Eventually Linearizable Shared Objects

Marco Serafini,* Dan Dobre, Matthias Majuntke, Péter Bokor and Neeraj Suri
CS Department, TU Darmstadt
Hochschulstr. 10
64289 Darmstadt, Germany
{marco, dan, majuntke, pbokor, suri}@cs.tu-darmstadt.de

ABSTRACT

Linearizability is the strongest known consistency property of shared objects. In asynchronous message passing systems, Linearizability can be achieved with $\diamond\mathcal{S}$ and a majority of correct processes. In this paper we introduce the notion of *Eventual Linearizability*, the strongest known consistency property that can be attained with $\diamond\mathcal{S}$ and *any number* of crashes. We show that linearizable shared object implementations can be augmented to support *weak* operations, which need to be linearized only eventually. Unlike *strong* operations that require to be always linearized, weak operations are live in worst case runs. However, there is a tradeoff between ensuring termination of weak and strong operations when processes have only access to $\diamond\mathcal{S}$. If weak operations terminate in the worst case, then we show that strong operations terminate only in the absence of concurrent weak operations. Finally, we show that an implementation based on $\diamond\mathcal{P}$ exists that guarantees termination of *all* operations.

Categories and Subject Descriptors

D.4.5 [**Operating Systems**]: Reliablity—*Fault-tolerance*; D.4.7 [**Operating Systems**]: Organization and Design—*Distributed systems*; F.2.m [**Theory of Computation**]: Analisys of Algorithms and Problem Complexity

General Terms

Algorithms, Design, Reliability, Theory

Keywords

eventual linearizability, graceful degradation, availability

1. INTRODUCTION

Shared objects are a useful abstraction in the design of concurrent systems. A concurrent system consists of a collection of sequential processes communicating through shared objects. A shared object can be made tolerant to process failures by storing a copy of the shared object at each process and by having the processes coordinate their actions to implement a certain degree of consistency. The more consistent the local copies are kept, the easier it is to design a distributed application using the replicated object.

The strongest consistency property is Linearizability [13], which provides the illusion that each operation applied to the shared object takes effect instantaneously at some point between its invocation and its response. In this way, the processes have the impression of interacting with a "centralized" object that executes all operations in a sequential order consistent with the real time ordering of operations. Linearizability, however, can be achieved if and only if consensus can be solved. In an asychronous message-passing system, consensus can be solved assuming a failure detector of class $\diamond\mathcal{S}$, or the equivalent class Ω, and a majority of correct processes [6]. If these conditions are not met, a linearizable implementation blocks, becoming unavailable [5].

In many real world applications, availability is imperative, and therefore blocking is often unacceptable [8, 9, 11]. In practice, processes often issue operations that do not need to be linearized. We call these operations *weak* as opposed to *strong* operations that must be linearized. Ideally, weak operations applied to a shared object should terminate irrespective of the failure detector output or the number of faulty processes. However, it is acceptable that weak operations violate Linearizability only if the system deviates from its "normal" behavior, and that such violations must cease when the anomalies terminate [12, 1]. We call this property *Eventual Linearizability.*

Shared objects with Eventual Linearizability can, for example, be used for master-worker applications. Consider a replicated real-time queue used to dispatch taxi requests to taxi cabs [12]. Some degree of redundant work, such as having multiple cabs respond to the same call, can be accepted if this prevents the system from becoming unavailable, for example by letting cabs dequeue requests even in presence of anomalies. However, no redundant work should take place when there is no anomaly.

In this paper we address the following question: Is it possible to achieve these desirable properties of weak operations without sacrificing linearizability and termination of strong operations? We answer this question in the negative. In fact, combining Linearizability and Eventual Linearizability requires using a stronger failure detector to complete strong operations than the one sufficient for Consensus.

We introduce the notion of Eventual Linearizability for weak operations, which is the strongest known consistency

*Marco Serafini is currently also with Yahoo! Research, Av. Diagonal 177, 08018 Barcelona, Spain

PODC'10, July 25–28, 2010, Zurich, Switzerland.

property that can be attained with $\diamond\mathcal{S}$ despite *any number* of crashes. Eventual Linearizability guarantees that Linearizability is violated only for a finite time window. It satisfies the same locality and nonblocking properties as Linearizability. We show that Eventual Linearizability for weak operations cannot be provided using existing notions of Eventual Consistency [20, 25, 10]. With Eventual Consistency, in fact, Linearizability can be violated whenever multiple operations are invoked concurrently. Therefore, Eventual Consistency never ensures Linearizability.

We introduce a primitive, called *Eventual Consensus*, that we prove to be necessary and sufficient to implement Eventual Linearizability. Eventual Consensus is strictly weaker than consensus, since it can be implemented with $\diamond\mathcal{S}$ despite any number of faulty processes. Inputs to Eventual Consensus are operations proposed by processes, and outputs are sequences of operations. Informally, Eventual Consensus requires that after some unknown time t, all operations proposed after t are totally ordered at each process *before* being output.

Beyond introducing and formalizing Eventual Linearizability and Eventual Consensus, we study whether Consensus implementations can be extended to provide Eventual Consensus without degrading their properties.

We present a shared object implementation, called Aurora, which provides Linearizability for strong operations and Eventual Linearizability for weak operations using the Eventual Consensus primitive. For high availability, Aurora ensures termination and Eventual Consistency for weak operations in asynchronous runs. Aurora also preserves causal consistency [14]. Unlike other weakly consistent implementations such as Lazy Replication [15] and Bayou [23], Aurora additionally implements Eventual Linearizability for weak operations in runs where processes have access to a failure detector of class $\diamond\mathcal{S}$. In this case, strong operations terminate in the absence of concurrent weak operations. Finally, if the processes have access to a failure detector $\mathcal{D}$ of class $\diamond\mathcal{P}$, then all operations terminate even in presence of concurrency. Aurora is a gracefully degrading algorithm because it requires different degrees of synchrony to achieve different consistency semantics. In particular, it ensures termination of weak operations even in asynchronous runs by gracefully degrading Eventual Linearizability to Eventual Consistency

It may seem unnecessary that Aurora requires a stronger failure detector than a Consensus algorithm to terminate strong operations. We show, perhaps unexpectedly, that this reflects a fundamental tradeoff. Specifically, we show that with $\diamond\mathcal{S}$, it is impossible to ensure termination of strong operations with a majority of correct processes and at the same time to achieve Eventual Consensus and termination of weak operations with a minority of correct processes.

Interestingly, at the heart of circumventing the impossibility lies the ability to eventually tell if consensus will terminate, which is possible with $\diamond\mathcal{P}$ but impossible with $\diamond\mathcal{S}$. This seems to be a fundamental and unexplored difference between the two classes of failure detectors. On the other hand, a strongly complete failure detector is sufficient to eventually detect that consensus will *not* terminate.

Summary of contributions and outline.

We distinguish between strong operations, which must be linearized, and weak operations, which need to be linearized only eventually. For the latter, we introduce and formalize the Eventual Linearizability correctness condition (Section 3). We show that Eventual Linearizability is stronger than Eventual Consistency but equivalent to Eventual Consensus (introduced in Section 4). Next, we study the inherent tradeoffs of combining Linearizability and Eventual Linearizability (Section 5). First we show an impossibility result that limits the design space of eventually linearizable implementations (Section 5.1). Finally, we present a shared object implementation called Aurora that combines Linearizability and Eventual Linearizability (Section 5.2). In asynchronous runs, Aurora gracefully degrades to Eventual Consistency. In this paper we present our main results, referring the reader to [21] for further details and proofs.

