(Almost) All Objects are Universal in Message Passing Systems
(Extended Abstract)

Carole Delporte-Gallet[1], Hugues Fauconnier[2], and Rachid Guerraoui[3]

[1] ESIEE-IGM Marne-La-Vallee, France
[2] LIAFA Univ Paris VII, France
[3] EPFL Lausanne, Switzerland

Abstract. This paper shows that all shared atomic object types that can solve consensus among $k > 1$ processes have the same weakest failure detector in a message passing system with process crash failures. In such a system, object types such as test-and-set, fetch-and-add, and queue, known to have weak synchronization power in a shared memory system are thus, in a precise sense, equivalent to universal types like compare-and-swap, known to have the strongest synchronization power. In the particular case of a message passing system of two processes, we show that, interestingly, even a register is in that sense universal.

1 Introduction

1.1 Atomic objects

A shared atomic object is a data structure exporting a set of operations that can be invoked concurrently by the processes of the system. *Atomicity* means that any object operation appears to execute at some individual instant between its invocation and reply time events [14, 11]. Thanks to atomicity, the type of an object can solely be defined according to its *sequential specification*: the set of all possible sequential executions of the object operations [11].

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

Some of these atomic object types have been shown to have more *synchronization power* than others in the sense that they can solve the seminal *consensus* problem [9] among more processes [12]. What is meant here by a *type solving consensus* is that instances of that type can be used in a deterministic algorithm that solves the consensus problem; we consider here the *uniform* variant of consensus where no two processes can decide differently: the problem can be casted as an atomic object type also called consensus.

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

The type register is in this sense weak as it can only solve consensus for exactly one process [16]. Test-and-set, fetch-and-add, or queue can solve consensus among exactly 2 processes. Interestingly, for any number k, there is a type that can solve consensus among exactly k processes [12]. This leads to a hierarchy of types, called the *consensus hierarchy*, classifying types according to their *consensus number (k)*. Types such as compare-and-swap can solve consensus among any number of processes, and are said to be *universal* [12]: their consensus number is ∞ and they are at the top of the hierarchy.

1.2 Atomic object implementations in message passing systems

This paper studies necessary and sufficient conditions for implementing atomic object types in a distributed system where processes communicate by exchanging messages: no physical shared memory is assumed. The processes are assumed to communicate through reliable channels but can fail by crashing. Through such implementations, algorithms based on shared atomic objects can be automatically ported into a message passing system prone to crash failures. We focus on *robust* [2] implementations where any process that invokes an object operation and does not crash eventually gets a reply.

Two fundamental results are known about such implementations in an asynchronous message passing system (with no synchrony assumptions). The type register can only be implemented if we assume that a majority of the processes do not crash [2], and most of the types cannot be implemented, including test-and-set, fetch-and-add, queue, and compare-and-swap, if at least one process may crash [9].

In most distributed systems however, certain synchrony assumptions can be made, and these can even be precisely expressed through axiomatic properties of a *failure detector* abstraction [5]: a distributed oracle that provides processes with hints about crashes, and which can itself be implemented based on synchrony assumptions, e.g., timeouts.

Two related results, of particular interest in this paper, have been recently established. First, the *weakest* failure detector to implement the basic type register in a message passing system (with any number of crashes) has been shown to be an oracle, denoted by Σ, and which outputs, at any time and at every process, a set of processes such that (1) any two sets always intersect and (2) eventually every set contains only correct processes [6, 8]. This result means that (a) there is a distributed algorithm that implements the type register using Σ, and (b) for every failure detector $\mathcal{D}$ such that some algorithm implements the type register using $\mathcal{D}$, there is an algorithm that implements Σ using $\mathcal{D}$. Failure detector $\mathcal{D}$ encapsulates information about failures that are at least as strong as those encapsulated by Σ.

Second, the weakest failure detector to solve consensus (with any number of crashes) has been shown to be an oracle, denoted by $\Sigma * \Omega$, and which outputs, at any time and at every process, both outputs of failure detector Σ and failure detector Ω. Failure detector Ω outputs, at any time and at every process, a single (leader) process, such that, eventually this process is the same at all processes and is correct [4, 6, 8]. The fact that $\Sigma * \Omega$ was established as the weakest failure detector to solve consensus directly implies that it is also the weakest to implement compare-and-swap, and more generally, any other universal type.

