A General Characterization of Indulgence

R. GUERRAOUI
Ecole Polytechnique Fédérale de Lausanne
and
N. LYNCH
Massachusetts Institute of Technology

An indulgent algorithm is a distributed algorithm that, besides tolerating process failures, also tolerates unreliable information about the interleaving of the processes. This article presents a general characterization of indulgence in an abstract computing model that encompasses various communication and resilience schemes. We use our characterization to establish several results about the inherent power and limitations of indulgent algorithms.

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 between models*

General Terms: Algorithms, Theory, Reliability

Additional Key Words and Phrases: Process failures, scheduling failures, agreement

ACM Reference Format:

Guerraoui, R. and Lynch, N. 2008. A general characterization of indulgence. ACM Trans. Autonom. Adapt. Syst. 3, 4, Article 20 (November 2008), 19 pages. DOI = 10.1145/1452001.1452010 http://doi.acm.org/10.1145/1452001.1452010

1. INTRODUCTION

1.1 Indulgence

The idea of *indulgence* is motivated by the difficulty for any process in a distributed system to accurately figure out, at any point of its computation, any new information about which of—and in what order—the rest of the processes will take steps. Informally, an *indulgent* algorithm is a distributed algorithm that tolerates such uncertainty.

Authors' address: R. Guerraoui, School of Computer and Communication Sciences, EPFL; email: rachid.guerraoui@epfl.eh.

The notion of indulgence is not new and has been implicitly considered in various forms, usually in specific computing and communication models (e.g., [Dwork et al. 1988; Chandra and Toueg 1996; Lamport 1998]). The goal of this work is to capture this notion in an abstract and general way, independently of specific computing models, be they time-based, round-based, message-passing or shared-memory.

An obvious class of indulgent algorithms are *asynchronous* ones [Fischer et al. 1985]. These do not make any assumption about communication delays and relative process speeds. As a consequence, a process can never distinguish the situation where another process has failed from the situation where the other process is simply slow, nor can it determine how processes will interleave their steps if they indeed perform some. The idea of indulgence is however more general. In particular, an algorithm that eventually becomes synchronous, after an unknown period of time [Dwork et al. 1988], is also indulgent. Even if the processes know that there is a time after which bounds on communication delays and process relative speeds do hold, the processes do not know when that time will occur, or whether or not that time has occurred. Similarly, algorithms that rely on an eventual leader election abstraction, such as Paxos [Lamport 1998], or an eventually accurate failure detector, such as the rotating coordinator algorithm of Chandra and Toueg [1996], are also indulgent. Other examples of indulgent algorithms include those that tolerate an unbounded number of timing failures [Taubenfeld 2007] or assume an eventual time after which processes execute steps in a certain order [Mostefaoui et al. 2004], an eventual bound on the ratio between the delay of the fastest message and the slowest message [Widder et al. 2005], or an eventual bound on the response time of the processes [Fetzer et al. 2005]. All of these algorithms are indulgent, though not asynchronous.

These examples indeed illustrate the idea of indulgence, but all refer directly to specific failure detector models or specific synchrony assumptions, typically assuming a message passing model [Guerraoui 2000; Dutta and Guerraoui 2002; Vicente and Rodrigues 2002; Guerraoui and Raynal 2004; Sampaio and Brasileiro 2005]. The goal of this work is to characterize the notion of indulgence in a way that helps determine the inherent power and limitation of indulgent algorithms, independently of specific models.

1.2 Murphy's Law

To seek a general characterization of indulgence, it is tempting to consider an abstract approach that looks at *runs* of an algorithm as sequences of *events* that occur at the *interface* between the processes executing the algorithm and the *distributed services*[1] used in the algorithm, each event representing a *step* of a process. This is in contrast to an approach where we would look into the internals of the individual services involved in the computation and the automata executed on the processes.

While appealing for its generality, the abstract approach is not straightforward, as we explain in this article. In particular, it is not immediate to devise

[1]Shared memory, broadcast primitive, message passing channel, failure detector, clock, etc.

an abstract characterization of indulgence without precluding algorithms that assume a threshold of correct (nonfaulty) processes. This would be unfortunate, since many seminal indulgent algorithms assume a majority of correct processes [Dwork et al. 1988; Lamport 1998; Chandra and Toueg 1996].

In short, we characterize indulgence by applying *Murphy's law* to partial runs of an indulgent algorithm: *whatever can go wrong will go wrong, and at the worst possible time, in the worst possible way*. Basically, we capture indulgence by requiring that if the interleaving I (the order according to which processes execute steps) of a partial run R of an algorithm A *could* be extended with steps of certain processes and not others, while still being tolerated by the algorithm, then the partial run R can *itself* be extended in A with such steps. More specifically, we say that an algorithm A is indulgent if, given any partial run R of A and the corresponding interleaving I of processes, if A tolerates an extension I' of I, then A does also have an extension of R with interleaving I'. In a sense, we capture the intuition of indulgence by requiring that partial run R does not provide the processes with enough information to predict the extension of the interleaving I: if some extension of I is tolerated by the algorithm, then this extension can also be associated with an extension of R.

1.3 Results

The first contribution of this article is thus a general characterization of indulgence. We show that our characterization is *monotonic* in the sense that, if an algorithm A that tolerates k failures is indulgent, then the restriction of A to runs with $k - 1$ failures is also indulgent.

We then show that every indulgent algorithm A is inherently *uniform*: if A ensures the *correct-restriction* of a safety property P (P restricted to correct processes), then A ensures the actual property P (required for all processes). A corollary of this, for instance, is that an indulgent algorithm cannot solve the correct-restriction of consensus [Fischer et al. 1985], also called *nonuniform* consensus (where a process can decide a different value from a value decided by a failed process) without solving consensus (where no two processes should ever decide different value-uniform agreement). This is not the case with non-indulgent algorithms.

