $s'(\gamma_m, \tau_m)$ and $\tau_{m+1} = s'(\tau_m)$.

By construction, γ_m is identical, up to time τ_m, to every γ'_m for $m' \geq m$. Let γ be such that for all m, γ is identical to γ_m up to time τ_m. By Lemma 29, γ is a run of A for failure detector history H. But, in γ, the output of A changes infinitely often for at least one correct process in S, contradicting the property of Ω_S.

Reductions

We now compare different failure detectors. We prove some reduction relations, which we summarize in Figure 6.

We consider in particular the failure detectors introduced in [Chandra and Toueg 1996; Chandra et al. 1996]. All have range $R = 2^{\Pi}$. That is, all output, at any time and any process, a subset of processes in the system. For any failure pattern F, and any history H, $H(p, \tau)$ is the set of processes that are said to be *suspected* by process p at time τ.

(1) The *Perfect* failure detector ($\mathcal{P}$) ensures *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 *Strong* failure detector ($\mathcal{S}$) ensures *strong completeness* and *weak accuracy*, i.e., some correct process is never suspected.
(3) The *Eventually Strong* failure detector ($\diamond\mathcal{S}$) ensures *strong completeness* and *eventual weak accuracy*, i.e., eventually, some correct process is never suspected.

We also consider here environments where less than a certain number t of process might fail, denoted by $\mathcal{E}_t$.

To simplify the presentation of our results, we denote by $\perp$ the constant failure detector that always output the same value $\perp$: the fact that this failure detector can be stronger than some failure detector $\mathcal{D}$ in some environment simply captures the fact that $\mathcal{D}$ can be implemented with an asynchronous algorithm (in that environment).

First we prove that Σ can be implemented if a majority of processes is correct (remember that we assume by default message passing):

Proposition 32. *The algorithm of Figure 7 emulates, within variable $Output_p$ for each p, the behavior of failure detector Σ, assuming a majority of correct processes.*

Proof. Consider a failure pattern F where the number of faulty processes is strictly less than $n/2$. In F, no correct process waits forever in line 7 and therefore r_p is unbounded for any such process.

(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 τ be the time for which variable r_p is greater than x. After time τ, q does not belong to $Output_p$.

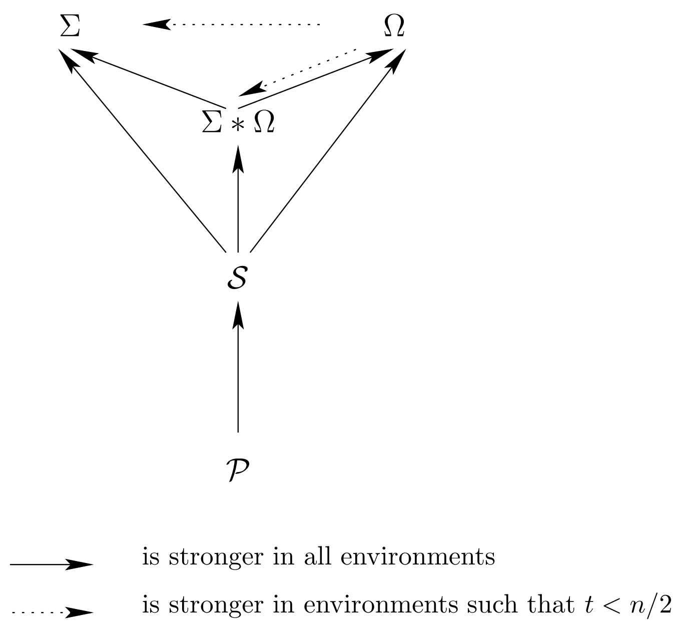

Fig. 6. Reductions

(2) **Intersection:** 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 $Output_q$.

□

```
Every process p executes the following code:
  Initialization:
    r_p := 0
  Task 1:
    repeat forever
      send(ARE_YOU_ALIVE, r) to all
      wait until receive (I_AM_ALIVE, r) from majority of processes
      Output_p := {q | a message (I_AM_ALIVE, r) from q received by p }
      r_p := r_p + 1
  Task 2:
    upon receive (ARE_YOU_ALIVE, x) from q
    send(I_AM_ALIVE, x) to q
```

Fig. 7. Implementing Σ with a correct majority

Proposition 33. *In every environment:*

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

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

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

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

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

(*5*) $\Sigma \preceq \bot$*;*

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

(*7*) $\Sigma * \Omega \preceq \Omega$.

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

(*8*) $\mathcal{S} \equiv \Sigma$.

