The Information Structure of Indulgent Consensus

Rachid Guerraoui, *Member, IEEE,* Michel Raynal, *Member, IEEE*

***Abstract*— To solve consensus, distributed systems have to be equipped with oracles such as a failure detector, a leader capability, or a random number generator. For each oracle, various consensus algorithms have been devised. Some of these algorithms are indulgent toward their oracle in the sense that they never violate consensus safety, no matter how the underlying oracle behaves. This paper presents a simple and generic indulgent consensus algorithm that can be instantiated with any specific oracle, and be as efficient as any ad hoc consensus algorithm initially devised with that oracle in mind. The key to combining genericity and efficiency is to factor out the *information structure* of indulgent consensus executions within a new distributed abstraction, which we call "Lambda".**

Interestingly, identifying this information structure also promotes a fine-grained study of the inherent complexity of indulgent consensus. We show that instantiations of our generic algorithm with specific oracles, or combinations of them, match lower bounds on oracle-efficiency, zero-degradation, and one-step-decision. We show however that no leader or failure detector-based consensus algorithm can be, at the same time, zero-degrading and configuration-efficient. Moreover, we show that leader-based consensus algorithms that are oracle-efficient are inherently zero-degrading, but some failure detector-based consensus algorithms can be both oracle-efficient and configuration-efficient. These results highlight some of the fundamental trade-offs underlying each oracle.

***Index Terms*— Asynchronous Distributed System, Consensus, Crash Failure, Fault-Tolerance, Indulgent Algorithm, Information Structure, Leader Oracle, Modularity, Random Oracle, Unreliable Failure Detector.**

Rachid Guerraoui is with EPFL, Switzerland.
Michel Raynal is with IRISA, France.

This paper is a revised and extended version of the paper *A Generic Framework for Indulgent Consensus*, that appeared in the proceedings of the *23rd IEEE Int. Conference on Distributed Computing Systems (ICDCS'03)*, IEEE Computer Press, Providence (RI), May 2003.

I. Introduction

a) Context: Understanding the deep structure and the basic design principles of algorithms solving fundamental distributed computing problems is an important and challenging task. This task has been undertaken for basic problems such as distributed mutual exclusion [17], [30] and distributed deadlock detection [6], [20]. Another such basic problem is consensus [2], [13], [23]. This problem consists for a set of n processes to propose each, an initial value, and eventually agree on one of the proposed values, even if some of the processes fail by crashing. Consensus is at the heart of reliable distributed computing and it is tempting to seek for the fundamental structure of its algorithms, in particular consensus algorithms that are optimal in terms of *resilience* and *performance*.

b) Resilience Optimality: Given that it is impossible to solve consensus deterministically in the presence of crash failures in a purely asynchronous system [13], several proposals have been made to augment the system with oracles that circumvent the impossibility. A first approach consists in introducing a *random oracle* [3] allowing to design consensus algorithms that provide eventual decision with probability 1. Another approach considers a *failure detector oracle* [7] encapsulating eventual synchrony assumptions [12]. In particular, failure detector $\diamond\mathcal{S}$ has received a lot of attention. It provides each process with a list of processes suspected to have crashed, in such a way that every process that crashes is eventually suspected (completeness property) and there is a time after which some correct process is no longer suspected (accuracy property). Another approach consists in equipping the system with a *leader* oracle [21]. This oracle, denoted Ω in [8], provides the processes with a function leader which eventually always delivers the same correct process identity to all processes.

Oracles $\diamond\mathcal{S}$ and Ω are in a precise sense *minimal* in that each provides a necessary and sufficient amount of information about failures to solve consensus with a deterministic algorithm. Oracles $\diamond\mathcal{S}$ and Ω have actually the same computational power [8], [10]. The random oracle solves a non-deterministic variant of consensus [3] and is, strictly speaking, incomparable with the other two. This oracle is however, in the sense of [9], as strong as consensus, and hence also somehow minimal. Interestingly, the algorithms that rely on any of those oracles have all the common inherent flavor that consensus safety is never violated, no matter how the oracle behaves: they are *indulgent* toward their oracle [15]. In other words, the oracles are only necessary for the liveness property of consensus. A price to pay for this indulgence is that the upper bound f on the number of processes that are allowed to crash has to be smaller than $n/2$ (where n is the total number of processes), and this is needed for each of these oracles [3], [7], [8].

c) Performance Optimality: This paper focuses on the performance of indulgent consensus algorithms in terms of time complexity (i.e., latency measured in terms of communication steps). That is, we consider the number of communication steps needed for the processes to reach a decision in certain runs of the execution. We overview here different optimality metrics along this direction, and we define them precisely later in the paper.

- Oracle-efficiency When an oracle behaves *perfectly*, the consensus decision can typically be expedited. More precisely, for all oracles discussed above, two communication steps are necessary and sufficient to reach consensus in *failure-free* runs where the oracle behaves perfectly, e.g., [19]. Algorithms that match this lower bound (e.g., [18], [26], [31]) are said to be *oracle-efficient* in the sense that they are optimized for the well behavior of the oracle.
- Zero-degradation This property extends oracle-efficiency from failure-free runs to runs with initial crashes [11]. Algorithms that have this property also match the two communication steps lower bound in runs with *initial* crashes. For instance, the consensus algorithms of [11] need only two communication steps to reach consensus when the oracle behaves perfectly, even if some processes had crashed initially. This is particularly important because consensus is typically used in a repeated form, and a process failure during one consensus instance appears as an initial failure in a subsequent consensus instance. In a zero-degrading algorithm, a failure in a given instance does not impact the performance of any future instances.
- One-step-decision If the processes exploit an initial knowledge on a privileged value, or on a specific subset of processes, they can even, sometimes, reach consensus in a single communication step [5], e.g., when all non-crashed processes propose that privileged value. This can for instance be very useful if that specific value has reasonable chances to be proposed more often than others. Algorithms that exploit such a knowledge to expedite a decision are called here *one-step-decision* algorithms.
- Configuration-efficiency Finally, when all processes propose the same initial value, no underlying oracle is actually necessary to obtain a decision. In that case, two communication steps are also necessary and sufficient to reach a consensus decision, no matter how the underlying

oracle behaves. Algorithms that match this lower bound are said to be *configuration-efficient*. (Such algorithms actually follow the *condition-based* approach introduced and investigated in [24].)

d) Motivation: In short, solving consensus goes through equipping distributed systems with additional oracles such as a failure detector, a leader oracle or a random number generator. Interestingly, algorithms relying on such oracles are all indulgent but they all require a correct majority. They do also have some inherent performance lower bounds in terms of time complexity. The objective of this work is to come up with a simple unified indulgent consensus algorithm that is generic and efficient. Genericity means here that we could easily instantiate the algorithm with any oracle, whereas efficiency means that the resulting algorithm should be as efficient as any ad hoc consensus algorithm designed for that specific oracle.

The first difficulty underlying this objective lies in factoring out the appropriate information structure that is common to efficient indulgent consensus algorithms, as each of which might be using a different oracle and making use of specific algorithmic techniques. In fact, it is not entirely clear whether such a common structure could be precisely defined, and whether the same generic algorithm could encompass the specific characteristics of a random oracle, a failure detector and a leader oracle. The second difficulty has to do with the possible conflicting nature of the lower bounds. We know of no algorithm that matches all lower bounds recalled above and it is not clear whether such an algorithm can indeed be devised.

e) Related Work: In [18], several consensus algorithms were unified within the same framework, all however relying on $\diamond\mathcal{S}$-like failure detectors. A similar unification was proposed in [4], for Ω-based consensus algorithms. In [1], [28], consensus algorithms that make use of several oracles at the same time were presented. In these *hybrid* algorithms however, efficiency was not the issue and the oracles are used in a hardwired manner, e.g., they cannot be interchangeable. A first attempt to build a common consensus framework, unifying a leader oracle, a random oracle, and a failure detector oracle, was proposed in [25]. Unfortunately, algorithms derived by instantiating that framework with a given oracle are clearly not as efficient as ad hoc algorithms devised directly with that oracle. Efficient indulgent consensus algorithms were presented in [11]. However, for each oracle, a specific consensus algorithm is given.

f) Contribution: This paper factors out the information structure of efficient indulgent consensus algorithms within a new distributed abstraction, which we call Lambda. This abstraction encapsulates the use of any oracle (random, leader or failure detector) during every individual round of indulgent consensus executions.

Using this abstraction, we construct a generic indulgent consensus algorithm that can be instantiated with any oracle, while assuming the highest number of possible failures, i.e., $f < n/2$, and be as efficient as any ad hoc algorithm initially devised with that oracle in mind. The generic algorithm and the Lambda abstraction are constructed as two pluggable *co-routines*, with the round number of the consensus execution acting as the actual glue. Interestingly, the generic algorithm also enables the composition of different oracles, in an interchangeable way, generalizing the idea of *hybrid* consensus algorithms [1], [28].

