Distributed Computing manuscript No.
(will be inserted by the editor)

The Weakest Failure Detector for Eventual Consistency

Swan Dubois · Rachid Guerraoui · Petr Kuznetsov · Franck Petit · Pierre Sens

Received: date / Accepted: date

Abstract In its classical form, a *consistent* replicated service requires all replicas to witness the same evolution of the service state. If we consider an asynchronous message-passing environment in which processes might fail by crashing, and assume that a majority of processes are correct, then 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.

Keywords Eventual consistency · Failure detectors · Consensus · Total-order broadcast

A previous version of this work appears in the proceedings of the 2015 ACM Symposium on Principles of Distributed Computing [13].

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

S. Dubois, F. Petit, and P. Sens
Sorbonne Universités, UPMC Univ Paris 06, CNRS, Inria, LIP6 UMR 7606
Paris, France
E-mail: firstname.lastname@lip6.fr

R. Guerraoui
École Polytechnique Fédérale de Lausanne
Lausanne, Suisse
E-mail: rachid.guerraoui@epfl.ch

P. Kuznetsov
Télécom ParisTech
Paris, France
E-mail: petr.kuznetsov@telecom-paristech.fr

1 Introduction

State machine replication [24,29] is the most studied technique to build a highly-available and consistent distributed service. 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 crash failures of replicas. This is typically captured using the abstraction of *total order broadcast* [9,20], where messages represent invocations of the service operations from clients to replicas. Assuming that the state machine is deterministic, delivering the invocations in the same total order ensures that the replicas behave like a single state machine. Total order broadcast is, in turn, typically implemented by having the processes agree on which message (or a batch of messages) to execute next, using the *consensus* abstraction [25,5]. The two abstractions, consensus and total order broadcast, were shown to be equivalent in [5].

Replicas behaving like a single one is a property generally called *consistency*. The 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*, 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 [15,4,1], and the number of communication steps needed to deliver an invocation [25,26].

An appealing approach to circumvent these costs is to trade consistency with what is sometimes called *eventual consistency* [28,33]: 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.*, Cassandra [23] and Dynamo [10]. 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 [5,4], it addresses the question of the minimal information about failures needed to implement an eventually consistent replicated state machine.

It has been shown in [4] that, in a message-passing environment with a majority of correct processes, the weakest failure detector to implement consensus (and, thus, total order broadcast [9,7]) is the *eventual leader* failure detector, denoted Ω. In short, Ω outputs, at every process and at all times, 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, 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 [4]. 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 repeated 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, 17,11]. Therefore, in a system with a correct majority of processes, weakening consistency does not allow for a weaker failure detector: (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 [11] 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 [1,17] 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 repeated 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 [10,8,6].
- We show how to extend the celebrated CHT proof [4], initially establishing that Ω is necessary for solving consensus, to the context of eventual repeated consensus. Through this extension, we indirectly highlight a hidden power of the technique proposed in [4] 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.

[1] Note that three communication steps are, in the worst case, necessary when strong consistency is required [26].

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 repeated 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 repeated 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.

2 Preliminaries

We adopt the classical model of distributed systems provided with the failure detector abstraction proposed in [5,4]. In particular we employ the simplified version of the model proposed in [18,21].

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.

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.

Inputs and outputs of processes are modeled via *input histories* H_I and *output histories* H_O that specify the sequences of inputs each process receives from its application and the sequences of outputs each process returns to the application over time: At each time t and for every process p_i, $H_I(p_i, t)$ (resp., $H_O(p_i, t)$) denotes the input (resp., the output) received by p_i (resp., produced by p_i) at time t, which can be a distinct value $\perp$ in case no input (resp., output) occurs. Typically, inputs and outputs represent invocations and responses of *operations* exported by the implemented abstraction.

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 (F, H_I, H_O) where F is a failure pattern, 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 for every admissible run R of $\mathcal{A}$ in $\mathcal{E}$, $R = (F, H, H_I, H_O, S, T)$, we have $(F, H_I, H_O) \in P$. 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 $\mathcal{E}$* [20] 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 $T_{P\rightarrow P'}$ 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 $\mathcal{E}$ [4,21]. 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 repeated consensus. We show that the two abstractions are equivalent: each of them can be used to implement the other.

