Update Consistency for Wait-free Concurrent Objects

Matthieu Perrin, Achour Mostefaoui, Claude Jard

To cite this version:

Matthieu Perrin, Achour Mostefaoui, Claude Jard. Update Consistency for Wait-free Concurrent Objects. IEEE. IPDPS - IEEE International Parallel & Distributed Processing Symposium, May 2015, Hyderabad, India. 2015, Proceedings of the 29th IEEE International Parallel & Distributed Processing Symposium. <hal-01101657>

HAL Id: hal-01101657

https://hal.archives-ouvertes.fr/hal-01101657

Submitted on 9 Jan 2015

Update Consistency for Wait-free Concurrent Objects

Matthieu Perrin, Achour Mostefaoui, and Claude Jard
LINA – University of Nantes, Nantes, France
Email: [firstname.lastname]@univ-nantes.fr

***Abstract*—In large scale systems such as the Internet, replicating data is an essential feature in order to provide availability and fault-tolerance. Attiya and Welch proved that using strong consistency criteria such as atomicity is costly as each operation may need an execution time linear with the latency of the communication network. Weaker consistency criteria like causal consistency and PRAM consistency do not ensure convergence. The different replicas are not guaranteed to converge towards a unique state. Eventual consistency guarantees that all replicas eventually converge when the participants stop updating. However, it fails to fully specify the semantics of the operations on shared objects and requires additional non-intuitive and error-prone distributed specification techniques.**

This paper introduces and formalizes a new consistency criterion, called *update consistency*, that requires the state of a replicated object to be consistent with a linearization of all the updates. In other words, whereas atomicity imposes a linearization of all of the operations, this criterion imposes this only on updates. Consequently some read operations may return out-dated values. Update consistency is stronger than eventual consistency, so we can replace eventually consistent objects with update consistent ones in any program. Finally, we prove that update consistency is universal, in the sense that any object can be implemented under this criterion in a distributed system where any number of nodes may crash.

***Keywords*-Abstract Data Types; Consistency Criteria; Eventual Consistency; Replicated Object; Sequential Consistency; Shared Set; Update Consistency;**

I. Introduction

Reliability of large scale systems is a big challenge when building massive distributed applications over the Internet. At this scale, data replication is essential to ensure availability and fault-tolerance. In a perfect world, distributed objects should behave as if there is a unique physical shared object that evolves following the atomic operations issued by the participants[1]. This is the aim of strong consistency criteria such as linearizability and sequential consistency. These criteria serialize all the operations so that they look as if they happened sequentially, but they are costly to implement in message-passing systems. If one considers a distributed implementation of a shared register, the worst-case response time must be proportional to the latency of the network either for the reads or for the writes to be sequentially consistent [1] and for all the operations for linearizability [2]. This generalizes to many objects [2]. Moreover, the availability of the shared object cannot be ensured in asynchronous systems where more than a minority of the processes of a system may crash [3]. In large modern distributed systems such as Amazon's cloud, partitions do occur between data centers, as well as inside data centers [4]. Moreover, it is economically unacceptable to sacrifice availability. The only solution is then to provide weaker consistency criteria. Several weak consistency criteria have been considered for modeling shared memory such as PRAM [1] or causality [5]. They expect the local histories observed by each process to be plausible, regardless of the other processes. However, these criteria do not impose that the data eventually converges to a consistent state. Eventual consistency [4] is another weak consistency criterion which requires that when all the processes stop updating then all replicas eventually converge to the same state.

This paper follows the long quest of the (a) strongest consistency criterion (there may exist several incomparable criteria) implementable for different types of objects in an asynchronous system where all but one process may crash (wait-free systems [6]). A contribution of this paper consists in proving that weak consistency criteria such as eventual consistency and causal consistency cannot be combined is such systems. This paper chooses to explore the enforcement of eventual consistency. The relevance of eventual consistency has been illustrated many times. It is used in practice in many large scale applications such as Amazon's Dynamo highly available key-value store [7]. It has been widely studied and many algorithms have been proposed to implement eventually consistent shared object. Conflict-free replicated data types (CRDT) [8] give sufficient conditions on the specification of objects so that they can be implemented. More specifically, if all the updates made on the object commute or if the reachable states of the object form a semi-lattice then the object has an eventually consistent implementation [8]. Unfortunately, many useful objects are not CRDTs.