The Lambda abstraction is defined by a set of precise properties that can be ensured in different ways according to the underlying oracle. The proposed generic consensus algorithm has a simple structure and its proof relies only on the properties of Lambda. As a convenient consequence, for any instantiation of the generic algorithm, it is sufficient to prove only that the particular oracle (or combination of oracles) ensures the properties defining Lambda. The genericity of the approach promotes a fine-grained composition of consensus optimizations. In particular, specific instantiations of Lambda have the novel, and noteworthy, feature to lead to indulgent consensus algorithms that are, at the same time, oracle-efficient, zero-degrading and one-step-deciding.

Through new impossibility results on indulgent consensus, we also show that (1) no leader-based or failure detector-based consensus algorithm can be, at the same time, zero-degrading and configuration-efficient, and (2) any leader-based consensus algorithm that is oracle-efficient is also zero-degrading (hence such algorithm cannot be configuration-efficient). We furthermore exhibit failure detector-based instantiations of our generic algorithm that are, at the same time, oracle-efficient and configuration-efficient (hence such algorithm cannot be zero-degrading). These results highlight some of the fundamental tradeoffs underlying each oracle.

To summarize, the contributions of this paper are:

- A new distributed programming abstraction, called Lambda, which captures the information structure of indulgent consensus algorithms. This abstraction promotes genericity without hampering efficiency.
- New impossibility results on the composability of consensus optimality properties. We show that some time complexity lower bounds on consensus cannot all be matched with the same algorithm.

g) Road-map: The paper consists of seven sections. Section II presents the computation model. Section III recalls the consensus problem and its oracles. Section IV describes our generic algorithm and gives the specification of the Lambda abstraction. Section V provides particular instances implementing Lambda, each with a specific oracle, and discusses the efficiency of the resulting algorithms. Section VI gives our impossibility results. Finally, Section VII concludes the paper with some final remarks.

II. Distributed Computation Model

A. Processes

The computation model we consider is basically the asynchronous system model of [2], [7], [13], [23]. The system consists of a finite set Π of $n > 1$ processes: $\Pi = \{p_1, \ldots, p_n\}$. A process can fail by *crashing*, *i.e.*, by prematurely halting. Until it possibly crashes, the process behaves according to its specification. If it crashes, the process never executes any other action. By definition, a *faulty* process is a process that crashes, and a *correct* process is a process that never crashes. Let f denote the maximum number of processes that may crash. We assume $f < n/2$ (i.e., a majority of processes are correct).

B. Channels

Processes communicate and synchronize by sending and receiving messages through channels. Channels are assumed to be reliable: messages are not altered or duplicated, and any message sent by a correct process to a correct process is eventually received. There is no assumption about the relative speed of processes nor on message transfer delays.

Our generic algorithm makes use of two communication abstractions that can be built on top of those channels: a (simple) *Broadcast* and a *Reliable Broadcast* abstractions [16]. Implementations of these communication abstractions using reliable channels are straightforward and described in [16], [29]. We simply recall their properties below.

The first abstraction is defined by two primitives: *Broadcast* and *Delivery*, the semantics of which are expressed by three properties, namely, *validity, integrity* and *termination*. When a process p executes *Broadcast*(m) (resp. *Delivery*(m)) we say that p Broadcasts m (resp. Delivers m). We assume that the messages are uniquely identified.

- Validity: If a process Delivers m, then some process has Broadcast m. (No spurious messages.)
- Integrity: A process Delivers a message m at most once. (No duplication.)
- Termination: If a correct process Broadcasts m, then all correct processes Deliver m. (No message Broadcast by a correct process is missed by any correct process.)

Reliable Broadcast is defined by two primitives: *R_Broadcast* and *R_Delivery*, the semantics of which are also expressed by the three properties, *validity, integrity* and *termination*. The only difference with the Broadcast abstraction is the termination property. The latter is stated here as follows.

- Termination: If (1) a correct process R-broadcasts m, or if (2) a process R-delivers m, then all correct processes R-deliver m. (No message R-broadcast by a correct process or R-delivered by a process is missed by a correct process.)

When there is no ambiguity, we use the term *receive* to mean *Deliver* or *R-Deliver*.

III. Consensus and its Oracles

A. The Consensus Problem

In the *Consensus* problem, every process p_i is supposed to *propose* a value v_i and the processes have to *decide* on the same value v, that has to be one of the proposed values. More precisely, the problem is defined by two safety properties (*validity* and *uniform agreement*) and a liveness property (*termination*) [7], [13]:

- Validity: If a process decides v, then v was proposed by some process.
- Uniform Agreement: No two processes decide differently.[1]
- Termination: Every correct process eventually decides on some value.

[1]We actually consider here the *uniform* variant of consensus. It is important to notice that this does not make any difference for indulgent algorithms with the non-uniform variant of consensus (which only requires that no two correct processes decide differently) [15].

For presentation simplicity, we consider only consensus in its binary form: 0 and 1 are the only values that can be proposed.

B. Leader Oracle

A *leader* oracle is a distributed entity that provides the processes with a function leader() that returns a process name each time it is invoked. A unique correct leader is eventually elected but there is no knowledge of when the leader is elected. Several leaders can coexist during an arbitrarily long period of time, and there is no way for the processes to learn when this "anarchy" period is over. The *leader* oracle, denoted Ω, satisfies the following property[2]:

- Eventual Leadership: There is a time t and a correct process p such that, after t, every invocation of leader() by any correct process returns p.

We say that the oracle Ω is *perfect* if, from the very beginning, it provides the processes with the same correct leader.

Ω-based consensus algorithms are described in [4], [11], [21], [27]. These algorithms are (or can be easily made) oracle-efficient: they can reach consensus in two communication steps when no process crashes and the oracle behaves perfectly. The Ω-algorithm of [11] is not only oracle-efficient but also zero-degrading: it reaches consensus in two communication steps when the oracle is perfect and no process crashes during the consensus execution, even if some processes had initially crashed. The notion of zero-degradation means here that a failure in one consensus instance does not impact the performance of future consensus instances (where the failure appears as an initial failure).

C. Failure Detector Oracle

A failure detector $\diamond S$ is defined as follows [7]. Each process p_i is provided with a set

[2]This property refers to a notion of global time. This notion is not accessible to the processes.

denoted by $suspected_i$. If $p_j \in suspected_i$, we say that "p_i suspects p_j". Failure detector $\diamond\mathcal{S}$ satisfies the following properties:

- Strong Completeness: Eventually, every process that crashes is permanently suspected by every correct process.
- Eventual Weak Accuracy: There is a time after which some correct process is never suspected by the correct processes.

A $\diamond\mathcal{S}$ oracle is *perfect* if it suspects exactly processes that have crashed and hence behaves as a perfect failure detector. That is, in addition to strong completeness, the oracle never suspects any non-crashed process.

Several $\diamond\mathcal{S}$-based consensus algorithms have been proposed in [7], [11], [18], [26], [31]. The algorithms of [18], [26], [31] reach consensus in two communication steps when the oracle is perfect and no process crashes: they are oracle-efficient. The algorithm of [11] is also zero-degrading.

D. Random oracle

A *random* oracle provides each process p_i with a function random that outputs a binary value randomly chosen. Basically, random() outputs 0 (resp. 1) with probability $1/2$.

A binary consensus algorithm based on such a random oracle is described in [3]. In the case where the processes, which have not initially crashed, have the same initial value, this algorithm reaches consensus in two communication steps. (Note that, in this case, the algorithm does not actually use the underlying *random* oracle. This situation does actually correspond to the notion of *configuration-efficiency* investigated in Section VI.)

IV. A Generic Consensus Algorithm

Our generic consensus algorithm is described in Figure 1. As announced in the Introduction, its combination of genericity and efficiency lies in the use of an appropriate information structure, called Lambda. This abstraction exports a single function, itself denoted lambda(), and this function encapsulates the use of any underlying oracle. The algorithm borrows its skeleton from [26].[3]

A. Two Phases per Round

A process p_i starts a consensus execution by invoking function consensus(v_i) where v_i is the value proposed by p_i for consensus. Function consensus() is made up of two concurrent tasks: $T1$ (the main task) and $T2$ (the decision dissemination task). The execution of statement *return*(v) by any process p_i terminates the consensus execution (as far as p_i is concerned) and returns the decided value v to p_i.

In their main task (i.e., $T1$), the processes proceed through consecutive asynchronous rounds. Each round is made up of two phases. During the first phase of a round, the processes strive to select the same value, called an *estimate* value. Then, the processes try to decide during the second phase. This occurs when they got the same value at the end of the selection phase.

