Eventually Linearizable Shared Objects

Marco Serafini, *Dan Dobre, Matthias Majuntke, Péter Bokor and Neeraj Suri
CS Department, TÜ 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 $S$ 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 $S$ 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 terminate in worst case runs. However, there is a tradeoff between ensuring termination of weak and strong operations when processes have only access to $S$. 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 $P$ exists that guarantees termination of all operations.

Categories and Subject Descriptors

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 set 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 asynchronous message-passing system, consensus can be solved assuming a failure detector of class $S$, 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 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 of the number of faulty processes. To this end, it is acceptable that weak operations violate Linearizability when the system deviates from its “normal” behavior, but only if such violations cease when the anomalies terminate [12, 1]. We call this property Eventual Linearizability.

Shared objects with Eventual Linearizability can be used, for example, in master-worker applications to implement the master. 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 in presence of anomalies, guaranteeing that cabs can always dequeue requests. 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
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 [19]. 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 $\Delta$ 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 graphically 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 consistabilty, 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 transactions taking snapshots of the state of multiple objects are presented in [26].

3. DEFINITIONS

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 \( \text{completed}(H) \) 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 of Eventual Linearizability

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 \( o \), \( \tau(i,o') \subseteq \tau(i,o) \).

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: \( o_1 <_{H,t} o_2 \) if \( o_2 \) 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 \( t \). However, not every well-formed history has a linearization as defined in [13].

Eventual Linearizability is a property of histories that may initially be weakly consistent but that eventually start behaving like in a linearization. We formalize this constraint as follows. 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 \( H \). Note that all well-formed histories having a linearization also have a \( t \)-linearization.

We can now define Eventual Linearizability as follows.

**Eventual Linearizability:** An 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 is nonblocking and satisfies locality.
4. IMPLEMENTATIONS

Eventual Linearizability only requires that operations are linearized eventually. It 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 $D$ 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 $\Omega$ 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 $C$, 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 $\diamond S$ (resp. $\diamond P$) 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 $P_i$ such that:
  - (C1) For any correct process $p_i$, $P_i$ is a prefix of $S(i, t')$ if $t' \leq t$;
  - (C2) $P_i$ is a prefix of $P_{i'}$ if $t \leq t'$;
  - (C3) Every operation $o$ submitted at time $t$ by a correct process is included in $P_{i'}$, 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 $S(j, t)$ for some $t' \geq t$.

This definition of Eventual Consistency is a relaxation of Consensus on sequences of operations [18]. 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 $P_i$ of operations. Property (C1) of prefix consistency says that a common prefix $P_i$ of operations has been delivered by each replica; (C2) constrains 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.

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 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 `write1(0)`
operation after $t$. After this operation is completed, process $p_1$ invokes a \texttt{read()} operation. Prefix consistency allows the consistency layer to delay convergence to a common prefix $P_t$ for an arbitrarily long time. Before completing \texttt{read()}, $p_1$ may thus not distinguish this run from a run where \texttt{write}(0) was never invoked. Therefore, \texttt{read()} 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 \texttt{write}(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 \texttt{write}(0). This implies that the consistency layer of $p_1$ has to deliver \texttt{write}(0) before \texttt{read()} 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 example, nontriviality 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 $\pi$, if $\pi$ is a prefix of $S(i,t)$ and $t' \geq t$ then $\pi$ 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 $\pi_i$ and $\pi_j$, 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 result

In this section we show that even if a $\Diamond S$ 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 \( \pi \cdot s \). Let a submit event for an operation \( w \not\in \pi \) occur at a correct process \( p_i \) before consensus on \( \pi \cdot s \) is reached. Process \( p_i \) cannot know whether consensus will terminate or not, as it accesses only failure detector \( \Diamond S \), but it must deliver weak operations in either case. Therefore, \( p_i \) cannot wait until consensus on \( \pi \cdot s \) is reached before delivering \( w \). \( p_i \) is thus forced to deliver \( w \) before consensus on \( \pi \cdot s \) is reached. When consensus on \( \pi \cdot s \) is reached, eventual stability forbids \( p_i \) to deliver \( \pi \cdot s \) because \( w \) is not in \( \pi \). Therefore, consensus needs to be reached on a new strong prefix \( \varphi \cdot s \) with \( w \in \varphi \). However, a new weak operation \( w' \) may be submitted before consensus on \( \varphi \cdot s \) is reached. This pattern can be repeated forever. As a result, 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 \( \Diamond S \) 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 \( submit(o) \). An output event occurs when a sequence \( \pi \) 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 \( \Diamond S \): (P1) termination of weak operations; (P2) termination of strong operations if \( f < n/2 \); and (P3) Eventual Consensus.

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

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

We first consider the case \( k = 0 \) and define \( \sigma_0 \) as follows. Let \( submit_b(s) \) be the first and only input event of the system. Assume that no process crashes in \( \sigma_0 \). Assume also that no message is received in \( \sigma_0 \) and that all enabled computation events occur. Let \( M_0 \) be set of initial messages sent in \( \sigma_0 \).

It is easy to see that (I) is satisfied in \( \sigma_0 \). 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 \( \sigma_0 \) and that consensus cannot be solved using \( \Diamond S \) in any extension satisfying conditions (i) and (ii) since \( f \geq \lfloor n/2 \rfloor \) (see proof in [6]).

For the inductive step, we now define how \( \sigma_{k+1} \) is constructed for \( k > 0 \) by extending \( \sigma_{k-1} \). Assume that no process crashes in \( \sigma_k \) and that \( \Diamond S \) permanently suspects \( \Pi_M \). Let an event \( submit(w_k) \) occur at a process \( p_i \in \Pi_m \) after \( \sigma_{k-1} \), where \( w_k \) is a weak operation that has never been submitted earlier. Let process \( p_i \) eventually deliver a sequence \( \varphi_k \) at a time \( t_k \) such that \( w_k \in \varphi_k \) and \( \varphi_k \not\subseteq \varphi_i \). Assume that no event occurs at any process in \( \Pi_M \) after \( \sigma_{k-1} \) and before \( t_k \). Assume that all messages in \( M_{k-1} \) sent by processes in \( \Pi_M \) and \( \sigma_{k-1} \) are received by processes in \( \Pi_m \) (resp. \( \Pi_M \)) in \( \sigma_k \) but after \( t_k \). Let all enabled computation events occur. Finally, assume that all messages sent after \( \sigma_{k-1} \) are included in \( \Pi_M \) and are not received in \( \sigma_0 \).

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

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

Consider first the case \( p_d \in \Pi_M \). Let \( \sigma_{E2} \) be an extension of \( \sigma_k \) where \( p_d \) delivers \( \pi \cdot s \cdot d \) and let \( t'_k \) be the time when this delivery occurs. Let all processes in \( \Pi_M \) crash immediately after \( \sigma_k \) and let all the messages sent by processes in \( \Pi_M \)
sent after \( \sigma_{k-1} \) to processes in \( \Pi_m \) be lost. Finally, let \( \diamondsuit \) return \( \Pi_M \) at all processes. From eventual stability and since \( p_i \) has already delivered at time \( t_i' < t_i'' \) a sequence \( \varphi \) such that \( w_k \notin \varphi \) but \( s \notin \varphi \), it follows \( w_k \in \pi \).

We now consider a run \( \sigma_{E2} \) where the same events as in \( \sigma_{E21} \) occur until time \( t_i' \) but no process crashes before \( t_i''. \) All processes in \( \Pi_m \) crash immediately after \( t_i''. \) All messages sent from processes in \( \Pi_m \) to processes in \( \Pi_M \) after \( \sigma_{k-1} \) are lost. Assume that after \( t_i'' \), \( \diamondsuit \) eventually returns \( \Pi_m \) at all processes in \( \Pi_M \). \( p_i \) cannot distinguish \( \sigma_{E21} \) and \( \sigma_{E22} \) until \( t_i'' \) so it delivers \( \pi \) at all \( \sigma_{E22} \) too. As all processes in \( \Pi_M \) are correct, they must eventually deliver a sequence containing \( s \) by (P2). From strong prefix consistency and strong prefix stability, this sequence must have \( \pi \) as prefix with \( w_k \in \pi \).

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

Next, consider the case \( p_i \in \Pi_M \). By assumption, (I) holds so \( p_i \) must deliver \( \pi \) at any time \( s \in \pi \) in \( \sigma_k \). Let \( t_i'' \) be the time when this occurs. Consider an extension \( \sigma_{E24} \) of \( \sigma_k \) 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 \( \pi \) as prefix. By eventual stability, since \( p_i \) has already delivered at time \( t_i'' \) a sequence \( \varphi \) including \( w_k \) and not \( s \), it must hold \( w_k \in \pi \). Before \( t_i'' \), process \( p_i \) cannot distinguish \( \sigma_k \) from a similar run \( \sigma_{E23} \) where \( submit(w_k) \) does not occur. In fact, \( p_i \) does not receive any message before \( t_i'' \) that is causally related with \( submit(w_k) \). At time \( t_i'' \), therefore, \( p_i \) delivers \( \pi \) at all \( \sigma_{E22} \) too, a violation of nontriviality. This ends our proof that \( \sigma_k \) satisfies (I).

The infinite run \( \sigma \) can be built iteratively by extending \( \sigma_k \) as it has been done with \( \sigma_{k-1} \). The resulting run is fair by construction because all messages in \( M_{k-1} \) are delivered in \( \sigma_k \) 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 \( \sigma \). By construction, however, each finite prefix \( \tau \) of \( \sigma \) 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 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 any number of correct processes using \( \diamondsuit \), still ensuring termination of weak operations and Eventual Consistency in worst-case asynchronous runs. The algorithm also shows that causal consistency can be combined with Eventual Consensus.

Failure detectors and communication primitives.

Aurora uses termination of weak operations and Eventual Consistency in asynchronous runs. To this end, Aurora uses a failure detector module \( D \in C \), which outputs the set of indices of the processes that have been suspected to crash. Virtually all failure detector implementations are of class \( C \) 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 \( D \in \diamondsuit \subseteq C \), while for termination of strong operations we assume \( D \in \diamond P \subseteq \diamondsuit \). 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 termination of weak operations in runs where Consensus would not terminate, including asynchronous runs.

The use of \( \diamond P \) to complete strong operations is a consequence of Theorem 4. For simplicity, we use \( \Omega P \) to denote a simulation of a leader election oracle ensuring the properties of \( \Omega \) on top of \( D \) in runs where \( D \in \diamondsuit \) similar to [7]. The simulation ensures that the leader trusted by \( \Omega P \) is not suspected by \( D \). We call the process that is permanently trusted by \( D \) when \( D \in \Omega P \) the permanent leader.

Processes use two communication primitives: a reliable channel providing \textit{send} and \textit{receive} primitives, and a (uniform) FIFO atomic broadcast primitive providing \textit{abcast} and \textit{abdeliver} primitives [3]. Implementing atomic broadcast is equivalent to solving consensus [6]. We consider atomic broadcast implementations that use a failure detector \( \Omega \) 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 \( C \) 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 \( D \in C \). If \( D \in \diamondsuit \), Eventual Consensus is implemented. Termination of strong operations is ensured if \( D \in \diamond P \) or, in absence of concurrent weak operations, if \( D \in \diamondsuit \). All proofs are available in [21].