[1] We use indifferently participant or process to designate the computing entities that invoke the distributed object.

The limitations of eventual consistency led to the study of stronger criteria such as strong eventual consistency [9]. Indeed, eventual consistency requires the convergence towards a *common state* without specifying which states are legal. In order to prove the correctness of a program, it is necessary to fully specify which behaviors are accepted for an object. The meaning of an operation often depends on the context in which it is executed. The notion of *intention* is widely used to specify collaborative editing [10], [11]. The intention of an operation not only depends on the operation and the state on which it is done, but also on the intentions of the concurrent operations. In another solution [12], it is claimed that, it is sufficient to specify what the concurrent execution of all pairs of non-commutative operations should give (e.g. an error state). This result, acceptable for the shared set, cannot be extended to other more complicated objects. In this case, any partial order of updates can lead to a different result. This approach was formalized in [13], where the concurrent specification of an object is defined as a function of partially ordered sets of updates to a consistent state leading to specifications as complicated as the implementations themselves. Moreover, a concurrent specification of an object uses the notion of *concurrent events*. In message-passing systems, two events are concurrent if they are produced by different processes and each process produced its event before it received the notification message from the other process. In other words, the notion of concurrency depends on the implementation of an object not on its specification. Consequently, the final user may not know if two events are concurrent without explicitly tracking the underlying messages. A specification should be independent of the system on which it is implemented.

Contributions of the paper: for not restricting this work to a given data structure, this paper first defines a class of data types called UQ-ADT for *update-query abstract data type*. This class encompasses all data structures where an operation either modifies the state of the object (update) or returns a function on the current state of the object (query). This class excludes data types such as a stack where the pop operation removes the top of the stack and returns it (update and query at the same time). However, such operations can always be separated into a query and an update (lookup_top and delete_top in the case of the stack) which is not a problem as, in weak consistency models, it is impossible to ensure atomicity anyway. This paper has three main contributions.

- It proves that in a wait-free asynchronous system, it is not possible to implement eventual and causal consistency for all UQ-ADTs.
- It introduces *update consistency*, a new consistency criterion stronger than eventual consistency and for which the converging state must be consistent with a linearization of the updates.
- Finally, it proves that for any UQ-ADT object with a sequential specification there exists an update consistent implementation by providing a generic construction.

The remainder of this paper is organized as follows. Section II formalizes the notion of consistency criteria and the type of objects we target in this paper. Section III recalls the definition of (strong) eventual consistency. Section IV proves that eventual consistency cannot be combined with causal consistency in wait-free systems. Section V introduces (strong) update consistency and compares it with (strong) eventual consistency. Section VI compares, through the example of the set, the expressiveness of strong update consistency and strong eventual consistency. Section VII presents a generic construction for any UQ-ADT object with a sequential specification. Finally, Section VIII concludes the paper.

II. Abstract Data Types and Consistency Criteria

Before introducing the new consistency criterion, this section formalizes the notion of object and how a consistency criterion is defined. In distributed systems, sharing objects is a way to abstract message-passing communication between processes. The abstract type of these objects has a sequential specification, defined in this paper by a transition system that characterizes the sequential histories allowed for this object. However, shared objects are implemented in a distributed system using replication and the events of the distributed history generated by the execution of a distributed program is a partial order [14]. The consistency criterion makes the link between the sequential specification of an object and a distributed execution that invokes it. This is done by characterizing the partially ordered histories of the distributed program that are acceptable. The formalization used in this paper is explained with more details in [15].

An abstract data type is specified using a transition system very close to Mealy machines [16] except that infinite transition systems are allowed as many objects have an unbounded specification. As stated in the Introduction, this paper focuses on "update-query" objects. On the one hand, the updates have a side-effect that usually affects the state of the object (hence all processes), but return no value. They correspond to transitions between abstract states in the transition system. On the other hand, the queries are read-only operations. They produce an output that depends on the state of the object. Consequently, the input alphabet of the transition system is separated into two classes of operations (updates and queries).