Each process p_i manages three local variables: r_i (current round number), $est1_i$ (estimate of the decision value at the beginning of the first phase), and $est2_i$ (estimate of the decision value at the beginning of the second phase). The specific value $\perp$ denotes a default value (which cannot be proposed by the processes).

h) First phase (selection): The aim of this phase (line 104) is to provide all processes with the same estimate ($est2_i$). When this occurs, a decision will be obtained during the second phase. To attain this goal, this phase is made up of a call to the function lambda (). For any process p_i, the function has two input parameters: a round number r (an integer) and $est1_i$ representing either a possible consensus value (i.e., a binary value in our case) or the specific

[3] More precisely, our algorithm has the same structure as the $\diamond\mathcal{S}$-based algorithm of [26] but differs in its first phase. This phase relies on the Lambda abstraction instead of the particular $\diamond\mathcal{S}$ failure detector.

value $\perp$. The function returns as an output parameter $est2_i$, representing either a possible consensus value or the specific value $\perp$. A fundamental property ensured by this function is the following. For any two processes, p_i and p_j that, during a round r, return from lambda$(r, -)$, $est2_i$ and $est2_j$ respectively, we have:
$((est2_i \neq \perp) \ \wedge \ (est2_j \neq \perp))$
$\Rightarrow \ (est2_i = est2_j = v)$.

i) Second phase (commitment): The second phase (lines 105-111) starts with an exchange of the new estimates (note that these are equal to the same value v or to $\perp$). Then, the behavior of a process p_i depends on the set rec_i of estimate values it has gathered. There are three cases to consider [26].

1) If $rec_i = \{v\}$, then p_i decides on v. Note that in this case, as p_i receives v from $(n - f) > n/2$ processes, any process p_j receives v from at least one process. (Obviously, the algorithm remains correct if a process waits for y messages, with $n/2 < y \leq n - f$.)
2) If $rec_i = \{v, \perp\}$, then p_i considers v as its new estimate value (this is because some process might have decided v) and proceeds to the next round.
3) if $rec_i = \{\perp\}$, then p_i adopts $\perp$ as estimate, and proceeds to the next round. The adoption of $\perp$ as estimate is transitory. (An estimate equal to $\perp$ will be updated to a non-$\perp$ value by the lambda() function called at the next round.)

It is important to notice that, at any round, lines 108 and 110 are mutually exclusive: if some process executes one of them, then no process can execute the other. This exclusion actually "locks" the decided value, thereby guaranteeing consensus agreement. It follows that when the processes start a new round $r > 1$, $est1_i$ variables whose values are different from $\perp$ are equal to the same value v: the value is indeed *locked*.

The use of Reliable Broadcast (line 108) has the following motivation. Given that any process that decides stops participating in the sequence of rounds, and all processes do not necessarily decide during the same round, it is possible that processes that proceed to round $r + 1$ wait forever for messages from processes that have terminated at r. By disseminating the decided value, the Reliable Broadcast primitive prevents such occurrences.

B. The Lambda *Abstraction*

This section defines the properties of our Lambda abstraction. It states the properties any infinite sequence of invocations of lambda () function has to satisfy. Section V will then show how these properties can be ensured using various underlying oracles.

- Validity: No process invokes lambda $(1, \perp)$. Moreover, if p_i returns $a \neq \perp$ from lambda $(r, -)$, then some process invoked lambda (r', a) with $r' \leq r$.
- Quasi-agreement: Let p_i and p_j be any two processes that invoke lambda $(r, -)$ and get the values a and b, respectively. If $(a \neq \perp)$ and $(b \neq \perp)$ then $a = b$.
- Fixed point: For any round number r, if all processes that invoke lambda $(r, -)$, invoke it with the same value a (i.e., lambda (r, a)), then a and $\perp$ are the only values that can be returned by any invocation of lambda $(r', -)$, $\forall r' \geq r$.[4]
- Termination: For any round r, if all correct processes invoke lambda $(r, -)$, then all correct processes return from the invocation.
- Eventual convergence: If all correct processes keep on repeatedly invoking lambda () with increasing round numbers, then there is a round r such that lambda $(r, -)$ returns the same non-$\perp$ value to all correct processes.

[4]Using the terminology of [7], this property means that the value v is *locked*: no different value v' can be decided. Using the terminology of [13], the configuration is *monovalent* as only v can be decided.

Function consensus(v_i)

Task $T1$:
(101)$est1_i \leftarrow v_i$; $r_i \leftarrow 0$; % r_i: current round number %

(102)**while** $true$ **do** % Sequence of rounds %
(103) $r_i \leftarrow r_i + 1$;
==
% PHASE 1 of round r_i: Select phase (Determine an estimate value) %
(104) $est2_i \leftarrow$ lambda $(r_i, est1_i)$; %
% Here, $((est2_i \neq \perp) \wedge (est2_j \neq \perp)) \Rightarrow (est2_i = est2_j = v)$ %
==
% PHASE 2 of round r_i: Commit phase (Decision and value locking) %
(105) *Broadcast* PHASE$2(r_i, est2_i)$;
(106) **wait until** (PHASE$2(r_i, est2)$ messages Delivered from $(n - f)$ processes);
(107) **let** $rec_i = \{\ est2 \mid$ PHASE$2(r_i, est2)$ has been Delivered $\}$;
(108) **case** $(rec_i = \{v\})$ **then** *R_Broadcast* DECIDE(v); *return*(v) % terminates %
(109) $(rec_i = \{v, \perp\})$ **then** $est1_i \leftarrow v$
(110) $(rec_i = \{\perp\})$ **then** $est1_i \leftarrow \perp$
(111) **endcase**;
==
(112)**enddo**

Task $T2$:
(113)**upon** *R_Delivery* **of** DECIDE(v): *return*(v) % terminates consensus %

Fig. 1. The Generic Consensus Algorithm

C. Correctness of the Generic Algorithm

Assuming $f < n/2$ and the Lambda abstraction, this section discusses how the algorithm of Figure 1 satisfies the validity, uniform agreement and termination properties of consensus.

Theorem 1: (Validity) Any decided value is a proposed value.

Proof The specific value $\perp$ cannot be decided (line 108). By the validity of the Lambda abstraction as well as the integrity and validity properties of the Broadcast primitives, the $est1_i$ and $est2_i$ variables can only contain proposed values or $\perp$. $\Box_{Theorem\ 1}$

Theorem 2: (Uniform Agreement) No two processes decide different values.

Proof This follows from the quasi-agreement and fixed point properties of the Lambda abstraction as well as the integrity and validity properties of the broadcast primitives.

Let r be the smallest round during which some process decides ("decide v during r" means "during r, execute line 108 with $rec_i = \{v\}$"). We first show that (1) the processes that decide during r decide v, and (2) all estimates are equal to v at the end of r. We then show from (2) that no other value can be decided in a subsequent round.

At the end of the first phase of r (just after line 104 and before line 105), we have $((est2_i \neq \perp) \wedge (est2_j \neq \perp)) \Rightarrow (est2_i = est2_j = v)$. This follows from the quasi-agreement property of the Lambda abstraction. As $\perp$ cannot be decided, it follows that, if two processes decide during r, they decide the same non-$\perp$ value at line 108.

Assuming that some process p_i decides v during r, we now prove that the estimate $est1_j$ of any process p_j that progresses to $r + 1$ is equal to v at the end of r. As there are more than $n/2$ PHASE2 messages carrying the same v (these are the messages that allowed p_i to decide v during r), then by the integrity and validity properties of the Broadcast primitive, p_j must have Delivered at least one of those

PHASE2(r, v) messages. Consequently, p_j executed line 109, and updated $est1_j$ to v. It follows that all the processes that start $r + 1$ have their estimate variables equal to v.

Consider now round $r + 1$. As the estimates of the processes that start $r + 1$ are equal to the same non-$\perp$ value, namely v, it follows from the fixed point property of the Lambda abstraction that no value different from v can be decided in a future round. $\square_{Theorem\ 2}$

Lemma 1: No correct process blocks forever in a round.

Proof This follows from (1) f being the maximum number of processes that can crash, (2) the termination properties of Lambda, as well as (3) the termination and integrity properties of the Broadcast primitives. We show this more precisely below.

If a process decides, then by the termination property of the Reliable Broadcast of the DECIDE message, all correct processes decide. Hence, no correct process blocks forever in a round. Assume by contradiction that no process decides. Let r be the smallest round number in which some correct process p_i blocks forever. So, p_i blocks at line 104 or 106. By the termination property of Lambda, no correct process blocks forever at line 104. Consider now the case of line 106: the fact that p_i cannot block follows directly from the assumption that there are at most f crashed processes, from which we conclude that at least $(n - f)$ processes Broadcast the corresponding messages: the integrity and termination properties of the Broadcast lead to a contradiction. $\square_{Lemma\ 1}$

Theorem 3: (Termination) Every correct process eventually decides.

Proof This follows from (1) Lemma 1, (2) f being the maximum number of processes that can crash, (3) the termination and convergence properties of Lambda, as well as (4) the integrity and termination properties of the Broadcast primitives.