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 \( \pi \) 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 \( \pi \) 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 \( p_{ld} \).

Aurora uses \textit{trust messages} to let \( p_{ld} \) know which processes trust it. Whenever \( \Omega P \) outputs a new leader \( p_i \) at a process \( p_i \), \( p_i \) sends a TRUST(i) message to all processes through FIFO reliable channels. Each process \( p_i \) keeps a trusted-by \( set TB \) 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 detec-

\footnotetext{1}{We call a stable majority a majority quorum that does not change over time. The weakest failure detector to solve consensus, which is \( \Omega \), requires that eventually all correct processes permanently trust the same correct process \( p_{ld} \). We show in [21] that \( \Omega \) can be simulated if eventually a stable majority of correct processes permanently trusts \( p_{ld} \).}
tor of class $C$ 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 $D$). Even in worst-case runs where $D \in C$, $T$ 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 $D \in \mathcal{P}$. 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.

**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 already 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 $w \not\in \pi_s$ 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 $\pi$ 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 $\pi$ 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 winner 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
Algorithm 3: Handling of weak operations

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

function merge(H', d', H, d)
    dnew = max(d, d');
    if d = dnew then Hnew = longest strong prefix of H;
    else Hnew = longest strong prefix of H';
    W = set of weak operations in (H' \cup H) \ \ Hnew;
    R = order Q according to \(<_H\cup<_{H'}\) and break cycles
    according to \(<_D\);
    append R onto Hnew in R order;
    return (Hnew, dnew);

Algorithm 4: Background dissemination and merge

upon submit (o) and o is weak
    ld ← ΩD;
    send WREQ(H, d, op) to pD;
upon receive WREQ(H', d', op') from j
    if wait-consensus and (H', d', op') \ \ W then
        add (H', d', op') into W;
    else
        (H, d) ← merge(H', d', H, d);
        if op' \ \ H then append op' onto H;
        send PUSH(H, d) to pJ;
upon receive PUSH(H', d')
    (H, d) ← 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') \ W do
        if op' \ \ H then append op' onto H;
    send PUSH(H, d) to pJ;
    remove (H', d', op') from W;

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) ← merge(H', d', H, d);
    add op into N;
upon must-propose-new-prefix
    S ← N \ H;
    Q ← H;
    T ← TB \ D;
    abcast PROP(Q, S, k);
upon abdeliver PROP(H', S, k') from pJ
    if from-round-winner then
        P ← (H', S, k', j);
    if proposal-stable then
        foreach op \ S in \(<_D\) order do
            d ← k;
        deliver(H);
    abcast CLOSE-RND(k');
upon suspect-round-winner
    abcast CLOSE-RND(k');
upon abdeliver CLOSE-RND(k') from pJ and P = (*, *, k', *)
    P ← Ω;
    Q ← Ω;
    k ← k' + 1;
    let H' and S' be such that P = (H', S', k', h);
    Hnew ← H';
    foreach op \ S' in \(<_D\) order do
        append op onto Hnew;
    (H, d) ← merge(Hnew, k', H, d);
    deliver(H);

Algorithm 5: Handling of strong operations

Figure 1: The Aurora algorithm for process pI.

reached. Furthermore, if a majority is present in the system
and D \in \Diamond 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 sub-
mitted 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 pI believes to be a leader, it can make a pro-
sposal 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 pI as a prefix of some new strong operation in
the current round. A proposal is done by pI only if pI has
not yet sent any proposal for the round, so Q = \perp,5 or if
a prefix has been proposed by pI, 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 \ H of new strong
operations.
If a new proposal message from the round winner is ab-
delivered, 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.
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.

5The symbol \perp denotes the value "undefined".
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 \( \Diamond S \), which is sufficient for Consensus. Finally, we have presented Aurora, a gracefully-degrading shared object implementation extending Consensus with Eventual Consensus. Aurora only degrades consistency in periods when Consensus would block. It uses a failure detector of class \( \Diamond P \) to tell if Consensus will terminate, and one of class \( C \) to detect that Consensus will not terminate.

Acknowledgments

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

7. REFERENCES

APPENDIX

In this Appendix we first show the locality and nonblocking properties of Eventual Linearizability (Appendix A). We then show that Eventual Consensus is necessary and sufficient to implement of Eventual Linearizability, while Eventual Consistency is not sufficient (Appendix B). Finally, we show the correctness of the Aurora protocol (Appendix C).

A. LOCALITY AND NONBLOCKING

In this section we show that Eventual Linearizability inherits the most relevant properties of Linearizability as it is both local and nonblocking. Locality ensures that if every object of a system is eventually linearizable, then the system itself is also eventually linearizable. Being nonblocking implies that the specification of Eventual Linearizability does not result in deadlocks, and such that the history obtained from appending all events of the system is nonblocking. We prove that Eventual Linearizability is nonblocking by showing that locally consist and t-linearizability are local properties, which imply the locality of Eventual Linearizability.

Lemma 1. If a history $H$ is weakly consistent then, for each object $x$, $H|x$ is weakly consistent. If $H|x$ is weakly consistent for each object $x$, then $H$ is weakly consistent.

Proof. Given that $H$ is weakly consistent, we know that, for every process $p_i$ and operation $o$ completed by $p_i$ in $H$, there is a legal sequential history $\tau(i,o)$ which fulfills (i)-(iii). If $o$ is an operation of $x$, then $H|x$ and $\tau(i,o)|x$ also fulfill (i)-(iii). Otherwise, $o$ is not invoked in $H|x$. Therefore, $H|x$ is also weakly consistent.

On the other hand, given that $H|x$ is weakly consistent and $\tau(i,o)$ fulfills (i)-(iii) for every process $p_i$ and operation $o$ completed by $p_i$ in $H|x$, $o$ is also completed in $H$ by the same process and $\tau(i,o)$ is a legal sequential history of $H$ too. Therefore, $H$ is also weakly consistent.

Lemma 2. If a history $H$ is t-linearizable then, for each object $x$, $H|x$ is t-linearizable. If $H|x$ is t-linearizable for each object $x$, then $H$ is t-max-linearizable with $t_{max} = \max_{\tau}(t_x)$.

Proof. It is evident from the definitions that if $H$ is $t$-linearizable then $H|x$ is $t$-linearizable for each object $x$. In fact, if $L$ is a $t$-linearization of $H$, then $L|x$ is a $t$-linearization of $H|x$ and all response events in $L|x$ after $t$ have the same results as in $H|x$. Therefore, $H|x$ is $(t,L|x)$-linearizable for each object $x$.

In order to prove the second implication, we assume that for each $x$, $H|x$ is $t_x$-linearizable. Let $R_x$ be the response events added to $H|x$ to build the $t_x$-linearization $L_x$ of $H|x$, and $H'$ the history obtained from appending all events of $R_x$ to $H$. Let $<\ell$ be the total order of all operations in $H|x$ defined by $L_x$, and $<\ell$ be a relation built as the transitive closure of $U_{\ell\in L_e} <\ell \cup <_{H,t_{max}}$. Assuming that $<\ell$ is a partial order, we can build a $t_{max}$-linearization $L$ of $H$ which respects $<\ell$. For each $x$, all operations on $x$ are ordered in $L$ as in $L_x$. This implies that the results of the response events in $L$ are the same as in $L|x$. Since $H|x$ is $(t_x,L|x)$-linearizable, all response events of $H$ after $t_x \leq t_{max}$ have the same results as in $L$, so $H$ is $(t_{max},L)$-linearizable and thus $t_{max}$-linearizable.

We now show that $<\ell$ is a partial order. Assume by contradiction that $o_i <\ell o_j$ and $o_n <\ell o_k$, where $<\ell$ can be either $<_{x}$ for some $x$ or $<_{H,t_{max}}$ and assume that this is a cycle with minimal length in $<\ell$. If all these operations are on the same object $x$, then they are totally ordered by $<_{x}$. The existence of a cycle implies that there must exist two operations $o_i$ and $o_j$ on object $x$ such that $o_i <_{x} o_j$ and $o_j <_{H,t_{max}} o_i$. This contradicts with (P2) as $<_{x}$ is the order of a $t_x$-linearization $L_x$ of $H|x$ and (P2) implies that $<_{H,t_{max}} \subseteq <_{x}$. If and $<_{H,t_{max}} \subseteq <_{x}$, imply that $<_{H,t_{max}} \subseteq <_{x}$, a contradiction.

The cycle must thus contain operations on at least two objects. Assume $o_i$ is an operation on object $x$. Let $o_k$ be an operation in the cycle on a different object than $x$ and such that $o(k+1 \mod n), \ldots, o((j-1 \mod n))$ are on $x$. Similarly, let $o_j$ be an operation in the cycle on the same object than $x$ and such that $o((j+1 \mod n), \ldots, o(j-1 \mod n))$ are on $x$. Since $o_k <_{x} o_j <_{x} o_j$, it follows that $o_k <_{H,t_{max}} a_j <_{H,t_{max}} o_j$, so $o_k <_{H,t_{max}} o_j$. It must thus hold $k \neq j$, which implies that a cycle exists $o_1 \ldots o_k <_{x} o_j \ldots <_{x} o_k$, where the cycle is shorter than the one with minimal length, a contradiction. □

We also prove that Eventual Linearizability is nonblocking by showing that weakly consistency and t-linearizability are nonblocking.

Lemma 3. Let inv be an invocation of a total operation $o$ on an object $x$. If inv on $x$ is invoked by a process $p_i$ in a weakly consistent history $H$, then there exists a response resp on $x$ of $p$ such that the history $H'$ obtained by appending resp to $H$ is weakly consistent.

Proof. Given an operation $o'$ completed by process $p_j$ in the weakly consistent history $H$ (resp. $H'$), the corresponding legal sequential history is denoted by $\tau_{H}(j,o')$ (resp. $\tau_{H'}(j,o')$). Let operation $o'$ be the last completed operation in $H$ invoked by process $p_j$. Then, resp is determined by the execution $\tau_{H}(j,o') \cdot \text{inv} \cdot \text{resp}$. In $H'$, $o$ is a completed operation. We choose $\tau_{H'}(j,o') \cdot \text{inv} \cdot \text{resp}$. For every other operation $o''$ completed by $p_j$ in $H'$, $\tau_{H'}(j,o'')$ equals $\tau_{H}(j,o')$. As a result, $\tau_{H'}(j,o')$ and every $\tau_{H}(j,o')$ fulfill (i)-(iii) because $H'$ is weakly consistent and $<_{H} \subseteq <_{H'}$. □

Lemma 4. Let inv be an invocation of a total operation on an object $x$. If inv on $x$ is invoked by a process $p$ in a $t$-linearizable history $H$, then there exists a response resp on $x$ of $p$ such that the history $H'$ obtained by appending resp to $H$ is $t$-linearizable.

Proof. Let $L$ be a $t$-linearization of $H$. If $L$ includes a response to inv we are done. If not, inv is not included in $L$ since $L$ only contains completed operations. Since the operation is total, there exists a result for a response event resp that is determined by the execution of $L' = L \cdot \text{inv} \cdot \text{resp}$. $L'$ is a $t$-linearization of $H'$. As resp has the same response in $H'$ and $L'$, $H'$ is $t$-linearizable for any value of $t$ such that $H$ is $t$-linearizable. □

Theorem 1. Eventual Linearizability is nonblocking and satisfies locality.