3.1 Eventual Total Order Broadcast (ETOB)

The *total order broadcast* (TOB) abstraction [9,20] 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 $t' > t$.

To match out formalism of input and output histories, we interprete a call of *broadcastTOB(m)* that took place at time t as an event in the input history H_I, and each change in the value of d_i that takes place at time t, i.e., $d_i(t) \neq d_i(t-1)$ as an event in the output history H_O.

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 on 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 broadcasts m_1 and then broadcasts m_2, (2) a process p_i receives m_1 and then broadcasts m_2, or (3) there exists m_3 such that $m_1 \rightarrow_R m_3$ and $m_3 \rightarrow_R m_2$.

3.2 Eventual Repeated Consensus (ERC)

The *consensus* abstraction (C) [15] 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 repeated consensus* (ERC) abstraction exports, to every process p_i, operations $proposeERC_1$, $proposeERC_2$, ... that take binary arguments and return binary responses. Assuming that, for all $\ell = 1, 2, \ldots$, every process, as soon as it returns a response to $proposeERC_\ell$, invokes $proposeERC_{\ell+1}$ or crashes, the abstraction guarantees that, for every admissible run, there exists $k \in \mathbb{N}$, such that the following properties are satisfied:

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

ERC-Integrity No process responds twice to $proposeERC_\ell$, for all $\ell \in \mathbb{N}$.

ERC-Validity Every value returned to $proposeERC_\ell$ was previously proposed to $proposeERC_\ell$, for all $\ell \in \mathbb{N}$.

ERC-Agreement No two processes return different values to $proposeERC_\ell$, for all $\ell \geq k$.

It is straightforward to transform the binary version of ERC into a multivalued one with unbounded set of inputs [27]. In the following, except for the necessity proof in Section 4, when we discuss an ERC algorithm, we mean a multivalued version of it.

3.3 Equivalence between ERC and ETOB

It is well known that, in their classical forms, the consensus and the total order broadcast abstractions are equivalent [5]. In this section, we show that a similar result holds for our eventual versions of these abstractions.

The intuition behind the transformation from ERC 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 ERC. This sequence is built by concatenating the last output of ERC(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 ERC is stored in d_i, i.e., at any time, each process delivers the last sequence of messages returned by ERC.

The correctness of this transformation follows from the fact that ERC eventually 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 ERC-Validity of ERC, 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 ERC is as follows. At each invocation of the ERC 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 repeated 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 ERC-instance it invokes. Thus, the transformation satisfies the ERC specification.

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

Proof We prove this result by providing two algorithms, one implementing ERC from ETOB and the other implementing ETOB from ERC.

From ERC 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 ERC in this environment. This transformation protocol $\mathcal{T}_{\mathsf{ERC}\rightarrow\mathsf{ETOB}}$ is stated in Algorithm 1. Now, we are going to prove that $\mathcal{T}_{\mathsf{ERC}\rightarrow\mathsf{ETOB}}$ 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 ERC and the construction of the function *NewBatch*, there exists ℓ such that m is included in any sequence submitted to $proposeERC_\ell$. By the ERC-Validity and the ERC-Termination properties, we deduce that p_i stably delivers m in a finite time, which proves that $\mathcal{T}_{\mathsf{ERC}\rightarrow\mathsf{ETOB}}$ 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 $proposeERC_\ell$. By construction and by the ERC-Validity property, this sequence contains only messages that appear in the set *toDeliver* of a process p_j at the time p_j invokes $proposeERC_\ell$. But the *toDeliver* set contains only previously messages broadcast. Therefore, $\mathcal{T}_{\mathsf{ERC}\to\mathsf{ETOB}}$ satisfies the TOB-No-creation.

As the sequence output at any time by any process is the response to its last invocation of *proposeERC* and as the sequence submitted to any invocation of this primitive contains no duplicated message (by definition of the function *NewBatch*), we can deduce from the ERC-Validity property that $\mathcal{T}_{\mathsf{ERC}\to\mathsf{ETOB}}$ 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 *proposeERC* to p_i after this time. As the ERC-Agreement property is eventually satisfied, we can deduce that m always appears in the response of *proposeERC* for any correct process after some time. Thus, any correct process stably delivers m, and $\mathcal{T}_{\mathsf{ERC}\to\mathsf{ETOB}}$ satisfies the TOB-Agreement.