The proof is by contradiction. Assume that no process decides. By Lemma 1, the correct processes progress from round to round. Hence, by the eventual convergence property of Lambda, there is a round r during which all processes have the same value v at the end of their first phase. It follows that the PHASE2$(r, -)$ messages carry the same value v. By the integrity and termination properties of the Broadcast, for any process p_i executing the second phase of r, we have $rec_i = \{v\}$, from which we conclude that the correct processes decide at line 108. $\square_{Theorem\ 3}$

V. ORACLE-BASED IMPLEMENTATIONS OF Lambda

This section describes modules that implement the Lambda abstraction using various forms of oracles, namely, a leader oracle, a failure detector oracle, and a random oracle. A module provides a piece of code implementing the function lambda$(r_i, est1_i)$ invoked by a process p_i and returning a value in $est2_i$. All these implementations use a local variable per process p_i, denoted $prev\_est1_i$. The aim of this local variable, is to ensure the fixed point property by keeping the last non-$\perp$ value of $est1_i$ over subsequent invocations of the lambda() function.

As observed in the Introduction, the generic consensus algorithm and the Lambda abstraction act as two software components pluggable together. More precisely, from the point of view of each process p_i, an execution of the generic algorithm together with an implementation of the Lambda abstraction is the conjunction of two "co-routines" that progress in turn. During each round, the control flow at p_i moves during the first phase from the main co-routine (i.e., the algorithm), to the co-routine implementing Lambda, and then returns to the main co-routine for the second phase before progressing to the next round.

A. Leader Module

An implementation of lambda (r_i, est_i) based on Ω is described in Figure 2. After resetting $est1_i$ to its last non-$\perp$ value (if $est1_i = \perp$), every process p_i first invokes the oracle Ω. The latter provides p_i with the identity of some process (i.e., the name of a leader - line 202). Then, p_i exchange with all other processes its current estimate value plus the name of the process it considers leader (line 203). When p_i has got such messages from at least $(n - f)$ processes (line 204), p_i checks if some process p_ℓ is considered leader by a majority of the processes. If there is such a process p_ℓ, and p_i has got its current estimate value $est1_\ell$, then p_i considers $est1_\ell$ as the value of $est2_i$. In the other cases, p_i sets $est2_i$ to $\perp$ (lines 206-208).

Theorem 4: The algorithm of Figure 2 implements Lambda using Ω.

Proof [5] We have to show that the LO module, described in Figure 2, satisfies the validity, quasi-agreement, fixed point, termination and eventual convergence properties defined in Section IV-B.

Validity and termination follow directly from the algorithm and f being the maximum number of processes that can crash. Quasi-agreement follows from the fact that an $est2_i$ variable is equal to $\perp$ or the $est1_\ell$ value of a process p_ℓ (let us notice that there is at most one process p_ℓ that is considered leader by a majority of processes). The fixed point property follows from the fact that, if all $est1_i$ are equal to some value v, then only v or $\perp$ can be output at line 208 (notice that the second phase of the consensus algorithm can set $est1_i$ only to v or $\perp$). Therefore, if all $est1_i$ are equal to v at the beginning of r, due to the management of the $prev\_est1_i$ variables, all processes (that have a value) will have the same value v after executing line 201 during the next round. The eventual convergence property follows from the fact that there is a time after which all processes have the same correct leader p_ℓ; when this occurs, p_ℓ imposes its estimate to all processes.

$\Box_{Theorem\ 4}$

The generic consensus algorithm, instantiated with such an implementation of the function lambda (), boils down to the Ω algorithm of [11] which, as we pointed out, is the most efficient Ω-based algorithm we know of. When Ω is perfect, it provides the processes with the same correct process as a leader from the very beginning. Moreover, due to line 205, the test of lines 206-207 is then satisfied as p_i has got the estimate value from the common leader p_ℓ (which is $leader_i$). It follows that, at line 208, $est2_i$ is set to the value of $est1_\ell$ (which is different from $\perp$). Thus, it is easy to see that, in this case, all processes get the same estimate value after the execution of lambda $(1, -)$, and the protocol terminates in two communication steps despite initial process crashes. So, the algorithm is zero-degrading (hence, also oracle-efficient).

B. Failure Detector Module

A $\diamond\mathcal{S}$-based implementation of lambda (r_i, est_i) is described in Figure 3. Its principle is particularly simple. Each round has a coordinator (line 302) that tries to impose its estimate value to all the processes (line 303). As the coordinator can crash, a process relies on the strong completeness property of failure detector $\diamond\mathcal{S}$ in order not to wait indefinitely (line 304). If a process p_i gets a value v from the round coordinator, p_i sets $est2_i$ to v. If p_i suspects the current round coordinator to have crashed, p_i sets $est2_i$ to $\perp$ (line 305). In order not to miss the correct process that is eventually no longer suspected (eventual weak accuracy), all processes have to be considered in turn as coordinator until a value is decided. This is realized with the help of the *mod* function at line 302.

Theorem 5: The algorithm of Figure 3 implements Lambda using $\diamond\mathcal{S}$.

Proof The proof is similar to the one used for Theorem 4. $\Box_{Theorem\ 5}$

[5] Line 205 is related to the zero-degradation property (see the discussion below). It concerns neither the safety nor the liveness of the Lambda abstraction, and is consequently not used in this proof.

LO

(201) **if** $(est1_i = \bot)$ **then** $est1_i \leftarrow prev\_est1_i$ **else** $prev\_est1_i \leftarrow est1_i$ **endif**;
(202) $leader_i \leftarrow$ leader();
(203) *Broadcast* PHASE1_LO $(r_i, est1_i, leader_i)$;
(204) **wait until** (PHASE1_LO $(r_i, -, -)$) messages Delivered from $\geq (n - f)$ processes);
(205) **wait until** $\big($(PHASE1_LO $(r_i, -, -)$) Delivered from $leader_i$) $\vee$ ($leader_i \neq$ leader())$\big)$;
(206) **if** $(\exists\ \ell$: PHASE1_LO$(r_i, -, \ell)$ Delivered from a majority of processes
(207) $\wedge$ PHASE1_LO $(r_i, est1_\ell, -)$ Delivered from p_ℓ)
(208) **then** $est2_i \leftarrow est1_\ell$ **else** $est2_i \leftarrow \bot$ **endif**

Fig. 2. Ω-Based Module

FD

(301) **if** $(est1_i = \bot)$ **then** $est1_i \leftarrow prev\_est1_i$ **else** $prev\_est1_i \leftarrow est1_i$ **endif**;
(302) **let** $c = (r_i \ mod\ n) + 1$; % p_c: coordinator process for that round %
(303) **if** $(i = c)$ **then** *Broadcast* PHASE1_FDO$(r_i, est1_i)$ **endif**;
(304) **wait until** ((PHASE1_FDO(r_i, v) Delivered from p_c) $\vee$ $(p_c \in suspected_i)$);
(305) **if** PHASE1_FDO(r_i, v) received **then** $est2_i \leftarrow v$ **else** $est2_i \leftarrow \bot$ **endif**

Fig. 3. $\diamond\mathcal{S}$-Based Module

The generic consensus algorithm instantiated with such an implementation of the Lambda abstraction boils down to the $\diamond\mathcal{S}$-based consensus algorithm described in [26]. It is easy to see that the resulting algorithm needs only two communication steps to decide when the first coordinator is correct and the $\diamond\mathcal{S}$ failure detector is perfect. So, this algorithm is oracle-efficient. However, it is not zero-degrading because if the first coordinator has crashed, the algorithm has to proceed to the second round. In this case, at least three communication steps are required to decide. (One to proceed from the first to the second round, and the others in the second round.) A simple way to get a zero-degrading $\diamond\mathcal{S}$-based consensus algorithm, despite the crash of the first coordinator, consists in customizing its first round. More precisely, the lambda () function is then implemented as follows:

- Round $r = 1$: This round uses a module similar to the one described in Figure 2 except for its line 202 (the line where the Ω oracle is used) which is replaced by the following statement:

$$leader_i \leftarrow \ \min\ (\Pi - suspected_i).$$

- Round $r > 1$: These rounds use the module described in Figure 3.

When the failure detector is perfect, all processes get the same correct process as the "leader" of the first round, do not suspect it, and consequently the decision is obtained during that round. When we consider this implementation of the lambda () function, the first round satisfies validity, quasi-agreement, fixed point and termination, whereas the other rounds additionally satisfy eventual convergence.

C. Random Module

A random based implementation of Lambda is described in Figure V-C. When a process starts a new round with $est1_i = \bot$, it sets $est1_i$ randomly to 0 or 1. The processes then exchange their current estimates values and each process p_i looks for a value that is a majority value. If such a value is obtained, process p_i assigns it to $est2_i$. If p_i does not see a majority estimate value, p_i sets $est2_i$ to

$\perp$. Note that, if $est1_i = \perp$ at the beginning of a round, process p_i can conclude that both values have been proposed. Note also that, at the beginning of the first round, no estimate value is equal to $\perp$.

Theorem 6: Using random, the algorithm of Figure V-C implements the validity, quasi-agreement and termination properties of Lambda.

Proof Straightforward from the algorithm. $\Box_{Theorem\ 6}$

The randomized consensus algorithm obtained when using this implementation of the Lambda abstraction boils down to the algorithm proposed in [3]. As noticed in Section III-D, in the particular case where the processes that have not initially crashed propose the same initial value, this algorithm does not use the underlying random oracle and reaches consensus in two communication steps. Actually, this algorithm uses the random oracle to allow the processes to "eventually" start a round with the same estimate value. When that round is reached, the process can decide without the help of the oracle. The algorithm satisfies the eventual convergence property with probability 1.

