Tight Failure Detection Bounds on Atomic Object Implementations

CAROLE DELPORTE-GALLET AND HUGUES FAUCONNIER

LIAFA Univ Paris-Diderot, France

AND

RACHID GUERRAOUI

EPFL Lausanne, Switzerland

Abstract. This article 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. We then 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

ACM Reference Format:

Delporte-Gallet, C., Fauconnier, H., and Guerraoui, R. 2010. Tight failure detection bounds on atomic object implementations. J. ACM 57, 4, Article 21, (April 2010), 32 pages.
DOI = 10.1145/1734213.1734216 http://doi.acm.org/10.1145/1734213.1734216

1. *Introduction*

A shared atomic object is a data structure exporting a set of operations that can be invoked by concurrent processes. *Atomicity* means that every object operation

---

Authors' addresses: C. Delporte-Gallet and H. Fauconnier, Laboratoire d'Informatique Algorithmique: Fondements et Applications, CNRS UMR 7089, Université Paris Diderot, Paris 7, Case 7014, 75205 Paris Cedex 13 France, e-mail: {Carole.Delporte, Hugues.Fauconnier}@liafa.jussieu.fr; R. Guerraoui, EPFL IC IIF LPD, INR 310 (Bátiment INR), Station 14, CH-1015 Lausanne, Switzerland, e-mail: rachid.guerraoui@epfl.ch.

DOI 10.1145/1734213.1734216 http://doi.acm.org/10.1145/1734213.1734216

appears to execute at some individual instant between its invocation and reply time events [Lamport 1986; Herlihy and Wing 1990]. Many distributed algorithms are designed assuming atomic objects as underlying synchronization primitives. These include objects of types register, test-and-set, fetch-and-add, queue, consensus, and compare-and-swap.

We study necessary and sufficient conditions for implementing atomic object types in software assuming processes can communicate by message passing, that is, with no actual shared physical memory. Through such implementations, algorithms based on shared atomic objects can be automatically emulated in a message passing system.

CONTEXT. We consider a distributed system where processes communicate through reliable channels but can fail by crashing. If it crashes, a process halts its activities. Otherwise, it does not deviate from the algorithm assigned to it. We study *robust* [Attiya et al. 1995] implementations where any process that invokes an object operation and does not crash eventually gets a reply.

If the distributed system provides no information about failures, then two fundamental results are known about atomic object implementations. (1) The type register can be implemented if and only if a minority of the processes can crash [Attiya et al. 1995], and (2) we cannot implement any of the types test-and-set, fetch-and-add, queue, compare-and-swap and consensus, even if only one process may crash [Herlihy 1991; Fischer et al. 1985; Loui and Abu-Amara 1987]. On the other hand, if we can assume a *perfect* failure detection mechanism that provides the processes with the ability to accurately detect crashes, then all these objects can be implemented irrespective of the number of processes that can crash.

It is natural thus to ask what amount of failure information is actually necessary and sufficient to implement such atomic objects. The question can be expressed precisely using the notion of *failure detector reduction* introduced in Chandra and Toueg [1996]. Failure detectors can indeed be viewed as abstract oracles that output information about crashes, and they can be precisely compared. Basically, 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 the output of $\mathcal{D}'$ ($\mathcal{D}'$ is said to be *weaker* than $\mathcal{D}$) [Chandra and Toueg 1996].

It was shown in Chandra et al. [1996] that, assuming only a minority of processes can crash, the *weakest* failure detector to implement consensus is an oracle which outputs, at any time and at every process, a single leader process such that, eventually, this leader does never crash and is permanently the same at all processes [Chandra et al. 1996]. The meaning that Ω is the weakest to implement consensus (assuming only a minority of processes can crash) is twofold: (a) there is a distributed algorithm that implements consensus using Ω (assuming only a minority of processes can crash), and (b) for every failure detector $\mathcal{D}$ such that some algorithm implements consensus using $\mathcal{D}$, $\mathcal{D}$ is stronger than Ω. Given that the compare-and-swap type can emulate the consensus type, and consensus can emulate any atomic object type if only a minority of processes can crash [Herlihy 1991], Ω is thus also the weakest to implement the compare-and-swap type if only a minority of processes can crash.