Let τ be the time after which the ERC primitive satisfies ERC-Agreement and ERC-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 $proposeERC_{\ell_1}$ (respectively $proposeERC_{\ell_2}$). By construction of the protocol and the ERC-Agreement and ERC-Validity properties, we know that, after time τ, the response of $proposeERC_\ell$ to correct processes is a prefix of the response of $proposeERC_{\ell+1}$. As we have $\ell_1 \leq \ell_2$, we can deduce that $\mathcal{T}_{\mathsf{ERC}\to\mathsf{ETOB}}$ 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 $proposeERC_\ell$. By the ERC-Agreement property, we know that the response of $proposeERC_\ell$ 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, $\mathcal{T}_{\mathsf{ERC}\to\mathsf{ETOB}}$ satisfies the ETOB-Total-order property.

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

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

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

Output variable:
  d_i: sequence of messages (initially empty) output at any time by p_i
Internal variables:
  $toDeliver_i$: set of messages (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
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 $toDeliver_i \setminus \{m | m \in d_i\}$
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 $proposeERC_{count_i}$
  $d_i := d$
  $count_i := count_i + 1$
  $proposeERC_{count_i}(d_i.NewBatch(d_i, toDeliver_i))$
On local timeout
  If $count_i = 0$ and $NewBatch(d_i, toDeliver_i)) \neq \emptyset$ then
    $count_i := 1$
    $proposeERC_1(NewBatch(d_i, toDeliver_i))$

The update of the variable $count_i$ to ℓ for any process p_i that invokes $proposeERC_\ell$ and the assumptions on operations *proposeERC* ensure us that p_i executes at most once the function $DecideEC(count_i, First(count_i))$. Hence, $\mathcal{T}_{\mathsf{ETOB}\to\mathsf{ERC}}$ satisfies the ERC-Integrity property.

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 $proposeERC_k$ 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 $proposeERC_\ell$ 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

Algorithm 2 $\mathcal{T}_{\mathsf{ETOB}\to\mathsf{ERC}}$: transformation from ETOB to ERC 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
  d_i: sequence of messages (initially empty) output 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 $proposeERC_\ell$
On invocation of $proposeERC_\ell(v)$
  $count_i := \ell$
  $broadcastETOB((\ell, v))$
On local time out
  If $First(count_i) \neq \perp$ then
  $\lfloor DecideEC(count_i, First(count_i))$

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 $proposeERC_\ell$ in d_j at the time of the return of $proposeERC_\ell$. After that, d_j cannot satisfy simultaneously the ETOB-Stability and the ETOB-Total-order properties. This contradiction shows that $\mathcal{T}_{\mathsf{ETOB}\to\mathsf{ERC}}$ satisfies the ERC-Agreement property.

Every value returned by $proposeERC_\ell$ at a process p_i comes from d_i, and d_i may only contain ETOB outputs, which, by the TOB-No-creation property, have been previously proposed. Thus, $\mathcal{T}_{\mathsf{ETOB}\to\mathsf{ERC}}$ satisfies the ERC-Validity property.

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

4 The Weakest Failure Detector for ERC

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

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

4.1 Ω is necessary for ERC

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 ERC 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.

A very brief CHT primer. First, we briefly recall the arguments used by Chandra et al. [4] in the original proof (known as the *CHT proof*) that Ω can be derived from any algorithm solving consensus. To get a more detailed survey of the proof please refer to [16, Chapter 3].

The basic observation there is that a run of any algorithm $\mathcal{A}$ 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 $[p_i, d, k]$ to vertex $[p_j, d', k']$ is thus interpreted as "p_i queried $\mathcal{D}$ for the kth time and obtained value d and *after that* p_j queried $\mathcal{D}$ for the k'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 by computing the union of the vertices and edges of the two graphs and making sure that the paths in the merged graph are transitively closed. 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. For each input history H_I, each path in DAG G_i represents a *stimulus* for the schedule in the simulated run: processes take steps and observe failure-detector values in the order they appear in the path. All these runs are organized in the form of a *simulation tree* Υ_i.