2. RELATED WORK

Previous work has studied how to extend Linearizability with weaker consistency properties. Eventual Serializability requires that "strict" operations and all operations preceding them be totally ordered at the time of their response, while other operations may only be totally ordered *after* their response [10]. Most existing systems implementing optimistic replication provide variations of this property, often called Eventual Consistency [20, 25]. As we show, Eventual Consistency is weaker than Eventual Linearizability. Timed Consistency strengthens sequential consistency by setting a real-time bound Δ after which operations must be seen by any other process [24]. If $\Delta = 0$ the specification is equivalent to Linearizability. If not, Timed Consistency allows completed operation to remain invisible to subsequent operations, similar to Eventual Consistency. In this case, our result can be easily extended to show that Timed Consistency is not stronger than Eventual Linearizability. Like Eventual Serializability, Hybrid Consistency requires strong operations to be linearizable with each other but relaxes the ordering between pairs of weak operations [2]. Zeno extends Byzantine-fault tolerant state machine replication to guarantee availability and Eventual Consistency for weak operations in presence of partitions [22]. Zeno appears to achieve Eventual Consensus in some "good" runs. However, Zeno relaxes Linearizability for strong operations. In fact, processes invoking weak operations are allowed to observe concurrent strong operations in different orders.

A number of distributed systems, including modern highly-available data center services such as Amazon's Dynamo [9], the Google File System [11] and Yahoo's PNUTS [8] allow trading Linearizability for availability in presence of partitions, which occur between geographically remote data centers as well as inside data centers [25]. A survey on many practical weakly consistent systems is [20]. A drawback of weakly consistent systems is that they are notoriously hard to program and to understand [4]. Authors of [1] argue, with motivations similar to ours, that many systems aim at being "usually consistent". They propose a quantitative measure, called consistability, to study the tradeoffs between performance, fault-tolerance and consistency.

There is a large body of work on weak consistency semantics for distributed shared memories having read/write semantics. For a survey we refer to [19]. Eventual Linearizability is an eventual safety property that can be combined with any of these safety properties. For example, Aurora has a causal consistency property that allows implementing causal memories [14]. Refined specifications of graceful degradation and corresponding implementations for trans-actions taking snapshots of the state of multiple objects are presented in [26].

3. DEFINITION OF EVENTUAL LINEARIZABILITY

In this section we first define a model of concurrent executions. Next, we define Eventual Linearizability and show that, like Linearizability, it is local and nonblocking.

3.1 Model of concurrent executions

We consider concurrent systems consisting of a set of processes $\{p_i \mid i \in [0, n-1]\}$ accessing a set of shared objects. Processes interact with objects through *operations*. An execution is a history consisting of a finite sequence of operation *invocation* and *response events* taking place at a process and referring to an object. Invocations contain the *arguments* of the operation, while responses contain the *results* of the operation. All operations are unique and are ordered in the history according to the time of their occurrence. We assume the presence of a global clock providing a time reference for the whole system, which starts from 0 and is often referred to as *real-time order*. Processes do not have access to this clock. Given a history H and a process p_j (resp. an object x), we denote $H|j$ (resp. $H|x$) as the restriction of H to call and response events of p_j (resp. on x).

A history is *sequential* if (i) the first event is an invocation, (ii) all invocation events, except possibly the last, are immediately followed by the response event for the same operation, and (iii) response events are immediately preceded by the invocation event for the same operation. A sequential history H is *legal* if, for each object x, $H|x$ is correct according to the sequential specification of x. We denote the order of operations defined by a sequential history H as $<_H$. A *sequential permutation* of a history H is a sequential history obtained by permuting the events of H. A history that is not sequential is called *concurrent*. An operation is called *completed* if the history includes an invocation and a completion event for it. For a history H, we denote *completed*(H) as the subsequence of events in H related to all completed operations. A history is *well-formed* if the subhistory of events of each process is sequential. We assume all histories to be well-formed.

3.2 Definition

Eventual linearizable implementations need to always ensure some minimal weak consistency property that rules out arbitrary behaviors. For each history H, we require that the response to every completed operation o of every process p_i is the result of a legal sequential history $\tau(i, o)$. The history $\tau(i, o)$ must terminate with o, it must consist only of operations invoked in H before o is completed, and it must include all operations observed by p_i before o.

Formally, a history H is *weakly consistent* if, for every process p_i and operation o completed by p_i in H, there exists a legal sequential history $\tau(i, o)$ such that: (i) the last event in $\tau(i, o)$ is a response event of o having the same result as the response event of o in H, (ii) every operation invoked in $\tau(i, o)$ is also invoked in H before o is completed, and (iii) for each operation o' invoked by p_i before o, $\tau(i, o') \subseteq \tau(i, o)$.[1]

This definition of weak consistency is very generic. It allows processes to ignore operations of other processes. Furthermore, subsequent serializations observed by a process can reorder previously-observed operations. Eventual Linearizability can be combined with stronger weak consistency semantic than this. For example, in Section 5.2 we show that it is possible to combine Eventual Linearizability with causal consistency [14].

Eventual Linearizability requires all operations that are invoked after a certain time t to be ordered with respect to all other operations according to their real-time order. Pairs of operations invoked before t can be ordered arbitrarily. This requirement on the order is formalized by the following relation. Let H be a history and t a value of the clock. We define the irreflexive partial order $<_{H,t}$ as follows: $o_1 <_{H,t} o_2$ iff o_2 is invoked after t and the response event of o_1 precedes the invocation event of o_2.

A *t-permutation* P of a history H is a legal sequential history that orders operations of H according to $<_{H,t}$. The results of operations in P do not have to match with those of the corresponding operations in H. Formally, the following two properties must hold for a legal sequential history P to be a t-permutation of H: (P1) an operation o is invoked in P if and only if o is invoked in H; (P2) $<_{H,t} \subseteq <_P$. It is worth noting that every well-formed history H has a t-permutation P for each value of t because results of operations in H and P do not need to match. However, not every well-formed history has a *linearization* as defined in [13].

A *t-linearization* L of a history H is defined as a t-permutation where the results of all operations invoked after t are the same as in H. Operations invoked before t may have observed inconsistent histories that do not correspond to any single legal sequential history. A history H is *t-linearizable* if there exists a t-linearization of H. t-linearizability is a property of histories that may initially be weakly consistent but that eventually start behaving like in a linearization.

We can now define Eventual Linearizability as follows.

Eventual Linearizability: *The implementation of a shared object is eventually linearizable if all its histories are weakly consistent and t-linearizable for some finite and unknown time t.*

Linearizability differs from Eventual Linearizability because the convergence time t is known and equal to zero. In general, any form of t-linearizability where t is known can be easily reduced to Linearizability in systems where processors have access to a local clock with bounded drift. This is why we consider more general scenarios where t exists but is unknown. It is worth noting that, different from t-linearizability, Eventual Linearizability is a property of implementations, not of histories. In fact, all finite histories are trivially t-linearizable for some value of t larger than the time of their last event. Showing Eventual Linearizability on an implementation entails identifying a single value of t for all histories.

We show that Eventual Linearizability has two fundamental properties of Linearizability. *Locality* implies that any composition of eventually linearizable object implementations is eventually linearizable. *Nonblocking* requires that there exist no history such that every extension of the history violates Eventual Linearizability.