Proof. (1) follows directly from the definitions;

(2) follows from the fact that $\Sigma * \Omega$ 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 Proposition 32;

(6) and (7) follow from (5);

(8) by definition, $\mathcal{S}$ ensures *strong completeness*. By definition of $\mathcal{S}$ too, some correct process is never suspected and so $\mathcal{S}$ ensures the *intersection* property. Then $\Sigma \preceq \mathcal{S}$.
Denote by p_1 and p_2 the two processes of the system. Consider any failure pattern F. If no process crashes in F, then by the *intersection* property of Σ, one correct process is trusted forever by p_1 and p_2: each process trusts itself and either p_1 trusts p_2 forever or p_1 does not trust p_2 at some time and then by intersection property p_2 trusts p_1 forever. If some process, say p_1, crashes, then by the *completeness* property of Σ, after some time τ, p_2 is the only process trusted by p_2. By the *intersection* property of Σ, p_2 has been trusted forever by p_2. Therefore, in all cases, at least one correct process is never suspected. This proves the *accuracy* property of $\mathcal{S}$. Hence $\mathcal{S} \preceq \Sigma$.

□

Proposition 34. *For* $n = 2$, $\Sigma \equiv \Sigma * \Omega$.

Proof. We have shown in Proposition 33 that for $n = 2$, $\Sigma \equiv \mathcal{S}$. As $\mathcal{S}$ implements consensus [Chandra and Toueg 1996], from proposition 16, we get: $\Sigma \preceq \Sigma * \Omega \preceq \mathcal{S}$. □

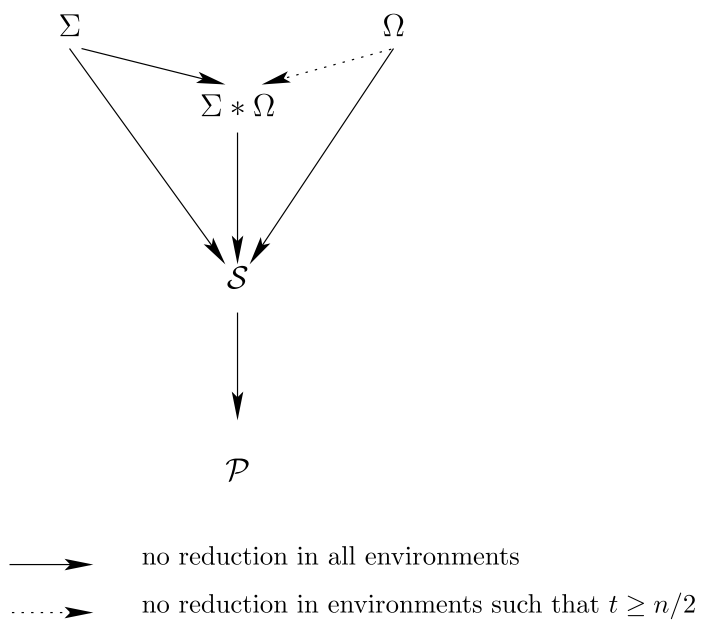

Fig. 8. No reductions

Non-reductions

We show here that some failure detectors are not reducible to others. Our results are summarized on Figure 8.

Lemma 35. *There is no environment $\mathcal{E}_t$, $0 < t < n$, $n > 2$, for which $\Omega \preceq \Sigma$.*

Proof. Consider $F_\emptyset$ the failure free failure pattern.

With Ω, at each time τ, each process p trusts only one process that is called the *leader* for p at time τ. The properties of Ω ensure that for each run there is a time after which all correct process have the same leader forever and this leader is a correct process.

By contradiction assume that there exists a reduction algorithm A from Σ to Ω.

Let $\alpha^p_\tau = \langle F_\emptyset, C, Sc, T\rangle$ with associated failure detector history H be a run of A in which H is a failure detector history of Σ such that, after time τ, all processes but p are trusted forever by all other processes, then we have:

Lemma 36. *For every such run $\alpha^p_\tau = \langle F_\emptyset, C, Sc, T\rangle$ with associated failure detector history H of A, there exists a run $\alpha'_\tau = \langle F_\emptyset, C, Sc', T\rangle$ with associated failure detector history H of A and a time $\tau' > \tau$ such that p is not leader for some correct process at time τ'.*

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

Proof. (of lemma) Consider a run $\alpha = \langle F, C, Sc, T\rangle$ of A with associated failure detector history H that is identical to α^p_τ, up to time τ, such that in F, all processes but p are correct and p crashes at time $\mu > \tau$. As α is a run of A, there is some time $\tau' > \mu$ such that p is not the leader for some correct process q. Let $\alpha' = \langle F_\emptyset, C, Sc', T\rangle$ with associated failure detector history H be a run identical to α up to time τ'. For q, α and α' and indistinguishable until time τ', and hence at time τ', 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 Σ.