1.3 Contributions

Naturally, this raises the following question. What about all other types like queue, test-and-set, or fetch-and-add? More generally, what about all types that can solve consensus among $k > 1$ processes but not $k+1$. This paper shows that the weakest failure detectors to implement these types do all boil down to the same one: $\Sigma * \Omega$.

In other words, we show that the weakest failure detector to implement any type that solves consensus among at least 2 processes is $\Sigma * \Omega$.

Our result conveys the interesting fact that, in a message passing system (unlike in a shared memory system), all these types are, in a precise sense, equivalent and universal. Hence, if we exclude types that cannot solve consensus among two processes such as register, the consensus hierarchy is thus *flat* in a message passing system. From a practical perspective, and given that many synchronization problems can be casted through atomic object types, our result suggests that, as far as failure detection is concerned, adopting an ad hoc approach focusing on each problem individually is not more economic than a generic approach where the failure detector $\Sigma * \Omega$ would be implemented in a message passing system as a common service underlying all problems, i.e., all type implementations.

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. Such a model is interesting by itself. To our knowledge, besides the model we introduce in this paper, the only model that captured these different abstractions in a unified framework has recently been defined (using I/O automata) by considering a restricted form of failure detectors [1]. In this paper, we establish the weakest failure detector to implement atomic object types among *all* failure detectors. Our model, and in particular our notion of implementation, is a slight generalization of both the notions of shared memory object implementations of [12] as well as failure detector reductions of [5].

To prove our result, we first consider the problem of solving consensus among a subset of processes S in the system. We observe that failure detector $\Sigma_S * \Omega_S$, obtained by restricting Σ and Ω to S, is the weakest to solve consensus in S. Then we show that the weakest failure detector to solve consensus among any subset of $k > 1$ processes is the same as the weakest to solve consensus among any subset of $k+1$ processes. The crucial technical step to establish this result is to show that the composition of Ω_S over all pairs S of processes in the system is Ω. (The analogous result for Σ_S is also needed but more easily obtained).

An interesting particular case is when the system simply consists of two processes. We show that, in this case, $\Sigma * \Omega$ is equivalent to Σ. This equivalence also has a surprising ramification: whereas no algorithm can solve consensus using a register in a system of two processes where at least one can crash [16], any failure detector that can be used to implement a register in a message passing system of two processes, where at least one can crash, can also be used to solve consensus.

1.4 Roadmap

To summarize, this paper shows that all atomic object types that can solve consensus among $k > 1$ processes have the same weakest failure detector in a message passing system with process crash failures. In the particular interesting case of a message passing system of two processes, this failure detector is also the weakest to implement a register.

The rest of the paper is organized as follows. Section 2 defines our model. Section 3 introduces failure detectors Σ_S and Ω_S and establishes some preliminary results. Section 4 determines the weakest failure detector to implement consensus among any subset of $k > 1$ processes in the system. Section 5 derives our main results on the weakest failure detector to implement atomic types. Section 6 relates our results with weakest failure detector results in the shared memory model. For space limitations, several proofs are omitted from this extended abstract and given in a companion technical report [7].

2 System model

We consider a distributed system composed of a finite set of n processes $\Pi = \{p_1, p_2, \ldots, p_n\}$; $|\Pi| = n \geq 3$. (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 detectors

Processes can fail by crashing. A process p is said to crash 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, i.e., 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 [5], 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 and hence we do not mention any specific environment.

Roughly speaking, a failure detector $\mathcal{D}$ is a distributed oracle which gives hints about failure patterns of a given environment $\mathcal{E}$. Each process p has a local failure detector module of $\mathcal{D}$, denoted by $\mathcal{D}_p$. Associated with each failure detector $\mathcal{D}$ is a range $R_{\mathcal{D}}$ (when the context is clear we omit the subscript) of values output by the failure detector. A failure detector history H with range

