The Weakest Failure Detector for Eventual Consistency

Swan Dubois* Rachid Guerraoui† Petr Kuznetsov‡§
Franck Petit* Pierre Sens*

Abstract

In its classical form, a *consistent* replicated service requires all replicas to witness the same evolution of the service state. Assuming a message-passing environment with a majority of correct processes, the necessary and sufficient information about failures for implementing a general state machine replication scheme ensuring consistency is captured by the Ω failure detector.

This paper shows that in such a message-passing environment, Ω is also the weakest failure detector to implement an *eventually consistent* replicated service, where replicas are expected to agree on the evolution of the service state only after some (*a priori* unknown) time.

In fact, we show that Ω is the weakest to implement eventual consistency in *any* message-passing environment, *i.e.*, under any assumption on when and where failures might occur. Ensuring (strong) consistency in any environment requires, in addition to Ω, the quorum failure detector Σ. Our paper thus captures, for the first time, an exact computational difference between building a replicated state machine that ensures consistency and one that only ensures eventual consistency.

1 Introduction

State machine replication [20, 25] is the most studied technique to build a highly-available and consistent distributed service. Roughly speaking, the idea consists in replicating the service, modeled as a state machine, over several processes and ensuring that all replicas behave like one correct and available state machine, despite concurrent invocations of operations and failures of replicas. This is typically captured using the abstraction of a *total order broadcast* [3], where messages represent invocations of the service operations from clients to replicas (server processes). Assuming that the state machine is deterministic, delivering the invocations in the same total order ensures indeed that the replicas behave like a single state machine. Total order broadcast is, in turn, typically implemented by having the processes agree on which batch of messages to execute next, using the *consensus* abstraction [21, 3]. (The two abstractions, consensus and total order broadcast, were shown to be equivalent [3].)

Replicas behaving like a single one is a property generally called *consistency*. The sole purpose of the abstractions underlying the state machine replication scheme, namely consensus and total order broadcast, is precisely to ensure this consistency, while providing at the same time *availability*,

*Sorbonne Universités, UPMC Université Paris 6, Équipe REGAL, LIP6, F-75005, Paris, France; CNRS, UMR 7606, LIP6, F-75005, Paris, France; Inria, Équipe-projet REGAL, F-75005, Paris, France; firstname.lastname@lip6.fr

†École Polytechnique Fédérale de Lausanne, Switzerland; rachid.guerraoui@epfl.ch

‡Télécom ParisTech; petr.kuznetsov@telecom-paristech.fr

§The research leading to these results has received funding from the Agence Nationale de la Recherche, under grant agreement N ANR-14-CE35-0010-01, project DISCMAT.

namely that the replicated service does not stop responding. The inherent costs of these abstractions are sometimes considered too high, both in terms of the necessary computability assumptions about the underlying system [11, 2, 1], and the number of communication steps needed to deliver an invocation [21, 22].

An appealing approach to circumvent these costs is to trade consistency with what is sometimes called *eventual consistency* [24, 28]: namely to give up the requirement that the replicas *always* look the same, and replace it with the requirement that they only look the same *eventually*, *i.e.*, after a finite but not *a priori* bounded period of time. Basically, *eventual consistency* says that the replicas can diverge for some period, as long as this period is finite.

Many systems claim to implement general state machines that ensure eventual consistency in message-passing systems, *e.g.*, [19, 7]. But, to our knowledge, there has been no theoretical study of the exact assumptions on the information about failures underlying those implementations. This paper is the first to do so: using the formalism of failure detectors [3, 2], it addresses the question of the minimal information about failures needed to implement an eventually consistent replicated state machine.

It has been shown in [2] that, in a message-passing environment with a majority of correct processes, the weakest failure detector to implement consensus (and, thus, total order broadcast [5]) is the *eventual leader* failure detector, denoted Ω. In short, Ω outputs, at every process, a *leader* process so that, eventually, the same correct process is considered leader by all. Ω can thus be viewed as the weakest failure detector to implement a generic replicated state machine ensuring consistency and availability in an environment with a majority of correct processes.

We show in this paper that, maybe surprisingly, the weakest failure detector to implement an *eventually consistent* replicated service in this environment (in fact, in *any* environment) is still Ω. We prove our result via an interesting generalization of the celebrated "CHT proof" by Chandra, Hadzilacos and Toueg [2]. In the CHT proof, every process periodically extracts the identifier of a process that is expected to be correct (the *leader*) from the *valencies* of an ever-growing collection of locally simulated runs. We carefully adjust the notion of valency to apply this approach to the weaker abstraction of *eventual consensus*, which we show to be necessary and sufficient to implement eventual consistency.

Our result becomes less surprising if we realize that a correct majority prevents the system from being *partitioned*, and we know that both consistency and availability cannot be achieved while tolerating partitions [1, 13, 8]. Therefore, in a system with a correct majority of processes, there is no gain in weakening consistency: (strong) consistency requires the same information about failures as eventual one. In an arbitrary environment, however, *i.e.*, under any assumptions on when and where failures may occur, the weakest failure detector for consistency is known to be $\Omega + \Sigma$, where Σ [8] returns a set of processes (called a *quorum*) so that every two such quorums intersect at any time and there is a time after which all returned quorums contain only correct processes. We show in this paper that ensuring eventual consistency does not require Σ: only Ω is needed, even if we do not assume a majority of correct processes. Therefore, Σ represents the exact difference between consistency and eventual consistency. Our result thus theoretically backs up partition-tolerance [1, 13] as one of the main motivations behind the very notion of eventual consistency.

We establish our results through the following steps:

- We give precise definitions of the notions of *eventual consensus* and *eventual total order broadcast*. We show that the two abstractions are equivalent. These underlie the intuitive notion of eventual consistency implemented in many replicated services [7, 6, 4].
- We show how to extend the celebrated CHT proof [2], initially establishing that Ω is necessary

for solving consensus, to the context of eventual consensus. Through this extension, we indirectly highlight a hidden power of the technique proposed in [2] that somehow provides more than was used in the original CHT proof.

- We present an algorithm that uses Ω to implement, in any message-passing environment, an eventually consistent replicated service. The algorithm features three interesting properties: (1) An invocation can be performed after the optimal number of two communication steps, even if a majority of processes is not correct and even during periods when processes disagree on the leader, i.e., partition periods; [1] (2) If Ω outputs the same leader at all processes from the very beginning, then the algorithm implements total order broadcast and hence ensures consistency; (3) *Causal* ordering is ensured even during periods where Ω outputs different leaders at different processes.

The rest of the paper is organized as follows. We present our system model and basic definitions in Section 2. In Section 3, we introduce abstractions for implementing eventual consistency: namely, eventual consensus and eventual total order broadcast, and we prove them to be equivalent. We show in Section 4 that the weakest failure detector for eventual consensus in any message-passing environment is Ω. We present in Section 5 our algorithm that implements eventual total order broadcast using Ω in any environment. Section 6 discusses related work, and Section 7 concludes the paper. In the optional appendix, we present some proofs omitted from the main paper, discuss an alternative (seemingly relaxed but equivalent) definition of eventual consensus, and recall basic steps of the CHT proof.

2 Preliminaries

We adopt the classical model of distributed systems provided with the failure detector abstraction proposed in [3, 2]. In particular we employ the simplified version of the model proposed in [14, 17].

We consider a message-passing system with a set of processes $\Pi = \{p_1, p_2, \ldots, p_n\}$ ($n \geq 2$). Processes execute steps of computation asynchronously, *i.e.*, there is no bound on the delay between steps. However, we assume a discrete global clock to which the processes do not have access. The range of this clock's ticks is $\mathbb{N}$. Each pair of processes are connected by a reliable link.

Processes may fail by *crashing*. A *failure pattern* is a function $F : \mathbb{N} \to 2^{\Pi}$, where $F(t)$ is the set of processes that have crashed by time t. We assume that processes never recover from crashes, *i. e.*, $F(t) \subseteq F(t+1)$. Let $\mathit{faulty}(F) = \bigcup_{t \in \mathbb{N}} F(t)$ be the set of *faulty* processes in a failure pattern F; and $\mathit{correct}(F) = \Pi - \mathit{faulty}(F)$ be the set of *correct* processes in F. An *environment*, denoted $\mathcal{E}$, is a set of failure patterns.

A *failure detector history H with range $\mathcal{R}$* is a function $H : \Pi \times \mathbb{N} \to \mathcal{R}$, where $H(p, t)$ is interpreted as the value output by the failure detector module of process p at time t. A *failure detector $\mathcal{D}$ with range $\mathcal{R}$* is a function that maps every failure pattern F to a nonempty set of failure detector histories. $\mathcal{D}(F)$ denotes the set of all possible failure detector histories that may be output by $\mathcal{D}$ in a failure pattern F.

For example, at each process, the *leader failure detector* Ω outputs the id of a process; furthermore, if a correct process exists, then there is a time after which Ω outputs the id of the same correct process at every correct process. Another example is the *quorum failure detector* Σ, which outputs a set of processes at each process. Any two sets output at any times and by any processes intersect, and eventually every set output at any correct process consists of only correct processes.

[1] Note that three communication steps are, in the worst case, necessary when strong consistency is required [22].

An *algorithm* $\mathcal{A}$ is modeled as a collection of n deterministic automata, where $\mathcal{A}(p)$ specifies the behavior of process p. Computation proceeds in *steps* of these automata. In each step, identified as a tuple $(p, m, d, \mathcal{A})$, a process p atomically (1) receives a single message m (that can be the empty message λ) or accepts an *input* (from the external world), (2) queries its local failure detector module and receives a value d, (3) changes its state according to $\mathcal{A}(p)$, and (4) sends a message specified by $\mathcal{A}(p)$ for the new state to every process or produces an *output* (to the external world). Note that the use of λ ensures that a step of a process is always enabled, even if no message is sent to it.

A *configuration* of an algorithm $\mathcal{A}$ specifies the local state of each process and the set of messages in transit. In the *initial* configuration of $\mathcal{A}$, no message is in transit and each process p is in the initial state of the automaton $\mathcal{A}(p)$. A *schedule* S of $\mathcal{A}$ is a finite or infinite sequence of steps of $\mathcal{A}$ that respects $\mathcal{A}(p)$ for each p.

Following [17], we model inputs and outputs of processes using *input histories* H_I and *output histories* H_O that specify the inputs each process receives from its application and the outputs each process returns to the application over time. A *run of algorithm $\mathcal{A}$ using failure detector $\mathcal{D}$ in environment $\mathcal{E}$* is a tuple $R = (F, H, H_I, H_O, S, T)$, where F is a failure pattern in $\mathcal{E}$, H is a failure detector history in $\mathcal{D}(F)$, H_I and H_O are input and output histories of $\mathcal{A}$, S is a schedule of $\mathcal{A}$, and T is a list of increasing times in $\mathbb{N}$, where $T[i]$ is the time when step $S[i]$ is taken. $H \in \mathcal{D}(F)$, the failure detector values received by steps in S are consistent with H, and H_I and H_O are consistent with S. An infinite run of $\mathcal{A}$ is *admissible* if (1) every correct process takes an infinite number of steps in S; and (2) each message sent to a correct process is eventually received.

We then define a distributed-computing *problem*, such as consensus or total order broadcast, as a set of tuples (H_I, H_O) where H_I is an input history and H_O is an output history. An algorithm $\mathcal{A}$ using a failure detector $\mathcal{D}$ solves a problem P in an environment $\mathcal{E}$ if in every admissible run of $\mathcal{A}$ in $\mathcal{E}$, the input and output histories are in P. Typically, inputs and outputs represent invocations and responses of *operations* exported by the implemented abstraction. If there is an algorithm that solves P using $\mathcal{D}$, we sometimes, with a slight language abuse, say that $\mathcal{D}$ *implements* P.