Proof. Directly follows from Lemmas 1, 2, 3 and 4. □
B. EVENTUAL CONSISTENCY, EVENTUAL CONSENSUS AND CONSENSUS

In this Section, we distinguish between high-level events, which are executed on the interface between the application and the execution layer, and low-level events, which are executed on the interface between the execution layer and the consistency layer. Given a run $\sigma$ of the system, we define as $top(\sigma)$ the history containing all high-level events and $bot(\sigma)$ the history containing all low-level events. Eventual Linearizability constraints the set of admissible high-level histories $top(\sigma)$ of a run $\sigma$. The specifications discussed in this Section constrain the low-level histories $bot(\sigma)$ of a run $\sigma$.

The results in this section consider a non-uniform notion of Eventual Linearizability, where operations invoked by faulty processes may never appear in the final $t$-linearization. We focus on non-uniformity for two reasons. The first reason is that this strengthens the impossibility results of this paper. The second reason is that, as it can be derived by using a simple partitioning argument, ensuring a uniform notion of Eventual Linearizability would require the existence of $f + 1$ correct processes to complete weak operations if $f$ replicas can crash. The availability of $f + 1$ correct replicas for completing weak operations is not assumed by most replication algorithms implementing Eventual Consistency [20]. For example, the specification of Eventual Serializability [10], which models the behavior of Lazy replication [15], does not distinguish between operations of correct and faulty processes. However, Lazy replication implements a non-uniform form of Eventual Serializability, where operations observed only by faulty replicas may never appear in the eventual serialization.

Some eventually consistent (or eventually serializable) algorithms ensure that all completed operations appear in the eventual serialization. The Zeno algorithm, for example, satisfies uniformity because it requires clients to contact a quorum of correct replicas in order to complete weak operations. This is needed to prevent clients from returning replies from Byzantine replicas [22]. Dynamo implements uniformity by writing values to "sloppy quorums" that might not intersect with read quorums [9]. If $f$ failures are to be tolerated without replicas, both these algorithms require that at least one quorum of $f + 1$ correct replicas is always available even in worst case runs.

Eventual Consistency is not sufficient to implement Eventual Linearizability for arbitrary objects. In fact, we show in the following Theorem 2 that it is not even sufficient to implement an eventually linearizable register.

Theorem 2. An eventually linearizable implementation of a single-writer, single-reader binary register cannot be simulated using only an eventually consistent consistency layer.

Proof. Consider a system with two processes $p_0$ and $p_1$, where $p_0$ is a writer and $p_1$ is a reader. The register stores an initial value 0. Assume by contradiction that there exists an implementation of a read/write register with Eventual Linearizability using only an eventually consistent consistency layer. Let $t_i$ be the time such that, for all runs $\sigma$ such that $bot(\sigma)$ satisfies Eventual Consistency, $t_i$-linearizability holds for $H = top(\sigma)$. We show the contradiction by using three finite runs. The last of these runs leads to a violation of $t_i$-linearizability.

In the first run $\sigma_0$, process $p_0$ writes the value 1 onto the register after time $t_i$. Let $t_w$ be the time when the write operation completes and $t_0$ be the time when the last event of $\sigma_0$ occurs. Process $p_1$ takes no actions in this run. Let $bot(\sigma_0)$ satisfy Eventual Consistency in this run.

In the second run $\sigma_1$, process $p_1$ invokes a read operation after time $t_w$. Let $t_r$ be the time when the read operation completes and $t_1$ be the time when the last event of $\sigma_1$ occurs. Process $p_0$ takes no action in this run. Since no write operation is invoked in this run, the read must return the initial value 0. Let $bot(\sigma_1)$ satisfy Eventual Consistency in this run too.

In the third run $\sigma_2$, $p_0$ and $p_1$ observe the same events until $t^*_{ij}$, as in $\sigma_0$ and $\sigma_1$, respectively. For indistinguishability, the read operation of $p_1$ returns 0 even if it is preceded by a write operation writing 1. At a time $t_2 > \max(t_0, t_1)$, the consistency layer delivers at both processes the same sequence $S$ including all the operations submitted before $t_2$. These delivery events are the last events of $\sigma_2$.

In every $t_i$-permutation $L$ of $H = top(\sigma_2)$, the write operation precedes the read so the read operation returns 1. This contradicts $t_i$-linearizability of $H$ since the write and read operations are invoked after $t_2$ but the result of the read in $H$ is 0. Therefore, $bot(\sigma_2)$ must violate Eventual Consistency. We show that it is not the case, which leads us to the final contradiction. It is easy to see that if nontriviality, set-stability and liveness hold for $bot(\sigma_0)$ and $bot(\sigma_1)$, then they also hold for $bot(\sigma_2)$. Prefix consistency holds if we define $P_i$ as follows. For $t \leq t_2$, $P_i$ is equal to the empty sequence. For $t > t_2$, $P_i$ is equal to the sequence $S$ delivered at time $t_2$. This definition of $P_i$ satisfies all properties (C1)-(C3) of prefix consistency.

We say that a consistency layer that satisfies Eventual Consensus satisfies $t$-stability if $t$ is the time defined in the definition of Eventual Stability. Combining Eventual Consistency with Eventual Stability implicitly strengthens consistency. Namely, $t$-stability ensures $t$-consistency, which is defined as follows. A consistency layer satisfies $t$-consistency if for any correct processes $p_i$ and $p_j$ delivering at any times $t_i, t_j > t$, one of the sequences $S(i, t_i)$ and $S(j, t_j)$ is prefix of the other.

We first show that Eventual Consensus satisfies $t$-consistency.

Lemma 5. If a consistency layer satisfies $t$-stability then it satisfies $t$-consistency.

Proof. Assume that a consistency layer satisfies Eventual Consistency and eventual stability but contradicts the Lemma. Let $t$ be the time after which stability holds. This implies that two delivery events occur at two processes $p_i$ and $p_j$ at times $t_i, t_j > t$ such that $S(i, t_i)$ and $S(j, t_j)$, which are the two sequences delivered at times $t_i$ and $t_j$, are not prefix of each other.

If $i = j$ a contradiction follows directly eventual stability. We thus consider the case $i \neq j$. There must exists an index $k$ and two different operations $o_i$ and $o_j$ that are the $k$-th elements of $S(i, t_i)$ and $S(j, t_j)$ respectively. It follows from eventual stability that for each $t_i' > t_i$ and $t_j' > o_j$, $o_j$ and $o_j$ that are the $k$-th elements of $S(i, t_i')$ and $S(j, t_j')$ respectively. From property (C3) of prefix consistency, there exists a time $t_{c_i}$ such that $P_{c_i} \subseteq S(i, t_{c_i}')$ with $t_{c_i}' > t_{c_i}$ so $o_i$ is the $k$-th element of $P_{c_i}$. Similarly, from property (C3) and (C1) it follows that there exists a time $t_{c_j}$ such that $P_{c_j} \subseteq S(j, t_{c_j}')$ with $t_{c_j}' > t_{c_j}$ so $o_j$ is the $k$-th element of $P_{c_j}$.
as the $k$-th element. However, $P_{ek}$ and $P_{ej}$ are not prefixes of each other. This violates (C2).

We are now ready to show the equivalence of implementing eventually linearizable arbitrary shared objects and of implementing a consistency layer solving Eventual Consensus.

**Lemma 6.** An eventually linearizable implementation of an arbitrary object can be implemented using only a consistency layer satisfying Eventual Consensus.

**Proof.** Assume that the consistency layer satisfies $t$-stability. From Lemma 5, the Eventual Consistency layer also satisfies $t$-consistency. The algorithm for the implementation is the one of Algorithm 1. High-level invocation events at each process $p_i$ for each operation $o$ are forwarded to the lower consistency layer. The implementation then waits for the first sequence delivered by the consistency layer at process $p_i$ containing $o$. The time when this delivery event takes place is denoted as $t(o)$. The implementation then executes the resulting sequence and returns the results as an upper-layer response event.

It is clear from the liveness of the consistency layer and from Algorithm 1 that each invoked operation is eventually completed. Weak consistency directly derives from the set stability and nontriviality properties of the consistency layer. We now show that $t$-linearizability for some time $t_l$ also holds.

From the prefix consistency (C3) property of the consistency layer, all operations submitted by correct processes are eventually included in a consistent prefix $P_l$. From prefix consistency (C2), consistent prefixes are prefixes of each other. Let $t_p$ be the minimum time such that all operations $o$ such that $t(o) \leq t_p$ are included in $P_{sp}$, and $t_c$ be the minimum time when all faulty processes have crashed. We define $t_l$ to be the minimum time greater than $\max(t_p, t_c)$ and show that the simulation of Algorithm 1 satisfies $t_l$-linearizability.

Assume by contradiction that $t_l$-linearizability is violated. Since $t_l > t_c$, this implies that there exists, for some run $\sigma$, a high-level operation $o_i$ of a correct process $p_i$ in $H = \text{top}(\sigma)$ which is invoked after $t_l$ and whose result is different than the result of $o_i$ in any $t_l$-linearization $L$ of $H$. Assume that there exists a $t_l$-linearization $L$ of $H$ having $S_i = S(i, t(o_i))$ as a prefix. It follows from the implementation of Algorithm 1 that the result of $o_i$ in $L$ is the same as in $H$, a contradiction. Therefore, there exists no such $L$. This implies that for some operation $o_k \in S_i$, there exists an operation $o_j$ such that $o_j <_{H,t_l} o_k$ and $o_j \notin S_i$. This in turns implies that either $o_j \notin S_i$ or $o_k <_{S_i} o_j$. Before contradicting these two cases, note that from $o_j <_{H,t_l} o_k$, the completion of $o_j$ precedes the invocation of $o_k$. From nontriviality, $S_j = S(j, t(o_j))$ cannot include $o_k$.

Assume the first condition holds and that there exists an operation $o_j$ with a prefix $P_{sp}$ such that $o_j <_{H,t_l} o_k$, $o_k \in S_i$ and $o_j \notin S_i$. We consider two cases based on the value of $t(o_j)$. If $t(o_j) \leq t_p$ then $o_j$ is included in $P_{sp}$. From prefix consistency (C1), $P_{sp}$ is a prefix of $S_i$ so $o_j \in S_j$. Therefore, $t(o_j) > t_p$ so it follows from $t_p$-consistency that one of $S_j$ and $S_j = S(j, t(o_j))$ is a prefix of the other. Since $o_j \notin S_i$ but $o_j \in S_j$, $S_j$ is a prefix of $S_j$, so $o_k \in S_j$. However, we have shown that $o_j \notin S_j$.