We use our uniformity result to show that certain problems are impossible with indulgent algorithms. In particular, we show that no indulgent algorithm can solve a *failure-sensitive* problem, even if only one process can fail, and can do so only initially. In short, a failure sensitive problem is one the specification of which depends on whether certain processes take steps after a decision is taken. Failure-sensitive problems include some classical ones, such as *nonblocking atomic commit* [Skeen 1981] and *terminating reliable broadcast* [Hadzilacos and Toueg 1993], also known as the *Byzantine Generals* problem [Lamport et al. 1982]. There are known algorithms that solve these problems, but these are not indulgent.

Our reduction from uniformity to impossibility with respect to failure-sensitive problems is, we believe, interesting in its own right. By showing that our impossibility applies only to initial failures, and holds even if the algorithm uses powerful services like consensus, we emphasize the fact that this impossibility is fundamentally different from the classical impossibility of consensus in an asynchronous system, when a process can fail during the computation [Fischer et al. 1985].

Finally, we prove that, given n the number of processes in the system, and assuming $n - \lfloor n/x \rfloor$ processes can fail ($x \leq n$), no indulgent algorithm can ensure a x*−divergent* property using only *timeless* services. In short, a x−divergent property is any property that can hold for partial runs involving a disjoint subset of processes, but not in the composition of these runs, whereas a timeless service is one that does not provide any real-time guarantee. We capture here, in a general way, the traditional partitioning argument that is frequently used in distributed computing. Corollaries of our result include the impossibility for an indulgent algorithm using message passing or sequentially consistent objects [Lamport 1979] to (a) implement a safe register [Lamport 1979] if half of the processes can fail, as well as (b) implement k-set agreement if $n - \lfloor n/k \rfloor$ processes can fail.

To conclude the article, we discuss how, using our notion of indulgence, we derive the first precise definition of the notion of *unreliable* failure detection [Chandra and Toueg 1996]. Whereas this elegant notion is now folklore in the distributed computing literature, it has never been precisely defined in a general model of distributed computation.

2. MODEL

2.1 Processes and Services

We consider a set Π of $n > 1$ processes each representing a Turing machine. Every process has a unique identity (id). Processes communicate through shared abstractions, including sequentially consistent or atomic objects [Lamport 1979; Herlihy 1991], as well as message passing channels and broadcast primitives [Hadzilacos and Toueg 1993]. The processes can also consult oracle abstractions such as failure detectors [Chandra and Toueg 1996] about the operational status of other processes, or specific devices that provide them with random values [Ben-Or 1983]. We call these oracles and communication abstractions *distributed services* or simply *services*. Each service exports a set of operations through which it is accessed. For instance:

— A message passing channel exports *send* and *receive* operations. The *send* takes an input parameter, that is, a message, and returns simply an *ok* indication that the message was sent. On the other hand, a *receive* does not take any input parameter and returns a message, possibly *nil* (empty message) if there is no message to be received. Message passing channels differ according to the guarantees on message delivery. Some might ensure that a message that is sent is eventually received by every correct process (the notion of *correct* is recalled more precisely in the following). Others ensure simply that the message is received if both the sender and the receiver are correct.

— An atomic queue exports *enqueue* and *dequeue* operations. The *enqueue* takes an input parameter (an element to enqueue) and returns an *ok* indication. On the other hand, a *dequeue* does not take any input parameter and returns an element in the queue (the oldest), if there is any, or simply *nil* if there is no element in the queue.

— A failure detector exports one *query* operation that typically does not take any input parameter, and returns a set of processes that are suspected to have failed and stopped their execution. In a sense, a failure detector provides information about the future interleaving of the processes. More generally, one could also imagine oracles that inform a process that certain processes will be scheduled before others.

2.2 Steps

Each process is associated with a set of possible states, some of which are *initial* states. A set of n states, each associated with a different process of the system, is called a *configuration*. A configuration composed of initial states is called an *initial configuration*. Besides a set of states, a process is also associated with an *automaton* that regulates the transition of the states of the processes according to a given *algorithm*.

The system starts from an initial configuration, among a set of possible initial configurations, and evolves to new configurations by having processes execute *steps* of their algorithm. A *step* is an atomic unit of computation that takes the system from a configuration to a new configuration.

Every step is associated with exactly one process. In every step, the associated process accesses at most one shared service by invoking one of the operations of the service and getting back a reply (we do not assume here any determinism). Based on this reply, the process modifies its local state before moving to the next step. The automaton of the process determines, given a state of a process and a reply from the invocation of an operation, the new state of the process and the operation to invoke in the next step of the process.

The visible part of a step, at the interface between a process and a service, is sometimes called an *event*. It is modeled by a process id, the name of a service, the name of an operation, as well as input and output parameters of the operation's invocation. By language abuse, we also call this a step when there is no ambiguity between the event and the corresponding step.

2.3 Schedules

A sequence of steps S is called a *schedule* and the corresponding sequence of process ids is called the *interleaving* of the schedule S and is denoted by $I(S)$. We talk about infinite schedules (respectively, infinite interleavings) if the sequence is infinite, and finite schedules (respectively, finite interleavings) otherwise.

If (the id of) a process p appears infinitely often in an infinite interleaving I, we say that p is *correct* in I. Otherwise, we say that p is *faulty* in I. We denote the set of faulty processes in I by *faulty(I)*. We say that a process p *initially fails* in I if p does never appear in I. We denote the set of processes that do not appear in I by $\mathit{faulty}^{\star}(I)$.

We say that a schedule S_2 (respectively, an interleaving I_2) is an *extension* of a schedule S_1 (respectively interleaving I_1) if S_1 (respectively, I_1) is a prefix of S_2 (resp. I_2). We write $S_2 \in E(S_1)$ (resp. $I_2 = E(I_1)$).