Consider two problems P and P'. A *transformation from P to P' in an environment $\mathcal{E}$* [16] is a map $T_{P \to P'}$ that, given any algorithm $\mathcal{A}_P$ solving P in $\mathcal{E}$, yields an algorithm $\mathcal{A}_{P'}$ solving P' in $\mathcal{E}$. The transformation is *asynchronous* in the sense that $\mathcal{A}_P$ is used as a "black box" where $\mathcal{A}_{P'}$ is obtained by feeding inputs to $\mathcal{A}_P$ and using the returned outputs to solve P'. Hence, if P is solvable in $\mathcal{E}$ using a failure detector $\mathcal{D}$, the existence of a transformation $T_{P \to P'}$ in $\mathcal{E}$ establishes that P' is also solvable in $\mathcal{E}$ using $\mathcal{D}$. If, additionally, there exists a transformation from P' to P in $\mathcal{E}$, we say that P and P' are *equivalent in $\mathcal{E}$*.

Failure detectors can be partially ordered based on their "power": failure detector $\mathcal{D}$ is *weaker than* failure detector $\mathcal{D}'$ in $\mathcal{E}$ if there is an algorithm that *emulates* the output of $\mathcal{D}$ using $\mathcal{D}'$ in $\mathcal{E}$ [2, 17]. If $\mathcal{D}$ is weaker than $\mathcal{D}'$, any problem that can be solved with $\mathcal{D}$ can also be solved with $\mathcal{D}'$. For a problem P, $\mathcal{D}^*$ is the *weakest* failure detector to solve P in $\mathcal{E}$ if (a) there is an algorithm that uses $\mathcal{D}^*$ to solve P in $\mathcal{E}$, and (b) $\mathcal{D}^*$ is weaker than any failure detector $\mathcal{D}$ that can be used to solve P in $\mathcal{E}$.

3 Abstractions for Eventual Consistency

We define two basic abstractions that capture the notion of eventual consistency: eventual total order broadcast and eventual consensus. We show that the two abstractions are equivalent: each of them can be used to implement the other.

Eventual Total Order Broadcast (ETOB) The *total order broadcast* (TOB) abstraction [16] exports one operation *broadcastTOB(m)* and maintains, at every process p_i, an output variable d_i. Let $d_i(t)$ denote the value of d_i at time t. Intuitively, $d_i(t)$ is the sequence of messages p_i *delivered* by time t. We write $m \in d_i(t)$ if m appears in $d_i(t)$.

A process p_i *broadcasts a message* m *at time* t by a call to *broadcastTOB(m)*. We say that a process p_i *stably delivers a message* m *at time* t if p_i appends m to $d_i(t)$ and m is never removed from d_i after that, *i.e.*, $m \notin d_i(t-1)$ and $\forall t' \geq t$: $m \in d_i(t')$. Note that if a message is delivered but not *stably* delivered by p_i at time t, it appears in $d_i(t)$ but not in $d_i(t')$ for some $t' > t$.

Assuming that broadcast messages are distinct, the TOB abstraction satisfies:

TOB-**Validity** If a correct process p_i broadcasts a message m at time t, then p_i eventually stably delivers m, i.e., $\forall t'' \geq t' : m \in d_i(t'')$ for some $t' > t$.

TOB-**No-creation** If $m \in d_i(t)$, then m was broadcast by some process p_j at some time $t' < t$.

TOB-**No-duplication** No message appears more than once in $d_i(t)$.

TOB-**Agreement** If a message m is stably delivered by some correct process p_i at time t, then m is eventually stably delivered by every correct process p_j.

TOB-**Stability** For any correct process p_i, $d_i(t_1)$ is a prefix of $d_i(t_2)$ for all $t_1, t_2 \in \mathbb{N}$, $t_1 \leq t_2$.

TOB-**Total-order** Let p_i and p_j be any two correct processes such that two messages m_1 and m_2 appear in $d_i(t)$ and $d_j(t)$ at time t. If m_1 appears before m_2 in $d_i(t)$, then m_1 appears before m_2 in $d_j(t)$.

We then introduce the *eventual total order broadcast* (ETOB) abstraction, which maintains the same inputs and outputs as TOB (messages are broadcast by a call to *broadcastETOB(m)*) and satisfies, in every admissible run, the TOB-Validity, TOB-No-creation, TOB-No-duplication, and TOB-Agreement properties, plus the following relaxed properties for some $\tau \in \mathbb{N}$:

ETOB-**Stability** For any correct process p_i, $d_i(t_1)$ is a prefix of $d_i(t_2)$ for all $t_1, t_2 \in \mathbb{N}$, $\tau \leq t_1 \leq t_2$.

ETOB-**Total-order** Let p_i and p_j be correct processes such that messages m_1 and m_2 appear in $d_i(t)$ and $d_j(t)$ for some $t \geq \tau$. If m_1 appears before m_2 in $d_i(t)$, then m_1 appears before m_2 in $d_j(t)$.

As we show in this paper, satisfying the following optional (but useful) property in ETOB does not require more information about failures.

TOB-**Causal-Order** Let p_i be a correct process such that two messages m_1 and m_2 appear in $d_i(t)$ at time $t \in \mathbb{N}$. If m_2 depends causally of m_1, then m_1 appears before m_2 in $d_i(t)$.

Here we say that a message m_2 *causally depends* on a message m_1 in a run R, and write $m_1 \rightarrow_R m_2$, if one of the following conditions holds in R: (1) a process p_i sends m_1 and then sends m_2, (2) a process p_i receives m_1 and then sends m_2, or (3) there exists m_3 such that $m_1 \rightarrow_R m_3$ and $m_3 \rightarrow_R m_2$.

Eventual Consensus (EC) The *consensus* abstraction (C) [11] exports, to every process p_i, a single operation *proposeC* that takes a binary argument and returns a binary response (we also say *decides*) so that the following properties are satisfied:

C-**Termination** Every correct process eventually returns a response to *proposeC*.

C-**Integrity** Every process returns a response at most once.

C-Agreement No two processes return different values.

C-Validity Every value returned was previously proposed.

The *eventual consensus* (EC) abstraction exports, to every process p_i, operations $proposeEC_1$, $proposeEC_2$, ... that take binary arguments and return binary responses. Assuming that, for all $j \in \mathbb{N}$, every process invokes $proposeEC_j$ as soon as it returns a response to $proposeEC_{j-1}$, the abstraction guarantees that, in every admissible run, there exists $k \in \mathbb{N}$, such that the following properties are satisfied:

EC-Termination Every correct process eventually returns a response to $proposeEC_j$ for all $j \in \mathbb{N}$.

EC-Integrity No process responds twice to $proposeEC_j$ for all $j \in \mathbb{N}$.

EC-Validity Every value returned to $proposeEC_j$ was previously proposed to $proposeEC_j$ for all $j \in \mathbb{N}$.

EC-Agreement No two processes return different values to $proposeEC_j$ for all $j \geq k$.

It is straightforward to transform the binary version of EC into a multivalued one with unbounded set of inputs [23]. In the following, by referring to EC we mean a multivalued version of it.

Equivalence between EC and ETOB It is well known that, in their classical forms, the consensus and the total order broadcast abstractions are equivalent [3]. In this section, we show that a similar result holds for our eventual versions of these abstractions.

The intuition behind the transformation from EC to ETOB is the following. Each time a process p_i wants to ETOB-broadcast a message m, p sends m to each process. Periodically, every process p_i proposes its current sequence of messages received so far to EC. This sequence is built by concatenating the last output of EC(stored in a local variable d_i) to the batch of all messages received by the process and not yet present in d_i. The output of EC is stored in d_i, i.e., at any time, each process delivers the last sequence of messages returned by EC.

The correctness of this transformation follows from the fact that ECeventually returns consistent responses to the processes. Thus, eventually, all processes agree on the same linearly growing sequence of stably delivered messages. Furthermore, every message broadcast by a correct process eventually appears either in the delivered message sequence or in the batches of not yet delivered messages at all correct processes. Thus, by EC-Validity of EC, every message ETOB-broadcast by a correct process is eventually stored in d_i of every correct process p_i forever. By construction, no message appears in d_i twice or if it was not previously ETOB-broadcast. Therefore, the transformation satisfies the properties of ETOB.

The transformation from ETOB to EC is as follows. At each invocation of the EC primitive, the process broadcasts a message using the ETOB abstraction. This message contains the proposed value and the index of the consensus instance. As soon as a message corresponding to a given eventual consensus instance is delivered by process p_i (appears in d_i), p_i returns the value contained in the message.

Since the ETOB abstraction guarantees that every process eventually stably delivers the same sequence of messages, there exists a consensus instance after which the responses of the transformation to all alive processes are identical. Moreover, by ETOB-Validity, every message ETOB-broadcast by a correct process p_i is eventually stably delivered. Thus, every correct process eventually returns from any EC-instance it invokes. Thus, the transformation satisfies the EC specification.

Theorem 1. *In any environment $\mathcal{E}$,* EC *and* ETOB *are equivalent.*

From EC to ETOB To prove this result, it is sufficient to provide a protocol that implements ETOB in an environment $\mathcal{E}$ knowing that there exists a protocol that implements EC in this environment. This transformation protocol $\mathcal{T}_{\mathsf{EC}\rightarrow\mathsf{ETOB}}$ is stated in Algorithm 1. Now, we are going to prove that $\mathcal{T}_{\mathsf{EC}\rightarrow\mathsf{ETOB}}$ implements ETOB. Assume that there exists a message m broadcast by a correct process p_i at time t. As p_i is correct, every correct process receives the message $push(m)$ in a finite time. Then, m appears in the set *toDeliver* of all correct processes in a finite time. Hence, by the termination property of EC and the construction of the function $NewBatch$, there exists ℓ such that m is included in any sequence submitted to $proposeEC_\ell$. By the EC-Validity and the EC-Termination properties, we deduce that p_i stably delivers m in a finite time, that proves that $\mathcal{T}_{\mathsf{EC}\rightarrow\mathsf{ETOB}}$ satisfies the TOB-Validity property. If a process p_i delivers a message m at time t, then m appears in the sequence responded by its last invocation of $proposeEC_\ell$. By construction and by the EC-Validity property, this sequence contains only messages that appear in the set *toDeliver* of a process p_j at the time p_j invokes $proposeEC_\ell$. But this set is incrementally built at the reception of messages *push* that contains only messages broadcast by a process. This implies that $\mathcal{T}_{\mathsf{EC}\rightarrow\mathsf{ETOB}}$ satisfies the TOB-No-creation. As the sequence outputted at any time by any process is the response to its last invocation of *proposeEC* and that the sequence submitted to any invocation of this primitive contains no duplicated message (by definition of the function $NewBatch$), we can deduce from the EC-Validity property that $\mathcal{T}_{\mathsf{EC}\rightarrow\mathsf{ETOB}}$ satisfies the TOB-No-duplication. Assume that a correct process p_i stably delivers a message m, i.e., there exists a time after which m always appears in d_i. By the algorithm, m always appears in the response of *proposeEC* to p_i after this time. As EC-Agreement property is eventually satisfied, we can deduce that m always appears in the response of *proposeEC* for any correct process after some time. Thus, any correct process stably delivers m, and $\mathcal{T}_{\mathsf{EC}\rightarrow\mathsf{ETOB}}$ satisfies the TOB-Agreement. Let τ be the time after which the EC primitive satisfies EC-Agreement and EC-Validity. Let p_i be a correct process and $\tau \leq t_1 \leq t_2$. Let ℓ_1 (respectively ℓ_2) be the integer such that $d_i(t_1)$ (respectively $d_i(t_2)$) is the response of $proposeEC_{\ell_1}$ (respectively $proposeEC_{\ell_2}$). By construction of the protocol and the EC-Agreement and EC-Validity properties, we know that, after time τ, the response of $proposeEC_\ell$ to correct processes is a prefix of the response of $proposeEC_{\ell+1}$. As we have $\ell_1 \leq \ell_2$, we can deduce that $\mathcal{T}_{\mathsf{EC}\rightarrow\mathsf{ETOB}}$ satisfies the ETOB-Stability property. Let p_i and p_j be two correct processes such that two messages m_1 and m_2 appear in $d_i(t)$ and $d_j(t)$ at time $t \geq \tau$. Let ℓ be the smallest integer such that m_1 and m_2 appear in the response of $proposeEC_\ell$. By the EC-Agreement property, we know that the response of $proposeEC_\ell$ is identical for all correct processes. Then , by the ETOB-Stability property proved above, that implies that, if m_1 appears before m_2 in $d_i(t)$, then m_1 appears before m_2 in $d_j(t)$. In other words, $\mathcal{T}_{\mathsf{EC}\rightarrow\mathsf{ETOB}}$ satisfies the ETOB-Total-order property. In conclusion, $\mathcal{T}_{\mathsf{EC}\rightarrow\mathsf{ETOB}}$ satisfies the ETOB specification in an environment $\mathcal{E}$ provided that there exists a protocol that implements EC in this environment.