We now consider the second condition, that is, that there exist two operations $o_j$ and $o_k$ in $S_i$ such that $o_j <_{H,t_l} o_k$ and $o_k <_{S_i} o_j$. We consider two cases. If $t(o_j) > t_s$, it follows from $t_s$-consistency that one of $S_j$ and $S_i$ are a prefix of each other. Since $o_k <_{S_i} o_j$ and $o_j \in S_j$, $o_k <_{S_i} o_j$. However, we have shown that $o_k \notin S_j$. If $t(o_j) \leq t_s$ then $P_{tp}$ includes $o_j$. From nontriviality and since $o_k$ is invoked after $t_l \geq t_p$, there exists no process $h$ such that $S(h, t_p)$ includes $o_k$. This and prefix consistency (C1) imply that $P_{tp}$ does not include $o_k$. Since $t(o_k) > t_s \geq t_p$, $P_{tp}$ is a prefix of $S_k = S(k, t(o_k))$ from prefix consistency (C1). This implies that $o_j <_{S_i} o_k$. However, one of $S_i$ and $S_k$ is prefix of the other, a contradiction with $o_k <_{S_i} o_j$. In fact, from the definition of $<_{H,t_l}$, $o_k$ is invoked after $t_l$. This and $t_s$-consistency imply that one of $S_k$ and $S_i$ are prefixes of the other.

**Lemma 7.** A consistency layer satisfying Eventual Consensus can be implemented using only an eventually linearizable arbitrary object implementation.

**Proof.** We show that the simulation of Algorithm 2 satisfies Eventual Consistency and $t_s$-stability for some time $t_s$. Let $t_l$ be the time such that $t_l$-linearizability holds for all histories, $t_d$ be the time when all operations submitted by invoked processes before $t_l$ are completed at all correct processes, and $t_s$ be the minimum time greater than $\max(t_l, t_d)$. The existence of $t_d$ is given by the liveness of the sequence implementation.

Set stability and nontriviality directly follow from the weak consistency property of the sequence. For liveness, it follows from the termination property of the sequence implementation that all append operations $o$ invoked by a correct process upon a submit($o$) event terminate. From $t_s$-linearizability, all appended operations are read by the first read operation $o'$ invoked by each correct process after $\max(o, t_l)$. All submitted operations are thus appended and eventually delivered by each correct process.

For prefix consistency, let $L$ be the $t_s$-linearization of the operations on the shared object, and let $P_l$ be defined as follows. $P_l$ is the empty sequence for $t \leq t_s$. For $t > t_s$, $P_l$ is the value returned by the last read operation of any correct process which is ordered in $L$ before all reads invoked by correct processes and ongoing at time $t$. Prefix consistency (C1) follows from the fact that every operation returned by a read invoked after $t_d$ is observed by any following read in a $t_s$-linearization. For prefix consistency (C2) it is sufficient to observe that for each $t \geq t_l$ and $t' > t$, either $P_l = P_{l'}$ or the read whose return value defines $P_{l'}$ observes a sequence which is an extension of the sequence observe by the read of $P_l$. In fact, both sequences are prefixes of $L$. Prefix consistency (C3) directly follows from the liveness of the sequence implementation, from the definition of $P_l$ and from the fact that sequences are periodically delivered.

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

**Proof.** The sufficiency of Eventual Consensus is shown by Lemma 6, the necessity is shown by Lemma 7.

C. CORRECTNESS OF THE AURORA PROTOCOL

We start the proof by providing some additional definitions, notations and conventions which will be used in the following correctness argument.
C.1 Definitions

**Time.**

Some proofs refer to a global time reference $t \geq 0$. Computation time is ignored, and the state of a process at time $t$ is the one after any event occurred at $t$. No two events occur at the same process and at the same time, and only a finite number of events occur in a finite time. We say that a message is **received** or **abdelivered** when the corresponding receipt or abdelivery event occurs.

**Sequences and histories.**

We define two sequence of operations to be **compatible** if one of the two is a prefix of the other. A **strong prefix** is a prefix of operations terminating in a strong operations. We abuse the terminology and say that a sequence $S_1$ is a **subset** of another sequence $S_2$ if all operations of $S_1$ are included in $S_2$.

We define $H(i, t)$ as follows: if $p_i$ not crashed at time $t$, then $H(i, t)$ is the local history stored by $p_i$ at time $t$; else, it is the last history stored by $p_i$ before crashing. The order induced on operations by the local history of a process $p_i$ at time $t$ is denoted as $<_i, t$. The order on operations determined by a sequence $S$ is denoted as $<_S$. Local variables and predicates of a process $p_i$ are denoted by a subscript $i$.

A process $p_i$ stores a variable $x = \text{val}$ upon receiving or abdelivering a message $m$ if $x = \text{val}$ in the local state of $p_i$ at the time of the local receipt or abdeliver event of $m$. A process $p_i$ stores an operation at a given time if it includes the operation in its local history $H_i$. A process $p_i$ stores a strong prefix $\pi$ for round $k$ when $p_i$ stores a local history $H_i$ containing the last operation of $\pi$ upon abdelivering a $\text{PROP}(\ast, \ast, k)$, a $\text{PUSH}(\ast, k)$ or a $\text{CLOSE-RND}(k)$ message. A process $p_i$ directly stores a strong prefix $\pi$ for round $k$ when $p_i$ stores $\pi$ for $k$ for the first time upon abdelivering a $\text{PROP}(\ast, \ast, k)$ or a $\text{CLOSE-RND}(k)$ message. Strong prefixes that are indirectly stored by $p_i$ are stored when $p_i$ receives a $\text{PUSH}$ message from some other process. We say that $\pi$ is a longest strong prefix for $p_i$ at a given time $t$ if $\pi$ is a strong prefix of the local history $H(i, t)$ and there exists no strong prefix $\pi'$ of $H(i, t)$ which is longer than $\pi$.

We need sometimes to show that the system converges to a common state after a certain time. Given a process $p_i$ and a finite set of operations $O$, we define $t_w(i, O)$ as the maximum time $t$ in a given run when the following holds: at time $t$ process $p_i$ appends an operation $op = H'$ onto its history or executes a merge $(H', H, \ast)$ such that either (i) there exists $o \in O \cap H'$ such that $o \not\in H(i, t)$, or (ii) there exist $o, o' \in H(i, t) \cap H'$ such that $o <_H o'$ and $o' <_H o$.

**Communication primitives.**

The reliable channel module has the following property: if a correct process $p_i$ sends a message $m$ to a correct process $p_j$, then $p_j$ eventually receives $m$. The atomic broadcast module has four properties: (validity) If a correct process abcasts a message $m$, then it eventually abdelivers $m$; (uniform agreement) If a process abdelivers a message $m$, then all correct processes eventually abdelivers $m$; (uniform integrity) For any message $m$, every process abdelivers $m$ at most once, and only if $m$ was previously abcast by its sender; (total order) If two correct processes $p_i$ and $p_j$ abdeliver two messages $m$ and $m'$, then $p_j$ abdelivers $m$ before $m'$ if and only if $p_j$ abdelivers $m$ before $m'$.

### Failure detectors and the quorum property.

The algorithm uses a failure detector $D$ and a leader oracle $\Omega_D$ implemented on top of $D$. We call a **leader oracle** any failure detector which outputs the id of a single **trusted** process. We say that a correct process $p_{id}$ is **perpetually trusted** at time $t$ if at each time $t' \geq t$ and for each correct process $p_i$, $\Omega_D(p_i) = ld < p_i$. We say that a correct process $p_{id}$ is **perpetually trusted** if it is perpetually trusted at some time $t$.

We say that a leader oracle is in class $\Omega_Q$ is it satisfies the following **quorum property**: there exists a quorum $Q$ of correct processes and a process $p_{id}$ such that eventually all processes in $Q$ perpetually trust $p_{id}$ and $|Q| > n/2$. Clearly, a leader oracle can satisfy this property only if a majority of correct processes exists. If this precondition is met, each leader oracle in $\Omega$ is trivially in $\Omega_Q$. Furthermore, the simple Lemma 8 shows that given a leader oracle in $\Omega_Q$, a leader oracle in $\Omega$ can be simulated using Algorithm 6, which relies on reliable FIFO channels. Therefore, classes $\Omega_Q$ and $\Omega$ are equivalent if a majority of correct processes exists.

**Causal consistency.**

We define causal consistency as follows. We first define the happens-before relation $<_C$ as follows. Let $o$ and $o'$ be two different operations, let $i$ the the process that invoked $o'$, and let $t$ the time when $o'$ is invoked. We say that $o <_C o'$ if and only if $o \in S(i, t')$ for some $t' < t$ or there exists a third different operation $o''$ such that $o <_C o'' <_C o'$. A consistency layer satisfies causal consistency if, for each process $i$ and time $t$ it holds that: (C1) If $o \in S(i, t)$ and $o <_C o$ then $o' \in S(i, t)$, and (C2) If $o, o' \in S(i, t)$ and $o$ precedes $o'$ in $S(i, t)$ then $o <_C o'$. It can be shown that this definition of causal consistency property is sufficient to implement causal memory [14].

### C.2 Correctness proof

**Lemma 8.** Algorithm 6 simulates a leader oracle in $\Omega$ using a leader oracle $\mathcal{L} \in \Omega_Q$. 

```
Initially: ld ← ⊥;
T[j] ← ⊥ for each j ∈ [0, n − 1];
\mathcal{L} outputs ⊥;

// A process calls this function to query its
// local instance of the leader oracle
function query()
if ld ≠ ⊥ then
    return ld;
else
    k ← \mathcal{L};
    return k;
upon \mathcal{L} changes its output to k
send TRUST_FD(k) to all processes;
upon receive TRUST_FD(k) from process $p_j$
T[j] ← k;
if \exists h, Q : T[l] = h for each l ∈ Q and |Q| ≥ n/2
then
    ld ← h;
else
    ld ← ⊥;
```

Algorithm 6: Implementing $\Omega$ on top of $\mathcal{L} \in \Omega_Q$
Proof. The proof is by contradiction. Assume that eventually the leader oracle $L$ in $\Pi_{<}$ permanently outputs the same process id $k$ at a quorum $Q$ of correct processes such that $|Q| > n/2$ and that $p_i$ is not permanently trusted by the local instance of the simulation of some correct process. Each process in $Q$ will eventually send a TRUST-FD($k$) to all other processes as last TRUST-FD message. Since the communication channel is FIFO and reliable, these messages are eventually received by each correct process and are the last messages received from any process in $Q$. This implies that for each correct process, eventually it permanently holds $T[j] = k$ for each $j \in Q$. For each correct process $p_i$, when the last TRUST-FD message from a process in $Q$ is received by $p_i$, $id$ is permanently set to $k$. The simulation thus permanently returns the same process id $k$ to each correct process, a contradiction. \qed

Lemma 9. If a process $p_i$ abcasts a PROP($H'$, $S$, $k$) message $m$, then $H'$ is an extension of a strong prefix $\pi_{k-1}$ stored by $p_i$ for round $k-1$ and $S \setminus H'$ is empty.