The general questions remained however open:

—What is the weakest failure detector for all other object types? For instance, types like queue, test-and-set, or fetch-and-add are, in a precise sense, *less* *powerful* than consensus: they can emulate consensus in a subsystem of two processes but cannot in any subsystem of more than two processes (unlike compare-and-swap [Herlihy 1991]). These types have *consensus number* 2 in the parlance of Herlihy [1991]. On other hand, compare-and-swap has *consensus number n* in any system of n processes. It is natural to seek for the weakest failure detector to implement objects with consensus number $k < n$; one would expect such a weakest failure detector to be strictly weaker than Ω.

—What if half of the processes can crash? As we pointed out, the basic type register cannot be implemented if there is no failure information and half of the processes might crash: it is thus natural to seek for the weakest failure detector to implement the register type in case half of the processes can crash? In this case, it is also known that Ω does not implement consensus [Chandra and Toueg 1996]. So what is the weakest failure detector to implement consensus (and other object types) if half of the processes might crash?

CONTRIBUTIONS. This article closes the general questions above. We determine the weakest failure detectors to implement the basic type register as well as any object type with consensus number $1 < k \leq n$, that is, including types like consensus, compare-and-swap, queue, test-and-set and fetch-and-add. We do so in *any environment* [Chandra and Toueg 1996], that is, given any assumption about the number and timing of process failures, and for any subset of processes in the system.

We proceed as follows. Considering *any* environment and *any* subset S of processes in the system:

(1) We first determine (1) the weakest failure detector to implement a register shared by processes in S, and then 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 processes that never crash.
   —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 does never crash.

(2) We then show that for any integer $1 < k \leq n$, the weakest failure detector to implement any type shared by processes in S that emulates consensus among k processes is also $\Sigma_S * \Omega_S$.

INTERPRETATIONS

—Failure detector Σ_S encapsulates the exact information about failures needed to implement a basic shared memory abstraction made by registers 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, assuming at most a minority of processes can crash, Σ, the restriction of Σ_S to the case where S is the entire system, can indeed be implemented directly with message passing (with no failure information).

—Identifying failure detector $\Sigma_S * \Omega_S$ generalizes, for any subset of processes and any environment, the fundamental result of Chandra et al. [1996]. Indeed, assuming at most a minority of processes can crash, $\Sigma * \Omega$, the restriction of $\Sigma_S * \Omega_S$ to the case where S is the entire system, is equivalent to Ω [Chandra et al. 1996].

—Our result that, for any $1 < k \leq n$, the weakest failure detector to implement any type that emulates consensus among k processes is also $\Sigma * \Omega$, reveals the interesting fact that the notion of consensus number [Herlihy 1991; Jayanti 1993] (as long as it is strictly higher than 1) of a type has no impact on the information about failures needed to implement this type. For instance, 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. More generally, and given that most synchronization problems can be cast 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, that is, all type implementations.

ROADMAP. The rest of the article is organized as follows. Section 2 defines our model. Section 3 introduces failure detectors Σ_S and Ω_S, then 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 compares failure detectors Σ and Ω. Section 7 concludes the article.

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. A process does never deviate from the algorithm assigned it (no Byzantine failures) except if it crashes, in which case it simply halts any activity. 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. Failures are permanent, that is, no process recovers after a crash. 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. The environment consisting of the set of failure patterns where at most t processes crash $(0 < t \leq n)$ is denoted $\mathcal{E}_t$.

Roughly speaking, a failure detector $\mathcal{D}$ is a distributed oracle that 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, that is, $H(p, \tau)$ 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 $seq = a_1 \cdots a_i a_{i+1}$ 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 $a_{i+1}((a_1 \ldots a_i)(C))$. To each schedule $Sc = a_1 \cdots a_i a_{i+1} \cdots$ and configuration C_0 correspond a unique sequence of configurations $C_0 C_1 \cdots C_i C_{i+1} \cdots$ 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 kth 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, by abuse of language, we simply denote the restrictions to the process p_i by $Sc|p_i$, $T|p_i$, and $\alpha|p_i$.

