Brief Announcement: Eventually Consistent Linearizability*

Maciej Kokociński Tadeusz Kobus Paweł T. Wojciechowski
Institute of Computing Science
Poznan University of Technology
{Maciej.Kokocinski, Tadeusz.Kobus, Pawel.T.Wojciechowski}@cs.put.edu.pl

ABSTRACT

Eventually consistent linearizability (ec-linearizability) is a new correctness condition for eventually consistent distributed systems (modeled as shared objects). Unlike the existing definitions of eventual consistency, ec-linearizability is suitable for describing the behaviour of some popular eventually consistent systems, such as Cassandra. It is because ec-linearizability allows for certain types of phenomena, such as lost updates. Similarly to linearizability, ec-linearizability is a safety property and is both local and nonblocking. Thus, ec-linearizability is a property that is both easy to use and reason about.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming; D.2.4 [**Software Engineering**]: Software/Program Verification

Keywords

Eventual consistency; correctness condition; linearizability

1. INTRODUCTION

Modern distributed applications often trade strong consistency for higher performance and increased availability. Development of such applications is difficult (and thus costly), because the programmer has to account for various artifacts caused by reading stale (inconsistent) data. Usually it is assumed that inconsistencies can occur only temporarily and over time servers running the distributed application converge to a consistent state. Hence, such an approach is often described as eventually consistent.

Surprisingly, relatively little attention has been devoted to formally describing the way in which such systems function and the guarantees they offer (for a broader discussion see Section 2). Some authors provide only a very vague specification of the system's behaviour [1] [14]. The more formal ones lack the description of the system properties from the client's point of view ([13] [5] [3] among others), or assume stronger guarantees than some popular eventually consistent systems, such as Cassandra ([4], [12]). Alternatively, some definitions assume that the system becomes strongly consistent at some point in the future [11] [7]. It means that systems that satisfy these properties are as difficult to implement, as if they were strongly consistent.

In this paper, we propose *eventually consistent linearizability (ec-linearizability)*, a new correctness condition for distributed systems (modeled as shared objects). Ec-linearizability relaxes linearizability [8] by allowing for a certain amount of inconsistency between operation executions on the same object. However, ec-linearizability retains the two fundamental properties of linearizability: locality and nonblocking. The former requires that the system is ec-linearizable if and only if every object is ec-linearizable. The latter states that every ec-linearizable history of some ec-linearizable object has an extension that is also ec-linearizable. Ec-linearizability is non-empty as well as prefix- and limit-closed. Therefore, it is a safety property [9].

Ec-linearizability is defined using a more basic property, called Δ-*ec-linearizability*. Roughly speaking, Δ corresponds to the level of inconsistency that each operation can encounter. For $\Delta = 0$, Δ-ec-linearizability reduces to linearizability.

2. RELATED WORK

Below we give a succinct survey on the literature relevant to our work. There is no single definition of *eventual consistency* that the community agrees upon. Informal definitions state that in a system that is eventually consistent, updates to the system's state are not necessarily immediately visible to subsequent read operations. On the other hand, if no new update operations are issued, eventually all read operations will return a value produced by the most recent update operation [1] [14] [2].

This formulation of eventual consistency is to some extent reflected by *eventual linearizability*, first defined in [11] and later refined in [7]. Eventual linearizability models intermittent inconsistencies (as in self-stabilizing systems) that may occur in a system only up to a certain point in time (measured in terms of elapsed time or the number of events from the beginning of computation). As noted by the authors, providing eventual linearizability is as difficult as providing linearizability itself, because, from some moment on, all pro-

*The project was funded from National Science Centre funds granted by decision No. DEC-2012/07/B/ST6/01230.

PODC'15, July 21–23, 2015, Donostia-San Sebastián, Spain.
ACM 978-1-4503-3617-8 /15/07.
http://dx.doi.org/10.1145/2767386.2767448.

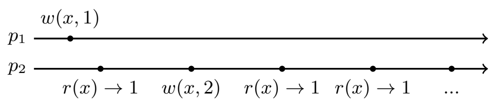


Figure 1: An example execution history in Cassandra with a lost update (p_1's clock drifts into future).

cesses have to work in a fully coordinated fashion (no further inconsistencies are allowed).

Contrary to eventual linearizability, ec-linearizability does not require the system to exhibit strong consistency at any point during the (possibly infinite) execution. Instead, ec-linearizability makes some weak assumptions on the history of operations already submitted to the system. In this sense, ec-linearizability is similar to approaches described below.

In some definitions, such as those in [13] [5] [3] [10] [4], processes in a system are guaranteed to eventually agree on some common prefix of (updating) operations. It means that the system *converges* over time to a consistent state. In some cases, as in [13], [3] or [4], all operations executed by any process have to be eventually visible to all operations that arrive after some point in the future. However, such a requirement can be deemed too strong. In fact, Cassandra, a popular eventually consistent replicated data storage system, allows some updates to be lost (and thus not visible) under certain circumstances (see Figure 1). Moreover, ensuring such a guarantee is not possible for some operations. For example, consider two concurrent *pop* operations executed on a stack. The stack has an invariant: no element can be returned when the stack is empty. If the stack has only one element and partition occurs, both *pop* operations will return the same last element. During convergence, the invariant will be broken, leading to a corrupt -1 sized stack. Inevitably, the next *push* operation performed on it will be lost (the pushed value will never be visible to any other operation). In principle, eventual-visibility of every operation is easy to ensure with updating operations that do not return a value or operations that commute [4] [12].