R is a function H from $\Pi \times \Phi$ to R. For every process $p \in \Pi$, for every time $\tau \in \Phi$, $H(p,\tau)$ denotes the value of the failure detector module of process p at time τ, i.e., $H(p,\tau)$ denotes the value output by $\mathcal{D}_p$ at time τ. A failure detector $\mathcal{D}$ is more precisely 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 $\mathcal{D} * \mathcal{D}'(\mathcal{F}) = \{(\mathcal{H}, \mathcal{H}') \mid \mathcal{H} \in \mathcal{H}(\mathcal{D}), \mathcal{H}' \in \mathcal{H}'(\mathcal{D}')\}$. 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 the $\mathcal{A}_{p_i}$ ($1 \leq i \leq n$). The state of a process after it executes action a in state s, is denoted $a(s)$. A configuration C is a function mapping each process to its local state. When applied to a configuration C, action a of $\mathcal{A}_{p_i}$ gives a new unique configuration denoted $a(C)$: for all $j \neq i$ $(a(C))(p_j) = C(p_j)$ and $(a(C))(p_i) = a(C(p_i))$.

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

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

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

2.3 Services

A service is defined by a pair $(Prim, Spec)$. Each element of $Prim$, denoted by $prim$, is a tuple $< s, p, arg, ret >$ representing an action of process p identified by a sort s, an input argument arg from some (possibly infinite) range In and an output argument (or return value) ret from some (possibly infinite) range Out. An empty argument is denoted by λ. The specification $Spec$ of a service

X is defined by a set of runs. In this paper, we consider three kinds of services: *message passing*, *atomic objects* and *failure detectors*.

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.[4] 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.

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 $R =< F, C, Sc, T >$ 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 R.

Atomic object. Atomic objects are services defined by a sequential specification, and which can be accessed through invocation and reply primitives associated with each operation of the object. It is common to call a pair of invocation and subsequent reply primitives the occurrence of the operation and identify an invocation and the associated operation. The sequential specification of an atomic object is defined by its type and an initial state. A type $\mathcal{T}$ is a tuple $< Q, Op, I, L >$: where Q is the set of states of the type, Op is a set of operations, I is a set of replies, and L is a relation that carries each state $st \in Q$ and operation op 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 returns λ, and a reply has λ as argument and returns a value in I. 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 $\sigma = (o_0r_0)(o_1r_1)\dots(o_jr_j)\dots$ where, for all l, o_l and r_l are respectively operations and replies, is legal from state s if there is a corresponding sequence of states $s = s_0, s_1, \dots s_j, \dots$ 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. [5]

Only well-formed schedules are considered. Consider a schedule Sc, and its restriction to a process $Sc|p$, we say that some occurrence of invocation is pending

[4] 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 $x \in M \cup \{\lambda\}$.

[5] The definition of the *Spec* part of an atomic object O is the same as in [12].

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

Let $R =< F, C, Sc, T >$ be any well-formed run, and $R|Prim$ be the history corresponding to object $O =< Prim, Spec >$ of type $\mathcal{T}$, a linearization of $R|Prim$ with respect to $\mathcal{T}$ and state s is a pair (H, T') such that: (1) H a sequential history of O from state s; (2) H includes all non pending invocations of operation in S; (3) If some invocation inv is pending in S, 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 $Sc[k]$ 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 R is linearizable for type $\mathcal{T}$ and state s if R 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 [11].

2.4 Algorithms and implementations

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

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

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

Note that, we implicitly consider robust [2] implementations of services: every correct process that executes a primitive of an implemented service should eventually get a reply from that invocation. We will sometimes focus on implementations of $S-$services: the primitives of such a service can only be invoked by processes of a subset S of the system. In such implementation, the only restriction is the fact that only the processes in S contain each one subtask per primitive sort of the S-service (but all processes contain implementation tasks). If we do not specify the subset S, we implicitly assume the set of all processes.