Definition 1 (Update-query abstract data type)**.** An update-query abstract data type (UQ-ADT) is a tuple $O = (U, Q_i, Q_o, S, s_0, T, G)$ such that:

- U is a countable set of *update* operations;
- Q_i and Q_o are countable sets called *input* and *output* alphabets; $Q = Q_i \times Q_o$ is the set of *query* operations. A query operation $(q_i, q_o) \in Q$ is denoted q_i/q_o (query q_i returns value q_o).
- S is a countable set of *states;*
- $s_0 \in S$ is the *initial state;*
- $T : S \times U \to S$ is the *transition function;*
- $G : S \times Q_i \to Q_o$ is the *output function.*

A sequential history is a sequence of operations. An infinite sequence of operations $(w_i)_{i \in \mathbb{N}} \in (U \cup Q)^\omega$ is recognized by O if there exists an infinite sequence of states $(s_i)_{i \geq 1} \in S^\omega$ (note that s_0 is the initial state) such that for all $i \in \mathbb{N}$, $T(s_i, w_i) = s_{i+1}$ if $w_i \in U$ or $s_i = s_{i+1}$ and $G(s_i, q_i) = q_o$ if $w_i = q_i/q_o \in Q$. The set of all infinite sequences recognized by O and their finite prefixes is denoted by $L(O)$. Said differently, $L(O)$ is the set of all the sequential histories allowed for O.

Along the paper, replicated sets are used as the key example. Three kinds of operations are possible: two update operation by element, namely insertion (I) and deletion (D) and a query operation read (R) that returns the values that belong to the set. Let *Val* be the support of the replicated set (it contains the values that can be inserted/deleted). At the beginning, the set is empty and when an element is inserted, it becomes present until it is deleted. More formally, it corresponds to the UQ-ADT given in Example 1.

Example 1 (Specification of the set)**.** Let *Val* be a countable set, called support. The set object $\mathcal{S}_{Val}$ is the UQ-ADT $(U, Q_i, Q_o, S, \emptyset, T, G)$ with:

- $U = \{\mathrm{I}(v), \mathrm{D}(v) : v \in \mathit{Val}\}$;
- $Q_i = \{\mathrm{R}\}$, and $Q_o = S = \mathcal{P}_{<\infty}(\mathit{Val})$ contain all the finite subsets of *Val*;
- for all $s \in S$ and $v \in \mathit{Val}$, $G(s, \mathrm{R}) = s$, $T(s, \mathrm{I}(v)) = s \cup \{v\}$ and $T(s, \mathrm{D}(v)) = s \setminus \{v\}$.

The set U of updates is the set of all insertions and deletions of any value of *Val*. The set of queries Q_i contains a unique operation R, a read operation with no parameter. A read operation may return any value in Q_o, the set of all finite subsets of *Val*. The set S of the possible states is the same as the set of possible returned values Q_o as the read query returns the content of the set object. $I(v)$ (resp. $D(v)$) with $v \in \mathit{Val}$ denotes an insertion (resp. a deletion) operation of the value v into the set object. R/s denotes a read operation that returns the set s representing the content of the set.

During an execution, the participants invoke an object instance of an abstract data type using the associated operations (queries and updates). This execution produces a set of partially ordered events labelled by the operations of the abstract data type. This representation of a distributed history is generic enough to model a large number of distributed systems. For example, in the case of communicating sequential processes, an event a precedes an event b in the *program order* if they are executed by the same process in that sequential order. It is also possible to model more complex modern systems in which new threads are created and destroyed dynamically, or peer-to-peer systems where peers may join and leave.

Definition 2 (Distributed History)**.** A distributed history is a tuple $H = (U, Q, E, \Lambda, \mapsto)$:

- U and Q are disjoint countable sets of *update* and *query* operations, and all queries $q \in Q$ are in the form $q = q_i/q_o$;
- E is a countable set of *events;*
- $\Lambda : E \to U \cup Q$ is a *labelling function;*
- $\mapsto \subset E \times E$ is a partial order called *program order*, such that for all $e \in E$, $\{e' \in E : e' \mapsto e\}$ is finite.