Recall that p_i periodically updates G_i by adding a vertex corresponding to a new query of $\mathcal{D}$ or merging with a DAG received from another process. Each time p_i updates G_i, it recomputes Υ_i. Therefore, the simulation trees Υ_i maintained by the correct processes converge to the same infinite simulation tree Υ. Every path in the simulation tree Υ_i represents a run of $\mathcal{A}$. [2]

In the example depicted in Figure 1, a DAG (a) induces a simulation tree a portion of which is shown in (b). There are three non-trivial paths in the DAG: $[p_1, d_1, k_1] \to [p_2, d_2, k_2] \to [p_1, d_3, k_3]$, $[p_2, d_2, k_2] \to [p_1, d_3, k_3]$, and $[p_1, d_1, k_1] \to [p_1, d_3, k_3]$.

[2] In [4], the simulated schedules form a *simulation forest*, where a distinct simulation tree corresponds to each initial configuration encoding consensus inputs. Recall that here we follow the more flexible model of Jayanti and Toueg [21]: 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 *propose* calls.

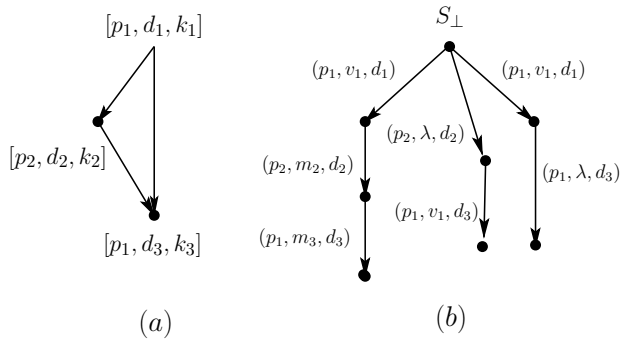

Fig. 1 A DAG and a tree

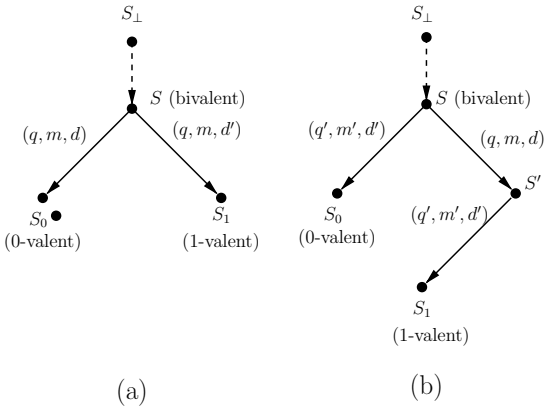

Fig. 2 A fork (a) and a hook (b)

Every path through the DAG and an input history induce at least one schedule in the simulation tree. Hence, assuming an input history initially providing p_1 with input v_1, the simulation tree has at least three leaves: (p_1, v_1, d_1) (p_2, m_2, d_2) (p_1, m_3, d_3), (p_2, λ, d_2) (p_1, v_1, d_3), and (p_1, v_1, d_1) $(p_1\lambda, d_3)$. Here $S_\perp$ denotes the empty schedule, and λ denotes the empty message: no non-empty message can be received in the first step of any schedule. Note that the simulation tree may contain other paths corresponding to other input histories, that is why Figure 1 (b) depicts only a portion of the possible tree.

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 determined as a set of decisions taken in all σ's *extensions* (descendants of σ in Υ_i): the valency tag of σ contains a value $v \in \{0, 1\}$ if σ has an extension in which some process decides v. A vertex is bivalent if its valency tag contains both 0 and 1. Note that no process can decide in a bivalent vertex.

It can be shown that every bivalent vertex in Υ has a *decision gadget*, i.e., finite subtree which is either a *fork* or a *hook* (Figure 2). Intuitively, in a decision gadget, a step of a fixed *deciding* process determines the univalent valency (and, thus, the decided values) in descendants S_0 and S_1. This process (e.g., q in Figure 2) must be correct. Otherwise, without its intervention, no correct process will be able to decide in any run extending S_0 or S_1.