From ETOB to EC To prove this result, it is sufficient to provide a protocol that implements EC in an environment $\mathcal{E}$ given a protocol that implements ETOB in this environment. This transformation protocol $\mathcal{T}_{\mathsf{ETOB}\rightarrow\mathsf{EC}}$ is stated in Algorithm 2. Now, we are going to prove that $\mathcal{T}_{\mathsf{ETOB}\rightarrow\mathsf{EC}}$ implements EC.

Let p_i be a correct process that invokes $proposeEC_\ell(v)$ with $\ell \in \mathbb{N}$. Then, by fairness and the TOB-Validity property, the construction of the protocol implies that the ETOB primitive delivers the message (ℓ, v) to p_i in a finite time. By the use of the local timeout, we know that p_i returns from $proposeEC_\ell(v)$ in a finite time, that proves that $\mathcal{T}_{\mathsf{ETOB}\rightarrow\mathsf{EC}}$ satisfies the EC-Termination property.

The update of the variable $count_i$ to ℓ for any process p_i that invokes $proposeEC_\ell$ and the assumptions on operations *proposeEC* ensure us that p_i executes at most once the function $DecideEC(\ell, received_i[\Omega_i, \ell])$. Hence, $\mathcal{T}_{\mathsf{ETOB}\rightarrow\mathsf{EC}}$ satisfies the EC-Integrity property.

Algorithm 1 $\mathcal{T}_{\mathsf{EC}\rightarrow\mathsf{ETOB}}$: transformation from EC to ETOB for process p_i

Proof. **Output variable:**

d_i: sequence of messages of M (initially empty) outputted at any time by p_i

Internal variables:

$toDeliver_i$: set of messages of M (initially empty) containing all messages received by p_i

$count_i$: integer (initially 0) that stores the number of the last instance of consensus invoked by p_i

Messages:

$push(m)$ with m a message of M

Functions:

$Send(message)$ sends $message$ to all processes (including p_i)

$NewBatch(d_i, toDeliver_i)$ returns a sequence containing all messages from the set $toDeliver_i \backslash \{m | m \in d_i\}$

On reception of $broadcastETOB(m)$ **from the application**

$Send(push(m))$

On reception of $push(m)$ **from** p_j

$toDeliver_i := toDeliver_i \cup \{m\}$

On reception of d **as response of** $\boldsymbol{proposeEC_\ell}$

$d_i := d$

$count_i := count_i + 1$

$proposeEC_{count_i}(d_i.NewBatch(d_i, toDeliver_i))$

On local timeout

If $count_i = 0$ then

  $count_i := 1$

  $proposeEC_1(NewBatch(d_i, toDeliver_i))$

Let τ be the time after which the ETOB-Stability and the ETOB-Total-order properties are satisfied. Let k be the smallest integer such that any process that invokes $proposeEC_k$ in run r invokes it after τ.

If we assume that there exist two correct processes p_i and p_j that return different values to $proposeEC_\ell$ with $\ell \geq k$, we obtain a contradiction with the ETOB-Stability, ETOB-Total-order, or TOB-Agreement property. Indeed, if p_i returns a value after time τ, that implies that this value appears in d_i and then, by the TOB-Agreement property, this value eventually appears in d_j. If p_j returns a different value from p_i, that implies that this value is the first occurrence of a message associated to $proposeEC_\ell$ in d_j at the time of the return of $proposeEC_\ell$. After that, d_j cannot satisfy simultaneously the ETOB-Stability and the ETOB-Total-order properties. This contradiction shows that $\mathcal{T}_{\mathsf{ETOB}\to\mathsf{EC}}$ satisfies the EC-Agreement property.

If we assume that there exists a process p_i that returns to $proposeEC_\ell$ with $\ell \in \mathbb{N}$ a value that was not proposed to $proposeEC_\ell$, we obtain a contradiction with the TOB-No-creation property. Indeed, the return of p_i from $proposeEC_\ell$ is chosen in d_i that contains the output of the ETOB primitive and processes broadcast only proposed values. This contradiction shows that $\mathcal{T}_{\mathsf{ETOB}\to\mathsf{EC}}$ satisfies the EC-Validity property.

In conclusion, $\mathcal{T}_{\mathsf{ETOB}\to\mathsf{EC}}$ satisfies the EC specification in an environment $\mathcal{E}$ provided that there exists a protocol that implements ETOB in this environment. □

Algorithm 2 $\mathcal{T}_{\mathsf{ETOB}\to\mathsf{EC}}$: transformation from ETOB to EC for process p_i

Internal variables:

$count_i$: integer (initially 0) that stores the number of the last instances of consensus invoked by p_i
d_i: sequence of messages (initially empty) outputted to p_i by the ETOB primitive

Functions:

$First(\ell)$: returns the value v such that (ℓ, v) is the first message of the form $(\ell, *)$ in d_i if such messages exist, $\perp$ otherwise
$DecideEC(\ell, v)$: returns the value v as response to $proposeEC_\ell$

On invocation of *proposeEC*$_\ell(v)$

$count_i := \ell$
$broadcastETOB((\ell, v))$

On local time out

If $First(count_i) \neq \perp$ then
$\lfloor\ DecideEC(count_i, First(count_i))$

4 The Weakest Failure Detector for EC

In this section, we show that Ω is necessary and sufficient for implementing the eventual consensus abstraction EC:

Theorem 2. *In any environment $\mathcal{E}$, Ω is the weakest failure detector for* EC.

Ω is necessary for EC Let $\mathcal{E}$ be any environment. We show below that Ω is weaker than any failure detector $\mathcal{D}$ that can be used to solve EC in $\mathcal{E}$. Recall that implementing Ω means outputting, at every process, the identifier of a *leader* process so that eventually, the same correct leader is output permanently at all correct processes.

First, we briefly recall the arguments use by Chandra et al. [2] in the original CHT proof deriving Ω from any algorithm solving consensus (to get a more detailed survey of the proof please rever to Appendix B or [12, Chapter 3]). The basic observation there is that a run of any algorithm using a failure detector induces a *directed acyclic graph* (DAG). The DAG contains a sample of failure detector values output by $\mathcal{D}$ in the current run and captures causal relations between them. Each process p_i maintains a local copy of the DAG, denoted by G_i: p_i periodically queries its failure detector module, updates G_i by connecting every vertex of the DAG with the vertex containing the returned failure-detector value with an edge, and broadcasts the DAG. An edge from vertex $[p_i, d, m]$ to vertex $[p_j, d', m']$ is thus interpreted as "p_i queried $\mathcal{D}$ for the mth time and obtained value d and after that p_j queried $\mathcal{D}$ for the m'th time and obtained value d'". Whenever p_i receives a DAG G_j calculated earlier by p_j, p_i merges G_i with G_j. As a result, DAGs maintained by the correct processes converge to the same infinite DAG G. The DAG G_i is then used by p_i to simulate a number of runs of the given consensus algorithm $\mathcal{A}$ for all possible inputs to the processes. All these runs are organized in the form of a *simulation tree* Υ_i. The simulation trees Υ_i maintained by the correct processes converge to the same infinite simulation tree Υ.

The outputs produced in the simulated runs of Υ_i are then used by p_i to compute the current estimate of Ω. Every vertex σ of Υ_i is assigned a valency tag based on the decisions taken in all its *extensions* (descendants of σ in Υ_i): σ is assigned a tag $v \in \{0, 1\}$ if σ has an extension in which some process decides v. A vertex is bivalent if it is assigned both 0 and 1. It is then shown in [2] that by locating the same bivalent vertex in the limit tree Υ, the correct process can eventually extract the identifier of the same correct process. (More details can be found in Appendix B and [2, 12].)

We show that this method, originally designed for consensus, can be extended to eventual consensus (i.e., to the weaker EC abstraction). The extension is not trivial and requires carefully adjusting the notion of valency of a vertex in the simulation tree.

Lemma 1. *In every environment $\mathcal{E}$, if a failure detector $\mathcal{D}$ implements* EC *in $\mathcal{E}$, then Ω is weaker than $\mathcal{D}$ in $\mathcal{E}$.*

Proof. Let $\mathcal{A}$ be any algorithm that implements EC using a failure detector $\mathcal{D}$ in an environment $\mathcal{E}$. As in [2], every process p_i maintains a failure detector sample stored in DAG G_i and periodically uses G_i to simulate a set of runs of $\mathcal{A}$ for all possible sequence of inputs of EC. The simulated runs are organized by p_i in an ever-growing *simulation tree* Υ_i. A vertex of Υ_i is the schedule of a finite run of $\mathcal{A}$ "triggered" by a path in G_i in which every process starts with invoking $proposeEC_1(v)$, for some $v \in \{0, 1\}$, takes steps using the failure detector values stipulated by the path in G_i and, once $proposeEC_\ell(v)$ is complete, eventually invokes $proposeEC_{\ell+1}(v')$, for some $v' \in \{0, 1\}$. (For the record, we equip each vertex of Υ_i with the path in G_i used to produce it.) A vertex is connected by an edge to each one-step extension of it. [2]

Note that in every admissible infinite simulated run, EC-Termination, EC-Integrity and EC-Validity are satisfied and that there is $k > 0$ such that for all $\ell \geq k$, the invocations and responses of $proposeEC_\ell$ satisfy the EC-Agreement.

Since processes periodically broadcast their DAGs, the simulation tree Υ_i constructed locally by a correct process p_i *converges* to an infinite simulation tree Υ, in the sense that every finite subtree of Υ is eventually part of Υ_i. The infinite simulation tree Υ, starting from the initial configuration of $\mathcal{A}$ and, in the limit, contains all possible schedules that can triggered by the paths DAGs G_i.

[2] In [2], the simulated schedules form a *simulation forest*, where a distinct simulation tree corresponds to each initial configuration encoding consensus inputs. Here we follow [17]: there is a single initial configuration and inputs are encoded in the form of input histories. As a result, we get a single simulation tree where branches depend on the parameters of $proposeEC_\ell$ calls.

Algorithm 3 Locating a bivalent vertex in Υ.