Let $H = (U, Q, E, \Lambda, \mapsto)$ be a history. The sets $U_H = \{e \in E : \Lambda(e) \in U\}$ and $Q_H = \{e \in E : \Lambda(e) \in Q\}$ denote its sets of update and query events respectively. We also define some projections on the histories. The first one allows to withdraw some events: for $F \subset E$, $H_F = (U, Q, F, \Lambda|_F, \mapsto \cap (F \times F))$ is the history that contains only the events of F. The second one allows to substitute the order relation: if $\to$ is a partial order that respects the definition of a program order ($\mapsto$), $H^{\to} = (U, Q, E, \Lambda, \to \cap (E \times E))$ is the history in which the events are ordered by $\to$. Note that the projections commute, which allows the notation $H_F^{\to}$.

Definition 3 (Linearizations)**.** Let $H = (U, Q, E, \Lambda, \mapsto)$ be a distributed history. A linearization of H corresponds to a sequential history that contains the same events as H in an order consistent with the program order. More precisely, it is a word $\Lambda(e_0) \dots \Lambda(e_n) \dots$ such that $\{e_0, \dots, e_n, \dots\} = E$ and for all i and j, if $i < j$, $e_j \not\mapsto e_i$. We denote by $\mathrm{lin}(H)$ the set of all linearizations of H.

Definition 4 (Consistency criterion)**.** A consistency criterion C characterizes which histories are allowed for a given data type. It is a function C that associates with any UQ-ADT O, a set of distributed histories $C(O)$. A shared object (instance of an UQ-ADT O) is C-consistent if all the histories it allows are in $C(O)$.

III. Eventual Consistency

In this section, we recall the definitions of eventual consistency [4] and strong eventual consistency [9]. Fig. 1 illustrates these two consistency criteria on small examples. In the remaining of this article, we consider an UQ-ADT $O = (U, Q_i, Q_o, S, s_0, T, G)$ and a history $H = (U, Q, E, \Lambda, \mapsto)$.

Eventual consistency: eventual consistency requires that, if all the participants stop updating, all the replicas eventually converge to the same state. In other word, H is eventually consistent if it contains an infinite number of updates (i.e. the participants never stop writing) or if there exists a state (the consistent state) compatible with all but a finite number of queries.

Definition 5 (Eventual consistency)**.** A history H is eventually consistent (EC) if U_H is infinite or there exists a state $s \in S$ such that the set of queries that return non consistent values while in the state s, $\{q_i/q_o \in Q_H : G(s, q_i) \neq q_o\}$, is finite.

All the histories presented in Fig. 1 are eventually consistent. The executions represent two processes sharing a set of integers. In Fig. 1a, the first process inserts value 1 and then reads twice the set and gets respectively $\{2\}$ and $\{1\}$; afterwards, it executes an infinity of read operations that return the empty set (ω in superscript denotes the operation is executed an infinity of times). In the meantime, the second process inserts a 2 then reads the set an infinity of times. It gets respectively $\{1\}$ and $\{2\}$ the two first times, and empty set an infinity of times. Both processes converge to the same state ($\emptyset$), so the history is eventually consistent. However, before converging, the processes can read anything a finite but unbounded number of times.

Strong eventual consistency: strong eventual consistency requires that two replicas of the same object converge as soon as they have received the same updates. The problem with that definition is that the notions of replica and message reception are inherent to the implementation, and are hidden from the programmer that uses the object, so they should not be used in its specification. A visibility relation is introduced to model the notion of message delivery. This relation is not an order since it is not required to be transitive.