2.4 Runs

A *run* R is a pair (S, C) composed of a schedule S and a configuration C, called the initial configuration of the run R. The interleaving of the schedule S, $I(S)$, is also called the interleaving of the run R, and is denoted by $I(R)$. We say that R is associated with I. We talk about *partial* runs for those associated with finite interleavings. We say that a (partial) run $R_2 = (S_2, C)$ is an *extension* of a partial run $R_1 = (S_1, C)$ (we write $R_2 \in E(R_1)$) if S_2 is an extension of S_1. We simply say that R_1 is a partial run of R_2. In this case, $I(R_2)$ is also an extension of $I(R_1)$ (we write $I(R_2) \in E(I(R_1))$). We also denote by $R/p = (S/p, C)$ the restriction of $R = (S, C)$ to the steps involving only process p.

2.5 Algorithms

We model an *algorithm* as a set of runs. The interleavings of the runs of an algorithm A are said to be *tolerated* by A, and the set of these interleavings is denoted by $I(A)$. This terminology conveys the idea that the interleaving is chosen by the operating system and not by the algorithm. In some sense, the operating system acts as an *adversary* that the algorithm needs to face. For instance, in *wait-free* computing [Herlihy 1991], an algorithm tolerates all possible interleavings: the algorithm has at least one run for every possible interleaving.

We focus on algorithms that ensure the following three properties:

(1) If R is a run of an algorithm A, then every partial run of R is also in A.
(2) Let A be any algorithm, and $R = (C, S)$ any run of A. If C' is an initial configuration similar to C, except for the initial states of the processes in $faulty^\star(I(R))$, then $R' = (C', S)$ is also a run of A. This property conveys the fact that we consider distributed algorithms where processes can only learn the states of other processes by communicating with them, that is, by performing steps.
(3) Any finite interleaving of some run of an algorithm A has a failure-free extension also associated with some run of A. That is, $\forall I_1 \in I(A)$, $\exists I_2 \in I(A) \cap E(I_1)$ such that $faulty(I_2) = \emptyset$. Requiring this property means that we preclude algorithms that assume that, eventually, a threshold of the processes do fail. On the other hand, we do encompass algorithms that tolerate a threshold of failures, as we precisely define in the following.

— We say that an algorithm A is a *k-resilient* algorithm if $I(A)$ is the set of all interleavings where at least $n - k$ processes appear infinitely often. That is, $I \in I(A)$ if and only if $faulty(I) \leq k$.

— We say that A is a *$k^\star$-resilient* algorithm if every process that appears once in any interleaving I of A appears infinitely often in I (we capture here the assumption of initial failures only). In other words, $I \in I(A)$ if and only if $faulty^{\star}(I) = faulty(I) < n - k$.

We also introduce two forms of algorithm extensions. Let A and A' be any two algorithms.

(1) A' is an *extension* of A if $A \subset A'$ (every run of A is a run of A').
We also say in this case that A is a *restriction* of A.

(2) A' is a *strict extension* of A if, (a) A' is an extension of A, and (b) $\forall R \in A'$ such that $I(R) \in I(A)$, $R \in A$ (every run of A' with an interleaving tolerated by A is also a run of A). We also say in this case that A is a *strict restriction* of A'.

3. INDULGENCE

3.1 Overview

Informally, an algorithm is *indulgent* if no process, at any point of its computation, can accurately predict the future interleaving of the processes. For instance, no process can ever determine if another process has stopped its execution, nor will perform any step in the future. In the same vein, no process can ever accurately determine the future order of events of some other couple of processes.

As we discuss in the following, it is not trivial to capture this intuition without precluding algorithms that tolerate certain interleavings and not others. Examples of these algorithms are t-(or $t^{\star}$-) resilient algorithms. In such algorithms, certain interleavings are known to be impossible in advance, that is, before the processes start any computation. Hence a process can, in some sense, predict the future interleaving. As we will explain, a naive definition of indulgence would preclude such algorithms. We discuss three such candidate definitions in the following.

(1) Consider a definition (*characterization 1*) that would declare an algorithm A indulgent if, for any partial run R of A, for any process q, A has an extension of R with an infinite number of steps by q. This clearly captures the idea that no process can, at any point of its computation (say after any partial run R) declare that some other process q is faulty, since q could still take an infinite number of steps (after R) and thus be correct. Although intuitive, this characterization is fundamentally flawed, as we discuss in the following. With this characterization, we might consider as indulgent an algorithm that relies on the ability of a process to accurately learn that at least *one out of two* processes have failed, or learn that certain processes will perform their steps in a round-robin manner if they indeed perform future steps. Indeed, characterization 1 simply says that *any* process q can still take steps in *some* extension of the partial run R. For some pair of processes q_1 and q_2, there might be no extension of R with both q_1 and q_2 taking an infinite number of steps.

(2) We would like indulgence to express the very fact that any *subset* of processes can still take steps after any point of the computation, that is, after

any partial run R, and in every possible order. In fact, there is an easy fix to characterization 1 that deals with this issue. It is enough to require (*characterization 2*) that, for any partial run R of A, for any subset of processes Π_i, A has an extension of R with an infinite number of steps of all processes of Π_i in every order. As we will discuss however, this characterization raises other issues. This characterization might lead us to consider as indulgent an algorithm that relies on the ability for the processes to learn that some specific process *will* take steps in the future.

(3) A naive way to address this issue is to also require (*characterization 3*) that, for any partial run R of an indulgent algorithm A, for any subset of processes Π_i, A has an extension of R where no process in Π_i takes any step after R. Characterization 3, however, excludes algorithms that assume a threshold of correct processes. As we pointed out earlier, many indulgent algorithms [Chandra et al. 1996; Dwork et al. 1988; Lamport 1998] assume a correct threshold of processes: in particular, they assume that every partial run has an extension where a majority of processes take an infinite number of steps. We now introduce our characterization of indulgence which, we claim, addresses these issues.

3.2 Characterization

Very intuitively, we cope with the issues we have described by proposing a definition of indulgence inspired by Murphy's law, which we apply to partial runs. Basically, we declare an algorithm A indulgent if, whenever the finite interleaving $I(R)$ of any partial run R of A could be extended with a certain interleaving I' tolerated by A, R also could be extended with a run R' associated with I'. In other words, if the interleaving $I(R)$ of a partial run R has an extension I' in $I(A)$, then R does also have an interleaving R' in A with the interleaving $I(R') = I'$.