2.3. SERVICES. In this article, we consider three kinds of services: *message passing* channels, *atomic objects* and *failure detectors*. 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.

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 exporting a set of operations 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 an initial state of the object as well as a type.

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 *op* 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 $\sigma = (o_0r_0)(o_1r_1)\cdots(o_jr_j)\cdots$ 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.

We say that some occurrence of invocation is pending in a schedule if it has no matching reply in that schedule. Consider a schedule Sc, and its restriction $Sc|p$ to a process p. We say that 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. Only well formed schedules and runs are considered.

When reasoning about the atomicity of an object, we consider only operations that terminate, that is, both invocation *inv* and the 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 nonpending 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 $A =< A_1, \ldots, A_n,$ $Serv >$, 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 $Sc = Sc[1]Sc[2] \cdots Sc[k] \cdots$ is applicable to a configuration C if for each $k > 1$, $Sc[k]$ is applicable to configuration $(Sc[1] \cdots Sc[k-1])(C)$. 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 returns 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 $I(X, n, Serv) =< (X_1, (ps_1^1, \ldots, ps_m^1)), \ldots, (X_n, (ps_1^n, \ldots, ps_m^n)) >$ 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 $A =< A_1, \ldots, A_n, Serv' \cup \{X\} >$, the corresponding algorithm $A' =< A'_1, \ldots, A'_n, Serv \cup Serv' >$ 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 $A_1, \ldots, A_n$, are runs of A.

In this article, we study robust implementations of services [Attiya et al. 1995]: 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 implementations, 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 $\mathcal{D}1 \preceq_{\mathcal{E}} \mathcal{D}2$.

We say that a failure detector $\mathcal{D}_1$ and MP implement a given service (in environment $\mathcal{E}$) if there is an algorithm that uses $\mathcal{D}_1$ and MP 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, most of our results hold for all environments. Hence, unless explicitly stated otherwise, we will not assume any specific environment In particular, we use the notation $\mathcal{D}2 \preceq \mathcal{D}1$ to mean $\mathcal{D}2 \preceq_{\mathcal{E}} \mathcal{D}1$ in every environment $\mathcal{E}$. 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.

3. *The Quorum and Leader Failure Detectors*

We introduce here two new 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 failure detector Ω introduced in Chandra et al. [1996].

We prove some properties of the composition of these failure detectors, which will be useful in proving some of the main results of this article (Corollary 7 and Corollary 2).

3.1. FAILURE DETECTOR Σ_S. Basically, 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 (a) every list intersects with every other list, ever output at any time and any process, and (b) eventually, all lists contain only correct processes.

More precisely, failure detector Σ_S outputs, to processes in S, 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: $\forall F \in \mathcal{E}, \forall H \in \Sigma_S(F), \forall p \in S \cap \mathit{correct}(F), \exists \tau \in \Phi, \forall \tau' > \tau \in \Phi : H(p,\tau') \subseteq \mathit{correct}(F)$.

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 (we give a simple algorithm in Section 6).

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, that is, 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, assuming at least one correct process in the system, the following property is satisfied:

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

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 $V = \mathit{correct}(F)$, 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 $G|p =< W, F >$ the restriction of G to p: W 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 F 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 $q \in \mathit{Leader}_p^r$.

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

4. *The Weakest Failure Detector to Implement a Register*

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

Consider any subset S of processes in the system Π. We define a S-register as 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 is sometimes called a *multi-writer/multi-reader* register [Lamport 1986] (or simply a register).

We prove in this section the following result:

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

A direct corollary of this proposition is that Σ is the weakest failure detector to implement a register.

The rest of the section is about proving the proposition. Our proof is based on the existence of two algorithms.