D. Module Composition

Interestingly, it is possible to provide implementations of the Lambda abstraction based on combinations of the previous LO, FD and RO modules (or even any XY module satisfying the validity, quasi-agreement, fixed point, termination and eventual convergence properties of the Lambda abstraction). Such combinations provide *hybrid* consensus algorithms that generalize the specific combinations that have been proposed so far (namely, the algorithms that combine a failure detector oracle with a random oracle [1], [28]).

As examples, let us consider two module combinations that merge the LO and FD modules.

- The first combination is the LO_FD_1 module defined as follows:
  - The odd rounds of lambda() are implemented with the LO module (Figure 2),
  - The even rounds are implemented with the FD module (Figure 3) where the coordinator p_c is defined as follows: $c = ((r_i/2) \bmod n) + 1$.[6]
  (Note that this composition requires a slight modification of the round coordinator definition.)
- The second combination is the LO_FD_2 module defined as follows. Each round of lambda() is implemented by the the concatenation made up of the LO module immediately followed by the FD module.

It is easy to see that each of the resulting LO_FD_1 and LO_FD_2 modules satisfies the properties associated with the Lambda abstraction. Other combinations could be defined in a similar way. Such combinations have to be such that, given a round r, all processes use the same module composition during r.

Appropriate combinations merging the RO module to the LO and FD modules, provide implementations of Lambda that satisfies its validity, quasi-agreement, fixed point and termination properties. As far as the eventual convergence property is concerned, it is satisfied if the LO (or FD) module is involved in an infinite number of rounds. In the other cases, it is only satisfied with probability 1 (assuming RO is involved in an infinite number of rounds).

In that sense, the generic algorithm provides indulgent consensus algorithms that can benefit from the best of several "worlds" (leader oracle, failure detector oracle or random oracle).

E. One-Step Decision

This section considers two additional assumptions that, when satisfied by an actual run, allow the consensus algorithm to expedite

[6] In that way, no process is a priori prevented to be the correct process that eventually is not suspected by the other processes.

RO

if ($est1_i = \perp$) **then** $est1_i \leftarrow$ random() **endif**;
Broadcast PHASE1_RO$(r_i, est1_i)$
wait until (PHASE1_RO $(r_i, -)$) messages Delivered from $\geq (n - f)$ processes);
if ($\exists\ v$: PHASE1_RO(r_i, v) received from a majority of processes)
  then $est2_i \leftarrow v$ **else** $est2_i \leftarrow \perp$ **endif**

Fig. 4. Random-Based Module

the decision, i.e., to terminate in one communication step [5]. Each of these assumptions relies on a specific a priori agreement. More precisely, the first one assumes an a priori agreement on a particular value, and allows a one-step decision when enough processes do propose that value. The other assumes that there is a predefined majority set of processes, and allows a one-step decision when those processes do propose the same value.

Interestingly, these additional assumptions can be merged in any deterministic implementation of Lambda (i.e., LO, FD or SV - defined in the next section). In the following, we illustrate this idea by combining a specific implementation with the Ω-based implementation of Lambda given in Figure 2. For a specific initial configuration, the processes can reach a decision in one communication step. Interestingly, the one-step decision characteristic of the resulting algorithm does not impact the performance of the algorithm when the initial configuration is not a specific one.

When combined with the LO module, each of the modules introduced below (named PV -Figure 5-, and PS -Figure 6-), assumes that there is a leader p_ℓ defined in the LO module (line 202 of Figure 2). The merging of the LO module with the PV module (respectively, PS) concerns only the first round. This merging is achieved through the LO module of Figure 2. More precisely, we test if there is a leader as defined in the LO module [line 202 of Figure 2]). In that case, we invoke the PV (respectively, PS) module.

j) Existence of a Privileged Value: Some applications have the property that some predetermined value (α) appears much more often than other values. This means that α is usually much more often proposed than other values. The a priori knowledge of such a predetermined value can help expedite the decision. This can be done by plugging the following module PV, described in Figure 5, as described above. The idea underlying this module is the following. If there is a leader p_ℓ (Figure 2, line 206), and a majority of processes including p_ℓ have their current estimates equal to α, then p_i decides α (line 502). Otherwise, if p_i has Delivered a PHASE1_LO message carrying α, then p_i updates its $prev\_est1_i$ local variable to α. It is easy to see that, in any run where the processes that have not initially crashed propose α and the oracle is perfect, the processes decide in one communication step.

Theorem 7: The previous combination merging the LO and PV modules provides a correct implementation of the Lambda abstraction. When used in the consensus protocol described in Figure 1, it allows the processes to decide in one communication step when the privileged value α is the only proposed value and the oracle is perfect.

Proof Let us first notice that, if no process executes line 502, then the only difference between the LO+PV merging and LO lies in the fact that some processes possibly set their $prev\_est1_i$ variables to α (which is then a proposed value). This does not modify the output of the lambda () function for that round.

Let us now consider the case where a process decides at line 502. In this case, the process decides α which is then the estimate value of the unique leader p_ℓ of that round. Moreover, as in this case α has been sent by a majority of processes and $f < n/2$, it follows that all the processes that do not decide at line 502 execute line 503 updating $prev\_est1_i$ to α.

If a process decides during the second phase, it necessarily decides the current estimate of p_ℓ, namely α. Assume some process p_i does not decide during the second phase. There are two cases to consider. Process p_i executes line 109 or line 110 (Figure 1). If p_i executes line 109, then p_i sets $est1_i$ to α. If p_i executes line 110, then p_i sets $est1_i$ to $\perp$, but then at the beginning of the next round, p_i will reset $est1_i$ to $prev\_est1_i$ whose value is now α (it has been set to that value at line 503). It follows that if a process decides at line 502, (1) the processes that decide during the second phase of the round decide α, and (2) the processes that start the next round have their estimates $est1_i$ equal to α, and so, no other value can be decided in a later round.

Let us now consider the case where all the processes propose the privileged value α. No matter which process p_ℓ is considered leader, p_i receives α from a majority of processes including p_ℓ, and consequently decides at line 502. So, in that case, the processes decide in one communication step. $\Box_{Theorem\ 7}$

k) Existence of a Privileged Set of Processes: This specific case considers the situations where there is a predetermined set S of processes ($f < n/2 < |S|$), initially known by each process. When processes in S do not crash and propose the same value, it is possible to terminate in one communication step. The corresponding PV module is described in Figure 6. It is merged with the LO as described previously. The proof that this combination is correct is similar to the previous one.

l) Discussion: When we look at the PV and PS modules, we can observe that they are dual in the following sense. PV is "value" oriented: it considers the case where the processes propose the same predetermined value. On the other hand, PS is "control" oriented: it considers the case where a predefined set of processes propose the same non-predetermined value. In both cases, the improvement results from an a priori agreement, either on the value α or on the set S.

Let us notice that the introduction of module PV or module PS does not add any communication cost to the resulting consensus algorithm. Hence, defining a priori a privileged value α, or a majority set of processes S, and trying to exploit it to expedite a decision is an overhead-free operation (whatever the value chosen or the set selected, it entails no additional communication cost).

Let us finally observe that when the set of values does not allow the PV (or PS) module to terminate in one communication step, the consensus algorithm remains zero-degrading: it still terminates in two communication steps despite initial crashes if the underlying (Ω or $\Diamond\mathcal{S}$) oracle behaves perfectly. So, combining the additional assumption that "there is a privileged value", or "there is a predefined majority set of processes", with the use of an Ω or $\Diamond\mathcal{S}$ oracles does not prevent zero-degradation. In a precise sense, one-step decision and zero-degradation are compatible. This has to be contrasted with the main result of the next section (Theorem 9) which shows that configuration-efficiency cannot be combined with zero-degradation.

VI. Configuration Efficiency

As we have pointed out (Section III-D and Section V-C), when all the processes (that have not initially crashed) propose the same initial value, no underlying oracle is necessary to obtain a decision, and two communication steps are necessary and sufficient to get a decision. We call an algorithm that matches this lower bound each time the initial values are the same, and no matter how the underlying oracle behaves, a *configuration-efficient* algorithm.