Definition. Indulgence. An algorithm A is indulgent if, for any pair of interleavings I_1 and I_2 tolerated by A such that I_2 extends I_1, any partial run R_1 of A with interleaving I_1, has an extension R_2 in A with interleaving I_2. More specifically: $\forall I_1 \in I(A)$ s.t.$\forall I_2 \in I(A) \cap E(I_1)$, $\forall R_1 \in A$ s.t. $I(R_1) = I_1$, $\exists R_2 \in A$ such that $I(R_2) = I_1$ and $I_2 \in E(I_1)$.

Basically, the definition says that no partial run R_1 can preclude any extension R_2 with interleaving I_2, provided I_2 is indeed tolerated by A. The definition does not preclude t-resilient algorithms from being indulgent. This would not have been the case, for instance, with a definition that would only consider as indulgent an algorithm A such that, for any partial run R of A, for any subset of processes $\Pi_i \subset \Pi$, A has an extension R_1 of R where all processes of Π_i are correct, and an extension R_2 of R where no process in Π_i takes any step after R.

3.3 Examples

Clearly, our definition of indulgence precludes synchronous algorithms [Lynch 1996]. This is because, in a synchronous algorithm, a process that does not take a step within a round of a partial run R does never take any step in future rounds of extensions of R. With an appropriate communication abstraction that reveals the absence of a step, a process can hence predict the fact that the absent process will never take any further step. Similarly, an asynchronous algorithm that makes use of a perfect failure detector [Chandra and Toueg 1996] is not indulgent. If a process is detected to have failed in some partial run R, then R cannot be extended with an interleaving including steps of p.

In fact, even an algorithm relying on an anonymously perfect failure detector is not indulgent [Guerraoui 2002]. Such a failure detector might signal that *some* process has failed, without indicating which one. When the failure detector does so in some partial run R, all processes can deduce that it is impossible to extend R with a run where all processes are correct. Similarly, an algorithm that uses an oracle that declares some process correct, say from the start [Guerraoui 2001], would not be indulgent if the algorithm tolerates at least one interleaving where that process crashes.

An obvious class of indulgent algorithms are t-resilient asynchronous ones [Fischer et al. 1985]. Such algorithms do not have any partial run providing meaningful information about the future interleaving of the processes. However, the idea of indulgence is broader than that of asynchrony. Algorithms that rely (only) on eventual properties (i.e., that hold only after an unknown period of time) about the interleavings of the processes, are indulgent. These include eventually synchronous algorithms [Dwork et al. 1988], eventual leader-based algorithms [Lamport 1998], rotating coordinator-based algorithms [Chandra and Toueg 1996], as well as algorithms that tolerate an unbounded number of timing failures [Taubenfeld 2007], or assume eventual interleaving properties [Mostefaoui et al. 2004], an eventual bound on the ratio between the delay of the fastest and the slowest communication [Widder et al. 2005], or an eventual bound of the response time of the processes [Fetzer et al. 2005].

In the following, we prove two characteristics of indulgent algorithms: *monotony* and *uniformity*. We then use these to prove some inherent limitations of indulgent algorithms.

4. MONOTONY

In short, the monotony aspect (of our definition) of indulgence conveys the fact that removing interleavings of an algorithm does not impact its indulgence. In particular, if an algorithm A that tolerates t failures is indulgent, then the restriction of A to (runs with) $t - 1$ failures is also indulgent.

PROPOSITION 1. *Every strict restriction of an indulgent algorithm is also indulgent.*

PROOF. Consider any two algorithms A and A' such that A is a *strict restriction* of A'. We proceed by contradiction and assume that A' is indulgent whereas A is not.

By definition, the fact that A is not indulgent means that (a) there are two interleavings I_1 and I_2 tolerated by $I(A)$ such that $I_2 \in E(I_1)$ (I_2 is an extension of I_1) (b) there is a partial run R of A such that $I(R) = I_1$, and (c) A has no extension of R, R', such that $I(R') = I_2$.

The fact that I_1 and I_2 are tolerated by A means that A has two runs R_1 and R_2 such that $I(R_1) = I_1$ and $I(R_2) = I_2$.

Since A is a restriction of A', and R, R_1 and R_2 are runs of A, then R, R_1 and R_2 are also runs of A'. In particular, this means that I_1 and I_2 are tolerated by A'.

As A' is indulgent, $I_1 = I(R)$ is tolerated by A', I_2 is an extension of I_1 also tolerated by A', R is a partial of A', then A' has an extension R' of R such that $I(R') = I_2$.

Finally, because A is a strict restriction of A', I_2 is tolerated by A and A', and $R' \in A'$, then $R' \in A$: a contradiction with (c). Hence A is indulgent. □

Consider an algorithm A that is t-resilient ($t > 1$) and define $A(t-1)$ as the restriction of A to runs with $t-1$ failures. Remember that the fact that A is t-resilient means A tolerates all interleavings where at least $n-t$ processes are correct, that is, $n-t$ processes take an infinite number of steps. Algorithm $A(t-1)$ is thus a $t-1$-resilient algorithm, that is, where at least $n-t-1$ processes take an infinite number of steps: it is a strict restriction of A. We thus get the following corollary from Proposition 1 (the same reasoning applies to $t^{\star}$-resilient algorithms):

Corollary 2. *If a t-resilient algorithm A is indulgent then so is $A(t-1)$.*

5. UNIFORMITY

In the following, we show that indulgent algorithms are inherently *uniform*, in the intuitive sense that they are not sensitive to safety properties that restrict only the behavior of correct processes. More specifically, indulgent algorithms cannot satisfy a safety property for correct processes without satisfying it for all processes. We will illustrate the idea of uniformity through the consensus problem and point out the fact that uniformity does not hold for algorithms that are not indulgent. Later, we will use the notion of uniformity to prove that certain problems do not have indulgent solutions. First however, we recall the notions of safety, and introduce the notion of correct-restriction of a property.