Theorem 1. *Eventual Linearizability satisfies locality and nonblocking.*

[1] We abuse the $\subseteq$ notation to indicate that the set of operations of $\tau(i, o')$ is included in the set of operations of $\tau(i, o)$.

4. IMPLEMENTING EVENTUAL LINEARIZABILITY

Eventual Linearizability requires operations to be linearized only eventually and can thus be implemented using primitives that are weaker than Consensus. In this Section we identify which properties must be satisfied by these primitives. We focus on weak operations where Eventual Linearizability is sufficient. Strong operations are introduced in Section 5. Many weakly consistent implementations provide properties such as *Eventual Serializability* [10] or *Eventual Consistency* [20, 25]. We show that these properties are not sufficient to implement Eventual Linearizability, and therefore define a stronger problem, called *Eventual Consensus*, that is stronger than Eventual Consistency but weaker than Consensus. We finally show that Eventual Consensus is necessary and sufficient to implement Eventual Linearizability.

4.1 System model for implementations

In this section we consider shared object implementations using an underlying *consistency layer* to keep replicas consistent. If Linearizability is required for all operations then the consistency layer implements Consensus. The specifications defined in this section refer to properties of consistency layers, unlike Eventual Linearizability which is a property of shared object implementations. For simplicity, we restrict our discussion to implementations of a single shared object.

We model the interface of the consistency layer with two types of events: *submit events*, which are input events including as input value an operation on the shared objects, and *delivery events*, which are output events returning a sequence of operations on the shared object. We denote as $S(i,t)$ the last sequence delivered to process p_i at time $t>0$ and define $S(i,0)$ to be equal to the empty sequence for each i. We assume that the processes interacting with the shared object can fail by crashing. If p_i is crashed at time t, $S(i,t)$ is the last sequence delivered by p_i before crashing. We say that a submitted operation *terminates* when it is included in a sequence that is delivered at each correct process.

The consistency layer itself is implemented on top of an asynchronous message passing system with reliable channels. Implementations can use *failure detectors* [6, 5]. A failure detector $\mathcal{D}$ is a module running at each process that outputs at any time a set of process indices [6]. In this paper we consider four classes of failure detectors. The class Ω includes all failure detectors that output at most one process at each process p_i, which is said to be *trusted* by p_i, and ensures that eventually a single correct process is permanently trusted by all correct processes [5]. The class of *strongly complete* failure detectors, which we denote $\mathcal{C}$, includes all failure detectors that output a set of *suspected* processes and that ensure *strong completeness*, i.e., eventually every process that crashes is permanently suspected by every correct process [6]. The classes of *eventually strong* (resp. *eventually perfect*) failure detectors $\diamond\mathcal{S}$ (resp. $\diamond\mathcal{P}$) include all strongly complete failure detectors having *eventually weak accuracy* (resp. *eventually strong accuracy*), i.e., eventually some correct process is (resp. all correct processes are) not suspected by any correct process [6].

4.2 Eventual Consistency and Eventual Consensus

Our formalization of Eventual Consistency builds upon the properties of Eventual Serializability [10] and Eventual Consistency [20] and is expressed in terms of a weakened form of Consensus. Like Eventual Serializability, we allow processes to temporarily diverge from each other on the order of operations and to eventually converge to a total order. Eventual Serializability supports defining precedence relations with each operation to constraint their execution order. These relations are typically used to specify causal consistency [10, 15]. Since we focus here on Eventual Consistency properties, these aspects are orthogonal to our discussion and are abstracted away.

Eventual Consistency: *A consistency layer satisfies Eventual Consistency if the following properties hold.*

- **Nontriviality:** *For any process p_i and time t, every operation in $S(i,t)$ has been invoked at a time $t' \leq t$ and appears only once in $S(i,t)$;*
- **Set stability:** *For any process p_i, if $t \leq t'$ then each operation in $S(i,t)$ is included in $S(i,t')$;*
- **Prefix consistency:** *For any time t there exists a sequence of operations P_t such that:*
  (C1) *For any correct process p_i, P_t is a prefix of $S(i,t')$ if $t \leq t'$;*
  (C2) *P_t is a prefix of $P_{t'}$ if $t \leq t'$;*
  (C3) *Every operation o submitted at time t' by a correct process is included in $P_{t''}$ for some $t'' \geq t'$.*

Note that property (C3) of prefix consistency implies *Liveness*, i.e., for any correct processes p_i and p_j and time t, every operation submitted by p_i at time t is included in $S(j,t_j)$ for some $t_j \geq t$.

This definition of Eventual Consistency is a relaxation of Consensus on sequences of operations [18].[2] Consensus requires the same nontriviality and liveness properties as Eventual Consistency, but requires stronger stability and consistency properties. *Stability* requires that for any process p_i, $S(i,t)$ is a prefix of $S(i,t')$ if $t<t'$. *Consistency* requires that for any processes p_i and p_j and time t, one of $S(i,t)$ and $S(j,t)$ is a prefix of the other.

Set stability allows reordering the sequence of operations returned as an output, provided that all operations returned previously are included in the new sequence. Prefix consistency allows replicas to temporarily diverge in a suffix of operations. However, it requires eventual convergence among all replicas on a common prefix P_t of operations. Property (C1) of prefix consistency says that a common prefix P_t of operations has been delivered by each replica; (C2) constraints this prefix to be monotonically increasing; (C3) ensures that all completed operations are eventually included in the common prefix.

Eventual Consistency is not sufficient to implement Eventual Linearizability not even for simple read/write registers, as shown in Theorem 2.

Theorem 2. *An eventually linearizable implementation of a single-writer, single-reader binary register cannot be simulated using only an eventually consistent consistency layer in a system with more than one process.*

The intuition for this result can be given by a simple example. Consider two processes p_0 and p_1 that share one single-writer, single-reader binary register holding a current

[2]We consider here the case where all processes are proposers and learners. We also trivially modify nontriviality to rule out sequences with duplicates.

value 1 at a given time t. Assume that p_0 is the writer of the register and p_1 is the reader. Process p_0 invokes a $write_0(0)$ operation after t. After this operation is completed, process p_1 invokes a $read_1()$ operation. Prefix consistency allows the consistency layer to delay convergence to a common prefix P_t for an arbitrarily long time. Before completing $read_1()$, p_1 may thus not distinguish this run from a run where $write_0(0)$ was never invoked. Therefore, $read_1()$ returns the previous value 1. A consistent ordering P_t of these two operations can be delivered by the consistency layer of both processes *after* both operations are completed. This is sufficient to satisfy Eventual Consistency. Such a pattern can occur after any finite time, making t-linearizability impossible for any t.

The key to achieve Eventual Linearizability is in strengthening stability. Assume in the previous example that the consistency layer is not allowed to change the order of the operations it has delivered after t. p_0 can complete its operation only after the consistency layer delivers a sequence containing $write_0(0)$. In order to prevent the consistency layer of p_0 from reordering its delivered sequence, the first non-empty consistent prefix $P_{t'}$ must include $write_0(0)$. This implies that the consistency layer of p_1 has to deliver $write_0(0)$ before $read_1()$ in order to preserve stability. p_1 can thus execute this sequence and return 0, respecting linearizability. In other words, an Eventually Consistent consistency layer satisfying eventual stability must eventually start to deliver all operations in a total order *before* the operations are completed. This total order also includes all the operations that have been submitted before t.

The previous example gives us the insight for the definition of Eventual Consensus. Different from Eventual Consistency, the delivered sequences eventually stop reordering operations that were previously delivered.