This section first presents a simple module that, when used to implement the first round of the lambda() function (the other rounds being implemented with the LO, FD

PV

(501) **if** (PHASE1_LO$(r_i, \alpha, -)$ Delivered from a majority of processes including p_ℓ)
(502) **then** *R_Broadcast* DECIDE(α); *return*(α) % terminates %
(503) **else** **if** (PHASE1_LO$(r_i, \alpha, -)$ has been Delivered) **then** $prev\_est1_i \leftarrow \alpha$ **endif**
(504) **endif**

Fig. 5. Privileged Value Module

PS

(601) **if** ($\forall p_j \in S :$ PHASE1_LO$(r_i, v, -)$ Delivered from $p_j \ \wedge \ p_\ell \in S$)
% all processes of S have sent the same value v %
(602) **then** *R_Broadcast* DECIDE(v); *return*(v) % terminates %
(603) **else** **let** x **be** the estimate Delivered from a $p_j \in S$; $prev\_est1_i \leftarrow x$
(604) **endif**

Fig. 6. Privileged Set of Processes Module

or RO[7] module, or a combination of them), provides a configuration-efficient consensus algorithm. Then, it is shown (Theorem 9) that no Ω or $\Diamond\mathcal{S}$-based consensus protocol can be, at the same time, configuration-efficient and zero-degrading. In a precise sense, these optimization techniques are incompatible. On the contrary, and as we have seen, a *random*-based consensus algorithm can be at the same time configuration-efficient and zero-degrading when a single value is proposed (we have also seen that, in this case, the random oracle is not used). Finally, we show that it is possible to trade zero-degradation with configuration-efficiency in the case of $\Diamond\mathcal{S}$ (in the sense that both cannot be simultaneously provided), but not in the case of Ω (Theorem 10).

A. Same Value Module

We present here a simple implementation of the lambda () function (Figure 7) that is only based on the values proposed by the processes: no oracle is used. The processes exchange their current estimates values and look for a value that is a majority value. If such a value exists, process p_i assigns it to $est2_i$. If a process p_i does not see a majority estimate value, it sets $est2_i$ to $\perp$. Except for its first line, this module, denoted by is SV, is the same as the RO module: the only difference lies in the fact that SV does not use any underlying oracle. When used in the first round, SV and RO are actually the same module (this is because, when the first round starts, we have $est1_i \neq \perp$).

Theorem 8: The algorithm of Figure 7 ensures the validity, quasi-agreement, fixed point and termination properties of Lambda.

Proof Straightforward from the algorithm. $\Box_{Theorem\ 8}$

This implementation does not satisfy eventual convergence, so the termination of the consensus algorithm is not guaranteed in all cases. This implementation is particularly interesting when there is a high likelihood that the processes do propose the same value. In such cases, the resulting consensus algorithm is zero-degrading and terminates in two communication steps. Interestingly, this module can be used in combination with other oracle-based modules to provide Lambda implementations.

m) Remark: Assuming no more than f processes can crash, the implementation of Lambda based on the SV (resp. RO, LO and FD) module ensures the validity, quasi-

[7]In the latter case, the eventual convergence property of Lambda can only be guaranteed with probability 1.

```
SV

if (est1_i = ⊥) then est1_i ← prev_est1_i else prev_est1_i ← est1_i endif;
Broadcast PHASE1_SV(r_i, est_i);
wait until (PHASE1_SV (r_i, −)) messages Delivered from ≥ (n − f) processes);
if (∃ v: PHASE1_SV(r_i, v) Delivered from a majority of processes)
   then est2_i ← v else est2_i ← ⊥ endif
```

Fig. 7. Same Value Module

agreement, fixed point and termination properties of Lambda. SV does not ensure eventual convergence, while (due to their powerful underlying oracles) LO and FD ensure it. RO can be seen as at an intermediate level as it ensures eventual convergence only probabilistically. So, RO can be seen as SV enriched with a "relatively weak oracle" whose aim is to help obtain eventual convergence.

B. Impossibility Result

Assuming $f < n/2$ and A being any Ω-based or $\diamond\mathcal{S}$-based consensus algorithm, we show here that A cannot be simultaneously zero-degrading and configuration-efficient. The proof technique uses indistinguishability arguments, both (1) among runs without crashes and runs with crashes, and (2) among runs where the oracle behaves perfectly and runs where the oracle does not.

As our impossibility result is a lower bound on a number of communication steps, it is stated and proved assuming a round-based full-information algorithm [14], i.e., we assume that processes exchange the maximum information they can exchange within every message. That is, whenever a process transmits a message, it transmits it to all and includes its current state. Processes proceed in rounds. In every round, a process sends a message to all processes. Before moving to the next round, the process waits for messages from a majority and, depending on the oracle, it waits for other messages. Basically, if the algorithm is based on Ω, the process also waits for a message from the leader. If the algorithm is based on $\diamond\mathcal{S}$, the process also waits for a message from every non-suspected process. Assuming this strong model strengthens our impossibility result.

As we defined it, the notion of zero-degradation means that the algorithm reaches consensus in 2 communication steps (rounds) in any run where no process crashes, except possibly initially, and the oracle behaves perfectly. Let us recall here that Ω behaves *perfectly* in a run if it always outputs the same correct process to all processes in that run; $\diamond\mathcal{S}$ behaves *perfectly* when every process that crashes is eventually suspected (in a permanent way) by all correct processes and no process is suspected before it crashes. Note that when we say here that a run *reaches* consensus, we mean that all correct processes have decided.

Theorem 9: Assuming $f < n/2$, no Ω or $\diamond\mathcal{S}$-based consensus algorithm can be zero-degrading and configuration-efficient.

Proof We need to prove that if consensus can be reached in 2 rounds in any run of A where the oracle behaves perfectly and no process crashes, except initially (i.e., A is zero-degrading), then there exists a run of A that does not reach consensus in two communication steps even if all processes have the same initial values (i.e., A cannot be configuration-efficient).

The proof first considers the case of 3 processes (i.e., $n = 3$ and $f = 1$). Then, it considers the case $n > 3$.[8] Furthermore, we first consider a communication-closed model [14]:

[8]In a sense, we consider a reduction proof technique similar to that of [22], where the case $n = 3$ and $f = 1$ is first considered, and then generalized.

If a process does not deliver, in a round r, a message Broadcast to it in that round, the process does never deliver that message. We shall later discuss the generalization of our proof argument for the communication-open model.

Case $n = 3$. We prove our result using simple indistinguishability arguments among four runs: R_1-R_4. We depict the important messages of these runs in Figure 8. Messages that are Broadcast and not Delivered or sent by a process to itself are not indicated; the value proposed by a process is indicated inside square brackets "[ ]" and the value decided inside parenthesis "( )".

1) At least until the second round. run R_1 is similar for processes p_2 and p_3 to a run where p_1 has initially crashed. Without loss of generality, if A is zero-degrading, then A must have a run such as run R_1. In this run, Ω would output the same leader process, say p_2, at all processes and $\diamond\mathcal{S}$ would output, say p_1, at all processes. In both cases, messages from p_1 are missed by p_2 and p_3, because they consider p_1 to have initially crashed: messages received by p_1 are hence not relevant for our purpose. Processes p_2 and p_3 decide, say 0, after two rounds (zero-degradation), and hence p_1 eventually decides 0 as well.
2) Process p_3 cannot distinguish R_1 from run R_2 up to the second round: p_3 receives exactly the same messages from p_2 and gets the same output from its oracle. Hence, p_3 decides 0 after 2 rounds in run R_2 as well. Processes p_1 and p_2 have to eventually decide 0 in R_2, even if p_3 crashes immediately after deciding at round 2.
3) Consider now run R_3. After two rounds, p_1 and p_2 cannot distinguish run R_3 from R_2 where p_3 might have decided 0 after 2 steps. Assume again that p_3 crashes immediately after the second round in R_3. Hence, p_1 and p_2 have also to eventually decide 0 in run R_3 as well. Process p_3 cannot distinguish run R_3 from R_4.
4) In run R_4, all processes might have the same initial value and if we assume that A is configuration-efficient, then p_3 must decide 1 after 2 steps. A contradiction as p_3 cannot distinguish R_4 from R_3.

Case $n > 3$. Let us divide the set of processes into three subsets P_1, P_2 and P_3, each of size less than or equal to $\lceil n/3 \rceil$. Moreover, let f be such that $n/2 > f \geq \lceil n/3 \rceil$. Given that $f \geq \lceil n/3 \rceil$, all processes of a given subset can crash in a run. The generalization of the proof for any n is then straightforward. We replace process p_1 with the set of processes P_1, process p_2 with set P_2 and process p_3 with set P_3, i.e., if we crash p_i, we crash all processes in P_i, if p_i proposes a value v, we have that value proposed by all processes in P_i, and so forth. We then follow the same reasoning as for the proof with $n = 3$ to construct four runs, as R_1-R_4, and make them contradictory.

Consider now a communication open model. Any message that is sent by a correct process is eventually received by all correct processes. The point here is that, given the asynchronous characteristic of the channels, these messages might arrive after the decision was made. They do simply not change the contradiction argument.