2.5 Weakest failure detector

The notion of failure detector $\mathcal{D}2$ being reducible to $\mathcal{D}1$ in a given environment $\mathcal{E}$ ($\mathcal{D}1$ is said to be stronger than $\mathcal{D}2$ in $\mathcal{E}$ and written $\mathcal{D}2 \preceq_{\mathcal{E}} \mathcal{D}1$) of [5], means in our context that there is an algorithm that implements $\mathcal{D}2$ using $\mathcal{D}1$ and MP in $\mathcal{E}$. All the implementation subtasks use only MP and $\mathcal{D}$. For every run $R =< F, C, Sc, T >$, and failure detector history $H \in \mathcal{D}1(F)$ such that F is in $\mathcal{E}$, the output of the algorithm in R is a history of $\mathcal{D}2(F)$. We say that $\mathcal{D}1$ is equivalent to $\mathcal{D}2$ in $\mathcal{E}$ ($\mathcal{D}1 \equiv_{\mathcal{E}} \mathcal{D}2$), if $\mathcal{D}2 \preceq_{\mathcal{E}} \mathcal{D}1$ and $\mathcal{D}1 \preceq_{\mathcal{E}} \mathcal{D}2$ in $\mathcal{E}$.

We say that a failure detector $\mathcal{D}_1$ 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 algorithm that implements the service using $\mathcal{D}_1$ 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 $\mathcal{E}$. As pointed out earlier, given that most of our results hold for all environments, we will generally not mention (and implicitly assume) any environment when stating and proving results.

3 The Quorum and Leader failure detectors

We introduce here two failure detectors: the *Quorum* and the *Leader*. Both are defined relatively to a subset S of processes in the system. The first one, denoted by Σ_S, is a generalization of Σ [6, 8]. The second one, denoted by Ω_S, generalizes Ω [4].

3.1 Failure detector Σ_S

Given any subset S of processes in Π, failure detector Σ_S outputs, at each process in S, and at any time, a list of processes, called *trusted* processes, such that every list intersects with every other list, and eventually, all lists contain only correct processes. For presentation simplicity, we consider that, at any process of S that has crashed, the list that is output is simply Π. More generally, the lists that are output 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 every correct process contains only correct processes: $\forall F \in \mathcal{E}, \forall H \in \Sigma_S(F), \forall p \in S \cap correct(F), \exists \tau \in \Phi, \forall \tau' > \tau \in \Phi : H(p,\tau') \subseteq correct(F)$

Failure detector Σ introduced in [6, 8] is simply Σ_Π.

In the following, we state a result on a register shared by the processes of a subset S, and denoted by $S-$register: the *read()* and *write()* operations can only be invoked by the processes in S. (The sequential specification of a register stipulates that the *read()* returns the last value *written*.) The proof of this proposition is in [7].

Proposition 1. *Σ_S is the weakest failure detector to implement a $S-$register.*

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 the following property is satisfied:

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

Intuitively, the guarantee here is that all processes inside S eventually get the same correct leader. Processes outside S might never get the same leader. However, the leader process that is output does not need to be in S: it can be any process in Π. Failure detector Ω corresponds to Ω_Π.

We state now a result on the consensus type (an abstraction of the consensus problem) shared by the processes of a subset S, denoted by $S-$consensus: the *propose()* operation can only be invoked by the processes of S. (The sequential specification of consensus stipulates that all *propose()* operations return the first value proposed.) The proof of this proposition is in [7].

Proposition 2. *$\Sigma_S * \Omega_S$ is the weakest failure detector to implement $S-$consensus.*

4 From k-consensus to $(k+1)$-consensus

To prove our main result on the weakest failure detector to implement types with a given consensus number k, we address the question of the weakest failure detector to implement k-process consensus (we simply write k-consensus). In short, an algorithm implements k-consensus if it implements S-consensus for any subset S of size k. We show that, for any k s.t. $1 < k < n$, the weakest failure detector to implement k-consensus is also the weakest to implement $(k+1)$-consensus.

To prove this, we go through intermediate results about the composition, over a family of subsets S, of all Σ_S and of all Ω_S.

The following proposition is a direct consequence of the definitions:

Proposition 3. *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 some set $L \in \mathcal{L}$ such that p and q belong to L. We have: $\Sigma_S \equiv *\{\Sigma_X | X \in \mathcal{L}\}$.*

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

Corollary 1. *For all $S \subseteq \Pi$, $\Sigma_S \equiv *\{\Sigma_{\{p,q\}} | p, q \in S\}$.*

Concerning Ω_S, we get the following:

Proposition 4. *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 $q \in L$. $\Omega \equiv *\{\Omega_L | L \in \mathcal{L}\}$.*

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

The opposite inequality is more involved.