Therefore, by locating the *same* bivalent vertex in the limit tree Υ, the correct process can eventually extract the identifier of the *same* correct process. The vertices in each simulation tree Υ_i are ordered by p_i in a specific deterministic way, which guarantees that each vertex in Υ is eventually assigned the same position in the order in all local trees Υ_i computed by correct processes. The identifier of the deciding process of the the "first" (according to this order) decision gadget in Υ_i is then returned as the current output of Ω. This way, the correct processes eventually agree on the "first" decision gadget and, thus, on the deciding process in this gadget. Note that the time when such agreement is reached is unknown to the processes. But this is good enough for extracting Ω.

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

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

Proof Let $\mathcal{A}$ be any algorithm that implements ERC using a failure detector $\mathcal{D}$ in an environment $\mathcal{E}$. As in [4], 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 sequences of inputs of ERC. The simulated runs are organized by p_i in the 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 $proposeERC_1(v)$, for some $v \in \{0, 1\}$, takes steps using the failure detector values stipulated by the path in G_i and, once $proposeERC_\ell(v)$, for some $\ell \geq 1$, is complete, eventually invokes $proposeERC_{\ell+1}(v')$, 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.

Note that in every (infinite) admissible simulated run, ERC-Termination, ERC-Integrity, ERC-Validity, and ERC-Agreement are satisfied.

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.

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 $proposeERC_{k-1}$ 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 $proposeERC_k$ 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 $proposeERC_k$, 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*

Recall that $\mathcal{A}$ ensures ERC-Termination in all admissible simulated runs that extend σ. Thus, if σ is k-enabled, then the k-tag of σ is non-empty: $proposeERC_k$ must return some values in all admissible extensions of σ. Moreover, ERC-Termination and ERC-Validity imply that a vertex in which no process has invoked $proposeERC_k$ yet has a descendant in which $proposeERC_k$ returns 0 and a descendant in which $proposeERC_k$ 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 the k-tag of σ' contains $\{0, 1\}$.

We show now that the "limit tree" Υ contains a k-bivalent vertex for some k. Consider the abstract procedure described in Algorithm 3 that intends to locate such a vertex in Υ, starting with the root of the tree.

Inductively, let σ be the currently considered k-enabled vertex such that its k-tag contains $\{0, 1\}$. Note that, initially, σ is the root of Υ, which is 1-enabled and either contains $\{0, 1\}$ or is 1-invalid. Moreover, each complete iteration of the loop in Algorithm 3, starting with some k-enabled vertex σ, computes a k-enabled descendant of σ whose k'-tag for some $k' > k$ contains $\{0, 1\}$.

Algorithm 3 Locating a bivalent vertex in Υ.

```
k := 1
σ := root of Υ
while true do
    if σ is k-bivalent then break
    σ1 := a descendant of σ in which
          ERC-Agreement does not hold for proposeERC_k
    σ2 := a descendant of σ1 in which every correct process
          completes proposeERC_k and receives
          all messages sent to it in σ
    choose k' > k and σ3, a descendant of σ2, such that
          k'-tag of σ3 contains {0,1}
    k := k'
    σ := σ3
```

Let the currently considered k-enabled vertex σ be *not* k-bivalent (if it is k-bivalent, we are done). Inductively, σ must be k-invalid, and hence it must have a descendant σ_1 in which ERC-Agreement does not hold for $proposeERC_k$. We then locate σ_2, a descendant of σ_1, in which every correct process completes $proposeERC_k$ and receives every message addressed to it in the message buffer of σ. By the very way Υ is constructed, every vertex in Υ has infinitely many descendants corresponding to every correct process, so such a descendant exists.

Now we use (*) to locate σ_3, a descendant of σ_2, such that (1) in σ_3, two processes return different values in $proposeERC_k$ in σ_3, (2) in σ_3, every correct process has completed $proposeERC_k$ 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 ERC-Agreement property of ERC—a contradiction.

Thus, for some k, there is a k-bivalent vertex in Υ. We now can apply the arguments of [4] to extract Ω. Indeed, we can simply let every process locate the "first" such k-bivalent 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.