(1) (Necessary condition) Our first algorithm, denoted R, emulates the output of failure detector Σ_S using any algorithm A that implements a S-register, that is, R extracts Σ_S from A. It is important at this point to notice that R does not use a S-register as a black-box, but it actually uses the algorithm implementing it. In some sense, R uses an *open* register that reveals information about its message passing implementation. The basic idea of algorithm R is the following. Every process $p \in S$ periodically writes in the S-register, triggering executions of A. For every such write w, process p tracks the processes that *participate* in w; namely, processes that send a A message that causally [Lamport 1978] follow the invocation of w and precede the return of w. As we will explain, this enables p to extract from A quorums of processes and emulate the output of failure detector Σ_S at p.

(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 simply use Σ_S.

4.2. PRELIMINARIES ABOUT REGISTERS. 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.

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 (respectively read) if the corresponding write (respectively 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], and Herlihy and Wing [1990], the correctness of an implementation of an atomic (p, S)-register can be conveniently expressed through three properties.

—(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.

4.3. NECESSARY CONDITION. We describe in the following our extraction algorithm R: this uses any algorithm A that implements a S-register 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 a 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 ensures that variable *Trust* satisfies the *completeness* and *intersection* properties of Σ_S.

When executing R, 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. Unless it crashes, process p goes through an infinite number of epochs: $1, 2, \ldots, k, \ldots$ At every epoch, p performs a *write* phase and then a *read* phase. The goal of these phases is to select a list of processes that are used to update $Trust_p$. We give a high-level description of these phases as well as their pseudo-code.

(1) *Write Phase*. Process p periodically initiates the writing in Reg_p of the current epoch number k (together with a specific value that we will discuss below). In turn, this writing (which we denote *write(k,*)*) triggers the execution of an instance of algorithm A (implementing Reg_p). Process p then tracks the messages it receives on behalf of A in order to select a list of *participating* processes denoted by $P_p(k)$. This set is determined by having every process that receives some message m in the context of *write(k,*)* from p, tags every message that causally [Lamport 1978] follows m, with (a) k, (b) p, as well as with (c) the list of processes from which messages have been received with those tags. When p terminates *write(k,*)*, it looks at all messages it received and gathers from those tagged with k the set $P_p(k)$ (to which p also belongs). There are two important properties of sets $P_p(k)$.
   - —If there is at least one correct reader, then $P_p(k)$ contains at least one correct process, for otherwise the value written could disappear and the reader would not be able to read it. Thus, if p is correct or at least one reader is correct, then $P_p(k)$ contains at least one correct process.
   - —Eventually, if p is correct, then there is a k after which every $P_p(k)$ contains only correct processes. This is because, after all faulty processes have crashed, the processes that participate in new write operations are necessarily correct.

(2) *Read Phase*. Every process p maintains the sequence, denoted E_p, of participating sets $P_p(k)$. Basically, before *write(k,*)* is performed on Reg_p, $E_p := \{P_p(0), P_p(1), P_p(2), \ldots, P_p(k-1)\}$. E_p is also the value written by process p in Reg_p together with epoch number k. (Initially, E_p contains exactly one set: the set of all processes Π, i.e., we assume that $P_p(0) = \Pi$.) After a process p writes E_p in Reg_p, p reads every register Reg_q written by every process q in S. Process p then sends $(ping, k)$ messages to all processes in every set $P_q(l)$ of E_q and waits for at least one ack message for each set

```
Code for every process p
/* each Σ_S failure detector query 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 is a temporary value of Trust */
   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(ping, 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(ping, k) from q send(k, ok) to q
```

FIG. 1. Implementing Σ_S using an *open* S-register.

$P_q(l)$. Process p then selects all processes that send such an ack message. We denote this set by $Q_p(k)$.

There are also two important properties to highlight here.

—If p is correct, then there is at least one correct reader of all registers and, as previously pointed out, every $P_q(k)$ contains at least one correct process. So, p indeed receives a message from one process in every set E_q and does not block forever. If process p is correct, then it terminates every *read* phase of every epoch k and determines a set $Q_p(k)$.

—Eventually, if p is correct, then there is an epoch k after which every $Q_p(k)$ contains only correct processes. This is because, after all faulty processes have crashed, the processes that respond to $(ping, k)$ messages are all correct.