Proof. Assume by contradiction that the thesis does not hold. It follows from the predicate must-propose-new-prefix that if $p_i$ abcasts the PROP($H'$, $S$, $k$) message then $H'$ is the local history of $p_i$, $S \setminus H'$ is empty and $k_i = k$. If $p_i$ has already stored a strong prefix $\pi_{k-1}$ for round $k-1$, $H'$ is an extension of $\pi_{k-1}$, a contradiction. So $p_i$ has not yet stored a strong prefix $\pi_{k-1}$. If $p_i$ has set its local variable $k_i$ to $k$ then it has abdelivered a CLOSE-RND($k-1$) message when $P_i = (\ast, \ast, k-1, \ast)$. If $d_i < k-1$ upon abdelivering CLOSE-RND($k-1$), then $p_i$ stored a strong prefix for $\pi_{k-1}$ by doing the following merge, a contradiction. Therefore, $P = (\ast, \ast, k-1, \ast)$ and $d_i >= k-1$. This implies that $p_i$ has already stored a strong prefix for $k-1$ upon abdelivering a PROP($\ast, \ast, k-1$) message or upon receiving a PUSH($\ast, k-1$) message, the final contradiction. \qed

Lemma 10. If a process $p_i$ directly stores a strong prefix for round $k$ then $p_i$ has exactly one strong prefix for round $k-1$ and $d = k-1$ when the strong prefix is stored for round $k$.

Proof. We first show that if $p_i$ directly stores a strong prefix for round $k$ at a certain time $t_k$, it stores $k_i = k$ and $d_i < k$ immediately before $t_k$. Two events can induce $p_i$ to directly store a strong prefix. If $p_i$ stores a strong prefix upon abdelivering a PROP($\ast, \ast, k$) message, then from the definition of proposal-stable it must hold $k_i = k$ and $d_i <= k-1$, so we are done. If the strong prefix is stored upon abdelivering a CLOSE-RND($k$) message $m$, it must hold $P_i = (\ast, \ast, k, \ast)$ and, from the merge, $d_i < k$. From the definition of from-round-winner, $P_i$ was assigned this value only if a PROP($\ast, \ast, k$) message $m'$ is abdelivered before $m$ and thus if $k_i = k > d_i$ at that time. We now only need to show that the values of $k_i$ and $d_i$ are not modified between receiving $m$ and $m'$. This is easy to see for $k_i$. By contradiction, the value of $d_i$ would be set to a value higher than $k-1$ before storing the strong prefix only if a PUSH($\ast, d$) message with $d > k - 1$ is received. In this case $p_i$ would not directly store a strong prefix for round $k$, a contradiction.

We now show that at least one strong prefix has been stored by $p_i$ for round $k-1$ and that $d_i \geq k-1$ immediately before $t_k$. The value of $k_i$ is set to $k$ only upon abdelivering a CLOSE-RND($k-1$) message. When this occurs, a new strong prefix for round $k - 1$ is included in the new history $H_{new}$ built by $p_i$. If this strong prefix is stored by $p_i$ in the subsequent merge, we are done since $d_i$ is set to $k-1$ by the merge and it holds $d_i \geq k-1$ until $t_k$ since $d_i$ monotonically grows. Else, this implies that $p_i$ has already set $d_i = k-1$. This happens only if $p_i$ has abdelivered PROP($\ast, \ast, k-1$) message and has stored a new strong prefix for $k-1$, or if it has received a PUSH($\ast, d$) message with $d = k-1$. In both cases process $p_i$ stores a strong prefix for round $k-1$ and sets $d_i = k-1$, so we are done.

We now only need to show that no other strong prefix is stored for round $k-1$. This follows from the fact that $d_i \geq k-1$ after storing the first prefix for $k-1$ and that $d_i$ monotonically grows. In fact, no following PROP($\ast, \ast, k-1$) message will lead $p_i$ to the delivery of a strong prefix nor will any merge executed upon receiving a PUSH($\ast, k-1$) or a CLOSE-RND($k-1$) message do it. \qed

Lemma 11. The relation $<_i,t$ is a partial order for each process $p_i$ and time $t$.

Proof. Transitivity and reflexivity are trivial because histories are sequences. We now show that the relation is anti-symmetrical, that is, it never induces cycles. Since a history is a sequence, it is sufficient to show that no local history has duplicates. This is trivially true for the initial empty history.

Histories are modified either by appending operations or by merging other histories. Assume by contradiction that an append or a merge creates a duplicate on a history for the first time. Append of weak operations are always preceded by a check that an operation is not already present in the history. Appends of strong operations in a new strong prefix for round $k$ do not create cycles because strong operations are always stored according to a proposal message. From Lemmas 10 and 9, this contains no duplicates. Merging two histories does not create duplicates unless the merged histories have duplicates, and this would imply that some other prior history contains duplicates, a contradiction. \qed

Lemma 12. If before a time $t$ a process $p_i$ abdelivers a message $m_i$ and a process $p_j$ abdelivers a message $m_j$, then some of the two processes abdelivers both $m_i$ and $m_j$ before $t$.

Proof. Assume by contradiction that this would not be the case. This implies that abcast never satisfies uniform agreement and total order in runs where $D \in \Delta S$ and a majority of correct processes is present. In fact, if uniform agreement holds, $p_i$ and $p_j$ will abdeliver $m_i$ and $m_j$ at some time after $t$. Therefore $p_i$ will deliver $m_i$ before $m_j$ and $p_j$ will do the opposite. This represents a violation of total order. \qed

Lemma 13. For each processes $p_i$ and $p_j$, if $p_i$ stores $P_i = P'$ and $k_i = k'$ upon abdelivering a message $m$ and $p_j$ abdelivers $m$ then $p_j$ stores $P_j = P'$ and $k_j = k'$ upon receiving $m$.

Proof. We show this by induction on the delivery order of $m$ at $p_i$. In the base case, all processes $p_i$ have initially the same value of $P_i = \bot$. Let $m'$ be the last message abdelivered by $p_i$ prior to $m$. For the inductive step, if $p_i$ and $p_j$ abdeliver $m'$ they both store $P_i = P_j = P_{prev}$ and $k_i = k_j = k_{prev}$ upon abdelivering $m'$. Assume $p_i$ stores
Lemma 14. For each $k'$, processes $p_i$ and $p_j$ and times $t_i$ and $t_j$, if $p_i$ stores $P_i = (\ast, \ast, k', i)$ at time $t_i$ and $p_j$ stores $P_j = (\ast, \ast, k', l)$ at time $t_j$, then $h = l$.

Proof. By contradiction, assume $h \neq l$ for some times $t_i$ and $t_j$. $p_i$ must have set $P_i = (\ast, \ast, k', h)$ upon abdelivering a PROP($\ast, \ast, k'$) message $m_i$ with $k' = k_i$ before $t_i$ and $p_j$ must have set $P_j = (\ast, \ast, k', l)$ upon abdelivering a PROP($\ast, \ast, k'$) message $m_j$ with $k' = k_j$ before $t_j$. From Lemma 12 some process, assume wlog $p_i$, has received both $m_i$ and $m_j$. Also assume wlog that $m_i$ is abdelivered by $p_j$ before $m_j$ in the total order. From Lemma 13, $p_i$ stores $P_i = P_j = P_0$ upon receiving $m_i$ and has $k_i = k_j = k'$. After this time and before $p_i$ receives $m_j$, $p_i$ must have set $P_i = \bot$ because it has changed the third field of $P_j$. This follows from the definition of from-round-winner. Whenever $P_j$ is set to $\bot$, however, $k_j$ is set to $k_j + 1 = k' + 1$. From predicate from-round-winner, process $p_i$ will thus never set $P_j$ to a value ($\ast, \ast, k', \ast$), a contradiction.

Lemma 15. If a process $p_i$ delivers its local history and stores $P_i = P' = (\ast, \ast, k', i)$ upon abdelivering a message $m$ and a process $p_j$ stores $P_j = (\ast, \ast, k', \ast)$ upon abdelivering $m$ or afterwards, then $P_j = P'$.

Proof. It follows from proposal-stable that is $p_i$ delivers its local history when $P_i = P'$, then $p_i$ does this upon abdelivering a PROP($\ast, \ast, k'$) message $m$ from itself. $d_i$ is set to $k'$ upon the abdelivery of $m$. After this time, $p_i$ only abcasts PROP($\ast, \ast, k_i$) messages with $k_i > d_i = k'$. From Lemma 13, process $p_j$ stores $P_j = P'$ upon abdelivering $m$. After this time, $p_j$ modifies $P_j$ only if it abdelivers a PROP($\ast, \ast, k'$) from $p_i$ but no such message is received after $m$ because of the FIFO property of abcast, or if $p_i$ sets $P_j$ to $\bot$, then but then $p_i$ sets $k_j$ to $k' + 1$ and, by definition of from-round-winner, will never set $P_j$ to ($\ast, \ast, k', \ast$) again.

Lemma 16. If two processes $p_i$ and $p_j$ store longest strong prefixes $\pi_i$ and $\pi_j$ for round $k$, then $\pi_i = \pi_j$ and every $\phi_{k'}$ stored by any process for round $k' < k$ is a prefix of $\pi_i$ and $\pi_j$.

Proof. We show this by induction on $k$. The property trivially holds for $k = 0$ when the strong prefixes of all processes are empty.

For $k > 0$, if by contradiction $p_i$ and $p_j$ would store different strong prefixes $\pi_i$ and $\pi_j$ for round $k$ upon receiving a PUSH message, then some other process would have directly stored those prefixes. Therefore, we reduce the problem to showing the thesis if $p_i$ and $p_j$ directly store $\pi_i$ and $\pi_j$. Assume by contradiction that processes $p_i$ and $p_j$ directly store different strong prefixes $\pi_i$ and $\pi_j$ upon abdelivering PROP($\ast, \ast, k$) or CLOSE-RND($k$) messages $m_i$ and $m_j$. From Lemma 10, $p_i$ and $p_j$ have stored exactly one strong prefix, $\phi_{k-1}$ and $\phi_{j-1}$ respectively $d_i = d_j = k - 1$ upon abdelivering these messages. By induction, $\phi_{k-1} = \phi_{j-1} = \phi_{k-1}$ is the current longest strong prefix stored by both $p_i$ and $p_j$ immediately before abdelivering $m_i$ and $m_j$.

We consider now two different cases. First we assume that at least one of $m_i$ and $m_j$ is a PROP($H, S, k$). Then we consider the case when both $m_i$ and $m_j$ are CLOSE-RND($k$) messages.

If at least one of $p_i$ and $p_j$ say wlog $p_i$, stores $\pi_i$ upon abdelivering a PROP($H', S, k$) message $m_i$, then from proposal-stable this was sent by $p_i$ and, as we have shown, $\phi_{k-1}$ is a prefix of $H'$. $\pi_i$ is then obtained by $p_i$ by appending elements of $S$ to $H'$ in $<_D$ order. Since $\phi_{k-1}$ is a prefix of $H'$, it is also a prefix of $\pi_i$. Let $P = (H', S, k, i)$ the value of $P_i$ stored by $p_i$ when $\pi_i$ is stored.

From proposal-stable, $p_j$ does not store $\pi_j$ before abdelivering $m_j$. Assume by contradiction that $m_j$ precedes $m_i$ in the total order of abcast. $p_j$ would have stored $k_j = k + 1$ upon abdelivering $m_j$. Since $p_i$ abdelivers $m_i$ which follows $m_j$ in the total order, it follows from Lemma 12 that $p_i$ abdelivers $m_i$ before $m_j$. From Lemma 13, $p_j$ would also set $k_j = k + 1$, upon receiving $m'$ and, from proposal-stable, it would thus store a strong prefix for round $k$ upon receiving $m_i$, a contradiction. Therefore, $m_j$ follows $m_i$ in the total order of the abcast.