Eventual Consensus: *A consistency layer satisfies Eventual Consistency if Eventual Consistency and the following additional property hold:*

Eventual Stability: There exists a time t such that for any times t' and t'' with $t \leq t' \leq t''$ and for any process p_i, $S(i, t')$ is a prefix of $S(i, t'')$.

Implementing Eventual Consensus is both necessary and sufficient to achieve Eventual Linearizability for generic objects as shown in Theorem 3. This result reduces the problem of obtaining eventually linearizable shared object implementations to the problem of implementing a consistency layer satisfying Eventual Consensus.

Theorem 3. *Eventual Consensus is a necessary and sufficient property of a consistency layer to implement arbitrary shared objects respecting Eventual Linearizability.*

Algorithm 1 shows the sufficiency part of the result. Whenever an operation is invoked, it is submitted to the consistency layer. The operation is then completed as soon as a sequence containing the operation is delivered. The returned sequence is executed and the result is returned in a completion event. Before stability eventually holds, nontriviality and set stability are sufficient to satisfy weak consistency. As discussed in the previous register example, eventual stability ensures that processes eventually start delivering operations in the same total order, which is identified by the consistent prefix P_t, *before* the operations are completed. This allows implementing Eventual Linearizability.

Necessity is shown by Algorithm 2, which uses a shared sequence having an append and a read operation. Whenever an operation is submitted, it is appended onto the sequence. The object is periodically read and its value is delivered. The weak consistency property of the sequence is sufficient to ensure nontriviality and set stability. When the object starts to be eventually linearizable, all reads and appends are totally ordered in a legal sequential history. This ensures that eventually all operations are included in the same total order, as required by prefix consistency, and that read sequences that are delivered are never reordered in the future, as required by eventual stability.

5. COMBINING LINEARIZABILITY AND EVENTUAL LINEARIZABILITY

We distinguish between strong operations that need to be linearized and weak operations that require to be eventually linearized. Strong operations are delivered only if Consensus is reached on the prefix including them as last operation. This is called a *strong prefix.* We extend the specification of Eventual Consensus accordingly.

Strong prefix stability: *For any process p_i, time t, strong operation s and sequence π, if $\pi\, s$ is a prefix of $S(i,t)$ and $t' \geq t$ then $\pi\, s$ is a prefix of $S(i,t')$.*

Strong prefix consistency: *For any processes p_i and p_j, time t, strong operations s_i and s_j and prefixes π_i and π_j, if $\pi_i\, s_i$ is a prefix of $S(i,t)$ and $\pi_j\, s_j$ is a prefix of $S(j,t)$ then one of $\pi_i\, s_i$ and $\pi_j\, s_j$ is prefix of the other.*

If all operations are strong, Eventual Consensus is equivalent to Consensus. One would desire to achieve termination of weak operations in all runs together with termination of strong operations in runs where Linearizability can be achieved. In this Section we discuss impossibility and possibility results on this topic.

5.1 Impossibility in combining Linearizability and Eventual Linearizability

In this section we show that even if a $\diamond\mathcal{S}$ failure detector is given for termination of weak operations, strong operations cannot terminate in runs where consensus can be solved (see Theorem 4).

The intuition behind the impossibility lays in the concurrency between weak and strong operations. We construct an infinite run where some strong operation s is never completed. For this, we consider an Eventual Consensus layer ensuring stability after a time t in a run where all events occur after the time t. Assume that a strong operation s is submitted by a correct process and that the processes are trying to reach consensus on a strong prefix $\pi\, s$. Let a submit event for an operation $w \notin \pi$ occur at a correct process p_i before consensus on $\pi\, s$ is reached. Process p_i cannot know whether consensus will terminate or not, as it accesses only failure detector $\diamond\mathcal{S}$, but it must deliver weak operations in either case. Therefore, p_i cannot wait until consensus on $\pi\, s$ is reached before delivering w. p_i is thus forced to deliver w before consensus on $\pi\, s$ is reached. When consensus on $\pi\, s$ is reached, eventual stability forbids p_i to deliver $\pi\, s$ because w is not in π. Therefore, consensus needs to be reached on a new strong prefix $\varphi\, s$ with $w \in \varphi$. However, a new weak operation w' may be submitted before consensus on $\varphi\, s$ is reached. This pattern can be repeated forever. As a result, *execute(o, H)*: returns the result of executing the sequence H up to and including the operation o;

upon invoke (o)
  $curr \leftarrow o$;
  submit(o);

upon deliver(H)
  if $curr \neq \bot \wedge curr \in H$ **then**
    $r \leftarrow$ execute($curr$, H);
    $curr \leftarrow \bot$;
    complete (o,r);

Algorithm 1: An eventually linearizable implementation of a generic object using Eventual Consensus.

append(o): appends an operation o at the end of the sequence;
read(): returns the current value of the sequence;

upon submit (o)
  append(o);

upon periodic tick
  $H \leftarrow$ read();
  deliver (H);

Algorithm 2: Solving Eventual Consensus using an eventually linearizable implementation of an append/read sequence object.

the strong operation s is never completed even if consensus can be solved.

This result highlights an implicit tradeoff in implementing Eventual Linearizability. As a consequence of our impossibility result, shared object implementations using $\diamond\mathcal{S}$ can ensure Eventual Linearizability and give up termination of strong operations in presence of concurrent weak operations. Alternatively, they can choose to violate Eventual Linearizability in order to ensure termination of both weak and strong operations. In the latter case, it follows from our result that Eventual Linearizability can be violated whenever there are concurrent weak and strong operations.

In the proof of the following theorem we describe asynchronous computations in terms of events as in [3]. *Input* events submitting operation o at p_i are denoted as $submit_i(o)$. An *output* event occurs when a sequence π is delivered. An operation is *delivered* when a sequence containing it is delivered. *Message receipt* events occur when a process receives a message. The occurrence of these events at a process p_i might *enable* the occurrence of *computation* events at p_i, which might in turn result in p_i sending new messages.[3] We say that a message m is *causally dependent on an event e* if the computation event that generated m is causally dependent on e according to the classical definition of Lamport [16].

Theorem 4. *In a system with $n \geq 3$ processes out of which f can crash, it is impossible to implement a consistency layer that satisfies the following properties using a failure detector $\diamond\mathcal{S}$: (P1) termination of weak operations; (P2) termination of strong operations if $f < n/2$; and (P3) Eventual Consensus if $f < n/2$.*

Proof. Assume by contradiction that a consistency layer satisfying properties (P1), (P2) and (P3) exists. Let processes be partitioned into two sets, Π_m of size $\lfloor (n-1)/2 \rfloor$ and Π_M of size $\lceil (n+1)/2 \rceil$. Consider all runs where no process fails and where the $\diamond\mathcal{S}$ modules of all processes suspect Π_M. By (P3), there exists a time t after which eventual stability holds for each of these runs. We build one such run σ that begins with an event $submit_h(s)$, with $p_h \in \Pi_M$ occurring after time t, where s is a strong operation. σ is an infinite and fair run that is built using an infinite number of finite runs σ_k with $k \geq 0$ in which s is never delivered by any process, thus violating (P2). Each run σ_k with $k > 0$ is built by extending σ_{k-1}. The run σ is the result of an infinite number of such extensions. Run σ is fair by construction because all messages sent in σ_{k-1} are received in σ_k, and because all enabled computation events occur.

[3]If a process sends a message to itself, then the receipt of this message is considered as a local computation event.