At the end of epoch k, process p updates $Trust_p$ with the union of $P_p(k-1)$ and $Q_p(k)$. We thus have:

—*Completeness*. If p is correct, then p keeps permanently updating variable $Trust_p$. Eventually, $Trust_p$ contains only correct processes.

—*Intersection*. In $Q_p(k)$, for all processes q in S, there is at least one process of each set $Trust_q$ previously output by process q.

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 A. For modularity purposes, we present our algorithm R in two parts. Figure 1 shows how to emulate variable *Trust* using a specific *open* S-register with

```
/* 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.

a specification customized to our needs. Then, Figure 2 shows how to implement this specific *open* register with any algorithm that implements a S-register in a message passing system.

The *open* S-register has a traditional read but a nontraditional write operations. (The register has one writer so this nontraditional write can be performed by only one process.) The write has, besides any possible input parameter that the writer might want to store 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, that is, the epoch number. Furthermore, the write returns an output, which is the list of processes that *participated* in the write, that is, the processes that replied to messages sent on behalf of the underlying message passing algorithm A implementing the register.

We first define here more precisely what *participate* means. Let w be some write operation invoked by some writer process p in the white-box (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 A implementing this register (possibly using some failure detector). We tag every message causally after the beginning of the kth 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 kth write.

The following lemma states that the set of processes returned by the algorithm of Figure 2 at the end of the kth write by 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 kth write, we have:

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

PROOF. Let w the kth 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 jth write, with $j > k$, before the end of the kth 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 kth 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 store 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.

  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.

PROOF. Let p be any process in S. 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 kth 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 $(ping, *)$ from process p and process p cannot block on Line 18. Hence, p updates infinitely often variable $Trust_p$ and the sequences $Trust_p^1, \ldots Trust_p^m \ldots$ as well as $\tau_p^1, \ldots, \tau_p^m, \ldots$ 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 $(ping, m')$ 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 kth write of the register Reg_q by q is before the lth read of this register by process p, or the l-write of register Reg_p is before the kth read of this register by process q. Assume without loss of generality that p performs its lth read of register Reg_q after the kth 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 lth read of Reg_q, and (2) is such that p has received an (l, OK) answer from s. As we assume that the lth read is after the end of the kth 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 $\Sigma_S \preceq \mathcal{D}$.*

4.4. SUFFICIENT CONDITION. 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 describe 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] and 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 (respectively, write) operation, the reader p_r (respectively, the writer p_w) sends a message to all processes and waits until it receives acknowledgments from every process trusted by

p_r (respectively, p_w), that is, output by its failure detector module. It is important to notice that the set of processes trusted by the reader (respectively, the writer) might change between the time the reader (respectively, the writer) sends its message and the time it receives acknowledgments. We implicitly assume here that the reader (respectively, 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 (respectively, 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 (respectively, 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 kth one.

Assume that the kth 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 sth 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

```
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 $p_w, p_r \in S$.

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 an (*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 jth 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 kth 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 belongs 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 ath write has begun before read R has terminated, and by (C) the value returned by R is the value of the ath 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 jth 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 nondecreasing, we have $ry \geq rx$, hence, p_w wrote y after x. $\square$

Using the register transformations of Israeli and Li [1993] and 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 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, we define S-consensus as a consensus object 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 both Ω_S and Σ_S and then (sufficient condition) we exhibit an algorithm that implements S-consensus using $\Sigma_S * \Omega_S$.

(1) We prove the necessary condition in two steps. We show first how to extract Ω_S (necessary condition (a)) and then how to extract Σ_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), we use the traditional fault-tolerant state machine replication approach [Lamport 1998; Schneider 1986], 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 that implements S-consensus using $\Sigma_S * \Omega_S$. The algorithm can be viewed as a variant of the rotating coordinator algorithm of Chandra and Toueg [1996] where the notion of majority is replaced by Σ_S. 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, p 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 of S 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.

(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 $\mathcal{E}$ be any environment, $\mathcal{D}$ 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 $\mathcal{D}(F)$.