Thus, the correct processes will eventually locate the same k-bivalent vertex and then, as in [4], stabilize extracting the same correct process identifier. This gives a reduction algorithm emulating Ω.

4.2 Ω is sufficient for ERC

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 [5]. In this section, we extend this result to the eventual repeated consensus abstraction for any environment.

The proposed implementation of ERC is very simple. Each process has access to an Ω failure detector module. Upon each invocation of the ERC 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 ERC 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 any environment $\mathcal{E}$,* ERC *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 ERC-Termination, ERC-Integrity, ERC-Agreement, and ERC-Validity properties.

Assume that a correct process never returns from an invocation of *proposeERC* 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 $proposeERC_\ell$. 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 $proposeERC_0$,..., $proposeERC_{\ell-1}$ 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 ERC-Termination property.

Algorithm 4 ERC 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 ⊥)
Functions:
    DecideEC(ℓ, v) returns the value v as a response to
    proposeERC_ℓ
Messages:
    promote(v, ℓ) with v ∈ {0, 1} and ℓ ∈ ℕ
On invocation of proposeERC_ℓ(v)
    count_i := ℓ
    Send promote(v, ℓ) to all
On reception of promote(v, ℓ) from p_j
    received_i[j, ℓ] := v
On local time out
    If received_i[Ω_i, count_i] ≠ ⊥ then
    ⌊ DecideEC(count_i, received_i[Ω_i, count_i])
```

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

Let τ_Ω be the time from which the local outputs of Ω are identical and constant for all correct processes in r. Let k be the smallest integer such that any process that invokes $proposeERC_k$ 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 $proposeERC_\ell$. 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 $received_i[p_l, \ell] = received_j[p_l, \ell]$, which ensures the ERC-Agreement property.

Assume that p_i is a process that responds to $proposeERC_\ell$, for some $\ell = 1, 2, \ldots$. 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 ERC-Validity property is satisfied.

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

5 An Eventual Total Order Broadcast Algorithm

We have shown in the previous section that Ω is the weakest failure detector for the ERC 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 [26]. 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 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.

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

Output variable:
  d_i: sequence of messages m (initially empty) output by p_i
Internal variables:
  $promote_i$: sequence of messages (initially empty) promoted by p_i when $\Omega_i = p_i$
  CG_i: directed graph on messages (initially empty) that contains causality dependencies known by p_i
Messages:
  $update(CG_i)$ with CG_i a directed graph on messages
  $promote(promote_i)$ with $promote_i$ a sequence of messages
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
  $\lfloor\ d_i := promote_j$
On local time out
  If $\Omega_i = p_i$ then
  $\lfloor$ Send $promote(promote_i)$ to all

Theorem 3 *In any environment $\mathcal{E}$, Algorithm $\mathcal{A}_{\mathcal{ETOB}}$ implements* ETOB *using Ω.*

Proof First, we prove that any run r of $\mathcal{A}_{\mathcal{ETOB}}$ 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 output by Ω, p_i adopts the promotion sequence of p_j in a finite time and this sequence contains m, which proves that $\mathcal{A}_{\mathcal{ETOB}}$ satisfies the TOB-Validity property.

Any sequence output 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 $\mathcal{A}_{\mathcal{ETOB}}$ satisfies the TOB-No-creation property.

Any sequence output by any process is built by a call to $UpdatePromote$ that ensures that any message appears only once. Then, we can deduce that $\mathcal{A}_{\mathcal{ETOB}}$ 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 imply that any correct process eventually stably delivers m, and $\mathcal{A}_{\mathcal{ETOB}}$ satisfies the TOB-Agreement property.

We now prove that, for any environment $\mathcal{E}$, for any run r of $\mathcal{A}_{\mathcal{ETOB}}$ 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 $\mathcal{A}_{\mathcal{ETOB}}$ 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 process 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, $\mathcal{A}_{\mathcal{ETOB}}$ 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 process 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 $d_j(t)$, which proves that $\mathcal{A}_{\mathcal{ETOB}}$ 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 $d_i(t)$, which proves that $\mathcal{A}_{\mathcal{ETOB}}$ satisfies the TOB-Causal-Order property.