Consider a run R with a failure pattern F, and let τ_0 be a time such that (1) after time τ_0 no more process crashes and (2) the output of failure detectors Ω_L, $L \in \mathcal{L}$, does not change after τ_0.

In the following, we show how to implement failure detector $\Diamond\mathcal{S}$, which is equivalent to Ω [4]. Failure detector $\Diamond\mathcal{S}$ outputs subsets of *suspected* processes and ensures: (1) completeness, i.e. eventually every faulty process is permanently suspected by every correct process; and (2) accuracy, i.e. eventually, some correct process is never suspected.

Consider the digraph $G =< V, E >$ for which $V = \mathit{correct}(F)$, and $(p, q) \in E$ if and only if q is leader for p for some Ω_L such that $p \in L$.

Now consider $G' =< V', E' >$ 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 $p \in C$ and one $q \in C'$ such that $(p, q) \in E$. We say that $C \in V'$ is a *sink* if there is no edge going out of C: note that this means that (a) if p belongs to some sink S, and $(p, q) \in E$ then $q \in S$.

First, there exists at least one sink in G'. Indeed, assume the contrary and let C_0 be any vertex in G'; by induction, we construct a sequence (C_u) $(u > 0)$ of vertices such that (i) $(C_{u-1}, C_u) \in E'$ and (ii) $C_{u-1} \neq C_u$. As we assume that there is no sink, this sequence is infinite. Moreover this sequence is cycle free: if $C_u = C_m$ for some $m < u$, then a direct induction proves that all C_k ($m \le k \le u$) are the same strongly connected component contradicting (ii). This implies an infinite number of different C_i contradicting the fact that G is finite.

Consider process p in some C, and process q in some sink S. By definition of $\mathcal{L}$, there is at least one L of $\mathcal{L}$ such that p and q both belong to L. By (a), l, the common leader for p and q, belongs to S. Proving that (b) for every p and every sink S, there exists a process $l \in S$ such that $(p, l) \in E$.

Moreover, let S and S' be two sinks, by (b) there is an edge from S to S' and an edge from S' to S in G', proving that S and S' are in the same strongly connected component and then $S = S'$. Hence, there is only one sink in G'. In the following S will denote this unique sink of G'.

In order to implement $\Diamond\mathcal{S}$, each process p proceeds as follows.

Process p maintains (1) a set $Leader_p^p$ of all its leaders, (2) for each q, a set $Leader_p^q$ of the known leaders of q and, (3) a digraph $G_p =< V_p, E_p >$ for which $V_p = \Pi$, and $(k, q) \in E_p$ if and only if $q \in Leader_p^k$ and, (4) $Trust_p$, the output of the emulated failure detector: this output will be the set of processes q such that there is a path from p to q in G_p.

Process p updates its variables as follows:

– $Leader_p^p$ is always the set of processes output as leader from Ω_L for all L on $\mathcal{L}$ such that $p \in L$. p broadcasts forever $Leader_p^p$.
– If p receives from some q a set X of processes, then p replaces $Leader_p^q$ by X.
– $G_p =< V_p, E_p >$ for which $V_p = \Pi$, and $(k, q) \in E_p$ if and only if $q \in Leader_p^k$. Variable $Trust_p$ holds the set of processes q such that there is a path from p to q in G_p. If $Leader_p^p$ or $Leader_p^q$ change, then p computes again G_p and $Trust_p$.

As (1) any change in $Trust$ variables comes from changes in the output of Ω_L's, and (2) no message is lost, there is a time $\tau_1 \ge \tau_0$ after which no variable $Trust_p$ changes.

Consider a correct process p. Observe that if (q, r) is an edge of G_p then r is a leader for q, and hence if q is a correct process then r is correct too. Then by an easy induction, after time τ_1, every process on a path from p in G_p is a correct process and $Trust_p$ contains only correct processes. This proves the *completeness* property of required for $\Diamond\mathcal{S}$.