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 same 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 $\mathcal{D}$. 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 $\Sigma_S \preceq \mathcal{D}$. □

*(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

P1: VLM
JACM5704-21 Journal of the ACM April 6, 2010 2:44


```
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.

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 process 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.

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.

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* $L_p = \{\bot, x\}$ *for some* $x \neq \bot$;
   (2.2) *if* $L_p = \{x\}$ *for some* $x \neq \bot$ *then* $L_q = \{x\}$ *or* $L_q = \{\bot, x\}$,
   (2.3) *if* $L_p = \{\bot, x\}$ *for some* $x \neq \bot$ *then* $L_q = \{x\}$ *or* $L_q = \{\bot, x\}$ *or* $L_q = \{\bot\}$.

```
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_C ≠ 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 using Σ_S and Ω_S.

PROOF.

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

If *estFromC*$_p = x$ for some $x \neq \perp$, 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*$_q = \perp$, 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 come from *estFromC*$_q$ values, hence (2.1) is a direct consequence of (1.1).

(2.2) and (2.3): If $L_p = \{\perp, x\}$ 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, $\perp$ 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 $a = \perp$. Hence, for any process q that ends round r either $L_q = \{x\}$ or $L_q = \{\perp, x\}$ or $L_q = \{\perp\}$. This concludes the proof of (2.3). For (2.2), it remains to show that $L_q \neq \{\perp\}$.

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 $y = \perp$ 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 round 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 $\{\perp\}$ and does not change its v_p or with $L_p = \{d\}$ and decides d or with $L_p = \{d, \perp\}$ and sets v_p to d. □

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

PROOF. It is impossible for all processes to decide by Task 2. Hence at least one process decides by Task 1. Consider the first round r in which a process, say p, of S sends a $(DECIDE, *)$ message in Task 1. Let $(DECIDE, d)$ be this message. 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 L_q is $\{d, \perp\}$ and q ends 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 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. Assume by contradiction that no correct process decides. This means that no correct process decides by Task 2. The *completeness* property 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 reach τ and let r be the greatest element 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 round r_0 by Ω_S. Then, the processes 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. $\square$

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 $2 \le k \le n$, 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 $T_1, \ldots, T_n$ emulate k-consensus if there is an implementation of S-consensus using only $T_1, \ldots, T_n$ 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). $\square$

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,

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

FIG. 6. Implementing Σ_S with a majority of correct processes.

and h_m^r means many instances and many registers.[1]

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. *Failure Detectors Comparisons*

In this section, we compare failure detectors Σ_S and Ω_S. We assume that S contains at least two processes for, otherwise, the failure detectors are trivial. We show that, in a system of at least three processes with a majority of correct processes Σ_S is strictly weaker than Ω_S. Without the majority assumption, but still in a system of at least three processes, the two failure detectors are incomparable. In a system of two processes, the two failure detectors are equivalent.

Consider first a system with a majority of correct processes: that is, consider environment $\mathcal{E}_t$ with $t \leq (n-1)/2$. In this case, Σ_S can directly be implemented using the algorithm of Figure 6 (without any failure detector). In this algorithm, for each round r, each correct process $p \in S$ is ensured to receive (*ARE_YOU_ALIVE*, r) messages from a majority of processes. The completeness and intersection properties of Σ_S follow directly from the algorithm of Figure 6 and the majority assumption.

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

We have:

PROPOSITION 29. *If $n > 2$, (1) $\Sigma_S * \Omega_S$ is strictly stronger than Σ_S in every environment $\mathcal{E}_t$ with $t > 0$ and (2) $\Sigma_S * \Omega_S$ is strictly stronger than Ω_S in every environment $\mathcal{E}_t$ with $t > (n-1)/2$.*

PROOF. For any pair of failure detectors (A, B) $A * B$ is stronger than A and stronger than B. In particular, $\Sigma_S * \Omega_S$ is stronger than Σ_S and stronger than Ω_S. We thus need to show that there is no algorithm that implements $\Sigma_S * \Omega_S$ using solely Σ_S nor Ω_S.