$k := 1$

$\sigma :=$ root of Υ

while true do
  if σ is k-bivalent then break
  $\sigma_1 :=$ a descendant of σ in which
    EC-Agreement does not hold for $proposeEC_k$
  $\sigma_2 :=$ a descendant of σ_1 in which every correct process
    completes $proposeEC_k$ and receives
    all messages sent to it in σ
  choose $k' > k$ and σ_3, a descendant of σ_2, such that
    k'-tag of σ_3 contains $\{0, 1\}$
  $k := k'$
  $\sigma := \sigma_3$

Consider a vertex σ in Υ identifying a unique finite schedule of a run of $\mathcal{A}$ using $\mathcal{D}$ in the current failure pattern F. For $k > 0$, we say that σ is *k-enabled* if $k = 1$ or σ contains a response from $proposeEC_{k-1}$ at some process. Now we associate each vertex σ in Υ with a set of *valency tags* associated with each "consensus instance" k, called the *k-tag* of σ, as follows:

- If σ is k-enabled and has a descendant (in Υ) in which $proposeEC_k$ returns $x \in \{0, 1\}$, then x is added to the k-tag of σ.
- If σ is k-enabled and has a descendant in which two different values are returned by $proposeEC_k$, then $\perp$ is added to the k-tag of σ.

If σ is not k-enabled, then its k-tag is empty. If the k-tag of σ is $\{x\}$, $x \in \{0, 1\}$, we say that σ is *(k, x)-valent* (*k-univalent*). If the k-tag is $\{0, 1\}$, then we say that σ is *k-bivalent*. If the k-tag of σ contains $\perp$, we say that σ is *k-invalid*

Since $\mathcal{A}$ ensures EC-Termination in all admissible runs extending σ, each k-enabled vertex σ, the k-tag of σ is non-empty. Moreover, EC-Termination and EC-Validity imply that a vertex in which no process has invoked $proposeEC_k$ yet has a descendant in which $proposeEC_k$ returns 0 and a descendant in which $proposeEC_k$ returns 1. Indeed, a run in which only v, $v \in \{0, 1\}$ is proposed in instance k and every correct process takes enough steps must contain v as an output. Thus:

(*) For each vertex σ, there exists $k \in \mathbb{N}$ and σ', a descendant of σ, such that k-tag of σ' contains $\{0, 1\}$.

If the "limit tree" Υ contains a k-bivalent vertex, we can apply the arguments of [2] to extract Ω. Now we show that such a vertex exists in Υ. Then we can simply let every process locate the "first" such vertex in its local tree Υ_i. To establish an order on the vertices, we can associate each vertex σ of Υ with the value m such that vertex $[p_i, d, m]$ of G is used to simulate the last step of σ (recall that we equip each vertex of Υ with the corresponding path). Then we order vertices of Υ in the order consistent with the growth of m. Since every vertex in G has only finitely many incoming edges, the sets of vertices having the same value of m are finite. Thus, we can break the ties in the m-based order using any deterministic procedure on these finite sets.

Eventually, by choosing the first k-bivalent vertex in their local trees Υ_i, the correct processes will eventually stabilize on the same k-bivalent vertex $\tilde{\sigma}$ in the limit tree Υ and apply the CHT extraction procedure to derive the same correct process based on k-tags assigned to $\tilde{\sigma}$'s descendants.

It remains to show that Υ indeed contains a k-bivalent vertex for some k. Consider the procedure described in Algorithm 3 that intends to locate such a vertex, starting with the root of the tree.

For the currently considered k-enabled vertex σ that is *not* k-bivalent (if it is k-bivalent, we are done), we use (*) to locate σ_3, a descendant of σ, such that (1) in σ_3, two processes return different values in $proposeEC_k$ in σ_3, (2) in σ_3, every correct process has completed $proposeEC_k$ and has received every message sent to it in σ, and (3) the k'-tag of σ_3 contains $\{0, 1\}$.

Thus, the procedure in Algorithm 3 either terminates by locating a k-bivalent tag and then we are done, or it never terminates. Suppose, by contradiction, that the procedure never terminates. Hence, we have an infinite admissible run of $\mathcal{A}$ in which no agreement is provided in infinitely many instances of consensus. Indeed, in the constructed path along the tree, every correct process appears infinitely many times and receives every message sent to it. This admissible run violated the EC-Agreement property of EC—a contradiction.

Thus, the correct processes will eventually locate the same k-bivalent vertex and then, as in [2], stabilize extracting the same correct process identifier to emulate Ω. □

Ω is sufficient for EC Chandra and Toueg proved that Ω is sufficient to implement the classical version of the consensus abstraction in an environment where a majority of processes are correct [3]. In this section, we extend this result to the eventual consensus abstraction for any environment.

The proposed implementation of EC is very simple. Each process has access to an Ω failure detector module. Upon each invocation of the EC primitive, a process broadcasts the proposed value (and the associated consensus index). Every process stores every received value. Each process p_i periodically checks whether it has received a value for the current consensus instance from the process that it currently believes to be the leader. If so, p_i returns this value. The correctness of this EC implementation relies on the fact that, eventually, all correct processes trust the same leader (by the definition of Ω) and then decide (return responses) consistently on the values proposed by this process.

Lemma 2. *In every environment $\mathcal{E}$,* EC *can be implemented using Ω.*

Proof. We propose such an implementation in Algorithm 4. Then, we prove that any admissible run r of the algorithm in any environment $\mathcal{E}$ satisfies the EC-Termination, EC-Integrity, EC-Agreement, and EC-Validity properties.

Assume that a correct process never returns from an invocation of *proposeEC* in r. Without loss of generality, denote by ℓ the smallest integer such that a correct process p_i never returns from the invocation of $proposeEC_\ell$. This implies that p_i always evaluates $received_i[\Omega_i, count_i]$ to $\perp$. We know by definition of Ω that, eventually, Ω_i always returns the same correct process p_j. Hence, by construction of ℓ, p_j returns from $proposeEC_0$,..., $proposeEC_{\ell-1}$ and then sends the message $promote(v, \ell)$ to all processes in a finite time. As p_i and p_j are correct, p_i receives this message and updates $received_i[\Omega_i, count_i]$ to v in a finite time. Therefore, the algorithm satisfies the EC-Termination property.

The update of the variable $count_i$ to ℓ for any process p_i that invokes $proposeEC_\ell$ and the assumptions on operations *proposeEC* ensure us that p_i executes at most once the function $DecideEC(\ell, received_i[\Omega_i, \ell])$. Hence, the EC-Integrity property is satisfied.

Let τ_Ω be the time from which the local outputs of Ω are identical and constants for all correct processes in r. Let k be the smallest integer such that any process that invokes $proposeEC_k$ in r invokes it after τ_Ω.

Let ℓ be an integer such that $\ell \geq k$. Assume that p_i and p_j are two processes that respond to $proposeEC_\ell$. Then, they respectively execute the function $DecideEC(\ell, received_i[\Omega_i, \ell])$ and

$DecideEC(\ell, received_j[\Omega_j, \ell])$. By construction of k, we can deduce that $\Omega_i = \Omega_j = p_l$. That implies that p_i and p_j both received a message $promote(v, \ell)$ from p_l. As p_l sends such a message at most once, we can deduce that $received_i[p_l, \ell] = received_j[p_l, \ell]$, that proves that ensures the EC-Agreement property.

Let ℓ be an integer such that $\ell \geq k$. Assume that p_i is a process that respond to $proposeEC_\ell$. The value returned by p_i was previously received from Ω_i in a message of type *promote*. By construction of the protocol, Ω_i sends only one message of this type and this latter contains the value proposed to Ω_i, hence, the EC-Validity property is satisfied.

Thus, Algorithm 4 indeed implements EC in any environment using Ω. □

Algorithm 4 EC using Ω: algorithm for process p_i

Local variables:

$count_i$: integer (initially 0) that stores the number of the last instances of consensus invoked by p_i
$received_i$: two dimensional tabular that stores a value for each pair of processes/integer (initially $\perp$)

Functions:

$DecideEC(\ell, v)$ returns the value v as a response to $proposeEC_\ell$

Messages:

$promote(v, \ell)$ with $v \in \{0, 1\}$ and $\ell \in \mathbb{N}$

On invocation of *proposeEC*$_\ell(v)$

$count_i := \ell$
Send $promote(v, \ell)$ to all

On reception of $promote(v, \ell)$ **from** p_j

$received_i[j, \ell] := v$

On local time out

If $received_i[\Omega_i, count_i] \neq \perp$ do
$\lfloor$ $DecideEC(count_i, received_i[\Omega_i, count_i])$

5 An Eventual Total Order Broadcast Algorithm

We have shown in the previous section that Ω is the weakest failure detector for the EC abstraction (and, by Theorem 1, the ETOB abstraction) in any environment. In this section, we describe an algorithm that directly implements ETOB using Ω and which we believe is interesting in its own right.

The algorithm has three interesting properties. First, it needs only two communication steps to deliver any message when the leader does not change, whereas algorithms implementing classical TOB need at least three communication steps in this case. Second, the algorithm actually implements total order broadcast if Ω outputs the same leader at all processes from the very beginning. Third, the algorithm additionally ensures the property of TOB-Causal-Order, which does not require more information about faults.

The intuition behind this algorithm is as follows. Every process that intends to ETOB-broadcast a message sends it to all other processes. Each process p_i has access to an Ω failure detector module and maintains a DAG that stores the set of messages delivered so far together with their causal dependencies. As long as p_i considers itself the leader (its module of Ω outputs p_i), it periodically sends to all processes a sequence of messages computed from its DAG so that the sequence respects

Algorithm 5 $\mathcal{ETOB}$: protocol for process p_i

Output variable:

d_i: sequence of messages $m \in M$ (initially empty) output by p_i

Internal variables:

$promote_i$: sequence of messages $m \in M$ (initially empty) promoted by p_i when $\Omega_i = p_i$

CG_i: directed graph on messages of M (initially empty) that contains causality dependencies known by p_i

Messages:

$update(CG_i)$ with CG_i a directed graph on messages of M

$promote(promote_i)$ with $promote_i$ a sequence of messages $m \in M$

Functions:

$UpdateCG(m, C(m))$ adds the node m and the set of edges $\{(m', m) | m' \in C(m)\}$ to CG_i

$UnionCG(CG_j)$ replaces CG_i by the union of CG_i and CG_j

$UpdatePromote()$ replaces $promote_i$ by one of the sequences of messages s such that $promote_i$ is a prefix of s, s contains once all messages of CG_i, and for every edge (m_1, m_2) of CG_i, m_1 appears before m_2 in s

On $broadcastETOB(m, C(m))$ **from the application**

$UpdateCG(m, C(m))$

Send $update(CG_i)$ to all

On reception of $update(CG_j)$ **from** p_j

$UnionCG(CG_j)$

$UpdatePromote()$

On reception of $promote(promote_j)$ **from** p_j

If $\Omega_i = p_j$ then

  $d_i := promote_j$

On local time out

If $\Omega_i = p_i$ then

  Send $promote(promote_i)$ to all

the causal order and admits the last delivered sequence as a prefix. A process that receives a sequence of messages delivers it only if it has been sent by the current leader output by Ω. The correctness of this algorithm directly follows from the properties of Ω. Indeed, once all correct processes trust the same leader, this leader promotes its own sequence of messages, which ensures the ETOB specification.

The pseudocode of the algorithm is given in Algorithm 5). Below we present the proof of its correctness, including the proof that the algorithm additionally ensures TOB-Causal-Order.

Lemma 3. *In every environment $\mathcal{E}$, Algorithm $\mathcal{ETOB}$ implements* ETOB *using* Ω.