In conclusion, $\mathcal{A}_{\mathcal{ETOB}}$ 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 are intended to offer highly available services by replicating the services over several server processes. In order to tolerate process failures as well as partitions, forms of eventual consistency are typically considered [28,33,31].

Assuming read and write operations, we can distinguish two main kinds of guarantees [2]: *basic eventual consistency* ensures that the effect of every write operation will eventually become visible to all replicas and *ordering guarantees* impose conditions on the order in which operations may be performed.

Data stores such as Amazon's Dynamo [10] or Cassandra [23] rely on basic eventual consistency alone to support a large number of read/write operations per second on a set of replicas. In their eventual consistency model, read and write operations may be performed on non-overlapping subsets of nodes. Once a write is acknowledged from a subset, the new value is asynchronously propagated to the remaining replicas. This *update propagation property* [3] ensures that all replicas are eventually updated. However, stale values may be observed if readers fetch data from replicas that have not yet received the ongoing updates.

Basic eventual consistency is often too weak for the users and many systems provide additional stronger guarantees on the ordering of operations. We can then distinguish *session* and *prefix* guarantees.

Session guarantees [32] preserve order of operations issued from a user in the same *session*, *i.e.*, informally, a sequence of read and write operations performed during an execution of an application. *Read-your-writes* is a session consistency model [33] commonly used where a user after having written an object, always accesses the updated value and never sees an older value. In *monotonic read consistency* when a user has seen a value for an object, any subsequent accesses to the same object by that user will never return any previous values whereas *monotonic write consistency* serializes the writes by the same process.

Prefix guarantees define orders on the set of update operations applied to each replica. For instance, in [28],

Saito and Shapiro define eventual consistency in data stores. Assuming that the replicas start from the same initial state, they agree on the a *committed prefix* of operations, and this prefix should grow monotonically over time. Moreover, every submitted operation must eventually be included in the committed prefix. However, for the purpose of conflict resolution, some operation may be included in the prefix but not executed.

The committed prefix ensures that starting from the same initial state, all replicas produce the same final state. Note that the prefixes of different replicas are equivalent but not necessarily identical. For instance, two consecutive commuting operations may appear in different orders in the prefixes of two replicas, since this difference does not affect the final state.

ETOB properties ensure both the update propagation and order (session and prefix) guarantees where messages represent operations from clients to replicas. With TOB-Validity and TOB-Agreement all messages will be delivered to every correct replica thus ensuring update propagation. Session consistency models are examples of causal consistency satisfying by the TOB-Causality property of ETOB. On the other hand, our algorithm implementing ETOB ensures that after a time τ when all processes stabilize on trusting the same leader all replicas apply the same sequence of messages while the ETOB-Stability ensures the growth of this prefix after τ.

In [14], the intuition behind eventual consistency was captured through the concept of eventual serializability. Two types of operations were defined: (1) "strict" operations need to be a stable part of a total order at the time of acknowledgement and (2) "weak" operations can be later re-ordered preserving the causal order of operations invoked by the same client. 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 [30] and [19]. It is shown in [30] 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 [19], 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 [19]: 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.

The notion of *eventual consensus* was introduced in [22]. It refers to one instance of consensus which stabilizes at the end; not multiple instances as we consider in this paper. In [12], 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. This abstraction is close to our ERC, but the authors of [22] focused on the shared-memory model and did not address the question of the weakest failure detector.

7 Concluding Remarks

This paper defined the abstraction of eventual total order broadcast and proved its equivalence to eventual repeated 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 [11]. In terms of time complexity, the gap is exactly one message delay: An operation on the strongly consistent replicated state machine must, in the worst case, incur three communication steps [26], while one built 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 [10, 8, 6]. 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. 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.