From Lemma 13, $p_i$ stores $P_i = (H', S, k, i)$ upon abdelivering $m_i$. From Lemma 15, $P_j = P$ upon abdelivering $m_i$. From proposal-stable, $m_i$ can not be a PROP($\ast, \ast, k$) message so it must be a CLOSE-RND($k$). When $m_j$ is abdelivered by $p_j$, $p_j$ builds the same strong prefix $H'_{new} = \pi_i$ as stored by $p_i$, since $P_j = P$. $\pi_i$ is obtained by merging the current local history of $p_j$ with $H_a$. From Lemma 10, $d_j = k - 1$ so $k > d_j$ and the merge returns $\pi_j = \pi_i$. Also from Lemma 10, $\phi_{k-1}$ is a prefix of $\pi_j$ and of $\pi_i$, so the result of the merge is the longest strong prefix stored by $p_j$. This contradiction concludes the proof for the first case.

We now consider the second case where both $p_i$ and $p_j$ store $\pi_i$ and $\pi_j$ upon abdelivering CLOSE-RND($k$) messages $m_i$ and $m_j$. Assume wlog that $m_i$ precedes $m_j$ in the total order of abcast. Let ($H', S, k, h$) be the value of $P_i$ when before $m_i$ is abdelivered. From round-winner, $P_i$ was set to a value ($\ast, \ast, k, h$) for the first time only after $p_i$ abdelivers a PROP($\ast, \ast, k$) message $m'$ from process $p_j$. $m_i$ is the first CLOSE-RND($k$) message abdelivered after $m_i$ in the total order of abcast. If this would not be the case, $p_i$ would have set $k_j > k$ and would not have stored a strong prefix upon abdelivering $m_i$, a contradiction. $p_i$ obtains $H'_{new} = \pi_i$ by appending operations of $S$ onto $H'$ in $<_D$ order. From Lemma 9 and by the induction hypothesis, $\phi_{k-1}$ is a prefix of the local history $H'$ stored by a process $p_k$. This implies that $\phi_{k-1}$ is a prefix of $\pi_i$ so $\pi_i$ is a new longest strong prefix of $p_i$. From Lemma 12 and total order of abcast, $p_j$ also delivers $m'$ before $m_i$ and before $m_j$. From Lemma 13, $p_j$ also sets $P_j$ to ($\ast, \ast, k, h$) for the first time upon abdelivering $m'$. We have already shown that $m_i$ is the first CLOSE-RND($k$) message which is abdelivered after $m'$. From Lemma 13, $p_j$ also stores a strong prefix $\pi_j$ for round $k$ and builds $\pi_j = \pi_i$ upon abdelivering $m_i$. This is the new longest prefix since $\phi_{k-1}$ is a prefix of $\pi_j$. This is the final contradiction.

Lemma 17. For each processes $p_i$ and $p_j$ and times $t_i$ and $t_j$, if $p_i$ is a strong prefix of $H(i, t_i)$ and $p_j$ is a strong prefix of $H(j, t_j)$ then $\pi_i$ and $\pi_j$ are compatible.

Proof. When a process $p_i$ stores a strong prefix for round $k$, it sets $d_i = k$ and stores no other strong prefixes for
Lemma 18. For each process $p_i$ and times $t$ and $t'$, if $t' > t$ and $\pi$ is a strong prefix of $H(i,t)$, then $\pi_j$ is a strong prefix of $H(i,t')$.

Proof. The result directly follows from Lemma 16 if $p_i = p_j$.

Lemma 19. For each times $t$ and $t_i$ and correct processes $p_i$ and $p_j$ and for each operation $op$ submitted by any process before $t$ and included in $H(i,t_i)$, if $t' \geq \text{ord}(t)$ then $op \in H(j,t')$.

Proof. Assume by contradiction that there exists an operation $op$ submitted before $t$ and included in $H(i,t)$ such that $op$ not in $H_j$. Since $op$ not in $H(j,t')$ and $t' \geq \text{ord}(t)$, $p_j$ never includes $op$ into its history by definition of $\text{ord}(t)$.

Assume that $op$ is a weak operation. Since $op$ is stored by $p_j$, $p_j$ eventually sends a PUSH$(H,*)$ message including $op \in H$ to $p_j$. Since $p_i$ and $p_j$ are correct, $p_j$ eventually receives the PUSH message and calculates its new local history as a merge between $H$ and its previous local history. The resulting history contains $op$, a contradiction.

Assume now that $op$ is strong and let $k'$ be the round number where $p_j$ stores the first strong prefix $\pi_i$ including $op$. After storing $\pi_i$, $p_j$ stores $d_i \geq k'$. If $p_j$ stores a strong prefix for round $k'' \geq k'$, it also stores $\pi_i$ from Lemma 16, a contradiction. Therefore, $p_j$ never stores a strong prefix for a round $k'' \geq k'$ and thus never sets $d_i \geq k'$. However, $p_j$ eventually sends a PUSH$(*,d_i)$ message with $d_i \geq k'$ to $p_j$. Since both $p_i$ and $p_j$ are correct, $p_j$ eventually receives the PUSH message. After the subsequent merge, $p_i$ stores $d_j \geq k'$, a contradiction.

Lemma 20. If there exists a time $t_{ld}$ when $p_{ld}$ is perpetually trusted, then for each $t' \geq \text{ord}(\text{ord}(t_{ld}))$ and for each correct process $p_i$, $H(i,t')$ is a subset of $H(Id, t')$.

Proof. We show that $H_i = H(i,t')$ is a subset of $H_{ld} = H(Id, t')$ by contradiction. Assume that there exists an operation $op$ submitted by a process $p_j$ such that $op \in H_i$ and $op \notin H_{ld}$. If $op$ is submitted before $t_{ld}$, thesis follows from $t' \geq \text{ord}(t_{ld})$ and Lemma 19. Therefore, $op$ is submitted after $t_{ld}$.

We distinguish two cases. If $op$ is a weak operation, $p_j$ trusts $p_{ld}$ when $op$ is submitted and sends a WREQ msg only to $p_{ld}$. $p_{ld}$ is the first process to add $op$ to its history and all other processes store $op$ in their history after directly or indirectly merging their history with the one of $p_{ld}$. Therefore, if $op \in H_i$ then $op \in H_{ld}$.

If $op$ is a strong operation, let $k'$ be the round number where $p_j$ stores the first strong prefix $\pi_i$, including $op$. Since $op$ is submitted after $t_{ld}$ and $p_{ld}$ is perpetually trusted, it follows from must-propose-prefix that $p_{ld}$ is the only process which abcasts a PROP$(*,*,k')$ message. This implies that no process $p_j \neq p_{ld}$ but over sets $P_j = (*,*,k',j)$. Therefore, any process $p_j \neq p_{ld}$ that directly stores $\pi_i$ for round $k'$ does it upon abdelivering a CLOSE-RND$(k')$ message $m$. Since $p_{ld}$ is the perpetual leader, no process $p_j \neq p_{ld}$ abcasts a CLOSE-RND$(k')$ message. Therefore $m$ is sent by $p_{ld}$ after having stored $\pi_d$ in its history. From Lemma 16, $t_{ld}$ is equal to $t_d$ and thus includes $op$. Any other process, like $p_n$, which stores $\pi_i$ for round $k'$ does it event after $p_{ld}$. This implies that if $op \in H_i$ then $op \in H$.

Lemma 21. For each time $t$ and $t' \geq t$, if $op \not<_{i,t} op'$, $op$ and $op'$ are not in a strong prefix of $H(i,t)$ or of $H(i,t')$ and $op \not<_{D} op'$, then $op \not<_{i,t} op'$.

Proof. Since operations are never removed from a history and $op \not<_{i,t} op'$, $p_i$ stores $op$ and $op'$ for any time $t' \geq t$. Assume by contradiction that for some time $t'' \geq t$, it performs $op'$ before $op$ for the first time in its local history. The order of two operations is changed in a local history only by making a merge. However, any merged history always keeps $op \not<_{i,t} op'$ as $op \not<_{D} op$ and $op$ and $op'$ are not in a strong prefix of $H(i,t')$.

Lemma 22. For each time $t$, if $p_i$ and $p_j$ are correct processes, $op \not<_{i,t} op'$ and $op \not<_{j,t} op$ and $op'$ are submitted before $t$, and $op$ and $op'$ are not in a strong prefix of $H(i,t_i)$ or $H(j,t_j)$ and $op \not<_{D} op$ in the deterministic order, then $t_i \not< \text{ord}(t)$.

Proof. Assume by contradiction $t_i \geq \text{ord}(t)$. Assume that $p_i$ receives at time $t' \leq \text{ord}(t)$ a PUSH$(H_p,*)$ message sent by $p_j$ at time $t'' \leq t'$. Since both $p_i$ and $p_j$ are in the strong prefix of $H(i,t'')$ or $H_p$, they would also be in a strong prefix of the local history of $p_i$ at time $\text{ord}(t') \geq t'$ from the definition of the merge operation and from Lemma 18. When $m$ is received, $p_i$ merges the $H_p$ in its local history. The resulting history orders $op' \not< as op' \not<_{D} op$ and as $op$ and $op'$ are not in a strong prefix of $H(i,t'')$ or $H_p$. From Lemma 21, $op' \not<_{i,t} op$ for each time $t_i \geq t''$ so also for each time $t_i \geq \text{ord}(t)$, a contradiction.

We now need to show that $p_i$ receives a PUSH$(H_p,*)$ message from $p_j$ with a history containing $op' \not< at a time t'' \leq \text{ord}(t)$. Assume that $p_i$ does not receive any history where $op'$ precedes $op$ before $\text{ord}(t)$. By definition of $\text{ord}(t)$ and since $op$ and $op'$ are both submitted before $t$, $p_i$ never receives a history containing where $op'$ precedes $op$. Since $op' \not<_{i,t} op$, process $p_j$ eventually sends a PUSH$(H_p,*)$ message to $p_i$. From Lemma 21, $H(j,t')$ still orders $op'$ before $op$ and so does $H_p$. Since $p_i$ and $p_j$ are correct, $p_i$ eventually receives $H_p$, a contradiction.

Lemma 23. For each time $t$, if $t_i,t_j \geq \text{ord}(t)$, $p_i$ and $p_j$ are correct processes, $op$ and $op'$ are submitted before $t$ and are not in a strong prefix of $H(i,t_i)$ or $H(j,t_j)$, then it never holds $op \not<_{i,t_i} op'$ and $op' \not<_{j,t_j} op$.

Proof. Assume by contradiction that $op \not<_{i,t_i} op'$ and $op' \not<_{j,t_j} op$. If $op <_{D} op'$ it follows from Lemma 22 that $t_j < \text{ord}(t_i)$, a contradiction. Similarly, if $op' <_{D} op$ then $t_i < \text{ord}(t)$, a contradiction.

Lemma 24. For each time $t$, if $t_i,t_j \geq \text{ord}(\text{ord}(t))$, $p_i$ and $p_j$ are correct processes, $op$ is submitted before $t$ and $op$ and $op'$ are not in a strong prefix of $H(i,t_i)$ or $H(j,t_j)$, then $op' \not<_{j,t_j} op$.