Proof. First, we prove that any run r of $\mathcal{ETOB}$ in any environment $\mathcal{E}$ satisfies the TOB-Validity, TOB-No-creation, TOB-No-duplication, and TOB-Agreement properties.

Assume that a correct process p_i broadcasts a message m at time t for a given $t \in \mathbb{N}$. We know that Ω outputs the same correct process p_j to all correct processes in a finite time. As p_j is correct, it receives the message $update(CG_i)$ from p_i (that contains m) in a finite time. Then, p_j includes m in its causality graph (by a call to $UnionCG$) and in its promotion sequence (by a call to $UpdatePromote$). As p_j never removes a message from its promotion sequence and is outputted by Ω, p_i adopts the promotion sequence of p_j in a finite time and this sequence contains m, that proves that $\mathcal{ETOB}$ satisfies the TOB-Validity property.

Any sequence outputted by any process is built by a call to $UpdatePromote$ by a process p_i. This function ensures that any message appearing in the computed sequence appears in the graph CG_p. This graph is built by successive calls to $UnionCG$ that ensure that the graph contains only messages received in a message of type $update$. The construction of the protocol ensures us that

such messages have been broadcast by a process. Then, we can deduce that $\mathcal{ETOB}$ satisfies the TOB-No-creation property.

Any sequence outputted by any process is built by a call to *UpdatePromote* that ensures that any message appears only once. Then, we can deduce that $\mathcal{ETOB}$ satisfies the TOB-No-duplication property.

Assume that a correct process p_i stably delivers a message m at time t for a given $t \in \mathbb{N}$. We know that Ω outputs the same correct process p_j to all correct processes after some finite time. Since m appears in every $d_i(t')$ such that $t' \geq t$, we derive that m appears infinitely in $promote_j$ from a given point of the run. Hence, the construction of the protocol and the correctness of p_j implies that any correct process eventually stably delivers m, and $\mathcal{ETOB}$ satisfies the TOB-Agreement property.

We now prove that, for any environment $\mathcal{E}$, for any run r of $\mathcal{ETOB}$ in $\mathcal{E}$, there exists a $\tau \in \mathbb{N}$ satisfying ETOB-Stability, ETOB-Total-order, and TOB-Causal-Order properties in r. Hence, let r be a run of $\mathcal{ETOB}$ in an environment $\mathcal{E}$. Let us define:

- τ_Ω the time from which the local outputs of Ω are identical and constant for all correct processes in r;
- Δ_c the longest communication delay between two correct processes in r;
- Δ_t the longest local timeout for correct processes in r;
- $\tau = \tau_\Omega + \Delta_t + \Delta_c$

Let p_i be a correct process and p_j be the correct elected by Ω after τ_Ω. Let t_1 and t_2 be two integers such that $\tau \leq t_1 \leq t_2$. As the output of Ω is stable after τ_Ω and the choice of τ ensures us that p_i receives at least one message of type *promote* from p_j, we can deduce from the construction of the protocol that there exists $t_3 \leq t_1$ and $t_4 \leq t_2$ such that $d_i(t_1) = promote_j(t_3)$ and $d_i(t_2) = promote_j(t_4)$. But the function *UpdatePromote* used to build $promote_j$ ensures that $promote_j(t_3)$ is a prefix of $promote_j(t_4)$. Then, $\mathcal{ETOB}$ satisfies the ETOB-Stability property after time τ.

Let p_i and p_j be two correct processes such that two messages m_1 and m_2 appear in $d_i(t)$ and $d_j(t)$ at time $t \geq \tau$. Assume that m_1 appears before m_2 in $d_i(t)$. Let p_k be the correct elected by Ω after τ_Ω. As the output of Ω is stable after τ_Ω and the choice of τ ensures us that p_i and p_j receive at least one message of type *promote* from p_j, the construction of the protocol ensures us that we can consider t_1 and t_2 such that $d_i(t) = promote_k(t_1)$ and $d_j(t) = promote_k(t_2)$. The definition of the function *UpdatePromote* executed by p_k allows us to deduce that either $d_i(t)$ is a prefix of $d_j(t)$ or $d_j(t)$ is a prefix of $d_i(t)$. In both cases, we obtain that m_1 appears before m_2 in $d_j(t)$, that proves that $\mathcal{ETOB}$ satisfies the ETOB-Total-order property after time τ.

Let p_i be a correct process such that two messages m_1 and m_2 appear in $d_i(t)$ at time $t \geq 0$. Assume that $m_1 \in C(m_2)$ when m_2 is broadcast. Let p_j be the process trusted by Ω_i at the time p_i adopts the sequence $d_i(t)$. If m_2 appears in $d_i(t)$, that implies that the edge (m_1, m_2) appears in CG_j at the time p_j executes *UpdatePromote* (since p_j previously executed *UnionCG* that includes at least m and the set of edges $\{(m', m) | m' \in C(m)\}$ in CG_j). The construction of *UpdatePromote* ensures us that m_1 appears before m_2 in $d_i(t)$, that proves that $\mathcal{ETOB}$ satisfies the TOB-Causal-Order property.

In conclusion, $\mathcal{ETOB}$ is an implementation of ETOB assuming that processes have access to the Ω failure detector in any environment. □

6 Related Work

Modern data service providers such as Amazon's Dynamo [7], Yahoo's PNUTs [6] or Google Bigtable distributed storage [4] are intended to offer highly available services. They consequently replicate those services over several server processes. In order to tolerate process failures as well as partitions, they consider eventual consistency [24, 28, 27].

The term *eventual* consensus was introduced in [18]. It refers to one instance of consensus which stabilizes at the end; not multiple instances as we consider in this paper. In [9], a self-stabilizing form of consensus was proposed: assuming a self-stabilizing implementation of $\diamond S$ (also described in the paper) and executing a sequence of consensus instances, validity and agreement are eventually ensured. Their consensus abstraction is close to ours but the authors focused on the shared-memory model and did not address the question of the weakest failure detector.

In [10], the intuition behind eventual consistency was captured through the concept of eventual serializability. Two kinds of operations were defined: (1) a "stable" operation of which response needs to be totally ordered after all operations preceding it and (2) "weak" operations of which responses might not reflect all their preceding operations. Our ETOB abstraction captures consistency with respect to the "weak" operations. (Our lower bound on the necessity of Ω naturally extends to the stronger definitions.)

Our perspective on eventual consistency is closely related to the notion of *eventual linearizability* discussed recently in [26] and [15]. It is shown in [26] that the weakest failure detector to boost eventually linearizable objects to linearizable ones is $\Diamond P$. We are focusing primarily on the weakest failure detector to *implement* eventual consistency, so their result is orthogonal to ours.

In [15], eventual linearizability is compared against linearizability in the context of implementing specific objects in a shared-memory context. It turns out that an eventually linearizable implementation of a *fetch-and-increment* object is as hard to achieve as a linearizable one. Our ETOB construction can be seen as an *eventually linearizable universal construction*: given any sequential object type, ETOB provides an eventually linearizable concurrent implementation of it. Brought to the message-passing environment with a correct majority, our results complement [15]: we show that in this setting, an eventually consistent replicated service (eventually linearizable object with a sequential specification) requires exactly the same information about failures as a consistent (linearizable) one.

7 Concluding Remarks

This paper defined the abstraction of eventual total order broadcast and proved its equivalence to eventual consensus: two fundamental building blocks to implement a general replicated state machine that ensures eventual consistency. We proved that the weakest failure detector to implement these abstractions is Ω, in any message-passing environment. We could hence determine the gap between building a general replicated state machine that ensures consistency in a message-passing system and one that ensures only eventual consistency. In terms of information about failures, this gap is precisely captured by failure detector Σ [8]. In terms of time complexity, the gap is exactly one message delay: an operation on the strongly consistent replicated must, in the worst case, incur three communication steps [22], while one build using our eventually total order broadcast protocol completes an operation in the optimal number of two communication steps.

Our ETOB abstraction captures a form of eventual consistency implemented in multiple replicated services [7, 6, 4]. In addition to eventual consistency guarantees, such systems sometimes produce indications when a prefix of operations on the replicated service is *committed*, i.e., is not

subject to further changes. A prefix of operations can be committed, *e.g.*, in sufficiently long periods of synchrony, when a majority of correct processes elect the same leader and all incoming and outgoing messages of the leader to the correct majority are delivered within some fixed bound. We believe that such indications could easily be implemented, during the stable periods, on top of ETOB. Naturally, our results imply that Ω is necessary for such systems too.

Our EC abstraction assumes eventual agreement, but requires integrity and validity to be always ensured. Other definitions of eventual consensus could be considered. In particular, we have studied an eventual consensus abstraction assuming, instead of eventual agrement, *eventual integrity*, *i.e.*, a bounded number of decisions in a given consensus instance could be revoked a finite number of times. In Appendix A, we define this abstraction of eventual *irrevocable* consensus (EIC) more precisely and show that it is equivalent to our EC abstraction.

References

[1] E. A. Brewer. Towards robust distributed systems (abstract). In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '00, pages 7–, 2000.

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

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

[4] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, June 2008.

[5] B. Charron-Bost and G. Tel. Approximation d'une borne inférieure répartie. Technical Report LIX/RR/94/06, Laboratoire d'Informatique LIX, École Polytechnique, Sept. 1994.

[6] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1(2):1277–1288, Aug. 2008.

[7] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM.

[8] C. Delporte-Gallet, H. Fauconnier, and R. Guerraoui. Tight failure detection bounds on atomic object implementations. *J. ACM*, 57(4), 2010.

[9] S. Dolev, R. I. Kat, and E. M. Schiller. When consensus meets self-stabilization. *Journal of Computer and System Sciences*, 76(8):884 – 900, 2010.

[10] A. Fekete, D. Gupta, V. Luchangco, N. Lynch, and A. Shvartsman. Eventually-serializable data services. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '96, pages 300–309, New York, NY, USA, 1996. ACM.

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

[12] F. C. Freiling, R. Guerraoui, and P. Kuznetsov. The failure detector abstraction. *ACM Comput. Surv.*, 43(2):9:1–9:40, Feb. 2011.

[13] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002.

[14] R. Guerraoui, V. Hadzilacos, P. Kuznetsov, and S. Toueg. The weakest failure detectors to solve quittable consensus and nonblocking atomic commit. *SIAM J. Comput.*, 41(6):1343–1379, 2012.

[15] R. Guerraoui and E. Ruppert. A paradox of eventual linearizability in shared memory. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing*, PODC '14, pages 40–49, 2014.

[16] V. Hadzilacos and S. Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical Report TR 94-1425, Department of Computer Science, Cornell University, May 1994.

[17] P. Jayanti and S. Toueg. Every problem has a weakest failure detector. In *PODC*, pages 75–84, 2008.

[18] F. Kuhn, Y. Moses, and R. Oshman. Coordinated consensus in dynamic networks. In *Proceedings of the 30th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 1–10. ACM, 2011.

[19] A. Lakshman and P. Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, Apr. 2010.

[20] L. Lamport. Proving the correctness of multiprocessor programs. *Transactions on software engineering*, 3(2):125–143, Mar. 1977.

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

[22] L. Lamport. Lower bounds for asynchronous consensus. *Distributed Computing*, 19(2):104–125, 2006.

[23] A. Mostefaoui, M. Raynal, and F. Tronel. From binary consensus to multivalued consensus in asynchronous message-passing systems. *Inf. Process. Lett.*, 73(5-6):207–212, Mar. 2000.

[24] Y. Saito and M. Shapiro. Optimistic replication. *ACM Comput. Surv.*, 37(1):42–81, Mar. 2005.