5.1 Safety

The specifications of problems are typically expressed in terms of predicates over runs, also called properties of runs. An algorithm solves a problem if those predicates hold over all runs of the algorithm.

Informally, a safety property states that *nothing bad should happen*, whereas a liveness property states that *something good should eventually happen* [Lamport 1977; Alpern and Schneider 1985] .

Consider a predicate P over runs and a specific run R. We say that P *holds in* R if $P(R) = true$; P does not hold in R if $P(R) = false$.

A safety property P is a predicate that satisfies the two following conditions:

(1) any run for which P does not hold has a partial run for which P does not hold;

(2) P does not hold in every extension of a partial run where P does not hold.

A liveness property P, on the other hand, is one such that:

—any partial run has an extension for which P holds.

It was shown in Lamport [1977] and in Alpern and Schneider [1985] that any property can be expressed as the intersection of a safety property and a liveness property. Given a property P, possibly a set of properties (i.e., a problem), we denote by $S(P)$ the safety part of P and $L(P)$ the liveness part of P.

5.2 Correct Restriction of a Property

We now introduce the notion of a *correct-restriction* of a property.

Informally, the *correct-restriction* of a property P, denoted $C[P]$, is the restriction of P to correct processes. In the following, we denote by $R\,[\mathit{Correct}(R)]$ the restriction of R to correct processes.

Definition. Correct-restriction. Let P be any property. We define the *correct-restriction* of P, denoted $C[P]$ as the property that is true for any run R if and only if P is true for $R\,[\mathit{Correct}(R)]$.

PROPOSITION 3. *Let P be any safety property and A any indulgent algorithm. If A satisfies $C[P]$ then A satisfies P.*

PROOF. Let P be any safety property and A any indulgent algorithm that satisfies $C[P]$.

Assume by contradiction that A does not satisfy P. This implies that there is a run of A, say R, such that $P(R)$ is false. Because P is a safety property, there is a partial run of R, R', such that $P(R')$ is false.

By the indulgence of A, and our assumption that any interleaving has a failure-free extension, A has an extension of R', say R'', where all processes are correct.

Because P is a safety property and $P(R')$ is false, $P(R'')$ is also false. Hence $C[P](R'')$ is false because all processes are correct in R'' and $C[P](R'') = P(R'')$. A contradiction with the fact that A satisfies $C[P]$. □

5.3 Example: Consensus

An immediate corollary of our proposition concerns, for instance, the *consensus* [Fischer et al. 1985] and *uniform consensus* problems (respectively, *total order broadcast* and *uniform total order broadcast*) [Hadzilacos and Toueg 1993]. Before stating our corollary, we recall the consensus problem.

We assume here a set of values V. For every value $v \in V$ and every process $p \in \Pi$, there is an initial state e_p of p associated with v and e_p is not associated with any other value $v' \neq v$. v is called the initial value of p (in state e_p). Hence each vector of n values correspond to an initial configuration of the system. We also assume that, among other distributed services used by the processes, a specific one models the act of *deciding* on a value. The service, called the *output* service, has an operation *output()*; when a process p invokes that operation with an input parameter v, we say that p decides v.

An algorithm A solves the consensus problem if, in any run $R = (C, S)$ of A, the three following properties are satisfied.

—*Validity:* the value decided by any process p_i in R is the initial value of some process p_j in C.
—*Agreement:* no two processes decide different values in R;
—*Termination:* every correct process in R eventually decides in R.

Clearly, agreement and validity are safety properties whereas termination is a liveness property. Two weaker, yet orthogonal, variants of *consensus* have been studied in the literature. One, called *nonuniform consensus*, only requires that no two *correct* processes decide different values (interestingly, this is a liveness property, since it can always be eventually ensured by crashing processes). Another variant, called k-set-*agreement* [Chaudhuri 1993], requires that the number of different values decided by all processes (in any run) is at most k.

The following is a corollary of Proposition 3.

COROLLARY 4. *Any indulgent algorithm that solves consensus also solves uniform consensus.*

This is not the case with nonindulgent algorithms, as we will explain. Consider a system of two processes $\{p_1, p_2\}$ using two services: an atomic shared register and a perfect failure detector. The latter service ensures that any process is eventually informed about the failure of the other process, and only if the other process, has indeed failed. The idea of a nonindulgent algorithm solving nonuniform consensus is the following: process p_1 decides its initial value and then writes it in the shared register; process p_2 keeps periodically consulting its failure detector and reading the register until either (a) p_1 is declared faulty by the failure detector, or (b) p_2 reads p_1's value. In the first case, (a) p_2 decides its own value, and in the second, (b) p_2 decides the value read in the register. If both processes are correct, they both decide the value of p_1. If p_1 fails after deciding, p_2 might decide a different value.

6. FAILURE SENSITIVITY

In the following, we show that no indulgent algorithm can solve certain problems if at least one process can fail, even if this process can do so only initially, that is, if the algorithm is $1^\star$-resilient. We call a $1^\star$-resilient indulgent algorithm simply a $1^\star$-indulgent algorithm.

The problems we show to be impossible are those we call *failure-sensitive*. In short, these are decision problems that resemble consensus with the particularity that the decision value might be considered valid depending on whether certain processes have failed. These problems include several classical problems in distributed computing, such as *terminating reliable broadcast*, *interactive consistency* and *nonblocking atomic commit* [Hadzilacos and Toueg 1993].

To prove our impossibility, we proceed as follows: we first define a simple failure sensitive problem, which we call *failure signal*, and which we show cannot be solved with a $1^\star$-indulgent algorithm. Then we show that any solution to *terminating reliable broadcast*, *interactive consistency* or *non-blocking atomic commit* solves *failure signal*: in this sense, *failure signal* is weaker than all those problems that are thus impossible with a $1^\star$-indulgent algorithm.

6.1 The Failure Signal Problem

In failure signal, just as in consensus, the goal is for processes to decide on a value based on some initial value. As we explain however, unlike consensus, no agreement is required, and a process can decide different values.