We construct by induction a strictly increasing infinite sequence of times $(\tau_i)_{i\in\mathcal{N}}$ and an infinite sequence of $(\alpha_i = \langle F_\emptyset, C, Sc_i, T\rangle)_{i\in\mathcal{N}}$ runs of A with associated failure detector history H_i such that for all i:

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

. (Q2) $p_{1+(i \mod n)}$ is not leader for some correct process at time τ_i in run α_i

. (Q3) if $i > 0$ then α_i and α_{i-1} are identical up to time τ_{i-1}

. (Q4) if $i > 0$ then every message sent by time τ_{i-1} in α_i is received by time τ_i and, between time τ_{i-1} and time τ_i, every process makes at least one step

—(basis: $i = 0$) Let $\alpha'_0 = \langle F_\emptyset, C, Sc'_0, T\rangle$ with associated failure detector history H_0 be any run of A such that for all τ and all p $H_0(p,\tau) = E_1$.
By the previous lemma, there is a run $\alpha_0 = \langle F_\emptyset, C, Sc_0, T\rangle$ with associated failure detector history H_0 of A such that at some time τ_0, p_1 is not leader for some correct process.
(Q1) and (Q2) are ensured by construction, (Q3) and (Q4) are trivially verified.

—(induction: $i > 0$) Assume we have α_i and τ_i verifying (Q1), (Q2), (Q3) and (Q4).
Consider $\alpha_i = \langle F_\emptyset, C, Sc_i, T\rangle$ with associated failure detector history H_i, there is a time $\tau' > \tau_i$ such that by time τ' every message sent before τ_i is received and each process takes at least one step between τ_i and τ'.
Let $\alpha' = \langle F_\emptyset, C, Sc', T\rangle$ with associated failure detector history H_{i+1} be a run of A identical to α_i with associated failure detector H_i up to time τ', but after time τ', H_{i+1} outputs always $E_{1+((i+1) \mod n)}$. By the previous lemma, there is a run $\alpha_{i+1} = \langle F_\emptyset, C, Sc_{i+1}, T\rangle$ with associated failure detector history H_{i+1} identical to α' up to time τ' such that, at time $\tau_{i+1} > \tau'$, $p_{1+(i \mod n)}$ is suspected by some correct process. Clearly (Q1), (Q2), (Q3) and (Q4) hold.

Let $\alpha_{lim} = \langle F_\emptyset, C, Sc_{lim}, T\rangle$ with associated failure detector history H_{lim} identical up to τ_i to run α_i with associated failure detector history H_i for all i. By (Q1) the output of failure detectors in α_{lim} is always one of the E_i, and then $H_{lim} \in \Sigma(F_\emptyset)$. From (Q3) and (Q4) and Lemma 29, α_{lim} is a run of A for failure detector history H_{lim}.

But by (Q2) every process in Π is infinitely often not a leader for at least one correct process, contradicting the property of Ω.

LEMMA 37. *In every environment $\mathcal{E}_t$, $0 < t < n$, $n > 2$, $(\Sigma * \Omega) \prec \mathcal{S}$.*

PROOF. As in any environment, $\Sigma*\Omega$ is the weakest failure detector for consensus and, in any environment, consensus can be implemented with $\mathcal{S}$, then $\Sigma * \Omega \preceq \mathcal{S}$.

Now, we prove that there is no reduction from $\Sigma * \Omega$ 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 A such that for each failure pattern $F \in \mathcal{E}_t$ and failure detector history $H \in \Sigma * \Omega(F)$, outputs a failure detector history $H_{\mathcal{S}} \in \mathcal{S}(F)$.

In the following, if $\alpha = \langle F, C, Sc, T\rangle$ with associated failure detector history $(H^0, H^1) \in \Sigma * \Omega(F)$ is a run of a reduction algorithm A, the failure detector history in $\mathcal{S}(F)$ outputs by this run is denoted by $A(\alpha)$.

Consider the following finite sequence of runs α_i $(1 \leq i \leq n)$ and sequence of times τ_i $(1 \leq i \leq n)$ such that for all $1 \leq i \leq n$:

$$\text{if } i > 1 \text{ then } \alpha_i \text{ is identical up to time } \tau_{i-1} \text{ to } \alpha_{i-1} \tag{1}$$

$$p_i \text{ is suspected by all live processes at time } \tau_i \text{ in } A(\alpha_i) \tag{2}$$

—(basis) In failure pattern F_1, p_1 crashes at time 0 and no other process crashes; in run $\alpha_1 = \langle F_1, C, Sc_1, T\rangle$ with associated failure detector history (H_1^0, H_1^1) for H_1^0 every alive process trusts all processes in Π but p_1. By the *completeness* property of $\mathcal{S}$, there is a time τ_1 after which, in $A(\alpha_1)$, p_1 is suspected by all processes. H_1^1 outputs any process, say p_1, up to time τ_1 and then the same correct process forever. (1) is trivially verified and (2) is true by construction.