$\Box_{Theorem\ 9}$

C. Trading Zero-Degradation

This section discusses the possibility of trading zero-degradation with configuration-efficiency. The issue is to devise algorithms that are oracle-efficient and configuration-efficient instead of zero-degrading. Recall that oracle-efficiency (that concerns crash-free runs) is a weaker property than zero-degradation (that concerns runs with only initial crashes).

n) The case of Ω: We show below that any Ω-based consensus algorithm that is oracle-efficient is also zero-degrading. As a consequence of our previous impossibility result

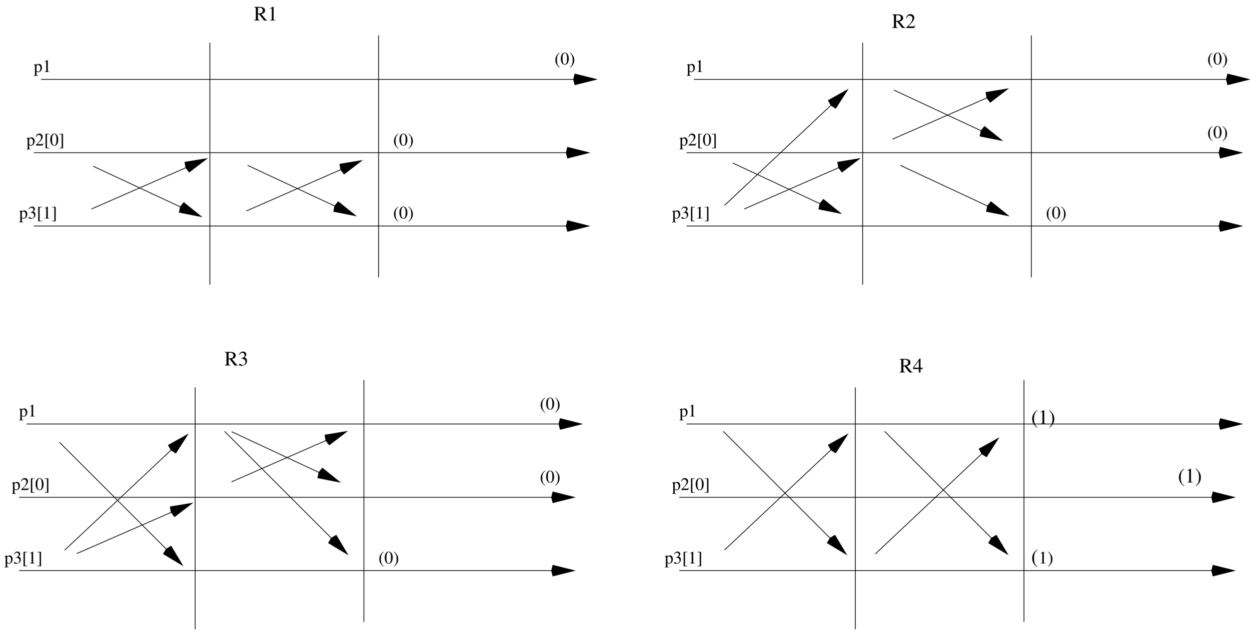

Fig. 8. Runs *R1-R4* used in the proof of Theorem 9

(Theorem 9), there is no way to trade the zero-degrading characteristic of oracle-efficient Ω-based consensus algorithms with configuration-efficiency.

Theorem 10: Assuming $f < n/2$, any oracle-efficient Ω-based consensus algorithm is also zero-degrading.

Proof Let A be any Ω-based consensus algorithm assuming $f < n/2$. We need to show that if any run of A, where the oracle behaves perfectly and no process crashes, reaches consensus in 2 steps, then any run of A where the oracle behaves perfectly and no process crashes, except initially, also reaches consensus in 2 steps. Our proof argument is by contradiction.

For presentation simplicity, we simply consider the case of 3 processes, i.e., $f = 1$ in a communication-closed model and exhibit two contradictory runs, depicted in Figure 9. A being oracle-efficient, let us assume without loss of generality that (1) A has a run R_1 where some process, say p_1, crashes initially, (2) the oracle behaves perfectly, say by permanently electing p_2 and (3) either p_2 or p_3 decides after round 2. Let us observe that the oracle Ω might also output p_2 in a run R_2 that is similar to R_1, except for p_1 that does not crash. Processes $p2$ and p_3 cannot distinguish R_1 from R_2, as in both runs they get the same information from Ω. But then, in R_2, Ω behaves perfectly, no process crashes, and some process decides after round 2: a contradiction as A is an oracle-efficient Ω-based consensus algorithm. $\Box_{Theorem}$ 10

o) The case of $\Diamond\mathcal{S}$: We give here a $\Diamond\mathcal{S}$-based consensus algorithm assuming $f < n/2$ that is oracle-efficient and configuration-efficient. The algorithm is however not zero-degrading (this would contradict Theorem 9).

The idea in this $\Diamond\mathcal{S}$-based consensus algorithm consists in customizing its first round. More precisely, the lambda () function is implemented as follows:

- Round $r = 1$: The processes exchange their current estimates values and every process waits until it receives (a) a majority of estimates and (b) an estimate from all non-suspected processes. If a process (1) receives the same value v, or (2) receives values from all processes and v is

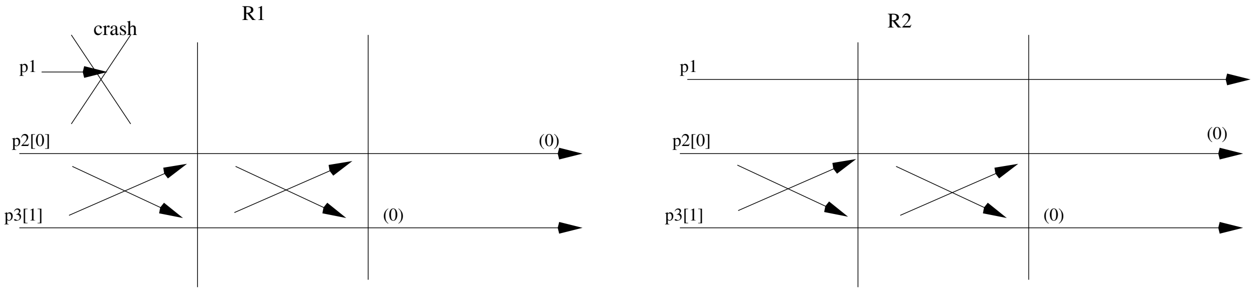

Fig. 9. Runs *R1-R2* used in the proof of Theorem 10

(2.1) the majority among those, or (2.2) p_1's value if there is no such majority, then the process returns v. Otherwise, the process returns $\perp$. (Let us notice that this first round relies only on the majority of correct processes assumption and the strong completeness property of $\diamond\mathcal{S}$.)

- Round $r > 1$: These rounds use the module described in Figure 3.

When we consider this implementation of the lambda () function, the first round satisfies validity, quasi-agreement, fixed point and termination and the other rounds additionally satisfy eventual convergence. Consider a consensus algorithm using this implementation of lambda. Clearly, if all processes propose the same value, the processes return that value within lambda and all correct processes decide in 2 steps (configuration-efficiency). Similarly, if the oracle behaves perfectly and no process crashes, then all processes get all values and return the same value within lambda: thus, the processes decide in 2 steps (oracle-efficiency).

p) On perfect Ω and $\diamond\mathcal{S}$ oracles: While Ω and $\diamond\mathcal{S}$ have the same computational power [8], [10], it is important to notice that perfect Ω and perfect $\diamond\mathcal{S}$ are not equivalent. This observation might help get an intuition of why any oracle-efficient Ω-based consensus algorithm is also zero-degrading, which is not the case with $\diamond\mathcal{S}$.

VII. Conclusion

This paper dissects the information structure of consensus algorithms that rely on oracles such as the random oracle [3], the leader oracle [21], and the failure detector oracle [7]. The algorithms we consider are indulgent toward their oracle: no matter how the underlying oracle behaves, consensus safety is never violated.

We encapsulate the information structure of indulgent consensus algorithms within a new distributed abstraction, called Lambda. Basically, this abstraction is invoked in the first phase of every round of our generic consensus algorithm. It highlights a deep unity in the design principles of consensus solutions and allows to state, in an abstract way, the properties the oracles equipping the underlying asynchronous system have to satisfy. This not only allows to provide a single proof of a family of algorithms (whatever the oracles effectively used), but also promotes the design of new oracles appropriately defined according to the practical setting in which the system has to run.

The genericity of the approach helps devise new consensus algorithms that are, at the same time, oracle-efficient, zero-degrading and one-step-deciding. We could also derive new lower bounds such as the impossibility of having an algorithm that is zero-degrading and configuration-efficient at the same time.

It is important to notice that our approach does not aim at unifying all algorithmic principles underlying consensus. In particular, we focused on indulgent consensus algorithms [15]. Figuring out how to include for instance the *synchronous* dimension in our general information structure might be feasible along the lines of [14], but requires a careful study. Similarly, we did not consider the *memory* dimension of consensus algorithms to unify models with crash-stop message passing, crash-recovery message passing, and shared memory [4]. Integrating this dimension to the oracle one is yet another non-trivial challenge.

Acknowledgements

We are grateful to the referees for their useful comments that helped us improve the presentation of this paper. We would also like to thank Partha Dutta, Achour Mostéfaoui, Bastian Pochon and Sergio Rasjbaum for discussions on consensus algorithms and lower bounds.

References

[1] Aguilera M.K. and Toueg S., Failure Detection and Randomization: a Hybrid Approach to Solve Consensus. *SIAM Journal of Computing*, 28(3):890-903, 1998.