More specifically, in failure signal, a specific designated process p has an initial binary value, 0 or 1, as part of p's initial state. The two following properties need to be satisfied: (1) Every correct process eventually decides and (2) no process (a) decides 1 if p proposes 0, nor (b) decides 0 if p proposes 1 and p is correct.

Interestingly, we prove the impossibility of *failure signal* by reduction to our *uniformity* result (Proposition 3). We prove by contradiction that if there is a $1^\star$-indulgent algorithm that solves failure signal, then there is an algorithm that ensures the corrected-restricted variant of a safety property without ensuring the actual property.

PROPOSITION 5. *There is no solution to failure signal using a $1^\star$-indulgent algorithm.*

PROOF. Assume by contradiction that there is a $1^\star$-indulgent algorithm that solves failure signal. Consider the designated process p and some other process q (remember that we assume a system of at least two processes).

Define property P such that (a) $P(R)$ is *false* in every run R where p proposes 1 and q decides 0, and (b) $P(R)$ is *true* in all other runs. By definition of a correct-restriction, $C[P]$ is *false* in runs where p proposes 1, q decides 0 and all processes are correct, and *true* in all other runs.

We now show that, if there is a $1^\star$-indulgent algorithm that solves failure signal, then A ensures $C[P]$ but not P.

It is easy to show that A ensures $C[P]$. Indeed, because A solves *failure signal*, in any run R where p proposes 1 and all processes are correct, all processes decide 1.

We now show that A does not ensure P. Remember that A is a $1^\star$-resilient algorithm, that is, A tolerates at least one initial failure. Consider a run R where p proposes 0 and does not take any step whereas all other processes are correct (p initially fails). Any $1^\star$-resilient algorithm that solves the failure signal problem has such a run R—in this run, every process that decides decides 0.

Consider now a run R' with the same schedule as R, except that p initially proposes 1 (and fails before taking any step). Such a run R is also a run of A and, because no process other than p, which fails initially, can distinguish R from R', all processes but p decide 0. This run R' is thus a run of A, and $P(R')$ is false. This contradicts the uniformity of A. □

6.2 Example 1: Terminating Reliable Broadcast

In *terminating reliable broadcast*, also called *Byzantine generals* [Lamport et al. 1982], a specific designated process is supposed to *broadcast* one message $m \neq \perp$ that is a priori unknown to the other processes. (In our model, the process invokes a specific service with m as a parameter.) In a run R where the sender p does not fail, all correct processes are supposed to eventually receive m. If the sender fails, then the processes might or not receive m. If they do not, then they receive a specific message $\perp$ indicating that the sender has failed. More specifically, the following properties need to be satisfied. (1) Every correct process eventually receive one message; (2) No process receives more than one message; (3) No process receives a message different from $\perp$ or the message broadcast by the sender; (4) No two processes receive different messages; and (5) No process receives $\perp$ if the sender is correct.

The following is a corollary of Proposition 5.

COROLLARY 6. *No $1^\star$-resilient algorithm solves terminating reliable broadcast.*

PROOF. We show how any solution to *terminating reliable broadcast* can be used to solve *failure signal*. Assume there is an algorithm A that solves terminating reliable broadcast. Whenever the designated process p (in failure signal) proposes a value, 0 or 1, p broadcasts a message with that value to all, using *terminating reliable broadcast*. Any process that receives the message delivers the value in the message (0 or 1). A process that delivers $\perp$ decides 0. □

6.3 Example 2: Nonblocking Atomic Commit

In nonblocking atomic commit, processes do all start with initial values 0 or 1, and are supposed to eventually decide one of these values. The following properties need to be satisfied. (1) Every correct process eventually decides one value (0 or 1); (2) no process decides two values; (3) No two processes decide different values; (4) No process decides 1 if some process proposes 0, and no process decides 0 if all processes propose 1 and no process fails.

The following is a corollary of Proposition 5.

COROLLARY 7. *No $1^\star$-resilient algorithm solves nonblocking atomic commit.*

PROOF. Assume there is a solution to nonblocking atomic commit. We show how to obtain a solution to failure signal. All processes but p propose 1. Process p proposes exactly its initial value (of failure signal) to nonblocking atomic commit. The processes decide the output of nonblocking atomic commit. Because all processes but p propose 1, the decision can be 1 only if p proposes 1, and can be 0 only if p fails or proposes 0. □

6.4 Example 3: Interactive Consistency

In interactive consistency, processes do all start with initial values, and are supposed to eventually decide an n-vector of values. The following properties need to be satisfied. (1) Every correct process eventually decides one vector; (2) No process decides two vectors; (3) No two processes decide different vectors; (4) If a process decides a vector v, then $v[i]$ should contain the initial value of p_i if p_i is correct. Otherwise, if p_i is faulty, $v[i]$ can be the initial value of p_i or $\perp$.

The following is a corollary of Proposition 5.

COROLLARY 8. *No $1^\star$-indulgent algorithm solves interactive consistency.*

PROOF. Assume there is a solution to interactive consistency. Assume p is p_i. We show how to obtain a solution to failure signal. All processes propose to interactive consistency their identity, except p which proposes its initial value of failure signal. If a process q outputs a vector v such that $v[i] \neq \perp$, then q decides $v[i]$. Otherwise, q decides 0. □

7. DIVERGENCE

We now capture, in a general way, the traditional partitioning argument that is frequently used in distributed computing, for example, [Attiya et al. 1995]. This argument was traditionally used for message-passing asynchronous algorithms where half of the processes can fail. In this case, the system can partition into two disjoint subsets that progress concurrently. We precisely state it here in the general context of indulgent algorithms using timeless services which, as we have pointed out, is a wider class than the class of asynchronous algorithms using message passing, and for systems with several possible partitions (the case with two partitions is just a special case).

Definition. Divergent property. We call a *$k-$divergent* property P a property such that, for any k disjoint nonempty subsets of processes $\Pi_1, \Pi_2,..\Pi_k$, there is a configuration C such that every k runs $R_1, R_2,..R_k$ of A, such that R_i involves only processes from Π_i, have respective partial runs $R'_1, R'_2,..,R'_k$ for which $S(P(R'_1.R'_2...R'_k))$ is false.