Observe also that, after time τ_1, for every correct process p, the set of processes q such that there is a path from p to q in G is equal to $Trust_p$. Moreover, if $(x, y) \in E$ then $Trust_y \subseteq Trust_x$ and, by an easy induction, (c) if there is a path from x to y in G then $Trust_y \subseteq Trust_x$. This proves that, if x and y are in the same strongly connected component of G, then $Trust_x = Trust_y$. In particular, for all q in the sink S of G', $Trust_q = S$. By (b) and (c), for every correct process p, $Trust_q \subseteq Trust_p$ for at least one process q in S, therefore $S \subseteq Trust_p$. This proves the *accuracy* property of $\Diamond\mathcal{S}$. Hence we get $\Omega \preceq *\{\Omega_L | L \in \mathcal{L}\}$ and then $\Omega \equiv *\{\Omega_L | L \in \mathcal{L}\}$.

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

Corollary 2. $\Omega \equiv *\{\Omega_{\{p,q\}} | p, q \in \Pi\}$.

It is important to notice a difference here between Proposition 3 and Proposition 4, and this conveys a fundamental difference between Σ and Ω. Consider a strict subset S of Π. If we can implement a $\{p, q\}$-register within every pair $\{p, q\}$ of S, then we can implement a S-register in S. This is not true with $\{p, q\}$-consensus and this follows from the fact that some leaders output by $\Omega_{\{p,q\}}$ might not belong to the set S. We prove the following in [7]:

Proposition 5. *There exists a system of n processes, a non-empty subset of Π, S, an environment $\mathcal{E}$ and a set of failure detectors $\Omega_{\{p,q\}}$, for all p, q in S, such that $\Omega_S \not\equiv_{\mathcal{E}} *\{\Omega_{\{p,q\}} \mid p, q \in S\}$.*

If we restrict ourselves however to the overall set of processes Π, the difference (i.e., the proposition above) does not hold. That is, to implement consensus (resp. a register), it is necessary and sufficient to implement consensus (resp. register) among all subsets of at least two processes.

Corollary 3. *For any $n \geq k \geq 2$, $\Omega \equiv *\{\Omega_S | |S| = k\}$ and $\Sigma \equiv *\{\Sigma_S | |S| = k\}$.*

Proof. We apply Proposition 3 and Proposition 4 to the family of all subsets of k ($n \geq k \geq 2$) processes.

We directly get from the previous Corollary and Proposition 2 the following:

Corollary 4. *For every k such that $2 \leq k \leq n$, for any failure detector $\mathcal{D}$, $\mathcal{D}$ implements* consensus *if and only if $\mathcal{D}$ implements S-*consensus *for all S such that $|S| = k$.*

It is important to notice again that the previous corollary holds only for consensus, and not for S-consensus if $S \neq \Pi$.

5 Implementing atomic object types

In the following, we will say that types $T_1, \cdots, T_n$ emulate k-consensus if there is an algorithm that uses only instances of types $T_1, \cdots, T_n$ to implements k-consensus.

Proposition 6. *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 and message passing that implements 2-consensus. Clearly, this algorithm with any failure detector $\mathcal{D}$ implementing T implements 2-consensus too and by Corollary 4 it implements consensus. Then, by Proposition 2 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 [12], we derive that $\Sigma * \Omega$ implements any type. Then by (a) any failure detector that implements T implements any type proving (2). Moreover, as $\Sigma * \Omega$ implements any type, it implements in particular T. Together with (a), this proves (1).

An interesting application of Proposition 6 concerns the environment where $n-1$ process might fail (which we call the wait-free environment) and the notion of consensus number, which we recall now.

In fact, several definitions of the notion of consensus number of a type T (sometimes also called consensus power) have been be considered [13]. 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. Hierarchy h_1 means one instance and no register, h_1^r means one instance and registers, h_m means many instances, no register, and h_m^r means many instances and many registers.[6]

From Proposition 6, 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 [3] that $h_m(T) = h_m^r(T)$. Hence we get the following:

Proposition 7. *In the wait-free environment, 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 $k = h_m^r(T)$.*

We finally consider the special case where $n = 2$. In this case, implementing a register is in some sense equivalent to implementing consensus. More precisely, we prove the following:

Proposition 8. *For $n = 2$, $\Sigma \equiv \Sigma * \Omega$.*

Proof. We actually prove a stronger result. We show that for $n = 2$, Σ is equivalent to $\mathcal{S}$ which is a failure detector introduced in [5], and which outputs subsets of *suspected* processes and ensures: (1) completeness, i.e. eventually every faulty process is permanently suspected by every correct process and (2) accuracy, i.e. some correct process is never suspected. As $\mathcal{S}$ implements consensus [5], from proposition 2, we get: $\Sigma \preceq \Sigma * \Omega \preceq \mathcal{S}$. Denote by p_1 and p_2 the two processes of the system. Consider a 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_2. Therefore, in all cases, at least one correct process is never suspected. This proves the *accuracy* property of $\mathcal{S}$. Hence $\mathcal{S} \preceq \Sigma$.