Definition 6 (Strong eventual consistency)**.** A history H is strong eventually consistent (SEC) if there exists an acyclic and reflexive relation $\xrightarrow{vis}$ (called *visibility* relation) that contains $\mapsto$ and such that:

- Eventual delivery: when an update is viewed by a replica, it is eventually viewed by all replicas, so there can be at most a finite number of operations that do not view it:
  $\forall u \in U_H, \{e \in E, u \not\xrightarrow{vis} e\}$ is finite;
- Growth: if an event has been viewed once by a process, it will remain visible forever:
  $\forall e, e', e'' \in E, (e \xrightarrow{vis} e' \wedge e' \mapsto e'') \Rightarrow (e \xrightarrow{vis} e'')$;
- Strong convergence: if two query operations view the same past of updates V, they can be issued in the same state s: $\forall V \subset U_H, \exists s \in S, \forall q_i/q_o \in Q_H$, $V = \{u \in U_H : u \xrightarrow{vis} q_i/q_o\} \Rightarrow G(s, q_i) = q_o$.

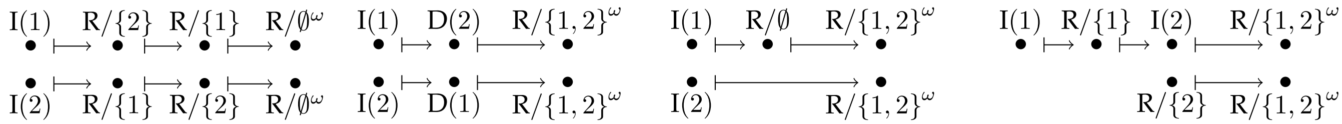
The history of Fig. 1a is not strong eventually consistent because the $\mathrm{I}(1)$ must be visible by all the queries of the first process (by reflexivity and growth), so there are only two possible sets of visible updates ($\{\mathrm{I}(1)\}$ and$\{\mathrm{I}(1), \mathrm{I}(2)\}$) for these events, but the queries are done in three different states ($\{1\}$, $\{2\}$ and $\emptyset$); consequently, at least two of these queries see the same set of updates and thus need to return the same value. Fig. 1c, on the contrary, is strong eventually consistent: the replicas that see $\{\mathrm{I}(1)\}$ are in state $\emptyset$ and those that see $\{\mathrm{I}(1), \mathrm{I}(2)\}$ are in state $\{1, 2\}$.

IV. Pipelined Convergence

A straightforward way to strengthen eventual consistency is to compose it with another consistency criterion that imposes restrictions on the values that can be returned by a read operation. Causality is often cited as a possible candidate to play this role [10]. As causal consistency is well formalized only for memory, we will instead consider Pipelined Random Access Memory (PRAM) [1], a weaker consistency criterion. As the name suggests, PRAM was initially defined for memory. However, it can be easily extended to all UQ-ADTs. Let's call this new consistency criterion *pipelined consistency (PC)*. In a pipelined consistent computation, each process must have a consistent view of its local history with all the updates of the computation. More formally, it corresponds to Def. 7. Pipelined consistency is local to each process, as different processes can see concurrent updates in a different order.

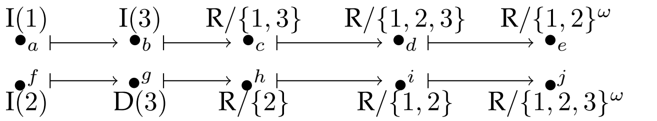
Definition 7. A history H is *pipelined consistent* (PC) if, for all maximal chains (i.e. sets of totally ordered events) p of H, $\mathrm{lin}\left(H_{U_H \cup p}\right) \cap L(O) \neq \emptyset$.

Pipelined consistency can be implemented at a very low cost in wait-free systems. Indeed, it only requires FIFO reception. However, it does not imply convergence. For example, the history given in Figure 2 is pipelined consistent but not eventually consistent. In this history, two processes p_1 and p_2 share a set of integers. Process p_1 first inserts 1 and then 3 in the set and then reads the set forever. Meanwhile, process p_2 inserts 2, deletes 3 and reads the set forever. The words w_1 and w_2 are correct linearizations for both processes, with regard to Definition 7 so the history is pipelined consistent, but after stabilization, p_2 sees the element 3 whereas p_1 does not.