Remember that $S(P)$ denotes the safety part of P. We call configuration C the *critical configuration* for $\Pi_1, \Pi_2,..\Pi_k$ with respect to P. Note that, by construction, any property that is $k-$divergent is also $k+1-$divergent.

To intuitively illustrate the idea of a $2-$divergent property, consider the specification of consensus in a system of two processes p_1 and p_2. Consider the initial configuration where p_1 has initial value 1 and p_2 has initial value 2. Starting from C, every run R_1 involving only p_1 eventually decides 1, and every run R_2 involving only p_2 eventually decides 2. Consider the partial run R'_1 of R_1 composed of all steps of R_1 until the decision of p_1 (1) is made, and the partial run R'_2 of R_2 until the decision of p_2 (2) is made. Clearly, the safety of consensus (in particular *agreement*) is violated in $R'_1.R'_2$.

Definition. Timeless service. We say that an algorithm A uses *timeless* services if, for any two partial runs R_1 and R_2 of A starting from the same initial configurations C and involving disjoint subsets of processes, if A has an extension of R_1, $R_1.R'_1$ such that $I(R'_1) = I(R_2)$, then $R_1.R_2$ is also a run of A. The idea is similar to that of Voelzer [2004].

Examples of timeless services include sequentially consistent shared objects [Lamport 1979] as well as reliable message passing or broadcast primitives [Hadzilacos and Toueg 1993]. To illustrate the underlying idea, consider an algorithm A in a system of two processes p_1 and p_2 using a message passing primitive that ensures that any message sent from process p_1 to process p_2 is eventually received by p_2, provided p_2 is correct. Assume that A has a partial run R_1 where p_1 executes steps alone, and a partial run R_2 where p_2 executes steps alone (clearly, p_2 cannot have received any message from p_1 in R_2). Provided that A does not preclude the possibility of p_2 to execute steps alone after R_1, and because there is no guarantee on the time after which the message of p_1 arrives at p_2, then $R_1.R_2$, the composition of both partial runs, is also a possible run of A. This captures the intuition that the message of p_1 can be arbitrarily delayed.

PROPOSITION 9. *No $(n - \lfloor n/x \rfloor)$-indulgent algorithm ensures a $x-$*divergent *property using $x-$*timeless *services.*

PROOF. Assume by contradiction that there is a $(n - \lfloor n/x \rfloor)$-resilient indulgent algorithm A that ensures a $x-$*divergent* property P using *timeless* services.

Divide the set of processes Π of the system into k subsets Π_1, Π_2,..Π_x of size at least $\lfloor n/x \rfloor$ such that all the subsets are disjoint and their union is Π. Consider the critical configuration C for Π_1, Π_2,..Π_x with respect to P.

Because the algorithm A is $(n - \lfloor n/x \rfloor)$-resilient, and each Pi_i is of size at least $\lfloor n/x \rfloor$, then A has x runs R_1, R_2,..R_x such that each such R_i involves only processes in Π_i, that is, only processes of P_i take steps in R_i and every such R_i start from C.

Because P is $x-$*divergent*, these runs have respective partial runs R'_1, R'_2, $\ldots$,R'_k such that $S(P(R'_1.R'_2...R'_k))$ is false. We need to show that $R'_1.R'_2 \cdots R'_k$ is also a partial run of A. Because $S(P(R'_1.R'_2 \cdots R'_k))$ is false, this would contradict the very fact that A ensures P.

We first show that $R'_1.R'_2$ is a partial run of A. By the assumption that A is $(n - \lfloor n/x \rfloor)$-resilient, there is a partial run R_0 of A such that $I(R_0) = I(R'_1.R'_2)$ (remember that an x-resilient algorithm is one that tolerates *all* interleavings where at least $n - x$ processes appear infinitely often).

By the indulgence of A, there is a partial run R''_2 such that $R'_1.R''_2$ is a partial run of A and $I(R'_1.R''_2) = I(R'_1.R'_2)$. By the assumption that A uses timeless services, $R'_1.R'_2$ is also a partial run of A. By a simple induction, $R'_1.R'_2 \cdots R'_k$ is also a run of A.

Because $S(P(R'_1.R'_2 \cdots R'_k))$ is false, P is false in every extension of $R'_1.R'_2 \cdots R'_k$: contradiction. □

The following is a corollary of Proposition 9.

COROLLARY 10. *No $(n - \lfloor n/2 \rfloor)$-indulgent algorithm using message passing or sequentially consistent objects can implement a safe register.*

There are nonindulgent algorithms that implement a safe register with any number of failures, using only message passing—for instance, an algorithm assuming a perfect failure detector. The idea is to make sure every value written is stored at all processes that are not detected to have crashed, and the value read can then simply be a local value. On the other hand, the previous result means that an algorithm using eventually perfect failure detectors, and possibly also sequentially consistent registers or message passing, cannot implement a safe register if two disjoint subsets of processes can fail. This clearly also applies to problems like consensus.

The following, assuming $k > 1$, is also a corollary of Proposition 9.

COROLLARY 11. *No* $(n - \lfloor n/k \rfloor)$*-indulgent algorithm using message passing or sequentially consistent objects can solve* k*-set agreement [Chaudhuri 1993].*

8. CONCLUDING REMARKS

Indulgent algorithms are algorithms that tolerate, besides process failures, unreliable information about the interleaving of the processes. This article presents a general characterization of indulgence. The characterization does not require any failure detector machinery [Chandra and Toueg 1996], or timing assumptions [Dwork et al. 1988]. We simply express indulgence in terms of the possibility of extending runs.

Our general characterization is furthermore not restricted to a specific communication model. Instead, we consider a general model of a distributed system, where processes might be communicating using any kind of services, including shared objects, be they simple read-write registers [Lamport 1979], or more sophisticated objects like compare-and-swap or consensus [Herlihy 1991], as well as message passing channels and broadcast primitives [Hadzilacos and Toueg 1993].