The folklore "CAP theorem" [1,17] states that no asynchronous system can combine (C)onsistency, (A)vailability, and (P)artition-tolerance. Our result complements this claim. Indeed, it shows that replacing consistency with eventual consistency, while still providing availability and partition-tolerance, still requires the information about failures encapsulated in Ω.

Acknowledgements We would like to thank Vassos Hadzilacos and the anonymous referees for their helpful comments.

References

1. Eric 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. Sebastian Burckhardt. Principles of eventual consistency. *Foundations and Trends in Programming Languages*, 1(1-2):1–150, 2014.
3. Sebastian Burckhardt, Daan Leijen, Jonathan Protzenko, and Manuel Fähndrich. Global sequence protocol: A robust abstraction for replicated shared state. In *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic*, pages 568–590, 2015.
4. Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722, July 1996.
5. Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.
6. Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, June 2008.
7. B. Charron-Bost and G. Tel. Approximation d'une borne inférieure répartie. Technical Report LIX/RR/94/06, Laboratoire d'Informatique LIX, École Polytechnique, September 1994.
8. Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1(2):1277–1288, August 2008.
9. F. Cristian, H. Aghili, R. Strong, and D. Dolev. Atomic broadcast. *Inf. Comput.*, 118(1):158–179, April 1995.
10. Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner 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.
11. Carole Delporte-Gallet, Hugues Fauconnier, and Rachid Guerraoui. Tight failure detection bounds on atomic object implementations. *Journal of the ACM*, 57(4), 2010.
12. Shlomi Dolev, Ronen I. Kat, and Elad M. Schiller. When consensus meets self-stabilization. *Journal of Computer and System Sciences*, 76(8):884 – 900, 2010.
13. Swan Dubois, Rachid Guerraoui, Petr Kuznetsov, Franck Petit, and Pierre Sens. The weakest failure detector for eventual consistency. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing, PODC 2015*, pages 375–384, 2015.
14. Alan Fekete, David Gupta, Victor Luchangco, Nancy Lynch, and Alex 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.
15. Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.
16. Felix C. Freiling, Rachid Guerraoui, and Petr Kuznetsov. The failure detector abstraction. *ACM Comput. Surv.*, 43(2):9:1–9:40, February 2011.
17. Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002.
18. Rachid Guerraoui, Vassos Hadzilacos, Petr Kuznetsov, and Sam Toueg. The weakest failure detectors to solve quittable consensus and nonblocking atomic commit. *SIAM J. Comput.*, 41(6):1343–1379, 2012.
19. Rachid Guerraoui and Eric 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.
20. Vassos Hadzilacos and Sam Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical Report TR 94-1425, Department of Computer Science, Cornell University, May 1994.
21. Prasad Jayanti and Sam Toueg. Every problem has a weakest failure detector. In *PODC*, pages 75–84, 2008.
22. Fabian Kuhn, Yoram Moses, and Rotem 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.
23. Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.
24. Leslie Lamport. Proving the correctness of multiprocessor programs. *Transactions on software engineering*, 3(2):125–143, March 1977.
25. Leslie Lamport. The Part-Time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.
26. Leslie Lamport. Lower bounds for asynchronous consensus. *Distributed Computing*, 19(2):104–125, 2006.
27. Achour Mostefaoui, Michel Raynal, and Frédéric Tronel. From binary consensus to multivalued consensus in asynchronous message-passing systems. *Inf. Process. Lett.*, 73(5-6):207–212, March 2000.
28. Yasushi Saito and Marc Shapiro. Optimistic replication. *ACM Comput. Surv.*, 37(1):42–81, March 2005.
29. Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.
30. Marco Serafini, Dan Dobre, Matthias Majuntke, Péter Bokor, and Neeraj Suri. Eventually linearizable shared objects. In Andréa W. Richa and Rachid Guerraoui, editors, *Proceedings of the 29th Annual ACM Symposium on Principles of Distributed Computing*, pages 95–104. ACM, 2010.

31. Atul Singh, Pedro Fonseca, Petr Kuznetsov, Rodrigo Rodrigues, and Petros 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.
32. Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike Spreitzer, Marvin Theimer, and Brent W. Welch. Session guarantees for weakly consistent replicated data. In *Proceedings of the Third International Conference on Parallel and Distributed Information Systems*, PDIS '94, pages 140–149, Washington, DC, USA, 1994. IEEE Computer Society.
33. Werner Vogels. Eventually consistent. *Commun. ACM*, 52(1):40–44, January 2009.