(a) EC but not SEC nor UC (b) SEC but not UC (c) SEC and UC but not SUC (d) SUC but not PC

Figure 1: Four histories for an instance of $\mathcal{S}_{\mathbb{N}}$ (cf. example 1), with different consistency criteria. The arrows represent the program order, and an event labeled ω is repeated an infinite number of times.


I(1) I(3) R/{1,3} R/{1,2,3} R/{1,2}^ω
a b c d e
f g h i j
I(2) D(3) R/{2} R/{1,2} R/{1,2,3}^ω

$$w_1 = \mathrm{I}(1)\cdot\mathrm{I}(3)\cdot\mathrm{R}/\{1,3\}\cdot\mathrm{I}(2)\cdot\mathrm{R}/\{1,2,3\}\cdot\mathrm{D}(3)\cdot\mathrm{R}/\{1,2\}^{\omega}$$

$$w_2 = \mathrm{I}(2)\cdot\mathrm{D}(3)\cdot\mathrm{R}/\{2\}\cdot\mathrm{I}(1)\cdot\mathrm{R}/\{1,2\}\cdot\mathrm{I}(3)\cdot\mathrm{R}/\{1,2,3\}^{\omega}$$

Figure 2: PC but not EC

Proposition 1 (Implementation)**.** *Pipelined convergence, that imposes both pipelined consistency and eventual consistency, cannot be implemented in a wait-free system.*

Proof: We consider the same program as in Figure 2, and we suppose the shared set is pipelined convergent. By the same argument as developed in [2], it is not possible to prevent the processes from not seeing each other's first update at their first reads. Indeed, if p_1 did not receive any message from process p_2, it is impossible for p_1 to make the difference between the case where p_2 crashed before sending any message and the case where all its messages were delayed. To achieve availability, p_1 must compute the return value based solely on its local knowledge, so it returns $\{1,3\}$. Similarly, p_2 returns $\{2\}$. To circumvent this impossibility, it is necessary to make synchrony assumption on the system (e.g. bounds on transmission delays) or to assume the correctness of a majority of processes.

If the first read of p_1 returns $\{1,3\}$, as the set is pipelined consistent, there must exist a linearization for p_1 that contains all the updates, $\mathrm{R}/\{1,3\}$ and an infinity of queries. As $2 \notin \{1,3\}$, the possible linearizations are defined by the ω-regular language $\mathrm{I}(1)\cdot\mathrm{I}(3)\cdot\mathrm{R}/\{1,3\}^{+}\cdot\mathrm{I}(2)\cdot\mathrm{R}/\{1,2,3\}^{\star}\cdot\mathrm{D}(3)\cdot\mathrm{R}/\{1,2\}^{\omega}$, so any history must contain an infinity of events labelled $\mathrm{R}/\{1,2\}^{\omega}$. Similarly, if p_2 starts by reading $\{2\}$, it will eventually read $\{1,2,3\}$ an infinity of times. This implies that pipelined convergence cannot be provided in wait-free systems. ■

Consequently causal consistency, that is stronger than pipelined consistency, cannot be satisfied together with eventual consistency in a wait-free system.

V. Update Consistency

In this section, we introduce two new consistency criteria: update consistency and strong update consistency[2], and we compare them to eventual consistency and strong eventual consistency. Fig. 1 illustrates these four consistency criteria on four small examples.

Update consistency: eventual consistency and strong eventual consistency are not interested in defining the states that are reached during the histories (the same updates have to lead to the same state whatever is the state). They do not depend on the sequential specification of the object, so they give very little constraints on the histories. For example, an implementation that ignores all the updates is strong eventually consistent, as all the queries return the initial state. In update consistency, we impose the existence of a total order on the updates, that contains the program order and that leads to the consistent state according to the abstract data type. Another equivalent way to approach update consistency is that, if the number of updates is finite, it is possible to remove a finite number of queries such that the history is sequentially consistent.

Definition 8 (Update consistency)**.** A history H is update consistent (UC) if U_H is infinite or if there exists a finite set of queries $Q' \subset Q_H$ such that $\mathrm{lin}\left(H_{E\setminus Q'}\right) \cap L(O) \neq \emptyset$.