Our characterization of indulgence also abstracts the essence of *unreliable failure detection*. The notion of *unreliable* failure detection, informally introduced in Chandra and Toueg [1996], captures the idea that failure detectors might not need to be accurate to be useful in solving interesting problems. Although the concept of failure detector was precisely defined in Chandra and Toueg [1996], the idea of an unreliable one was not, except in Guerraoui [2000] for the specific message passing context. Using our characterization of indulgence, we can precisely define it by simply stating that a failure detector is *unreliable* if any algorithm that uses that failure detector is indulgent.

Generalizing the notion of a failure detector, one can actually imagine oracles that inform a process that certain processes will be scheduled before others. In certain operating systems, for instance, processes can be not only informed about which processes have been swapped out, but also in which order these processes will be scheduled. For instance, an oracle could inform processes that a certain run is eventually synchronous. Indeed, a failure detector can be viewed as a particular case of an oracle that provides information about the interleaving of the processes (in the case of a failure detector, the information indicates when a process executes its last step, or will execute an infinite number of steps). Our characterization of indulgence thus helps capture what it means for such oracles to be unreliable.

To conclude, it is important to recall that we focused, in this article, on computability and not complexity. We studied what can it means for an algorithm to be indulgent, and what can be computed with such algorithms. We did not discuss the complexity of indulgent algorithms. One could generally expect that indulgent algorithms be less efficient than their nonindulgent counterparts to solve the same problem. There are many interesting open problems in measuring the inherent overhead of indulgence, but this goes through defining appropriate frameworks to measure the complexity of indulgent algorithms, for example, Dutta and Guerraoui [2002], Keidar and Shraer [2006], and Zielinski [2006].

ACKNOWLEDGMENTS

We are very grateful to the reviewers for their useful comments about the earlier draft of this article.

REFERENCES

Alpern, B. and Schneider, F. B. 1985. Defining liveness. *Inf. Process. Lett. 21,* 4, 181–185.

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

Ben-Or, M. 1983. Another advantage of free choice: completely asynchronous agreement protocols (extended abstract). In *Proceedings of the Annual ACM Symposium on Principles of Distributed Computing (PODC)*. ACM, New York, NY, 27–30.

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

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

Chaudhuri, S. 1993. More choices allow more faults: Set consensus problems in totally asynchronous systems. *Inform. Comput. 105,* 1, 132–158.

Dutta, P. and Guerraoui, R. 2002. The inherent price of indulgence. In *Proceedings of the Annual ACM Symposium on Principles of Distributed Computing (PODC)*. ACM, New York, NY, 88–97.

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

Fetzer, C., Schmid, U., and Susskraut, M. 2005. On the possibility of consensus in asynchronous systems with finite average response times. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS)*. IEEE Computer Society, Los Alamitos, CA. 271–280.

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

Guerraoui, R. 2000. Indulgent algorithms (preliminary version). In *Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing (PODC).* ACM, New York, NY, 289–297.

Guerraoui, R. 2001. On the hardness of failure sensitive agreement problems. *Inf. Process. Lett. 79*.

Guerraoui, R. 2002. Non-blocking atomic commit in asynchronous distributed systems with failure detectors. *Distrib. Comput. 15,* 1, 17–25.

Guerraoui, R. and Raynal, M. 2004. The information structure of indulgent consensus. *IEEE Trans. Comput. 53,* 4, 453–466.

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

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

Keidar, I. and Shraer, A. 2006. Timeliness, failure detectors and consensus peformance. In *Proceedings of the Annual ACM Symposium on Principles of Distributed Computing (PODC)*. ACM, New York, NY.

LAMPORT, L. 1977. Proving the correctness of multiprocessor programs. *IEEE Trans. Softw. Eng. 3,* 2, 125–143.

LAMPORT, L. 1979. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput. C-28,* 9, 690–691.

LAMPORT, L. 1998. The part-time parliament. *ACM Trans. Comput. Sys. 16,* 2, 133–169.

LAMPORT, L., SHOSTAK, R., AND PEASE, M. 1982. The Byzantine generals problem. *ACM Trans. Program. Lang. Syst. 4,* 3, 382–401.

LYNCH, N. A. 1996. *Distributed Algorithms*. Morgan Kaufmann.

MOSTEFAOUI, A., RAYNAL, M., AND TRAVERS, C. 2004. Crash-resilient time-free eventual leadership. In *Proceedings of the IEEE International Symposium on Reliable Distributed Systems (SRDS)*. IEEE Computer Society, Los Alamitos, CA, 208–217.

SAMPAIO, L. AND BRASILEIRO, F. 2005. Adaptive indulgent consensus. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*. 422–431.

SKEEN, D. 1981. Nonblocking commit protocols. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*. Y. E. Lien, Ed. ACM Press, New York, NY, 133–142.

TAUBENFELD, G. 2007. Computing in the presence of timing failures. In *Proceedings of the IEEE International Conference on Distributed Computing Systems (ICDCS)*. IEEE Computer Society, Los Alamitos, CA.

VICENTE, P. AND RODRIGUES, L. 2002. An indulgent uniform total order broadcast algorithm with optimistic delivery. In *Proceedings of the IEEE International Symposium on Reliable Distributed Systems (SRDS)*. IEEE Computer Society, Los Alamitos, CA, 92–80.

VOELZER, H. 2004. A constructive proof for flp. *Inf. Process. Lett. 92*.

WIDDER, J., LANN, G. L., AND SCHMID, U. 2005. Failure detection with booting in partially synchronous systems. In *Proceedings of the 5th European Dependable Computing Conference*. Lecture Notes in Computer Science, vol. 3464. Springer, Berlin, Germany.

ZIELINSKI, P. 2006. Optimistically terminating consensus. In *Proceedings of the IEEE International Symposium on Parallel and Distributed Computing (ISPDC)*. IEEE Computer Society, Los Alamitos, CA.

Received March 2007; revised August 2008; accepted September 2008