Proof. Assume by contradiction that $op' \not<_{j,t_j} op$. Also, assume that $op'$ is submitted before $\text{ord}(t)$. Since $p_i$ stores $op$ and $op'$, $p_j$ stores $op$ and $op'$ at time $t_j \geq \text{ord}(\text{ord}(t))$ from Lemma 19. This implies that $op \not<_{i,t_i} op'$. Since both $op$ and $op'$ are submitted before $\text{ord}(t)$ and are not in the strong prefix of $H(i,t_i)$ or $H(j,t_j)$, a contradiction follows from Lemma 23.

We now need to show that $op'$ is submitted before $\text{ord}(t)$. $op$ and $op'$ are weak operations because are not included
in a strong prefix. There are two ways for $p_i$ to store $op'$ before $op$. $p_i$ can directly append $op$ after $op'$ in its history or can merge its local history with another history $H$ such that $op' <_H op$ and contained in a PUSH($H, *$) message. In both cases, some process $p_k$ has directly appended $op$ after $op'$. By definition of $ord(t)$ and since $p_k$ stores $op$ these operations were already stored by $p_k$ at time $ord(t)$. Since $op$ is appended by $p_k$ in its local history after $op'$, $op'$ was already stored by $p_k$ at time $ord(t)$. Therefore, $op'$ is submitted before $ord(t)$.

**Lemma 25.** For each pair of operations $op$ and $op'$, times $t_i$ and $t_j$ and correct processes $p_i$ and $p_j$, if there exists a time $td$ when $p_{td}$ is perpetually trusted, $t_i, t_j \geq ord(ord(ord(t_{td})))$ and $op < op'$ are not in a strong prefix of $H(i, t_i)$ or $H(j, t_j)$ and $op' <_{i,t_i} op$ and $op \in H(j, t_j)$ then $op' <_{j,t_j} op$.

**Proof.** If $op$ is submitted before $td$, the result directly follows from Lemma 24. Therefore, $op$ and $op'$ are submitted after $td$.

If $op$ or $op'$ are in a strong prefix, since $p_{td}$ is the only process which trusts itself after $td$ and from $must-propose-new-prefix$, it follow stat $p_{td}$ is the only process which abcasts PROP($H, S, *$) messages with $op$ or $op'$ in $H \cup S$. Else, $p_{td}$ is the first process to establish an order for $op$ and $op'$. In both cases, if a process $p_i$ stores $op$ before $op'$, this is the order established by $p_{td}$. Therefore, each process $p_j$ storing $op$ also lets it precede by $op$ in its local history.

The last remaining case is the one where $op'$ is submitted before $td$ and $op$ is submitted after $td$. From Lemma 19 and the fact that $p_i$ stores $op'$, $p_j$ stores $op$ before $ord(t_{td})$. Also, $p_j$ stores $op$ by hypothesis, so $p_j$ has ordered $op$ and $op'$ at time $t_j$. Assume by contradiction that $op <_{i,t_j} op'$. This and $op' <_{i,t_i} op$ would contradict Lemma 24.

**Lemma 26.** For any pair of operations $op$ and $op'$, times $t'$ and $t''$, and correct process $p_i$, if there exists a time $td$ when $p_{td}$ is perpetually trusted such that $t_i, t_j, t'' \geq ord(ord(ord(t_{td})))$ and $op <_{i,t_i} op'$, then $op' <_{i,t''} op$.

**Proof.** Assume by contradiction that $op' <_{i,t''} op$. If $op$ (resp. $op'$) is strong, a contradiction directly follows from Lemma 17 and $op < _{i,t''} op'$ (resp. $op < _{i,t''} op'$). Therefore, $op$ and $op'$ are weak.

If $op$ and $op'$ are not in a strong prefix, a contradiction follows directly from Lemma 25. Therefore, both operations are in a strong prefix.

Let $k$ be the minimum round number such that $op$ or $op'$ are in a strong prefix $\pi$ of $H(ld, t')$ or $H(i, t'')$. $\pi$ either includes $op$ but not $op'$, or $op'$ but not $op$, else Lemma 17 would be violated by $p_{td}$ or $p_i$. Assume that $\pi$ includes $op'$ but not $op$. The argument in case $\pi$ includes $op$ but not $op'$ is similar and we discuss the main differences below.

Assume that $\pi$ has been submitted before $ord(ord(t_i))$. Since $p_i$ or $p_{td}$ have stored $\pi$, all other correct processes do the same before $ord(ord(ord(t_{td})))$ from Lemma 19. Therefore, $p_{td}$ stores $op'$ before $op$ at time $t' > ord(ord(ord(t_{td})))$ but this is inconsistent with $\pi$, a contradiction of Lemma 18. In case $\pi$ only includes $op'$, a similar contradiction is built with $p_i$.

We now need to show that $\pi$ has been submitted before $ord(ord(t))$. By definition, $\pi$ is built by a process after abdelivering a PROP($H', S', k$) message $m$ from a process $p_k$, and is the result of appending the strong operations of $S'$ onto $H'$. Since $op'$ is weak, $op' \in H'$. Let $t_h$ be the time when $p_h$ sends $m$. Since $op'$ is in $H'$ then $op' \in H(h, t_h)$. By definition of $k$, neither $op$ nor $op'$ are in a strong prefix of $H(h, t_h)$. Assume by contradiction that $t_h \geq ord(ord(t_{td}))$. It follows from this, Lemma 25, $op' \in H(h, t_h)$ and $op <_{td} op'$, $op' < op$. Therefore $H'$, and thus the strong prefix $\pi$ too, would include $op$ and $op'$, a contradiction. In case $\pi$ includes $op$ but not $op'$, a contradiction would follow from Lemma 25 and $op' <_{i,t_i} op$ since $\pi$ would contain $op$ and $op'$. This implies that $t_h < ord(ord(t_{td}))$ so $\pi$ has been submitted before $ord(ord(t_{td}))$.

**Lemma 27.** If there exists a time $td$ when a process $p_{td}$ is trusted by all processes, then there exists a time $t$ such that for each $t' \geq t$ and for each correct process $p_i$, it holds that $H(i, t)$ is a prefix of $H(i, t')$.

**Proof.** Let $H(i, t)$ be the history stored by process $p_i$ at time $t$. By contradiction, assume that $t = ord(ord(ord(t_{td})))$, and let $t_m \geq t$ be the minimum time such that $H = H(i, t)$ is not a prefix of $H_m = H(i, t_m)$.

$H$ is a subset of $H_m$ and $H_m$ is a subset of $H(ld, t_m)$. The first fact follows from the fact that histories are modified by appending operations or by merging that merges return the union of the merged histories. The second follows from Lemma 26. From Lemma 26, both $H$ and $H_m$ order their operations as in $H(ld, t_m)$, so $H$ is a prefix of $H_m$, a contradiction.

**Lemma 28.** If a correct process $p_{td}$ which is eventually permanently trusted by $\Omega_D$ abcasts a PROP($*, *, k$) message and eventually stops modifying $H_{td}$ until $k_{td} > k$, and if $\Omega_D \in \Omega$ and a majority of correct processes exists, then eventually $p_{td}$ sets $k_{td} > k$ and $Q_{td} = \bot$.

**Proof.** The proof is by contradiction. By hypothesis, $p_{td}$ abcasts a PROP($*, *, k$) message. Since $\Omega_D \in \Omega$ and a majority of correct processes exists, abcast terminates. This and the fact that $p_{td}$ is correct implies that some process will be the winner of round $k$ by having its proposal abdelivered.

If $p_{td}$ is the winner of round $k$, it sets $P_{td} = (*, *, k, ld)$ and $Q \neq \bot$. If $p_{td}$ later abdelivers a CLOSE-RND($k$) message, it sets $k_{td} > k$ and $Q_{td} = \bot$, a contradiction. Therefore, $p_{td}$ never abdelivers a CLOSE-RND($k$) message so, from validity of abcast, $p_{td}$ never abscasts such a message. This implies that $\Omega_D$ at $p_{td}$ always outputs $ld$. From $must-propose-new-prefix$, $p_{td}$ keeps sending proposal messages whenever its local history is modified. From validity of abcast, process $p_{td}$ abdelivers all the proposal messages that it abcasts. By hypothesis, $p_{td}$ eventually stops adding operations to its local history $H_{td}$ during round $k$. Therefore, process $p_{td}$ will eventually abdeliver a PROP($H', *$, $k$) message sent from itself with $H' = H_{td}$. It will therefore abcast a CLOSE-RND($k$) message, a contradiction.

If $p_j \neq p_{td}$ is the winner of round $k$, $p_{td}$ sets $P_{td} = (*, *, k, j)$. It is sufficient to show that $p_{td}$ abcasts or abdelivers a CLOSE-RND($k$) message to reach a contradiction like in the previous case. Therefore $p_{td}$ never abdelivers a CLOSE-RND($k$) message from the winner $p_j$. This implies that eventually $suspect-round-winner_{td}$ will hold since $p_{td}$ is the only process which is permanently trusted by $\Omega_D$. Therefore, $p_{td}$ will abcast a CLOSE-RND($k$) message, a contradiction.

**Lemma 29.** If a process $p_i$ stores a new history $H_n$ by merging its local history $H$ and another history $H'$ and both
H and H' satisfy properties (C1) and (C2) of causal consistency, then Hₙ satisfies (C1) and (C2).

Proof. It is trivial that Hₙ satisfies (C1) since Hₙ is the union of H and H'. For (C2), let M be the result of the merge and assume by contradiction that o <ₜ o'. Since M stores o', one of H and H', say H, stores o'. From (C1), H stores o too. From (C2), o <ₜ o'. Assume that o and o' are not in a strong prefix π of H or H'. Both o and o' are therefore weak operations. From the merge procedure it follows that if o <ₜ o and o <ₜ o' then o' <ₜ o. H' thus violates (C2), a contradiction.

We now show that o and o' are not in a strong prefix π of H or H'. Assume by contradiction that they are. From Lemmas 17 and 18 and the fact that M is stored by a process as new strong prefix, π is a prefix of M. If o ∈ π then either o' ∈ π or o <ₜ o' since (C1) and (C2) are not violated in π. For the same reason, if o' is in π then o <ₜ o'. In all these cases, since π is a prefix of M then o' ∉ₜ o, a contradiction. □

Theorem 5. Causal consistency is satisfied.

Proof. Assume that a process pᵢ is the first process to violate (C1) or (C2) at time t. A process violated these properties only when it modifies its local history. If pᵢ appends an operation it has submitted to its local history, a contradiction directly follows from the fact that the prior local history satisfies (C1) and (C2).

If pᵢ violates (C1) or (C2) upon receiving a PUSH or ORD message m at time t, the new history of pᵢ is the merge between the old history of pᵢ and the history contained in the message. Both merged histories are local histories of processes at a time preceding t so they satisfy (C1) and (C2). A contradiction follows from Lemma 29.

We now consider the case when pᵢ violates (C1) or (C2) upon receiving a WREQ(H, o) or SREQ(H, o) message m at time t. pᵢ merges its history with H and, similar to the previous case, the result satisfies (C1) and (C2). Also, H contains all operations o' such that o' <ₜ o. Appending o to the new local history of pᵢ preserves (C1) and (C2).