Let M_k be the set of messages that are sent, but not yet received, in σ_k. For each σ_k, we show by induction on k the following invariant (I): No process delivers s in σ_k or in any extension of σ_k where (i) all processes in Π_M crash immediately after σ_k, and (ii) all messages in M_k sent by processes in Π_M are lost.

We first consider the case $k = 0$ and define σ_0 is as follows. Let $submit_h(s)$ be the first and only input event of the system. Assume that no process crashes in σ_0. Assume also that no message is received in σ_0 and that all enabled computation events occur. Let M_0 be set of initial messages sent in σ_0.

It is easy to see that (I) is satisfied in σ_0. Since only a strong operation has been submitted, delivering s entails solving consensus on s by definition. Property (I) directly follows from the facts that no message is received in σ_0 and that consensus cannot be solved using $\diamond\mathcal{S}$ in any extension satisfying conditions (i) and (ii) since $f \geq \lceil n/2 \rceil$ (see proof in [6]).

We now define how σ_k is constructed for $k > 0$ by extending σ_{k-1}. Assume that no process crashes in σ_k and that $\diamond\mathcal{S}$ permanently suspects Π_M. Let an event $submit_h(w_k)$ occur at a process $p_i \in \Pi_m$ after σ_{k-1}, where w_k is a weak operation that has never been submitted earlier. Let process p_i eventually deliver a sequence φ_k at a time t_k such that $w_k \in \varphi_k$ and $s \notin \varphi_k$. Assume that no event occurs at any process in Π_M after σ_{k-1} and before t_k. Assume that no message in M_{k-1} sent by processes in Π_M is received by processes in Π_m before t_k. All remaining messages in M_{k-1} are received after t_k in σ_k. Let all enabled computation events occur. Finally, assume that all messages sent after σ_{k-1} are included in M_k and are not received in σ_k.

We first show that the construction of σ_k is valid by showing that t_k and φ_k exist. We construct an extension of σ_{k-1} called σ_{E1}. Assume that in σ_{E1} all processes in Π_M crash immediately after σ_{k-1} (i.e., before $submit_i(w_k)$) and $\diamond\mathcal{S}$ suspects Π_M at all processes. Assume that all messages in M_{k-1} that are sent by processes in Π_M are lost. By property (P1), and since $\diamond\mathcal{S}$ permanently satisfies weak accuracy, process p_i eventually delivers a sequence φ_k with $w_k \in \varphi_k$ at time t_k. Therefore, φ_k and t_k exist. As σ_{k-1} satisfies (I), process p_i cannot deliver s in σ_{E1} because all messages in M_{k-1} sent by processes in Π_M are lost. This implies that $s \notin \varphi_k$. Since process p_i cannot distinguish σ_k and σ_{E1} up to t_k, φ_k is delivered by p_i at time t_k in σ_k too.

We now show the inductive step, i.e., that σ_k satisfies (I). Assume by contradiction that a sequence $\pi\, s\, \varepsilon_d$ for some sequences π and ε_d is delivered for the first time by a process p_d in σ_k or in an extension of σ_k respecting (i)-(ii). As s was not delivered in σ_{k-1}, sequence $\pi\, s\, \varepsilon_d$ is delivered after σ_{k-1} and, by the argument above, also after t_k.

Consider first the case $p_d \in \Pi_m$. Let σ_{E21} be an extension of σ_k where p_d delivers $\pi\, s\, \varepsilon_d$ and let t'_k be the time when this delivery occurs. Let all processes in Π_M crash immediately after σ_k and let all the messages sent by processes in Π_M sent after σ_{k-1} to processes in Π_m be lost. Finally, let $\diamond\mathcal{S}$ return Π_M at all processes. From eventual stability and since p_i has already delivered at time $t_k < t'_k$ a sequence φ such that $w_k \in \varphi$ but $s \notin \varphi$, it follows $w_k \in \pi$.

We now consider a run σ_{E22} where the same events as in σ_{E21} occur until time t'_k but no process crashes before t'_k. All processes in Π_m crash immediately after t'_k. All messages sent from processes in Π_m to processes in Π_M after σ_{k-1} are lost. Assume that after t'_k, $\diamond\mathcal{S}$ eventually returns Π_m at all processes in Π_M. p_d cannot distinguish σ_{E21} and σ_{E22} until t'_k, so it delivers $\pi\, s\, \varepsilon_d$ at time t'_k in σ_{E22} too. By (P2), all processes in Π_M are correct and must thus eventually deliver a sequence containing s. From strong prefix consistency and strong prefix stability, this sequence must have $\pi\, s$ as prefix with $w_k \in \pi$.

Finally, consider a run σ_{E23} that is similar to σ_{E22} but where the $submit_i(w_k)$ event does not occur. Let all processes in Π_m crash at the same time as in σ_{E22}, and let all messages sent by processes in Π_m after σ_{k-1} be lost. Assume that no other process crashes. Let the outputs of $\diamond\mathcal{S}$ be at any time the same as in σ_{E21}. Runs σ_{E21} and σ_{E22} are indistinguishable for the processes in Π_M, which thus eventually deliver a sequence having $\pi\, s$ as a prefix with $w_k \in \pi$. However, w_k has never been submitted in σ_{E23}. This violates nontriviality, showing that $p_d \notin \Pi_m$.

Next, consider the case $p_d \in \Pi_M$. By definition of (I), p_d must deliver $\pi\, s\, \varepsilon_d$ in σ_k. Consider an extension σ_{E31} of σ_k where no process crashes. By (P2), all processes must eventually deliver a sequence containing s. By strong prefix consistency, all processes must eventually deliver a sequence having $\pi\, s$ as prefix. By eventual stability, since p_i has already delivered at time t_k a sequence φ_k including w_k and not s, it must hold $w_k \in \pi$. However, process p_d cannot distinguish σ_k from a similar run σ_{E32} where $submit_i(w_k)$ does not occur. In fact, p_d receives no message in σ_k that is causally related with $submit_i(w_k)$. Therefore, p_d delivers $\pi\, s\, \varepsilon_d$ with $w_k \in \pi$ in σ_{E32} too, a violation of nontriviality. This ends our proof that σ_k satisfies (I).

The infinite run σ can be built iteratively by extending σ_k as it has been done with σ_{k-1}. The resulting run is fair by construction because all messages in M_{k-1} are delivered in σ_k and no computation event is enabled forever without occurring. During the whole run no process crashes. According to (P2), s should be delivered in a finite prefix of σ. By construction, however, each finite prefix τ of σ is also prefix of a run $\sigma_{k'}$ for some k'. From the invariant (I), s is never delivered in $\sigma_{k'}$, a contradiction. □

5.2 Aurora: A gracefully degrading implementation

In this section we introduce Aurora (Figure 1), an algorithm implementing Eventual Consensus and thus, from Theorem 3, Eventual Linearizability. Aurora shows that Eventual Consensus can be implemented with $\diamond\mathcal{S}$ and any number of correct processes, still ensuring termination of weak operations and Eventual Consistency in worst-case asynchronous runs. The algorithm also shows that causal consistency can easily be combined with Eventual Consensus.

Failure detectors and communication primitives.