The history of Fig. 1c is update consistent because the sequence of operations $\mathrm{I}(1)\mathrm{I}(2)$ is a possible explanation for the state $\{1,2\}$. The history of Fig. 1b is not update consistent because any linearization of the updates would position a deletion as the last event. Only three consistent states are actually possible: state $\emptyset$, e.g. for the linearization $\mathrm{I}(1)\cdot\mathrm{I}(2)\cdot\mathrm{D}(1)\cdot\mathrm{D}(2)$, state $\{1\}$ for the linearization $\mathrm{I}(2)\cdot\mathrm{D}(1)\cdot\mathrm{I}(1)\cdot\mathrm{D}(2)$ and state $\{2\}$ for the linearization $\mathrm{I}(1)\cdot\mathrm{D}(2)\cdot\mathrm{I}(2)\cdot\mathrm{D}(1)$. Update consistency is incomparable with strong eventual consistency.

[2] These consistency criteria were previously presented as a brief announcement in DISC 2014 [17].

Strong update consistency: strong update consistency is a strengthening of both update consistency and strong eventual consistency. The relationship between update consistency and strong update consistency is analogous to the relation between eventual consistency and strong eventual consistency.

Definition 9 (Strong update consistency)**.** A history H is strong update consistent (SUC) if there exists (1) an acyclic and reflexive relation $\xrightarrow{vis}$ that contains $\mapsto$ and (2) a total order $\leq$ that contains $\xrightarrow{vis}$ such that:

- Eventual delivery:
  $\forall u \in U_H, \{e \in E, u \not\xrightarrow{vis} e\}$ is finite;
- Growth:
  $\forall e, e', e'' \in E, \left(e \xrightarrow{vis} e' \wedge e' \mapsto e''\right) \Rightarrow \left(e \xrightarrow{vis} e''\right)$;
- Strong sequential convergence: A query views an update if this update precedes it according to $\xrightarrow{vis}$. Each query is the result of the ordered execution, according to $\leq$, of the updates it views:
  $\forall q \in Q_H, \mathrm{lin}\left(H^{\leq}_{V(q)\cup\{q\}}\right) \cap L(O) \neq \emptyset$
  where $V(q) = \{u \in U_H : u \xrightarrow{vis} q\}$.

Fig. 1d shows an example of strong update consistent history: nothing prevents the second process from seeing the insertion of 2 before that of 1. Strong eventual consistency and update consistency does not imply strong update consistency: in the history of Fig. 1c, after executing event I(1), the only three possible update linearizations are I(1), I(1) · I(2) and I(2) · I(1) and none of them can lead to the state $\emptyset$ according to the sequential specification of a set object. So the history of Fig. 1c is not strong update consistent, while it is update consistent and strong eventually consistent.

Proposition 2 (Comparison of consistency criteria)**.** *If a history H is update consistent, then it is eventually consistent. If H is strong update consistent, then it is both strong eventually consistent and update consistent.*

Proof: Suppose H is update consistent. If H contains an infinite number of updates, then it is eventually consistent. Otherwise, there exists a finite set $Q' \subset Q_H$ and a word $w \in \mathrm{lin}\left(H_{E_H \setminus Q'}\right) \cap L(O)$. As the number of updates is finite, there is a finite prefix v of w that contains them all. $v \in L(O)$, so it labels a path between s_0 and a state s in the UQ-ADT. All the queries that are in w but not in v return the same state s, and the number of queries in Q' and v is finite. Hence, H is eventually consistent.