Assume by contradiction that there is an algorithm that can use only Σ_S to implement $\Sigma_S * \Omega_S$ in environment $\mathcal{E}_t$. Note that this algorithm would induce an algorithm that can use Σ_S to implement $\Sigma_S * \Omega_S$ in environment $\mathcal{E}_1$, the set of failure patterns with at most one faulty process. In environment $\mathcal{E}_1$, if the number of processes is greater than 2, we have a majority of correct processes and Σ_S can be implemented without any failure detector. This means $\Sigma_S * \Omega_S$ can be implemented in $\mathcal{E}_1$ without any failure detector. By Proposition 16, S-consensus would then be implementable without any failure detector. But if S contains more than one process and if at most one process can crash, with S-consensus it is easy to implement consensus for $t = 1$: processes in S send the decision value to all and processes outside S decide this value. We then get a contradiction with Fischer et al. [1985].

We prove now that no algorithm can use only Ω_S to implement $\Sigma_S * \Omega_S$ in environments $\mathcal{E}_t$ with $t > (n-1)/2$. Assume by contradiction that such algorithm exists. By Proposition 16, S-consensus would be implementable with only failure detector Ω_S. But this contradicts the following Lemma:

LEMMA 30. *There is no S-consensus algorithm with Ω_S in any environment $\mathcal{E}_t$ with $t > (n-1)/2$.*

PROOF OF LEMMA. We use the same partitioning technique as in Chandra and Toueg [1996]. Let by contradiction $\mathcal{A}$ be a S-consensus algorithm with Ω_S for such environments. Let A and B be any disjoint subsets of Π such that A and B contains each at least one process in S and the cardinalities of A and B are less than or equal to $\lceil n/2 \rceil$. Note that in this case, t is greater or equal to $\lceil n/2 \rceil$ and then all processes not in A or not in B may crash.

Consider run α_A in which all processes have initial value 0, all processes in A are correct, all other processes are initially crashed and let time t_A be the time at which all processes in $A \cap S$ decide (this decision is 0).

Consider run α_B in which all processes have initial value 1, all processes in B are correct, all other processes are initially crashed and let time t_B be the time at which all processes in $B \cap S$ decide (this decision is 1).

Consider run α in which (1) all processes in A and in B are correct, (2) all processes in A have 0 as initial value, (3) all processes in B have 1 as initial value, (4) the output of failure detector Ω is the same as in α_A for processes in A up to time t_A and as in α_B for processes in B up to time t_B, (5) the reception of all messages from processes in A to processes in B and the reception of all messages from processes in B to processes in A are delayed until after time $\max(t_A, t_B)$, (6) up to time $\max(t_A, t_B)$ processes in A, respectively in B, take steps at the same times as in α_A, respectively as in α_B.

Run α is indistinguishable from α_A to processes in $A \cap S$ and then processes in $A \cap S$ decide 0 in α. In the same way, however, α is indistinguishable from α_B for processes in $B \cap S$ and then processes in $B \cap S$ decide 1 in α, contradicting the agreement property. $\square$

COROLLARY 31. *For $n > 2$, in environments $\mathcal{E}_t$ with $0 < t < (n-1)/2$, Ω_S is strictly stronger than Σ_S and in environments $\mathcal{E}_t$ with $t \geq (n-1)/2$ Ω_S and Σ_S are incomparable.*

The $n > 2$ hypothesis is crucial in the proof above. Maybe surprisingly, in a system of 2 processes, $\Sigma * \Omega$ and Σ are equivalent. To prove this, we go through an intermediate failure detector: the *Strong* failure detector ($\mathcal{S}$) introduced in Chandra et al. [1996] and Chandra and Toueg [1996]. ($\mathcal{S}$) ensures *strong completeness*, that is, eventually every process that crashes is permanently suspected by every correct process, and *weak accuracy*, that is, some correct process is never suspected. This failure detector and MP implements consensus whatever the number of faulty processes. Furthermore, as shown in Chandra et al. [1996] and Chandra and Toueg [1996], $\mathcal{S}$ is stronger than Ω.

PROPOSITION 32. *For $n = 2$, $\mathcal{S} \equiv \Sigma$.*

PROOF.