Aurora ensures termination of weak operations and Eventual Consistency in asynchronous runs. To this end, Aurora uses a failure detector module $\mathcal{D} \in \mathcal{C}$, which outputs the set of indices of the processes that have been suspected to crash. Virtually all failure detector implementations are of class $\mathcal{C}$ in asynchronous runs. The key property of Eventual Consensus, eventual stability, is achieved by letting a leader order all operations. For this we require that $\mathcal{D} \in \diamond\mathcal{S} \subseteq \mathcal{C}$, while for termination of strong operations we assume $\mathcal{D} \in \diamond P \subseteq \diamond\mathcal{S}$. This models the fact that even if Aurora optimistically relies on additional synchrony in order to achieve Eventual Consensus, the algorithm falls back to Eventual Consistency to ensure liveness of weak operations in asynchronous runs. The use of $\diamond\mathcal{P}$ to complete strong operations is a consequence of Theorem 4. For simplicity, we use $\Omega_{\mathcal{D}}$ to denote a simulation of a leader election oracle ensuring the properties of Ω on top of $\mathcal{D}$ in runs where $\mathcal{D} \in \diamond\mathcal{S}$ similar to [7]. The simulation ensures that the leader trusted by $\Omega_{\mathcal{D}}$ is not suspected by $\mathcal{D}$. We call the process that is permanently trusted by $\mathcal{D}$ when $\mathcal{D} \in \Omega_{\mathcal{D}}$ the *permanent leader*.

Processes use two communication primitives: a reliable channel providing *send* and *receive* primitives, and a (uniform) FIFO atomic broadcast primitive providing *abcast* and *abdeliver* primitives [3]. Implementing atomic broadcast is equivalent to solving consensus [6]. We consider atomic broadcast implementations that use a failure detector Ω and a majority of correct processes for termination and that always respect their safety properties [17, 6]. The algorithm assumes that a predefined deterministic total order relationship $<_D$ exists. For simplicity, the algorithm sends and delivers whole histories although it is simple to optimize this away [10]. Garbage collection can be executed by periodically issuing strong operations for this purpose [22].

Properties of the Aurora algorithm.

Similar to weakly consistent implementations such as [15, 23], Aurora ensures termination of weak operations, causal consistency and Eventual Consistency if $\mathcal{D} \in \mathcal{C}$. If $\mathcal{D} \in \diamond\mathcal{S}$, Eventual Consensus is implemented. Termination of strong operations is ensured if $\mathcal{D} \in \diamond\mathcal{P}$ or, in absence of concurrent weak operations, if $\mathcal{D} \in \diamond\mathcal{S}$.

Checking if consensus will terminate.

A direct consequence of Theorem 4 is that if a leader p_{ld} has started consensus on a strong prefix $\pi\, s$ and it receives a weak operation w afterwards, it needs to distinguish whether consensus will terminate. If this is the case, w must wait to be ordered after $\pi\, s$ once consensus is reached. Else, w must be immediately be delivered since consensus will not terminate, and thus the strong operation will have to wait before being completed. Consensus will terminate if eventually there exists a stable majority of correct processes permanently trusting p_{ld}.[4]

Aurora uses *trust messages* to let p_{ld} know which processes trust it. Whenever $\Omega_{\mathcal{D}}$ outputs a new leader p_j at a process p_i, p_i sends a TRUST(j) message to all processes through FIFO reliable channels. Each process p_i keeps a *trusted-by*

[4]We call a stable majority a majority quorum that does not change over time. The weakest failure detector to solve consensus, which is Ω, requires that eventually *all* correct processes permanently trust the same correct process p_{ld}. We show, however, that Ω can be simulated if eventually a stable majority of correct processes permanently trusts p_{ld}.

set TB including the indices of all the processes p_j such that TRUST(i) is the last trust message received by p_i from p_j. This processing of trust messages is not included in Figure 1.

The leader uses the trusted-by set and a failure detector of class $\mathcal{C}$ to stop waiting for consensus unless consensus terminates. When a consensus instance is started, the leader remembers the subset T of TB that is composed only by correct processes (according to $\mathcal{D}$). Even in worst-case runs where $\mathcal{D} \in \mathcal{C}$, T will eventually include only correct processes. If T never changes and is a majority quorum, then there exists a majority of correct processes permanently trusting the leader. Consensus on $\pi\, s$ will thus eventually terminate, so the leader can wait to order and deliver w until this happens. The *wait-consensus* predicate is defined to reflect the aforementioned condition.

From Theorem 4, having a failure detector $\diamond\mathcal{S}$, so a single leader, and a majority of correct processes is not sufficient to implement the properties of Aurora. The leader needs to eventually detect that such majority exists, which is ensured if $\mathcal{D} \in \diamond\mathcal{P}$. This eventually lets the predicate *wait-consensus* be true whenever a consensus instance is ongoing, a sufficient condition for termination of strong operations. In fact, T will eventually be equal to the set of correct processes.

Note that if there is no concurrency between weak and strong operations, termination can be guaranteed for all operations without the need for distinguishing whether consensus can terminate or not.

Processing weak operations.

The processing of weak operations is described by Algorithm 3. When a weak operation o is submitted at a process p_i, p_i sends it in a *weak request* message to the current leader p_{ld} and waits for an answer from the leader. In order to preserve causal consistency, a weak request of p_i also contains its current history H and an associated round counter d which will be explained later. H contains all operations causally preceding o. When a weak request message m is received by p_{ld}, it merges its local history with the one received in m before adding o to its local history. This is done in order to preserve causal consistency. We will discuss the details of the merge operation (see Algorithm 4) later on.

If the leader has proposed a strong prefix and is waiting to deliver it, it might wait until consensus on it is completed. This occurs if the leader thinks that consensus can be solved and therefore *wait-consensus* is true. In this case, the leader stores the request in the set W and waits until the strong prefix is delivered or *wait-consensus* becomes false. When p_{ld} processes the weak request, it sends a *push* message containing its local history, including also o, back to p_i. When p_i receives the push message, it merges the history of p_{ld} with its own history to order o respecting the causal dependencies of all the operations ordered by the leader before o. The resulting history contains o and is now delivered by p_i.

As previously discussed, *wait-consensus* eventually becomes false unless consensus can be solved. Also, if p_{ld} is crashed, the failure detector will eventually suspect it. In the latter case, process p_i knows that no permanent leader is yet elected so eventual stability cannot yet be achieved. Therefore, p_i locally appends o to its current local history and delivers it without further waiting for a push message.

Processing strong operations - Overview.

The handling of strong operations is described by Algorithm 5 and is more complex. For eventual stability, if there is a permanent leader p_{ld} then strong operations should be delivered according to the order indicated by p_{ld}. However, we cannot rely on a leader to be permanent for strong prefix stability and consistency.

The properties of strong operations imply that delivering a strong prefix $\pi\, s$ requires solving consensus on $\pi\, s$. Equivalently, processes can propose strong prefixes by atomically broadcasting them and using some deterministic decision criteria to consistently choose one proposal. The main implication of Theorem 4, however, is that processes cannot just deliver the first strong prefix $\pi\, s$ proposed by a leader p_{ld}, even if this p_{ld} uses atomic broadcast. In fact, as long as p_{ld} believes that atomic broadcast will not terminate, it might have delivered some weak operation $w \notin \pi$ before being able to abdeliver $\pi\, s$. In this case, p_{ld} cannot deliver $\pi\, s$ for eventual stability and it needs to propose a new prefix for s.