—(induction step) $(1 \leq i < n)$
Assume that we have built $\alpha_i = \langle F_i, C, Sc_i, T\rangle$ with associated failure detector history (H_i^0, H_i^1) and τ_i:
Then $\alpha_{i+1} = \langle F_{i+1}, C, Sc_{i+1}, T\rangle$ with associated failure detector history (H_{i+1}^0, H_{i+1}^1) a run of A such (1) that α_i and α_{i+1} are identical up to time τ_i and (2) at time $\tau' > \tau_i$ p_{i+1} crashes and no other process crashes and (3) after time τ' in H_{i+1}^0 every process trusts forever every process in $\Pi - \{p_{i+1}\}$ and (4) H_{i+1}^1 is identical to H_i^1 up to time τ_i then from time τ_i to time τ_{i+1} outputs any process, say p_1 for every process and then a correct process, in this way H_{i+1}^1 is a failure detector history of $\Omega(F_{i+1})$.
Remark that in α_{i+1} only process p_{i+1} is not correct.
By an easy induction, we deduce that at each time τ 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 τ, τ' and all p, p' $H_{i+1}^0(p, \tau) \cap H_{i+1}^0(p', \tau') \neq \emptyset$ proving that $H_{i+1}^0 \in \Sigma(F_{i+1})$ and hence α_{i+1} is a run of A.
By construction (1) and (2) are true.

Up to time τ_i α_{i+1} and α_i are indistinguishable for all processes but p_{i+1}. Then, p_i being suspected by every other process at time τ_i in $A(\alpha_i)$ is also suspected at time τ_i in $A(\alpha_{i+1})$. This proves (2). 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 $\alpha_n = \langle F_n, C, Sc_n, T\rangle$ with associated failure detector history (H_n^0, H_n^1), α_n is a run of A. By an easy induction and (1), for all $0 \leq i < n$, α_n and α_i are identical up to time τ_i. Hence from (2) in $A(\alpha_n)$, each process p_i in Π is suspected at time τ_i, contradicting the *accuracy* property of $\mathcal{S}$.

□

PROPOSITION 38. *We have:*

(*1*) $\neg(\Omega \preceq \Sigma)$ *in any environment* $\mathcal{E}_t$ *such that* $0 < t < n$ *and* $n > 2$*;*
(*2*) $\neg(\mathcal{S} \preceq \Sigma)$ *in any environment* $\mathcal{E}_t$ *such that* $0 < t < n$ *and* $n > 2$*;*
(*3*) $\neg(\Sigma \preceq \Omega)$ *in any environment* $\mathcal{E}_t$ *such that* $n/2 \leq t < n$*;*
(*4*) $\neg(\mathcal{S} \preceq (\Sigma * \Omega))$ *in any environment* $\mathcal{E}_t$ *such that* $0 < t < n$ *and* $n > 2$*;*
(*5*) $\neg((\Sigma * \Omega) \preceq \Omega)$ *in any environment* $\mathcal{E}_t$ *such that* $n/2 \leq t < n$*;*
(*6*) $\neg((\Sigma * \Omega) \preceq \Sigma)$ *in any environment* $\mathcal{E}_t$ *for* $0 < t < n$ *and* $n > 2$.

PROOF. (1) from Lemma 35;
(2) As $\Omega \preceq \mathcal{S}$, if $\mathcal{S} \preceq \Sigma$, we would have $\Omega \preceq \Sigma$ contradicting (1);
(3) Recall that in environment $\mathcal{E}_t$ with $n/2 \leq t < n$, Ω is not sufficient to implement Consensus [Chandra and Toueg 1996]. If Ω were reducible to Σ in an environment $\mathcal{E}_t$ $(n/2 \leq t < n)$ then Ω could be reducible to $\Sigma * \Omega$, and then Consensus could be solvable with Ω in such an environment –a contradiction;
(4) From Lemma 37;
(5) If there were a reduction from Ω to $\Sigma * \Omega$ in environments $\mathcal{E}_t$ such that $n/2 \leq t < n$, we would get a reduction from Ω to Σ contradicting (3);
(6) If there were a reduction from Σ to $\Sigma * \Omega$ in environments $\mathcal{E}_t$ for $0 < t < n$, then we would have a reduction from Σ to Ω in such environments contradicting (4).

□