[25] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, Dec. 1990.

[26] M. Serafini, D. Dobre, M. Majuntke, P. Bokor, and N. Suri. Eventually linearizable shared objects. In A. W. Richa and R. Guerraoui, editors, *Proceedings of the 29th Annual ACM Symposium on Principles of Distributed Computing*, pages 95–104. ACM, 2010.

[27] A. Singh, P. Fonseca, P. Kuznetsov, R. Rodrigues, and P. Maniatis. Zeno: Eventually consistent byzantine-fault tolerance. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'09, pages 169–184, Berkeley, CA, USA, 2009. USENIX Association.

[28] W. Vogels. Eventually consistent. *Commun. ACM*, 52(1):40–44, Jan. 2009.

A Discussion on Eventual Consensus

Our definition of Eventual Consensus EC relaxes the Agreement property which holds after a finite number of operations. We could instead relax the Integrity property where processes can change their decisions a finite number of times. We discuss here the resulting abstraction.

A.1 Eventual Irrevocable Consensus (EIC)

The *eventual irrevocable consensus* (EIC) abstraction exports, to every process p_i, operations $proposeEIC_0$, $proposeEIC_1$, ... that take binary arguments and return binary responses. If a process p_i responds more than once to $proposeEIC_\ell$ for some $\ell \in \mathbb{N}$, we consider that the response of p_i to $proposeEIC_\ell$ at time $t \in \mathbb{N}$ is its last response to $proposeEIC_\ell$ before t.

Assuming that every process receives $proposeEIC_\ell$ as soon as it returns a (first) response to $proposeEIC_{\ell-1}$ for all $\ell \in \mathbb{N}$, the abstraction guarantees, for every run, there exists $k \in \mathbb{N}$ such that the following properties are satisfied:

EIC-Termination Every correct process eventually returns a response to $proposeEIC_\ell$ for all $\ell \in \mathbb{N}$.

EIC-Integrity No process responds twice to $proposeEIC_\ell$ for all $\ell \geq k$.

EIC-Agreement No two processes return infinitely different values to $proposeEIC_\ell$ for any $\ell \in \mathbb{N}$.

EIC-Validity Every value returned to $proposeEIC_j$ was previously proposed to $proposeEIC_j$ for all $j \in \mathbb{N}$.

Theorem 3. *In every environment $\mathcal{E}$, EC and EIC are equivalent.*

A.2 Transformation from EC to EIC

Lemma 4. *In every environment $\mathcal{E}$, there exists a transformation from EC to EIC.*

Proof. To prove this result, it is sufficient to provide a protocol that implements EIC in an environment $\mathcal{E}$ knowing that there exists a protocol that implements EC in this environment. This transformation protocol $\mathcal{T}_{\mathsf{EC}\to\mathsf{EIC}}$ is stated in Algorithm 6. Now, we are going to prove that $\mathcal{T}_{\mathsf{EC}\to\mathsf{EIC}}$ implements EIC.

As any invocation of $proposeEIC_\ell$ by a correct process p_i leads to an invocation of $ProposeEC_\ell$ by the same process, the EC-Termination property ensures us that p_i receives eventually a response (a sequence *decison*) from the EC primitive. Before this response, we have $decision_i[\ell] = \perp$. By the EC-Validity property, we know that $decision[\ell]$ is a value proposed by one process (hence not equal to $\perp$). Then, the construction of the protocol ensures us that $DecideEIC(\ell, decision[\ell])$ is executed in a finite time, that proves that $\mathcal{T}_{\mathsf{EC}\to\mathsf{EIC}}$ satisfies the EIC-Termination property.

Let k be the index after which the EC primitive satisfies EC-Agreement property. Let τ be the smallest time where all correct processes receive the response of $proposeEC_k$.

After time τ, we know that the sequences *decision* returned to all process are identical. Then, the construction of the protocol ensures us that every sequence submitted to the EC primitive is prefixed by the last sequence returned by this primitive. Hence, the EC-Agreement property ensures us that, after time τ, *DecideEIC* is executed only for the last value of the decision sequence and only when this sequence grows, that proves that $\mathcal{T}_{\mathsf{EC}\to\mathsf{EIC}}$ satisfies the EIC-Integrity property.

Assume that two processes p_i and p_j return forever two different values for $proposeEIC_\ell$ for some ℓ. By the EIC-Integrity property proved above, we know that p_i and p_j take at most one decision for $proposeEIC_\ell$ after time τ. That implies that p_i and p_j return different values at their

last decision. Then, we can deduce that $decision_i[\ell] \neq decision_j[\ell]$ forever, that is contradictory with the definition of τ or with the EC-Agreement property. This contradiction shows us that $\mathcal{T}_{\mathsf{EC}\to\mathsf{EIC}}$ satisfies the EIC-Agreement property.

The fact that $\mathcal{T}_{\mathsf{EC}\to\mathsf{EIC}}$ satisfies the EIC-Validity directly follows from the EC-Validity.

In conclusion, $\mathcal{T}_{\mathsf{EC}\to\mathsf{EIC}}$ satisfies the EIC specification in an environment $\mathcal{E}$ provided that there exists a protocol that implements EC in this environment. □

Algorithm 6 $\mathcal{T}_{\mathsf{EC}\to\mathsf{EIC}}$: transformation from EC to EIC for process p_i

Internal variables:

$decision_i$: sequence of values decided by p_i (initially ϵ)

Functions:

$DecideEIC(\ell, v)$ returns the value v as a response to $proposeEIC_\ell$

On invocation of $\boldsymbol{proposeEIC_\ell}(v)$

$proposeEC_\ell(decision_i.v)$

On reception of $decision$ **as response of** $\boldsymbol{proposeEC_\ell}$

For k from 0 to ℓ do
  If $decision[k] \neq decision_i[k]$ then
    $DecideEIC(k, decision[k])$
$decision_i := decision$

A.3 Transformation from EIC to EC

Lemma 5. *In every environment $\mathcal{E}$, there exists a transformation from* EIC *to* EC.

Proof. To prove this result, it is sufficient to provide a protocol that implements EC in an environment $\mathcal{E}$ knowing that there exists a protocol that implements EIC in this environment. This transformation protocol $\mathcal{T}_{\mathsf{EIC}\to\mathsf{EC}}$ is stated in Algorithm 7. Now, we are going to prove that $\mathcal{T}_{\mathsf{EIC}\to\mathsf{EC}}$ implements EC.

As any invocation of $proposeEC_\ell$ by a correct process p_i leads to an invocation of $proposeEIC_\ell$ by the same process, the EIC-Termination property ensures us that p_i receives eventually at least one response from the EIC primitive. The use of the counter $count_i$ allows us to deduce that only the first response from the EIC primitive leads to a decision for $proposeEC_\ell$ by p_i, that proves that $\mathcal{T}_{\mathsf{EIC}\to\mathsf{EC}}$ satisfies the EC-Termination and the EC-Integrity properties.

The construction of the protocol and the EIC-Agreement and the EIC-Validity properties trivially imply that $\mathcal{T}_{\mathsf{EIC}\to\mathsf{EC}}$ satisfies the EC-Agreement and the EC-Validity properties.

In conclusion, $\mathcal{T}_{\mathsf{EIC}\to\mathsf{EC}}$ satisfies the EC specification in an environment $\mathcal{E}$ provided that there exists a protocol that implements EIC in this environment. □

B Background on the CHT proof

Let $\mathcal{E}$ be any environment, $\mathcal{D}$ be any failure detector that can be used to solve consensus in $\mathcal{E}$, and $\mathcal{A}$ be any algorithm that solves consensus in $\mathcal{E}$ using $\mathcal{D}$. We determine a reduction algorithm $T_{\mathcal{D}\to\Omega}$ that, using failure detector $\mathcal{D}$ and algorithm $\mathcal{A}$, implements Ω in $\mathcal{E}$. Recall that implementing Ω means outputting, at every process, the id of a process so that eventually, the id of the same correct process is output permanently at all correct processes.

Algorithm 7 $\mathcal{T}_{\mathsf{EIC}\to\mathsf{EC}}$: transformation from EIC to EC for process p_i

Internal variables:

$count_i$: integer (initially 0) that stores the number of the last instance of consensus invoked by p_i

Functions:

$DecideEC(\ell, v)$ returns the value v as a response to $proposeEC_\ell$

On invocation of *proposeEC*$_\ell(v)$

$count_i := \ell$

$proposeEIC_\ell(v)$

On reception of v as response of *proposeEIC*$_\ell$

If $count_i = \ell$ then

$\lfloor$ $DecideEC(\ell, v)$

B.1 Overview of the reduction algorithm

The basic idea underlying $T_{\mathcal{D}\to\Omega}$ is to have each process locally *simulate* the overall distributed system in which the processes execute several runs of $\mathcal{A}$ that *could have happened* in the current failure pattern and failure detector history. Every process then uses these runs to extract Ω.

In the local simulations, every process p feeds algorithm $\mathcal{A}$ with a set of proposed values, one for each process of the system. Then all automata composing $\mathcal{A}$ are triggered locally by p which emulates, for every simulated run of $\mathcal{A}$, the states of all processes as well as the emulated buffer of exchanged messages.

Crucial elements that are needed for the simulation are (1) the values from failure detectors that would be output by $\mathcal{D}$ as well as (2) the order according to which the processes are taking steps. For these elements, which we call the stimuli of algorithm $\mathcal{A}$, process p periodically queries its failure detector module and exchanges the failure detector information with the other processes.

The reduction algorithm $T_{\mathcal{D}\to\Omega}$ consists of two tasks that are run in parallel at every process: the *commmuncation task* and the *computation task*. In the communication task, every process maintains ever-growing stimuli of algorithm $\mathcal{A}$ by periodically querying its failure detector module and sending the output to all other processes. In the computation task, every process periodically feeds the stimuli to algorithm $\mathcal{A}$, simulates several runs of $\mathcal{A}$, and computes the current emulated output of Ω.

B.2 Building a DAG

The communication task of algorithm $T_{\mathcal{D}\to\Omega}$ is presented in Figure 1. Executing this task, p knows more and more of the processes' failure detector outputs and temporal relations between them. All this information is pieced together in a single data structure, a directed acyclic graph (DAG) G_p. Informally, every vertex $[q, d, k]$ of G_p is a failure detector value "seen" by q in its k-th query of its failure detector module. An edge $([q, d, k], [q', d', k'])$ can be interpreted as "q saw failure detector value d (in its k-th query) *before* q' saw failure detector value d' (in its k'-th query)".

DAG G_p has some special properties which follow from its construction. Let F be the current failure pattern in $\mathcal{E}$ and H be the current failure detector history in $\mathcal{D}(F)$. Then:

(1) The vertices of G_p are of the form $[q, d, k]$ where $q \in \Pi$, $d \in \mathcal{R}_\mathcal{D}$ and $k \in \mathbb{N}$. There is a mapping $\tau :$ vertices of $G_p \mapsto \mathbb{T}$, associating a time with every vertex of G_p, such that:

(a) For any vertex $v = [q, d, k]$, $q \notin F(\tau(v))$ and $d = H(q, \tau(v))$. That is, d is the value output by q's failure detector module at time $\tau(v)$.

$G_p \leftarrow$ empty graph
$k_p \leftarrow 0$
while true **do**
  receive message m
  $d_p \leftarrow$ query failure detector $\mathcal{D}$
  $k_p \leftarrow k_p + 1$
  if m is of the form (q, G_q, p) **then** $G_p \leftarrow G_p \cup G_q$
  add $[p, d_p, k_p]$ and edges from all vertices of G_p to $[p, d_p, k_p]$ to G_p
  send (p, G_p, q) to all $q \in \Pi$

Figure 1: Building a DAG: process p

(b) For any edge (v, v') in G_p, $\tau(v) < \tau(v')$. That is, any edge in G_p reflects the temporal order in which the failure detector values are output.