Processes need to decide when a proposed strong prefix can be delivered because it is *stable*, i.e. it has been abdelivered by atomic broadcast and no weak operation has been delivered in the meanwhile. Establishing that a prefix is stable is a local decision of a leader p_{ld}. The problem now is how p_{ld} can communicate this local decision and let other processes agree on its decision in presence of concurrent proposals from multiple leaders. If p_{ld} just atomically broadcasts that a prefix is stable, this creates again the same problem as before: all processes would have to wait that a stability confirmation from the leader is successfully broadcast before delivering the strong prefix. In the meanwhile, p_{ld} might locally store and deliver some new weak operation.

The problem of multiple concurrent leaders is solved in Aurora by using *rounds* and identifying a single leader as the *winner* of each round. Processes store the current round k and deliver a single strong prefix at each round. Leader processes that receive a new strong operation atomically broadcast the strong operation in a *proposal* message for the current round. The leader whose proposal is the first one to be atomically delivered for a round is the winner of that round. The winner of a round can propose multiple new strong prefixes for the round. These are received in the same order as they are abcast by the leader since the broadcast primitive is FIFO.

Assume that a proposed strong prefix becomes stable at the winner of the current round, that is, the winner abdelivers the stable prefix and sees that it is consistent with its current local history. The winner can now safely decide to locally store the strong prefix in its local history, deliver it, and stop sending proposals for the round. The winner abcasts in this case a *close round* message indicating that the other processes can deliver its last proposed strong prefix for the round. A process abdelivering a close round message m for the current round delivers the last strong prefix proposed by the winner for that round and abdelivered before m. To ensure liveness in case a winner crashes, each process that suspects the winner of the current round can send a close round message.

Since proposal and close round messages are atomically broadcast, it is evident that all processes that did not win a round abdeliver the same strong prefix π for that round. Consistency with a winner of a round that has delivered a stable strong prefix based only on a local decision is ensured as follows. The prefix π is contained in the last proposal message m abdelivered by the winner, and thus by any other process, for the round, and it is not preceded by any close round message for the same round. Even if the win-

upon submit (o) and o is weak
  $ld \leftarrow \Omega_{\mathcal{D}}$;
  send WREQ$(H,\ d\ op)$ to p_{ld};
upon receive WREQ$(H',\ d',\ op')$ from j
  if *wait-consensus* and $(H', d', op') \notin W$ **then**
    add $(H',\ d',\ op')$ into W;
  else
    $(H, d) \leftarrow$ merge$(H',\ d',\ H,\ d)$;
    if $op' \notin H$ **then** append op' onto H;
    send PUSH$(H,\ d)$ to p_j;
upon receive PUSH$(H',\ d')$
  $(H, d) \leftarrow$ merge$(H',\ d',\ H,\ d)$;
  deliver(H);
upon *suspect-ld*
  append last locally submitted weak operation onto H;
  deliver(H);
upon *stop-waiting-consensus*
  foreach $(H',\ d',\ op') \in W$ **do**
    $(H, d) \leftarrow$ merge$(H',\ d',\ H,\ d)$;
    if $op' \notin H$ **then** append op' onto H;
    send PUSH$(H,\ d)$ to p_j;
    remove $(H',\ d',\ op')$ from W;

Algorithm 3: Handling of weak operations

upon periodic tick
  send PUSH$(H,\ d)$ to all other processes;

function merge$(H',\ d',\ H,\ d)$
  $d_{new} \leftarrow \max(d, d')$;
  if $d = d_{new}$ **then** $H_{new} \leftarrow$ longest strong prefix of H;
  else $H_{new} \leftarrow$ longest strong prefix of H';
  $O \leftarrow$ set of weak operations in $(H' \cup H) \setminus H_{new}$;
  $R \leftarrow$ order O according to $<_H \cup <_{H'}$ and break cycles according to $<_D$;
  append R onto H_{new} in R order;
  return (H_{new}, d_{new});

Algorithm 4: Background dissemination and merge

upon submit (o) and o is strong
  send SREQ$(H,\ d,\ op)$ to all processes;

upon receive SREQ$(H',\ d',\ op)$ from j
  $(H, d) \leftarrow$ merge$(H',\ d',\ H,\ d)$;
  add op into N;
upon *must-propose-new-prefix*
  $S \leftarrow N \setminus H$;
  $Q \leftarrow H$;
  $T \leftarrow TB \setminus \mathcal{D}$;
  abcast PROP$(Q,\ S,\ k)$;
upon abdeliver PROP$(H',\ S,\ k')$ from p_j
  if *from-round-winner* **then**
    $P \leftarrow (H', S, k', j)$;
  if *proposal-stable* **then**
    foreach $op \in S$ *in* $<_D$ *order* **do**
      append op onto H;
    $d \leftarrow k$;
    deliver(H);
    abcast CLOSE-RND(k');
upon *suspect-round-winner*
  abcast CLOSE-RND(k');

upon abdeliver CLOSE-RND(k') from p_j and $P = (*, *, k', *)$
  $P \leftarrow \perp$;
  $Q \leftarrow \perp$;
  $k \leftarrow k' + 1$;
  let H' and S' be such that $P = (H', S', k', h)$;
  $H_{new} \leftarrow H'$;
  foreach $op \in S'$ *in* $<_D$ *order* **do**
    append op onto H_{new};
  $(H, d) \leftarrow$ merge$(H_{new},\ k',\ H,\ d)$;
  deliver(H);

Algorithm 5: Handling of strong operations

wait-consensus	$\triangleq$	$Q \neq \perp$ and $T = TB \setminus \mathcal{D}$ and $\lvert T \rvert > n/2$	*must-propose-new-prefix*	$\triangleq$	$i = \Omega_{\mathcal{D}}$ and $N \setminus H \neq \emptyset$ and $(Q = \perp$ or $H \neq Q)$
suspect-ld	$\triangleq$	$ld \neq \Omega_{\mathcal{D}}$ and last locally submitted weak operation is not in H	*from-round-winner*	$\triangleq$	$(P = \perp$ and $k' = k)$ or $P = (*, *, k', j)$
stop-waiting-consensus	$\triangleq$	$W \neq \emptyset$ and $\neg$ *wait-consensus*	*proposal-stable*	$\triangleq$	$j = i$ and $P = (*, *, k', i)$ and $H' = H$ and $k' = k > d$
suspect-round-winner	$\triangleq$	$P = (*, *, k', j)$ and $j \neq \Omega_{\mathcal{D}}$			

Figure 1: The Aurora algorithm for process p_i.

ner crashes, all close round messages for the round will be abdelivered after m, ensuring consistency with the winner.

Eventually, only the permanent leader sends proposal and close round messages. This ensures that eventual stability is reached. Furthermore, if a majority is present in the system and $\mathcal{D} \in \diamond\mathcal{P}$, eventually *wait-consensus* will be true during ongoing rounds of strong prefixes. This ensures that the leader eventually only adds weak operations between two rounds, ensuring termination of strong operations.

Processing strong operations - Detailed description.

In Algorithm 5, all processes keep two round counters: k stores the last round number of a proposed strong prefix, or the next round number if a prefix has just been delivered for a round; d denotes the highest round number for which a strong prefix has been stored in the local history. A submitted strong operation o is sent to all processes in a *strong request* message. When a process receives such a message, it adds o to the set N containing all strong operations that have been received by the process.

If a process p_i believes to be a leader, it can make a proposal for a round if it has operations in N that have not yet been locally delivered and thus not yet inserted in the local history H. The sequence Q stores the last prefix that was proposed by p_i as a prefix of some new strong operation in the current round. A proposal is done by p_i only if p_i has not yet sent any proposal for the round, so $Q = \perp$,[5] or if a prefix has been proposed by p_i but some weak operations has been added to the local history H in the meanwhile so $H \neq Q$ (*must-propose-new-prefix* predicate). The proposal message contains H and the set $S = N \setminus H$ of new strong operations.