Ec-linearizability maintains a common prefix of operations but allows some operations to be omitted from it (and thus, it allows to model lost updates). Additionally, ec-linearizability is a correctness condition that, similarly to linearizability [8], concentrates on the interaction between the clients (processes) and the system, and intends to be as general as possible. In this way, it significantly differs from definitions in [13] [5] [3] [10], which put an emphasis on the properties of the systems' internal state, not on the correctness from the clients' point of view.

3. THE DEFINITION

We reuse the notation and basic definitions (such as completion of a history, legal history, etc.) from [6], but we only consider deterministic and finitely nondeterministic shared objects. Otherwise, neither linearizability, nor ec-linearizability, is a safety property.

For any complete sequential history S, S' is a *complete prefix* (or a *complete subsequence*) of S, if S' is a prefix (or a subsequence) of S, such that it contains an invocation event of some operation op_k, if and only if it contains the response event of op_k. We use S^k to denote a prefix consisting of the first k operation executions in S, and $S^{k,k+l}$ to denote a contiguous subsequence consisting of l consecutive operation executions in S starting from the k-th one. Let $fluctus(S)$ be a function that returns the set of all possible complete subsequences of any permutation of operation executions in S, e.g., $fluctus(\langle op_1, op_2\rangle) = \{\langle\rangle, \langle op_1\rangle, \langle op_2\rangle, \langle op_1, op_2\rangle, \langle op_2, op_1\rangle\}$, where $op_k = \langle inv[op_k], resp[op_k]\rangle$. Additionally, by $S_1 \cdot S_2$ we denote the concatenation of S_1 and S_2.

We are now ready to present our correctness condition. We define ec-linearizability using Δ-ec-linearizability. This way we ensure that the size of the inconsistency window, denoted Δ, is unknown but finite.

Definition 1. A history H is Δ*-ec-linearizable* for some natural number Δ, if:

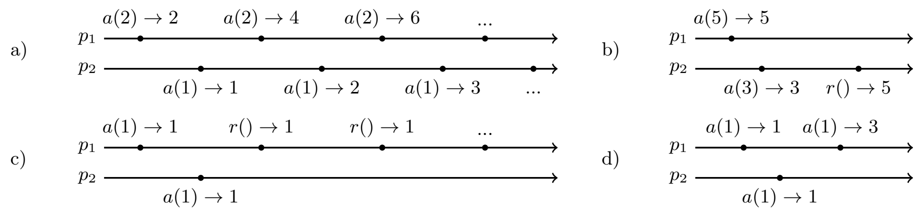
1. there exists a sequential history $S = \langle op_1, op_2, ...\rangle$ equivalent to some completion of H, such that $\to_H \subseteq \to_S$, and
2. there exists L, a legal complete subsequence of S, such that for every operation execution op_k in S, there exists a legal history $V = P \cdot F \cdot \langle op_k\rangle$, such that:
   - P is a complete prefix of L and P is a subsequence of $S^{k-1-\Delta}$ (or P is empty, if $\Delta \geq k-1$),
   - $F \in fluctus(S^{k-\Delta,k})$,
   - if $k > \Delta$, then there exists an operation op_l in F, such that op_l is in L, or op_k is in L.

Definition 2. A history is *ec-linearizable* if it is Δ-ec-linearizable for some Δ.

Δ-ec-linearizability defines a *view* V for every operation in a given history. The view V represents the subjective perspective (from the operation's point of view) on the operations processed by the system. More precisely, V is composed of two parts: the common prefix P and the *fluctus* part F. P is a prefix of the sequence L, which consists of operations that are agreed upon. Since L is shared between the views of operations (each operation observes a prefix of L), the amount of inconsistencies each operation observes is limited. The inconsistencies observed by the operations are modeled by the fluctus part F. It may consist of up to Δ operation executions that directly precede the considered operation in S. These operations may appear in F in any order (not consistent with the order they appear in S).

Note that our definition assumes that some operations may never be included in L. This way, we model the possibility that some conflicting operations are rolled-back or overwritten. More precisely, an operation may never appear in L even though it completed correctly and returned results to the issuing process. It can, however, appear in fluctus parts of views of the next Δ operations processed by the system. This trait reflects the *update semantics* of eventually consistent systems that ec-linearizability models. In such systems, when there are two operations performed on independent sites, they are executed locally, and their updates are propagated asynchronously. Whenever any process receives updates to the same object coming from two different sites, it may try to merge them or, if the updates do not commute (conflict), it may discard (or overwrite) one of them (the winning update is chosen deterministically).