(2) If $v' = [q, d, k]$ and $v'' = [q, d', k']$ are vertices of G_p, and $k < k'$, then (v, v') is an edge of G_p.

(3) G_p is transitively closed: if (v, v') and (v', v'') are edges of G_p, then (v, v'') is also an edge of G_p.

(4) For all correct processes p and q and all times t, there is a time $t' \geq t$, a $d \in \mathcal{R}_\mathcal{D}$ and a $k \in \mathbb{N}$ such that for every vertex v of $G_p(t)$, $(v, [q, d, k])$ is an edge of $G_p(t')$.[3]

Note that properties (1)–(4) imply that, for every correct process p, $t \in \mathbb{T}$ and $k \in \mathbb{N}$, there is a time t' such that $G_p(t')$ contains a path $g = [q_1, d_1, k_1] \rightarrow [q_2, d_2, k_2] \rightarrow \ldots$, such that (a) every correct process appears at least k times in g, and (b) for any path g' in $G_p(t)$, $g' \cdot g$ is also a path in $G_p(t')$.

B.3 Simulation trees

Now DAG G_p can be used to simulate runs of $\mathcal{A}$. Any path $g = [q_1, d_1, k_1], [q_2, d_2, k_2], \ldots, [q_s, d_s, k_s]$ through G_p gives the order in which processes q_1, $q_2, \ldots$, q_s "see", respectively, failure detector values d_1, $d_1, d_2, \ldots$, d_s. That is, g contains an activation schedule and failure detector outputs for the processes to execute steps of $\mathcal{A}$'s instances. Let I be any initial configuration of $\mathcal{A}$. Consider a schedule S that is applicable to I and *compatible with* g, i.e., $|S| = s$ and $\forall k \in \{1, 2, \ldots, s\}$, $S[k] = (q_k, m_k, d_k)$, where m_k is a message addressed to q_k (or the null message λ).

All schedules that are applicable to I and compatible with paths in G_p can be represented as a tree Υ_p^I, called the *simulation tree induced by G_p and I*. The set of vertices of Υ_p^I is the set of all schedules S that are applicable to I and compatible with paths in G_p. The root of Υ_p^I is the empty schedule $S_\perp$. There is an edge from S to S' if and only if $S' = S \cdot e$ for a step e; the edge is labeled e. Thus, every vertex S of Υ_p^I is associated with a sequence of steps $e_1\, e_2\, \ldots\, e_s$ consisting of labels of the edges on the path from $S_\perp$ to S. In addition, every descendant of S in Υ_p^I corresponds to an extension of $e_1\, e_2\, \ldots\, e_s$.

The construction of Υ_p^I implies that, for any vertex S of Υ_p^I, there exists a partial run $\langle F, H, I, S, T\rangle$ of $\mathcal{A}$ where F is the current failure pattern and $H \in \mathcal{D}(F)$ is the current failure detector history. Thus, if in S, correct processes appear sufficiently often and receive sufficiently many messages sent to them, then every correct (in F) process decides in $S(I)$.

[3] For any variable x and time t, $x(t)$ denotes the value of x at time t.

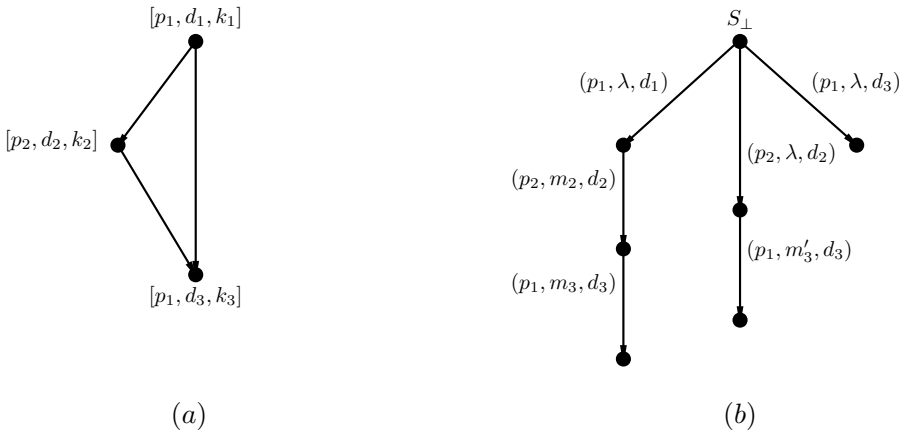

Figure 2: A DAG and a tree

In the example depicted in Figure 2, a DAG (a) induces a simulation tree a portion of which is shown in (b). There are three non-trivial paths in the DAG: $[p_1, d_1, k_1] \rightarrow [p_2, d_2, k_2] \rightarrow [p_1, d_3, k_3]$, $[p_2, d_2, k_2] \rightarrow [p_1, d_3, k_3]$, $[p_2, d_2, k_2] \rightarrow [p_1, d_3, k_3]$ and $[p_1, d_1, k_1] \rightarrow [p_1, d_3, k_3]$. Every path through the DAG and an initial configuration I induce at least one schedule in the simulation tree. Hence, the simulation tree has at least three leaves: (p_1, λ, d_1) (p_2, m_2, d_2) (p_1, m_3, d_3), (p_2, λ, d_2) (p_1, m'_3, d_3), and (p_1, λ, d_3). Recall that λ is the empty message: since the message buffer is empty in I, no non-empty message can be received in the first step of any schedule.

B.4 Tags and valences

Let I^i, $i \in \{0, 1, \ldots, n\}$ denote the initial configuration of $\mathcal{A}$ in which processes $p_1, \ldots, p_i$ propose 1 and the rest (processes $p_{i+1}, \ldots, p_n$) propose 0. In the computation task of the reduction algorithm, every process p maintains an ever-growing *simulation forest* $\Upsilon_p = \{\Upsilon_p^0, \Upsilon_p^1, \ldots, \Upsilon_p^n\}$ where Υ_p^i $(0 \leq i \leq n)$ denotes the simulation trees induced by G_p and initial configurations I^i.

For every vertex of the simulation forest, p assigns a set of *tags*. Vertex S of tree Υ_p^i is assigned a tag v if and only if S has a descendant S' in Υ_p^i such that p decides v in $S'(I^i)$. We call the set tags the *valence* of the vertex. By definition, if S has a descendant with a tag v, then S has tag v. Validity of consensus ensures that the set of tags is a subset of $\{0, 1\}$.

Of course, at a given time, some vertices of the simulation forest Υ_p might not have any tags because the simulation stimuli are not sufficiently long yet. But this is just a matter of time: if p is correct, then every vertex of p's simulation forest will eventually have an extension in which correct processes appear sufficiently often for p to take a decision.

A vertex S of Υ_p^i is 0-*valent* if it has exactly one tag $\{0\}$ (only 0 can be decided in S's extensions in Υ_p^i). A 1-valent vertex is analogously defined. If a vertex S has both tags 0 and 1 (both 0 and 1 can be decided in S's extensions), then we say that S is *bivalent*.[4]

It immediately follows from Validity of consensus that the root of Υ_p^0 can at most be 0-valent, and the root of Υ_p^n can at most be 1-valent (the roots of Υ_p^0 and Υ_p^n cannot be bivalent).

[4] The notion of valence was first defined in [11] as the set of values than are decided in *all* extensions of a given execution. Here we define the valence as only a subset of these values, defined by the simulation tree.

B.5 Stabilization

Note that the simulation trees can only grow with time. As a result, once a vertex of the simulation forest Υ_p gets a tag v, it cannot lose it later. Thus, eventually every vertex of Υ_p stabilizes being 0-valent, 1-valent, or bivalent. Since correct processes keep continuously exchanging the failure detector samples and updating their simulation forests, every simulation tree computed by a correct process at any given time will eventually be a subtree of the simulation forest of every correct process.

Formally, let p be any correct process, t be any time, i be any index in $\{0, 1, \ldots, n\}$, and S be any vertex of $\Upsilon_p^i(t)$. Then:

(i) There exists a non-empty $V \subseteq \{0, 1\}$ such that there is a time after which the valence of S is V. (We say that the valence of S *stabilizes* on V at p.)

(ii) If the valence of S stabilizes on V at p, then for every correct process q, there is a time after which S is a vertex of Υ_q^i and the valence of S stabilizes on V at q.

Hence, the correct processes eventually agree on the same tagged simulation subtrees. In discussing the stabilized tagged simulation forest, it is thus convenient to consider the *limit* infinite DAG G and the *limit* infinite simulation forest $\Upsilon = \{\Upsilon^0, \Upsilon^1, \ldots, \Upsilon^n\}$ such that for all $i \in \{0, 1, \ldots, n\}$ and all correct processes p, $\cup_{t \in \mathbb{T}} G_p(t) = G$ and $\cup_{t \in \mathbb{T}} \Upsilon_p^i(t) = \Upsilon^i$.

B.6 Critical index

Let p be any correct process. We say that index $i \in \{1, 2, \ldots, n\}$ is *critical* if *either* the root of Υ^i is bivalent *or* the root of Υ^{i-1} is 0-valent and the root of Υ^i is 1-valent. In the first case, we say that i is *bivalent critical*. In the second case, we say that i is *univalent critical*.

Lemma 6. *There is at least one critical index in* $\{1, 2, \ldots, n\}$.

Proof. Indeed, by the Validity property of consensus, the root of Υ^0 is 0-valent, and the root of Υ^1 is 1-valent. Thus, there must be an index $i \in \{1, 2, \ldots, n\}$ such that the root of Υ^{i-1} is 0-valent, and Υ^i is either 1-valent or bivalent. □

Since tagged simulation forests computed at the correct processes tend to the same infinite tagged simulation forest, eventually, all correct processes compute the same *smallest* critical index i of the same type (univalent or bivalent). Now we have two cases to consider for the smallest critical index: (1) i is univalent critical, or (2) i is bivalent critical.

(1) Handling univalent critical index

Lemma 7. *If i is univalent critical, then p_i is correct.*

Proof. By contradiction, assume that p_i is faulty. Then G contains an infinite path g in which p_i does not participate and every correct process participates infinitely often. Then Υ^i contains a vertex S such that p_i does not take steps in S and some correct process p decides in $S(I^i)$. Since i is 1-valent, p decides 1 in $S(I^i)$. But p_i is the only process that has different states in I^{i-1} and I^i, and p_i does not take part in S. Thus, S is also a vertex of Υ^{i-1} and p decides 1 in $S(I^{i-1})$. But the root of Υ^{i-1} is 0-valent — a contradiction. □

(2) Handling bivalent critical index

Assume now that the root of Υ^i is *bivalent*. Below we show that Υ^i then contains a *decision gadget*, i.e., a finite subtree which is either a *fork* or a *hook* (Figure 3).