(1) $\Sigma \preceq \mathcal{S}$: By definition, $\mathcal{S}$ ensures *strong completeness* and some correct process is never suspected. Hence, $\mathcal{S}$ ensures the *intersection property*. Then, $\Sigma \preceq \mathcal{S}$.
(2) $\mathcal{S} \preceq \Sigma$: 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. 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_1 and p_2. Therefore, in all cases, at least one correct process is never suspected. This proves the *accuracy* property of $\mathcal{S}$. By definition, Σ ensures *strong completeness*. Hence, $\mathcal{S} \preceq \Sigma$. $\square$

The following holds in any environment and is a direct corollary of the proposition above and the fact that $\Omega \preceq \mathcal{S}$ [Chandra et al. 1996; Chandra and Toueg 1996]:

COROLLARY 33. *For $n = 2$, $\Sigma \equiv \Sigma * \Omega$.*

7. *Concluding Remarks*

We show in this article 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. We show however that, according to this metric, these types are strictly harder to implement than the basic register type in a system of at least three processes. Maybe surprisingly, in a system of two processes, we prove that the necessary and sufficient information about failures to implement a register is the same as that necessary and sufficient to implement consensus. This contrasts with the fact that there is no asynchronous algorithm that implements consensus using registers even in a system of two processes [Loui and Abu-Amara 1987].

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

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*. Lecture Notes in Computer Science, vol. 1914. Springer-Verlag, Berlin, Germany, 268–283.

ATTIYA, H., BAR-NOY, A., AND DOLEV, D. 1995. Sharing memory robustly in message passing systems. *J. ACM 42*, 2 (Jan.), 124–142.

ATTIYA, H., AND WELCH, J. 1998. *Distributed Computing. Fundamentals, Simulations, and Advanced Topics.* McGraw-Hill, New York.

BAZZI, R. A., NEIGER, G., AND PETERSON, G. L. 1997. On the use of registers in achieving wait-free consensus. *Distrib. Comput. 10*, 3, 117–127.

CHANDRA, T. D., HADZILACOS, V., AND TOUEG, S. 1996. The weakest failure detector for solving consensus. *J. ACM 43*, 4 (July), 685–722.

CHANDRA, T. D., AND TOUEG, S. 1996. Unreliable failure detectors for reliable distributed systems. *J. ACM 43*, 2 (Mar.), 225–267.

FISCHER, M. J., LYNCH, N. A., AND PATERSON, M. S. 1985. Impossibility of distributed consensus with one faulty process. *J. ACM 32*, 2 (Apr.), 374–382.

HADZILACOS, V., AND TOUEG, S. 1993. Fault-tolerant broadcasts and related problems. In *Distributed Systems*. Addison-Wesley, Reading, MA, 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. *Distrib. Comput. 6*, 4 (July), 205–209.

JAYANTI, P. 1993. On the robustness of Herlihy's hierarchy. In *Proceedings of the 12th ACM Symposium on Principles of Distributed Computing*. ACM, New York, 145–157.

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

LAMPORT, L. 1986. On interprocess communication; part I and II. *Distrib. Comput. 1*, 2, 77–101.

LAMPORT, L. 1998. The Part-Time parliament. *ACM Trans. Comput. Syst. 16*, 2 (May), 133–169.

LOUI, M., AND ABU-AMARA, H. 1987. Memory requirements for agreement among unreliable asynchronous processes. *Adv. Comput. Res. 4*, 163–183.

SCHNEIDER, F. 1986. The state machine approach: A tutorial. In *Fault-Tolerant Distributed Computing*. Lecture Notes in Computer Science, vol. 448. Springer-Verlag, Berlin, Germany, 18–41.

VITANYI, P., AND AWERBUCH, B. 1986. Atomic shared register access by asynchronous hardware. In *Proceedings of the IEEE Symposium om Foundations of Computer Science*. IEEE Computer Society Press, Los Alamitos, CA, 233–243.

RECEIVED NOVEMBER 2005; REVISED AUGUST 2008; ACCEPTED DECEMBER 2009