Δ-ec-linearizability ensures progress by requiring that at least one operation from the fluctus part or the operation itself is in L. Without this requirement, satisfying Δ-ec-linearizability would be trivial: an implementation that never



history	ecl	wc	tl	el	bec
a	0	1	0	0	0
b	1	0	0	0	0
c	1	1	0	0	0
d	0	1	1	1	1

Figure 2: Incomparability of properties.

conducts any communication between processes would suffice. It is therefore imperative that L increases over time. Moreover, it is guaranteed that for every operation in L, only a finite number of operations can be excluded from L. In particular, given $\Delta + 1$ conflicting operations, at least one of them has to be recognized by all processes.

It is easy to show that ec-linearizability ensures eventual consistency in its most basic formulation: "if writes stop, then, processes will—at some point in time and forever afterwards—all agree on the same value for each object in the system."[2] Consider a history H. Let op_k be the last update operation in H included in L. If H is infinite, then there exists a finite Δ, such that the view of every operation in H, invoked Δ events after the response of op_k, contains op_k as the last update operation.

4. RESULTS

In this section, we present a few results regarding ec-linearizability. We begin with some easy observations. Firstly, a history that is Δ-ec-linearizable is also Δ'-ec-linearizable for some $\Delta' > \Delta$. Larger Δ means that we require even weaker consistency for the history. Secondly, Δ-ec-linearizability reduces to linearizability for $\Delta = 0$. In this case, the size of inconsistency window is zero, and the progress condition enforces every operation to be included in L. Additionally, as in case of linearizablilty, ec-linearizability requires that S respects the real-time order of H.

Now we compare ec-linearizability (ecl) with weak-consistency (wc), t-linearizablity for some t (tl), eventual linearizability (el) [7] and basic eventual consistency (bec) [4]. We show that ec-linearizability is incomparable with the rest of the properties by producing a few simple histories, which are shown in Figure 2. In each history two processes p_1 and p_2 execute operations on the same shared object that holds a single integer register and exports two operations: $a(u) \to v$ which adds u to the register and returns its current value, and $r() \to v$ which returns the current value of the register.

The infinite history a is trivially weakly consistent, because each process observes only all its previous operations. This history is not ec-linearizable, because processes never agree on a common prefix of operations (Δ has to be finite). On the other hand, in history b the opposite is true. From p_2's point of view, operation $a(5) \to 5$ overrides operation $a(3) \to 3$. It is an example of a lost update history acceptable by ec-linearizability, but not a weakly consistent one (p_2 does not recognize its own update operation).

History c also contains a lost update. It is not t-linearizable for any t, because p_1 never recognizes p_2's operation. For the same reason history c is neither eventually linearizable, nor (basically) eventually consistent. History d is not ec-linearizable, because there is no legal view of operation $a(1) \to 3$. However, it is t-linearizable for $t = 2$. It is also eventually linearizable and (basically) eventually consistent.

It is possible to show that ec-linearizability is both prefix- and limit-closed. It is also non-empty, therefore it is a safety property (unlike eventual linearizability) [9]. Two fundamental properties of linearizability are *locality* and *nonblocking* [8]. Both are preserved by ec-linearizability, what makes it an easy to reason about correctness condition.

5. REFERENCES

[1] P. Bailis and A. Ghodsi. Eventual consistency today: Limitations, extensions, and beyond. *Queue*, 11(3), Mar. 2013.

[2] P. Bailis, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Bolt-on causal consistency. In *Proc. of SIGMOD '13*, June 2013.

[3] A. Bouajjani, C. Enea, and J. Hamza. Verifying eventual consistency of optimistic replication systems. In *Proc. of POPL '14*, Jan. 2014.

[4] S. Burckhardt, A. Gotsman, and H. Yang. Understanding eventual consistency. Technical Report MSR-TR-2013-39, March 2013.

[5] A. Fekete, D. Gupta, V. Luchangco, N. Lynch, and A. Shvartsman. Eventually-serializable data services. In *Proc. of PODC '96*, May 1996.

[6] R. Guerraoui and E. Ruppert. Linearizability is not always a safety property. In *Proc. of NETYS '14*. May 2014.

[7] R. Guerraoui and E. Ruppert. A paradox of eventual linearizability in shared memory. In *Proc. of PODC '14*, July 2014.

[8] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM TOPLAS*, 12(3), 1990.

[9] N. A. Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers Inc., 1996.

[10] Y. Saito and M. Shapiro. Optimistic replication. *ACM Comput. Surv.*, 37(1), Mar. 2005.

[11] M. Serafini, D. Dobre, M. Majuntke, P. Bokor, and N. Suri. Eventually linearizable shared objects. In *Proc. of PODC '10*, July 2010.

[12] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. In *Proc. of SSS '11*, May 2011.

[13] D. Terry, M. Theimer, K. Petersen, A. Demers, M. Spreitzer, and C. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proc. of SOSP '95*, Dec. 1995.

[14] W. Vogels. Eventually consistent. *Commun. ACM*, 52(1), Jan. 2009.