[2] Attiya H. and Welch J., *Distributed Computing: Fundamentals, Simulations and Advanced Topics.* McGraw-Hill, 451 pages, 1998.

[3] Ben-Or M., Another Advantage of Free Choice: Completely Asynchronous Agreement Protocols. *Proc. 2nd ACM Symp. on Principles of Distributed Computing (PODC'83)*, ACM Press, pp. 27-30, Montréal (CA), 1983.

[4] Boichat R., Dutta P., Frolund S. and Guerraoui R., Deconstructing Paxos. *SIGACT News, Distributed Computing Column*, 2003.

[5] Brasileiro F., Greve F., Mostefaoui A. and Raynal M. Consensus in One Communication Step. *Proc. 6th Int. Conference on Parallel Computing Technologies (PaCT'01)*, Novosibirsk, Springer Verlag LNCS 2127, pp. 42-50, 2001.

[6] Brzezinsky J., Hélary J.-M., Raynal M. and Singhal M., Deadlock Models and a General Algorithm for Distributed Deadlock Detection. *Journal of Parallel and Distributed Computing*, 31(2):112-125, 1995.

[7] Chandra T. and Toueg S., Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM*, 43(2):225-267, 1996.

[8] Chandra T., Hadzilacos V. and Toueg S., The Weakest Failure Detector for Solving Consensus. *Journal of the ACM*, 43(4):685-722, 1996.

[9] Chor B. and Nelson L., Solvability in Asynchronous Environments: Finite Interactive Tasks. *SIAM Journal on Computing*, 29(2):351-377, 1999.

[10] Chu F., Reducing Ω to $\diamond W$. *Information Processing Letters*, 67(6):289-293, 1998.

[11] Dutta P. and Guerraoui R., Fast Indulgent Consensus with Zero Degradation. *Proc. 4th European Dependable Computing Conference (EDCC'02)*, Toulouse (France), Springer-Verlag LNCS 2485, pp. 191-208, 2002.

[12] Dwork C., Lynch N. and Stockmeyer L., Consensus in the Presence of Partial Synchrony. *Journal of the ACM*, 35(2):288-323, 1988.

[13] Fischer M.J., Lynch N. and Paterson M.S., Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, 32(2):374-382, 1985.

[14] Gafni E., A Round-by-Round Failure Detector: Unifying Synchrony and Asynchrony. *Proc. 17th ACM Symposium on Principles of Distributed Computing (PODC'98)*, ACM Press, pp. 143-152, Puerto Vallarta (Mexico), 1998.

[15] Guerraoui R., Indulgent Algorithms. *Proc. 19th ACM Symposium on Principles of Distributed Computing (PODC'00)*, ACM Press, pp. 289-298, Portland (OR), 2000.

[16] Hadzilacos V. and Toueg S., Reliable Broadcast and Related Problems. In *Distributed Systems*, ACM Press (S. Mullender Ed.), New-York, pp. 97-145, 1993.

[17] Hélary J.-M., Mostéfaoui A. and Raynal M., A General Scheme for Token- and Tree-Based Distributed Mutual Exclusion Algorithms. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 5(11):1185-1196, 1994.

[18] Hurfin M., Mostéfaoui A. and Raynal M., A Versatile Family of Consensus Protocols Based on Chandra-Toueg's Unreliable Failure Detectors. *IEEE Transactions on Computers*, 51(4):395-408, 2002.

[19] Keidar I. and Rajsbaum S., On the Cost of Fault-Tolerant Consensus when There are No Faults: a Tutorial. *SIGACT News, Distributed Computing Column*, 32(2):45-63, 2001.

[20] Kshemkalyani A.D. and Singhal M., On the Characterization and Correctness of Distributed Deadlock Detection. *Journal of Parallel and Distributed Computing*, 22(1):44-59, 1994.

[21] Lamport L., The Part-Time Parliament. *ACM Transactions on Computer Systems*, 16(2):133-169, 1998.

[22] Lamport L., Shostak R. and Pease L., The Byzantine General Problem. *ACM Transactions on Pro-*

gramming Languages and Systems, 4(3):382-401, 1982.

[23] Lynch N., Distributed Algorithms. *Morgan Kaufmann Pub.*, San Francisco (CA), 872 pages, 1996.

[24] Mostéfaoui A., Rajsbaum S. and Raynal M., Conditions on Input Vectors for Consensus Solvability in Asynchronous Distributed Systems. *Proc. 33rd ACM Symposium on Theory of Computing (STOC'01)*, ACM Press, pp. 153-162, Crete (Greece), 2001.

[25] Mostéfaoui A., Rajsbaum S. and Raynal M., A Versatile and Modular Consensus Protocol. *Proc. Int. IEEE Conference on Dependable Systems & Networks (DSN'02)*, IEEE Computer Press, pp. 364-373, Washington D.C., 2002.

[26] Mostéfaoui A. and Raynal M., Solving Consensus Using Chandra-Toueg's Unreliable Failure Detectors: a General Quorum-Based Approach. *Proc. 13th Int. Symposium on Distributed Computing (DISC'99)*, Bratislava (Slovaquia), Springer-Verlag LNCS 1693, pp. 49-63, 1999.

[27] Mostéfaoui A. and Raynal M., Leader-Based Consensus. *Parallel Processing Letters*, 11(1):95-107, 2001.

[28] Mostéfaoui A., Raynal M. and Tronel F., The Best of Both Worlds: a Hybrid Approach to Solve Consensus. *Proc. Int. IEEE Conference on Dependable Systems & Networks (DSN'00)*, New York, pp. 513-522, 2000.

[29] Rodrigues L. and Verissimo P., Topology-Aware Algorithms for Large Scale Communications. in *Advances in Distributed Systems*, Springer-Verlag LNCS 1752, pp. 127-156, 2000.

[30] Sanders B., The Information Structure of Distributed Mutual Exclusion Algorithms. *ACM Transactions on Computer Systems*, 5(3):284-299, 1987.

[31] Schiper A., Early Consensus in an Asynchronous System with a Weak Failure Detector. *Distributed Computing*, 10:149-157, 1997.

Rachid Guerraoui Rachid Guerraoui has been professor in computer science since 1999, at EPFL (Ecole Polytechnique Fédérale de Lausanne) in Switzerland, where he founded the distributed programming laboratory. Prior to that, Rachid Guerraoui has been with HP Labs in Palo Alto, the Center of Atomic Energy (CEA) in Saclay (France), and the Centre de Recherche de l'Ecole des Mines de Paris. He got his PhD in 1992 from the Univ of Orsay.

His research interests include distributed algorithms and distributed programming languages. In these areas, Rachid Guerraoui has been Principal Investigator of a number of research grants and has published papers in various journals (including ACM TOCS, ACM CS, Distributed Computing, IEEE TC, IEEE TSE, IEEE TPDS, IEEE Computer, IEEE Software, IPL, Theoretical Computer Science) and conferences (including ACM PODC, ACM OOPSLA, ECOOP, IEEE ICDCS,IEEE DSN, DISC, Europar, IEEE SRDS, IFIP TCS, etc.). Rachid Guerraoui has served in program committees for various conferences (including PODC, OOPSLA, ECOOP, DISC, ICDCS, DSN, SRDS, etc.) and chaired the program committees of ECOOP 1999, Middleware 2001, and SRDS 2002.

Michel Raynal Michel Raynal has been a professor of computer science since 1981. At IRISA (CNRS-INRIA-University joint computing research laboratory located in Rennes), he founded a research group on Distributed Algorithms in 1983. His research interests include distributed algorithms, distributed computing systems, networks and dependability. His main interest lies in the fundamental principles that underly the design and the construction of distributed computing systems. He has been Principal Investigator of a number of research grants in these areas, and has been invited by many universities to give lectures about operating systems and distributed algorithms in Europe, south and north America, Asia and Africa. He is serving as an editor for several international journals.

Professor Michel Raynal has published more than 75 papers in journals (Acta Informatica, Distributed Computing, Comm. of the ACM, Information and Computation, Journal of Computer and System Sciences, JPDC, IEEE TC, IEEE TSE, IEEE TKDE, IEEE TPDS, IEEE Computer, IEEE Software, IPL, PPL, Theoretical Computer Science, Real-Time Systems Journal, The Computer Journal, etc.); and more than 160 papers in conferences (ACM STOC, ACM PODC, ACM SPAA, IEEE ICDCS, IEEE DSN, DISC, IEEE IPDPS, Europar, FST-TCS, IEEE SRDS, etc.). He has also written seven books devoted to parallelism, distributed algorithms and systems (MIT Press and Wiley). Michel Raynal has served in program committees for more than 70 international conferences (including ACM PODC, DISC, ICDCS, DSN, SRDS, etc.) and chaired the program committee of more than 15 international conferences. He is currently serving as the chair of the steering committee leading the DISC symposium series. Michel Raynal got the IEEE ICDCS best paper Award three times in a row: 1999, 2000 and 2001.