The last case is that pᵢ violates (C1) or (C2) upon abdelivering a PROP or CLOSE-RND message. If the local history of pᵢ is modified upon receiving these messages, then pᵢ stores a new strong prefix for round k and sets Pᵢ = (H, S, k, h). Let Hₙ be the result of appending all operations of S onto H in a deterministic order. Since the previous local history of pᵢ satisfies (C1) and (C2), it is sufficient from Lemma 29 to show that Hₙ satisfies these properties.

If pᵢ has set Pᵢ = (H, S, k, h) then a process pᵢ has abcast a PROP(H, S, k) message. For each strong operation o ∈ S, pᵢ has received from the proposer processes histories including all operations o' such that o' <ₜ o. H is the local history of pᵢ has merged all these histories and, from Lemma 29, satisfies (C1) and (C2) and includes all operations causally dependent on operations in S. From Lemmas 11 and 16, all the operations in S has not yet been stored by any other process for any other round. This implies that none of the operations of S is causally dependent on each other, so Hₙ satisfies (C1) and (C2). □

Theorem 6. Nontriviality, set stability, strong prefix stability, prefix consistency, strong prefix consistency are always satisfied.

Proof. We prove that all properties of Eventual Consistency are met. For each process pᵢ and time t, the properties of S(i, t) are shown for local histories H(i, t). Since only the content of local histories is never delivered, and since local histories are delivered whenever they are modified, this is equivalent to show the properties for delivered sequences.

Nontriviality: It trivial from the algorithm and from Lemma 11.

Set stability: Directly follows from the fact that histories are modified either by appending operations or from merges. The latter operation returns the union of the merged histories, so no operation is removed from a history.

Strong prefix stability: Directly follows from Lemma 18.

Strong prefix consistency: Directly follows from Lemma 17.

Prefix consistency: We define Pᵢ as follows. For each operation op stored by a correct process, let t(op) be the time when op is submitted and p(op) the first correct process storing op. Pᵢ includes all operations stored by a correct process such that t ≥ ord(ord(t(op))). From strong prefix consistency and strong prefix stability, the longest strong prefix of Pᵢ is a prefix of H(i, tᵢ) for each i and tᵢ ≥ t. From Lemma 25, the prefix preceding each remaining operation of Pᵢ in H(i, tᵢ) is equal at each correct process pᵢ at time tᵢ ≥ t since t = (ord(ord(t(op)))) so Pᵢ is a prefix of H(i, tᵢ) with tᵢ ≥ t.

(C2) can be shown easily because, from (C1), Pᵢ and Pᵢ are both prefixes of H(i, tᵢ) for each i. Also, each operation of Pᵢ is included in Pᵢ by definition since t ≤ tᵢ. Therefore, Pᵢ is a prefix of Pᵢ.

As for (C3), it follows from Liveness that all operations invoked by a correct process are eventually stored by all other processes. From Lemma 19, all operations stored by a correct process are eventually stored by each correct process, so all operations stored by a correct process are included in some Pᵢ for some t.

□

Theorem 7. Eventual Stability is satisfied if D ⊆ S.

Proof. Eventual stability after for some t follows from Lemma 27.

□

Theorem 8. Each weak operation w submitted by a correct process is eventually stored by each correct process in its local history.

Proof. Assume a correct process pᵢ submits a weak operation w and some correct process pⱼ never adds it to its history. Let ld be value of Ωᵢ when the submit event occurs. The operation w is reliably sent to pᵢ in a WREQ message m.

If pᵢ is suspected by Ωᵢ, pᵢ appends w to its local history. Eventually pᵢ sends a PUSH(H, d) message with w in H. Since pᵢ and pⱼ are both correct, the PUSH message is eventually delivered. pⱼ then either adds w to its history or w is already in its history. Therefore, since by contradiction pⱼ never delivers w, Ωᵢ never suspects pᵢ. By the strong completeness of D, this implies that pᵢ is correct. The WREQ message m is thus eventually delivered by pᵢ.

If wait-consensusₙᵢ is false when m is received by pᵢ, or it is true, and thus w is included in Wᵢₙᵢ, but it eventually becomes false, and thus stop-waiting-consensusₙᵢ holds, pᵢₙᵢ
merges the history contained in \( m \) with its own, and the resulting \( H_{ld} \) contains \( w \). After this merge, \( p_{ld} \) eventually sends a PUSH message containing \( w \) to all correct processes, which eventually receive it and store it in their local history, a contradiction. Therefore, \( \text{wait-consensus} \) is always true. Therefore, it always holds that \( Q_d \neq \perp \) and that \( T_{ld} \) is a majority quorum equal to the current set \( TS_{id} \setminus D \).

If a majority of correct processes does not exist, then eventually \( |D_{ld}| \geq \lceil n/2 \rceil \) for strong completeness so \( |TS_{ld} \setminus D| < \lceil n/2 \rceil \), a contradiction. Therefore, there exists a majority of correct processes. From \( Q_{ld} \neq \perp \), \( p_{ld} \) has sent a \( \text{PROP}(*, *, k) \) message for some \( k = k_{ld} \). If \( \Omega_{\text{P}} \in \Omega \), it follows from Lemma 28 that eventually \( Q_{ld} = \perp \) and thus \( \text{wait-consensus}_{ld} \) stops holding, a contradiction. Therefore, \( \Omega_{\text{P}} \notin \Omega \).

Since \( \text{wait-consensus} \) always holds, it always holds that \( |T_{ld}| > n/2 \) and \( T_{ld} = TS_{ld} \setminus D \). From the strong completeness of \( D \), \( TS_{ld} \setminus D \) eventually only includes the ids of correct processes. Since \( T_{ld} = TS_{ld} \setminus D \) holds forever, \( T_{ld} \) contains the indexes of a majority of correct processes which permanently trust \( p_{ld} \). Therefore, \( \Omega_{\text{P}} \) satisfies the quorum property since \( \Omega_{\text{P}} \subset \Omega_{\text{Q}} \). Since there exists a majority of correct processes, \( \Omega \) and \( \Omega_{\text{P}} \) are equivalent from Lemma 8. This implies that \( \Omega_{\text{P}} \subset \Omega \), a contradiction.

**Theorem 9.** If a correct process \( p_i \) submits a strong operation \( s \), there exists a majority of correct processes, and either \( D \in \diamond P \) or \( D \in \diamond S \) and eventually no weak operation is submitted, then each correct process eventually stores \( s \) in their history.

**Proof.** Assume by contradiction that a correct process \( p_i \) submits a strong operation \( s \) and there exists a correct process \( p_j \) which never stores \( s \) in its history.

Let \( p_{ld} \) be the correct leader which is eventually perpetually trusted by \( \Omega_{\text{P}} \). If \( p_i \) or \( p_{ld} \) ever store \( s \) in their local history, we show a contradiction. Let \( k' \) be the round when \( p_i \) or \( p_{ld} \) first store a strong prefix \( \pi \) including \( s \). Each other correct process \( p_j \) will eventually receive a PUSH(\( H, d \)) message from \( p_i \) or \( p_{ld} \) with \( s \in H \) and \( d \geq k' \). By Lemma 16, if \( p_j \) never stores \( \pi \) then it never stores a strong prefix for round \( k' \) so \( d_j < k' \). When the PUSH message is received then the result of the merge has \( \pi \) as strong prefix, a contradiction.

Neither \( p_i \) nor \( p_{ld} \) thus ever store \( s \) in their local history. After \( s \) is submitted, \( p_i \) sends SREQ(\( *, *, s \)) to all processes. Since \( p_i \) and \( p_{ld} \) are correct, this message is eventually received by \( p_{ld} \). When this happens, \( p_{ld} \) adds \( s \) to \( N_{ld} \). However \( s \) is never added in \( H_{ld} \) of \( p_{ld} \) by contradiction. This implies that \( s \) is always in \( N_{ld} \setminus H_{ld} \).

Let \( k_s \) be the current value of \( k_{ld} \) when \( s \) is received by \( p_{ld} \). For each value \( k \geq k_s \) of \( k_{ld} \), eventually \( p_{ld} \) either sets \( k_{ld} = k + 1 \), and thus \( Q_{ld} = \perp \) too, or it abcasts a \( \text{PROP}(*, *, k) \) message with \( s \in S \). This follows by simple induction on the value of \( k_{ld} \) since \( N_{ld} \setminus H_{ld} \) always includes \( s \), since \( p_{ld} \) eventually trusts itself permanently, and from \( \text{must-propose-new-prefix} \). Assume that eventually \( p_{ld} \) sets \( k_{ld} = k + 1 \) in the both the aforementioned cases. Since \( p_{ld} \) is the permanent leader, it follows from \( \text{must-propose-new-prefix} \) that there exists a round \( k' \) such that \( p_{ld} \) is the only process abcasting a proposal messages \( \text{PROP}(*, S, k') \) for \( k' \). Furthermore, abcast terminates since \( D \in \diamond S \) implies that \( \Omega_{\text{P}} \subset \Omega \) and since a majority of correct processes exists. Since \( p_{ld} \) is correct, if follows from validity of abcast that it abdelivers it proposal message and, since this is the only proposal for \( k' \), that \( p_{ld} \) is the winner of round \( k' \). Since eventually \( k_{ld} = k' + 1 \), this implies that \( p_{ld} \) eventually stores its proposed strong prefix for round \( k' \). This strong prefix includes \( s \), a contradiction.

We now show that if \( p_{ld} \) abcasts a \( \text{PROP}(*, *, k) \) message then eventually \( k_{ld} = k + 1 \). This would follow from Lemma 28 if \( p_{ld} \) would eventually stop modifying its local history \( H_{ld} \) until \( k_{ld} = k + 1 \). Assume by contradiction that \( p_{ld} \) modifies \( H_{ld} \) infinitely often and that \( k_{ld} \) is always equal to \( k \). A contradiction is easy to see if eventually no weak operation is submitted. Therefore, it must hold that \( D \in \diamond P \) and that a majority of correct processes exists. In this case, infinitely many weak operations are received by \( p_{ld} \) and inserted in \( H_{ld} \). From \( \text{must-propose-new-prefix} \), this implies that the leader abcasts infinitely many \( \text{PROP}(*, *, k) \) messages since \( Q \neq \perp \) after sending the first \( \text{PROP}(*, *, k) \) message. However, since \( p_{ld} \) is perpetually trusted, eventually every correct process sends a \( \text{TRUST}(p_{ld}) \) message to \( p_{ld} \) as last trust message. This implies that the trust set \( TS_{ld} \) of \( p_{ld} \) eventually does not change any longer. Also, it follows from \( D \in \diamond P \) that eventually \( D \) outputs exactly the ids of the faulty processes, so eventually \( TS_{ld} \setminus D_{ld} > n/2 \) holds forever and \( D \) stops changing. Whenever a new proposal message is abcast by \( p_{ld} \), \( T_{ld} \) is set to be equal to \( TS_{ld} \setminus D \), so eventually \( T_{ld} \) is equal to \( TS_{ld} \setminus D \) forever. Therefore, eventually \( \text{wait-consensus}_{ld} \) holds forever and \( p_{ld} \) stops modifying \( H_{ld} \), a contradiction. \( \square \)