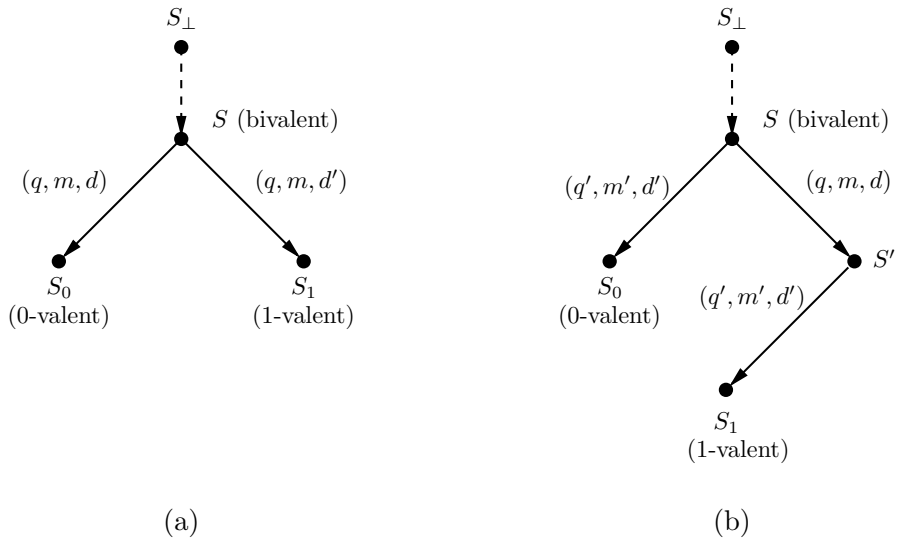

Figure 3: A fork and a hook

A fork (case (a) in Figure 3) consists of a bivalent vertex S from which two *different* steps by the *same* process q, consuming the same message m, are possible which lead, on the one hand, to a 0-valent vertex S_0 and, on the other hand, to a 1-valent vertex S_1.

A hook (case (b) in Figure 3) consists of a bivalent vertex S, a vertex S' which is reached by executing a step of some process q, and two vertices S_0 and S_1 reached by applying *the same* step of process q' to, respectively, S and S'. Additionally, S_0 must be 0-valent and S_1 must be 1-valent (or vice versa; the order does not matter here).

In both cases, we say that q is the *deciding process*, and S is the *pivot* of the decision gadget.

Lemma 8. *The deciding process of a decision gadget is correct.*

Proof. Consider any decision gadget γ defined by a pivot S, vertices S_0 and S_1 of opposite valence and a deciding process q. By contradiction, assume that q is faulty. Let g, g_0 and g_1 be the simulation stimuli of, respectively, S, S_0 and S_1. Then G contains an infinite path $\tilde{g}$ such that (a) $g \cdot \tilde{g}$, $g_0 \cdot \tilde{g}$, $g_1 \cdot \tilde{g}$ are paths in G, and (b) q does not appear and the correct processes appear infinitely often in g.

Let γ be a fork (case (a) in Figure 3). Then there is a finite schedule $\tilde{S}$ compatible with a prefix of $\tilde{g}$ and applicable to $S(I^i)$ such that some correct process p decides in $S \cdot \tilde{S}(I^i)$; without loss of generality, assume that p decides 0. Since q is the only process that can distinguish $S(I^i)$ and $S_1(I^i)$, and q does not appear in $\tilde{S}$, $\tilde{S}$ is also applicable to $S_1(I^i)$. Since $g_1 \cdot \tilde{g}$ is a path of G and $\tilde{S}$ is compatible with a prefix of $\tilde{g}$, it follows that $S_1 \cdot \tilde{S}$ is a vertex of Υ^i. Hence, p also decides 0 in $S_1 \cdot \tilde{S}(I^i)$. But S_1 is 1-valent — a contradiction.

Let γ be a hook (case (b) in Figure 3). Then there is a finite schedule $\tilde{S}$ compatible with a prefix of g and applicable to $S_0(I^i)$ such that some correct process p decides in $S_0 \cdot \tilde{S}(I^i)$. Without loss of generality, assume that S_0 is 0-valent, and hence p decides 0 in $S_0 \cdot \tilde{S}(I^i)$. Since q is the only process that can distinguish $S_0(I^i)$ and $S_1(I^i)$, and q does not appear in $\tilde{S}$, $\tilde{S}$ is also applicable to

$S_1(I^i)$. Since $g_1 \cdot \tilde{g}$ is a path of G and $\tilde{S}$ is compatible with a prefix of $\tilde{g}$, it follows that $S_1 \cdot \tilde{S}$ is a vertex of Υ^i. Hence, p also decides 0 in $S_1 \cdot \tilde{S}(I^i)$ But S_1 is 1-valent — a contradiction. □

Now we need to show that any bivalent simulation tree Υ^i contains at least one decision gadget γ.

Lemma 9. *If i is bivalent critical, then Υ^i contains a decision gadget.*

Proof. Let i be a bivalent critical index. In Figure 4, we present a procedure which goes through Υ^i. The algorithm starts from the bivalent root of Υ^i and terminates when a hook or a fork has been found.

$S \leftarrow S_\perp$
while *true* **do**
  $p \leftarrow \langle$choose the next correct process in a round robin fashion$\rangle$
  $m \leftarrow \langle$choose the oldest undelivered message addressed to p in $S(I^i)\rangle$
  if $\langle S$ has a descendant S' in Υ^i (possibly $S = S'$) such that, for some d,
      $S' \cdot (p, m, d)$ is a bivalent vertex of $\Upsilon^i\rangle$
  then $S \leftarrow S' \cdot (p, m, d)$
  else exit

Figure 4: Locating a decision gadget

We show that the algorithm indeed terminates. Suppose not. Then the algorithm locates an infinite *fair path* through the simulation tree, i.e., a path in which all correct processes get scheduled infinitely often and every message sent to a correct process is eventually consumed. Additionally, this fair path goes through bivalent states only. But no correct process can decide in a bivalent state $S(I^i)$ (otherwise we would violate the Agreement property of consensus). As a result, we constructed a run of $\mathcal{A}$ in which no correct process ever decides — a contradiction.

Thus, the algorithm in Figure 4 terminates. That is, there exist a bivalent vertex S, a correct process p, and a message m addressed to p in $S(I^i)$ such that

(*) For all descendants S' of S (including $S' = S$) and all d, $S' \cdot (p, m, d)$ is *not* a bivalent vertex of Υ^i.

In other words, any step of p consuming message m brings any descendant of S (including S itself) to either a 1-valent or a 0-valent state. Without loss of generality, assume that, for some d, $S \cdot (p, m, d)$ is a 0-valent vertex of Υ^i. Since S is bivalent, it must have a 1-valent descendant S''.

If S'' includes a step in which p consumes m, then we define S' as the vertex of Υ^i such that, for some d', $S' \cdot (p, m, d')$ is a prefix of S''. If S'' includes no step in which p consumes m, then we define $S' = S''$. Since p is correct, for some d', $S' \cdot (p, m, d')$ is a vertex of Υ^i. In both cases, we obtain S' such that for some d', $S' \cdot (p, m, d')$ is a 1-valent vertex of Υ^i.

Let the path from S to S' go through the vertices $\sigma_0 = S, \sigma_1, \ldots, \sigma_{m-1}, \sigma_m = S'$. By transitivity of G, for all $k \in \{0, 1, \ldots, m\}$, $\sigma_k \cdot (p, m, d')$ is a vertex of Υ^i. By (*), $\sigma_k \cdot (p, m, d')$ is either 0-valent or 1-valent vertex of Υ^i.

Let $k \in \{0, \ldots, m\}$ be the lowest index such that (p, m, d') brings σ_k to a 1-valent state. We know that such an index exists, since $\sigma_m \cdot (p, m, d')$ is 1-valent and all such resulting states are either 0-valent or 1-valent.

Now we have the following two cases to consider: (1) $k = 0$, and (2) $k > 0$.

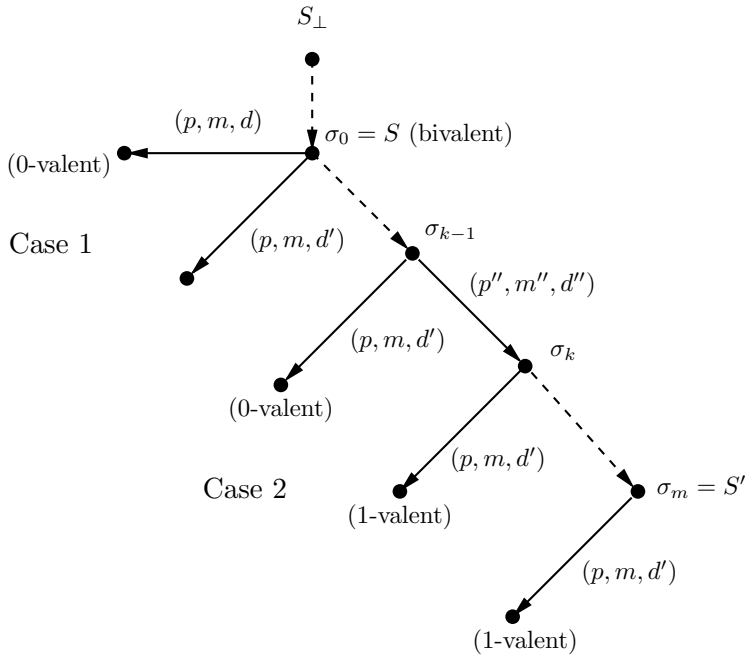

Figure 5: Locating a fork (Case 1) or a hook (Case 2)

Assume that $k = 0$, i.e., (p, m, d') applied to S brings it to a 1-valent state. But we know that there is a step (p, m, d) that brings S to a 0-valent state (Case 1 in Figure 5). That is, a fork is located!

If $k > 0$, we have the following situation. Step (p, m, d') brings σ_{k-1} to a 0-valent state, and $\sigma_k = \sigma_{k-1} \cdot (p', m', d'')$ to a 1-valent state (Case 2 in Figure 5). But that is a hook!

As a result, any bivalent infinite simulation tree has at least one decision gadget. □

B.7 The reduction algorithm

Now we are ready to complete the description of $T_{\mathcal{D}\to\Omega}$. In the computation task (Figure 6), every process p periodically extracts the current *leader* from its simulation forest, so that eventually the correct processes agree on the same correct leader. The current leader is stored in variable $\Omega\text{-}output_p$.

Initially, p elects itself as a leader. Periodically, p updates its simulation forest Υ_p by incorporating more simulation stimuli from G_p. If the forest has a univalent critical index i, then p outputs p_i as the current leader estimate. If the forest has a bivalent critical index i and Υ_p^i contains a decision gadget, then p outputs the deciding process of *the smallest* decision gadget in Υ_p^i (the "smallest" can be well-defined, since the vertices of the simulation tree are countable).

Eventually, the correct processes locate the same *stable* critical index i. Now we have two cases to consider:

(i) i is univalent critical. By Lemma 7, p_i is correct.

(ii) i is bivalent critical. By Lemma 9, the limit simulation tree Υ^i contains a decision gadget. Eventually, the correct processes locate the same decision gadget γ in Υ_i and compute the deciding process q of γ. By Lemma 8, q is correct.

Initially:
  for $i = 0, 1, \ldots, n$: $\Upsilon_p^i \leftarrow$ empty graph
  Ω-$output_p \leftarrow p$

while true **do**

  { Build and tag the simulation forest induced by G_p }
  for $i = 0, 1, \ldots, n$ **do**
    $\Upsilon_p^i \leftarrow$ simulation tree induced by G_p and I^i
    for every vertex S of Υ_p^i:
      if S has a descendant S' such that p decides v in $S'(I^i)$ **then**
        add tag v to S

  { Select a process from the tagged simulation forest }
  if there is a critical index **then**
    $i \leftarrow$ the smallest critical index
    if i is univalent critical **then** Ω-$output_p \leftarrow p_i$
    if Υ_p^i has a decision gadget **then**
      Ω-$output_p \leftarrow$ the deciding process of the smallest decision gadget in Υ_p^i

Figure 6: Extracting a correct leader: code for each process p

Thus, eventually, the correct processes elect the same correct leader — Ω is emulated!