[6] We implicitly assume here *n-ported* types, i.e., every instance of a type has n ports in our system of n processes [13].

As a direct consequence, we get: for $n = 2$, $\Sigma \equiv \Sigma * \Omega \equiv \mathcal{S}$.

6 Concluding remarks

The question we address in this paper is that of the weakest failure detector to implement atomic object types (of certain consensus numbers) in a message passing system. This question is complementary to the question of the weakest failure detector to solve consensus in a system of n processes, given object types of consensus number $k < n$ [15, 17, 10]. In this paper, the goal was to actually implement the types themselves.

It would be interesting to determine, for any $k > 1$, the weakest failure detector to implement any type with consensus number k, given any type of consensus number $j < k$. We conjecture that our proof technique could help show that Ω is the weakest failure detector to implement, with register objects (instead of message passing channels), any object type with a consensus number higher than 2. Going from any type with consensus number $k > 2$ to any type with consensus number $j < k < n$ would probably need a combination of our proof technique with that of [10].

Acknowledgments

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

References

1. P. Attie, R. Guerraoui, P. Kouznetsov, N. Lynch, and S. Rajsbaum. The impossibility of boosting distributed service resilience. In *Proceedings of the 25th International Conference on Distributed Computing Systems*. IEEE Computer Society Press, June 2005.
2. H. Attiya, A. Bar-Noy, and D. Dolev. Sharing memory robustly in message passing systems. *J. ACM*, 42(2):124–142, Jan. 1995.
3. R. A. Bazzi, G. Neiger, and G. L. Peterson. On the use of registers in achieving wait-free consensus. *Distributed Computing*, 10(3):117–127, 1997.
4. T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *J. ACM*, 43(4):685–722, July 1996.
5. T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, Mar. 1996.
6. C. Delporte-Gallet, H. Fauconnier, and R. Guerraoui. Shared memory vs message passing. Technical Report 200377, EPFL Lausanne, 2003.
7. C. Delporte-Gallet, H. Fauconnier, and R. Guerraoui. Implementing atomic objects in a message passing system. Technical report, EPFL Lausanne, 2005.
8. C. Delporte-Gallet, H. Fauconnier, R. Guerraoui, V. Hadzilacos, P. Koutnetzov, and S. Toueg. The weakest failure detectors to solve certain fundamental problems in distributed computing. In *23th ACM Symposium on Principles of Distributed Computing*, July 2004.

9. M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, Apr. 1985.
10. R. Guerraoui and P. Kouznetsov. On failure detectors and type boosters. In *Proceedings of the 17th International Symposium on Distributed Computing*, LNCS 2848, pages 292–305. Springer-Verlag, 2003.
11. M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.
12. M. P. Herlihy. Wait-free synchronization. *ACM Trans. Prog. Lang. Syst.*, 13(1):123–149, Jan. 1991.
13. P. Jayanti. On the robustness of herlihy's hierarchy. In *12th ACM Symposium on Principles of Distributed Computing*, pages 145–157, 1993.
14. L. Lamport. On interprocess communication; part I and II. *Distributed Computing*, 1(2):77–101, 1986.
15. W.-K. Lo and V. Hadzilacos. Using failure detectors to solve consensus in asynchronous shared memory systems. In *Proceedings of the 8th International Workshop on Distributed Algorithms*, LNCS 857, pages 280–295, Sept. 1994.
16. M. Loui and H. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. *Advances in Computing Research*, 4:163–183, 1987.
17. G. Neiger. Failure detectors and the wait-free hierarchy. In *14th ACM Symposium on Principles of Distributed Computing*, 1995.