If a new proposal message from the round winner is abdelivered, it is stored in the record P. If the winner decides that a proposal is stable, it stores it in H, delivers it, sends a close round message to all, and updates d. A close round message is also sent by any process that suspects the current round winner to be faulty. Whenever a close round message for the current round is received, the corresponding strong prefix is delivered. Before delivering a strong prefix, this is *merged* in the local history as described in Algorithm 4.

[5]The symbol $\perp$ denotes the value "undefined".

The merge operation gives as result a history containing the strong prefix delivered in the largest round. All remaining weak operations are ordered after this prefix.

Background dissemination and merge.

In order to eventually converge to the same history, processes periodically send push messages to all other processes (Algorithm 4). The push mechanism is not only used to achieve Eventual Consistency. The permanent leader of a run uses push messages to fetch the histories of all processes and to aggregate them in a single consistent history. This is the key to achieve eventual stability. Strong prefix consistency and strong prefix stability are preserved by merges because, by construction, the longest strong prefix stored in a history H for round d is a prefix of the longest strong prefix stored in a history H' for round d' if $d \leq d'$. Causal consistency is preserved because all merged histories preserve it by construction. The merge only reorders operations that are ordered inconsistently in the two input histories. These operations, however, cannot be causally dependent. Inconsistent orderings of operations are eventually propagated to all processes and deterministically ordered using the $<_D$ relation. This is the key to eventual stability and consistency.

6. CONCLUSIONS

In this paper, we have presented Eventual Linearizability and a related problem, Eventual Consensus. We have established that combining Eventual Consensus with Consensus comes at the price of using a stronger failure detector than $\Diamond\mathcal{S}$, which is sufficient for Consensus. Finally, we have presented Aurora, a gracefully-degrading shared object implementation extending Consensus with Eventual Consensus. Aurora uses a failure detector of class $\Diamond\mathcal{P}$ to tell if Consensus will terminate, and one of class $\mathcal{C}$ to detect that Consensus will not terminate.

Acknowledgments

The authors are very grateful for Fred Schneider's critiques that significantly helped shaping the paper. Christian Cachin, Rachid Guerraoui and Rodrigo Rodriguez also helped with useful discussions on the motivations of this paper.

7. REFERENCES

[1] A.S. Aiyer, E. Anderson, X. Li, M.A. Shah and J.J. Wylie, "Consistability: Describing Usually Consistent Systems," *Proc. of Fourth Workshop on Hot Topics in System Dependability*, 2008.

[2] H. Attiya and R. Friedman, "A Correctness Condition for High-Performance Multiprocessors," *Proc. twenty-fourth annual ACM symposium on Theory of computing*, pp. 679–690, 1992.

[3] H. Attiya and J. Welch, *Distributed Computing: Fundamentals, Simulations, and Advanced Topics.* J. Wiley & sons, 2004.

[4] K. Birman, G. Chockler and R. van Renesse, "Toward a Cloud Computing Research Agenda," *ACM SIGACT News*, 40(2), pp. 68-80, Jun. 2009.

[5] T.D. Chandra, V. Hadzilacos and S. Toueg, "The Weakest Failure Detector to Solve Consensus," *Journal of the ACM*, 43(4), pp. 685–722 , Jul. 1996.

[6] T.D. Chandra and S. Toueg, "Unreliable Failure Detectors for Reliable Distributed Systems," *Journal of the ACM*, 43(2), pp. 225–267, Mar. 1996.

[7] F. Chu, "Reducing Ω to $\Diamond\mathcal{W}$," *Information Processing Letters*, 67, pp. 289–193, 1998.

[8] 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. of the VLDB Endowment*, 1(2), pp. 1277–1288, Aug. 2008.

[9] 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," *Proc. of twenty-first ACM SIGOPS Symp. on Operating Systems Principles*, pp. 205–220, 2007.

[10] A. Fekete, D. Gupta, V. Luchangco, N. Lynch and A. Shvartsman, "Eventually-Serializable Data Services," *Proc. of the fifteenth annual ACM Symp. on Principles of Distributed Computing*, pp. 300-309, 1996.

[11] S. Ghemawat, H. Gobioff and S.T. Leung, "The Google File System," *Proc. of the nineteenth ACM Symp. on Operating Systems Principles*, pp. 29–43, 2003.

[12] M. P. Herlihy and J. M. Wing, "Specifying Graceful Degradation in Distributed Systems," *Proc. of the sixth annual ACM Symp. on Principles of Distributed Computing*, pp. 167–177, 1987.

[13] M. P. Herlihy and J. M. Wing, "Linearizability: A Correctness Condition for Concurrent Objects," *ACM Trans. on Programming Languages and Systems* , 12(3), pp. 463–492, Jul. 1990.

[14] P. W. Hutto and M. Ahamad, "Slow Memory: Weakening Consistency to Enhance Concurrency in Distributed Shared Memory," *Proc. of the tenth IEEE Int'l Conf. on Distributed Computing Systems*, 1990.

[15] R. Ladin, B. Liskov, L. Shrira and S. Ghemawat, "Lazy Replication: Exploiting the Semantic of Distributed Services," *ACM Trans. on Computers*, 10(4), pp. 360–391, Nov. 1992.

[16] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Comm. of the ACM*, 21(7), pp. 558–565, Jul. 1978.

[17] L. Lamport, "The Part-time Parliament," *ACM Trans. on Computers*, 16(2), pp. 133–169, May 1998.

[18] L. Lamport, "Generalized Consensus and Paxos," *Microsoft Research TR MSR-TR-2005-33.*

[19] M. Raynal and A. Schiper, "A Suite of Formal Definitions for Consistency Criteria in Distributed Shared Memories," *IRISA TR. 968*, 1995.

[20] Y. Saito and M. Shapiro, "Optimistic Replication," *ACM Computing Surveys*, 37(1), pp. 42–81, Mar. 2005.

[21] M. Serafini, D. Dobre, M. Majuntke, P. Bokor and N. Suri, "Eventually Linearizable Shared Objects," *TR-TUD-DEEDS-02-01-2010*, 2010.

[22] A. Singh, P. Fonseca, P. Kouznetsov, R. Rodrigues and P. Maniatis, "Zeno: Eventually Consistent Byzantine-Fault Tolerance," *Proc. of the sixth USENIX Symp. on Networked Systems Design and Implementation*, pp. 196–184, 2008.

[23] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer and C. H. Hauser, "Managing update conflicts in Bayou, a weakly connected replicated storage system," *Proc. of the fifteenth ACM Symp. on Operating Systems Principles*, pp 172–182, 1995.

[24] F.J. Torres-Rojas, M. Ahamad and M.Raynal, "Timed Consistency for Shared Distributed Objects," *Proc. of the eighteenth annual ACM Symp. on Principles of Distributed Computing*, pp. 163–172, 1999.

[25] W. Vogels, "Eventually consistent," *Comm. of the ACM*, 52(1), pp. 40–44, 2009.

[26] L. Zhou, V. Prabhakaran , V. Ramasubramanian, R. Levin and C.A. Thekkath, "Graceful Degradation via Versions: Specifications and Implementations," *Proc. of the twenty-sixth annual ACM Symp. on Principles of Distributed Computing*, pp. 264–273, 2007.