Suppose H is strong update consistent with a finite number of updates. $Q' = \bigcup_{u \in U_H} \{q \in Q_H, q \leq u\}$ is finite, and $\mathrm{lin}\left(E_H \setminus Q'\right)$ contains only one word that is also contained into $L(O)$. Obviously, H is update consistent

Now, suppose H is strong update consistent. Strong update consistency respects both eventual delivery and growth properties. Let $V \subset U_H$. As the relation $\leq$ is a total order, there is a unique word w in $\mathrm{lin}\left(H^{\leq}_{V}\right) \cap L(O)$. Let us denote s the state obtained after the execution of w. For all $q \in Q_H$ such that $V = \{u \in U_H : u \xrightarrow{vis} q\}$, $\mathrm{lin}\left(H^{\leq}_{E_q \cup \{q\}}\right) \cap L(O) = \{w \cdot \Lambda(q)\}$, so $q = q_i/q_o$ with $G(s, q_i) = q_o$. Consequently, H is strong eventually consistent. ■

VI. Expressiveness of Update Consistency: a Case Study

The set is one of the most studied eventually consistent data structures. Different types of sets have been proposed as extensions to CRDTs to implement eventually consistent sets even though the insert and delete operations do not commute. The simplest set is the Grow-Only Set (G-Set) [9], in which it is only possible to insert elements. As the insertion of two elements commute, G-Set is a CRDT. Using two G-Set, a white list for inserted elements and a black list for the deleted ones, it is possible to build a Two-Phases Set (2P-Set, a.k.a. U-Set, for Unique Set) [18], in which it is possible to insert and remove elements, but never insert again an element that has already been deleted. Other implementations such as C-Set [19] and PN-Set, add counters on the elements to determine if they should be present or not. The Observe-Remove Set (OR-Set) [9], [20] is the best documented algorithm for the set. It is very close to the 2P-Set in its principles, but each insertion is timestamped with a unique identifier, and the deletion only black-lists the identifiers that it observes. It guaranties that, if an insertion and a deletion of the same element are concurrent, the insertion will win and the element will be added to the set. Finally, the last-writer-wins element set (LWW-element-Set) [9] attaches a timestamp to each element to decide which operation should win in case of conflict. All these sets, and the eventually consistent objects in general, have a different behavior when they are used in distributed programs.

The above mentioned implementations are eventually consistent. However, as eventual consistency does not impose a semantic link between updates and queries, it is hazardous to say anything on the conformance to the specification of the object. Burckhardt *et al.* [13] propose to specify the semantics of a query by a function on its concurrent history, called *visibility*, that corresponds to the visibility relation in strong eventual consistency, and a linearization of this history, called *arbitration*. In comparison, sequential specifications are restricted to the arbitration relation. It implies that fewer update consistent objects than eventually consistent objects can be specified. Although the variety of objects with a distributed specification seems to be a chance that compensates the lower level of abstraction it allows, an important bias must be taken into account: from the point of view of the user, the visibility of an operation is not an *a priori* property of the system, but an *a posteriori* way to explain what happened. If one only focuses on the final state, an update consistent object is appropriate to be used instead of an eventually consistent object, since the final state is the same as if no operations were concurrent.

By adding further constraints on the histories, concurrent specifications strengthen the consistency criteria. Even if strong update consistency is stronger than strong eventual consistency, we cannot say in general that a strong update consistent object can always be used instead of its strong eventually consistent counterpart. We claim that this is true in practice for *reasonable* objects, and we prove this in the case of the Insert-wins set (the concurrent specification of the OR-set). The arbitration relation is not used for the OR-set, and the visibility relation has already been defined for strong eventual consistency. The concurrent specification only adds one more constraint on this relation: an element is present in the set if and only if it was inserted and is not yet deleted.

Definition 10 (Strong eventual consistency for the Insert-wins set)**.** A history H is strong eventually consistent for the Insert-wins set on a support *Val* if it is strong eventually consistent for the set $\mathcal{S}_{Val}$ and the visibility relation $\xrightarrow{vis}$ verifies the following additional property. For all $x \in Val$ and $q \in Q_H$, with $\Lambda(q) = R/s$, $x \in s \Leftrightarrow \left(\exists u \in vis(q, \mathrm{I}(x)), \forall u' \in vis(q, \mathrm{D}(x)), u \overset{vis}{\not\longrightarrow} u'\right)$, where for all $o \in U$, $vis(q, o) = \{u \in U_H : u \xrightarrow{vis} q \wedge \Lambda(u) = o\}$.

The OR-Set implementation of a set is not update consistent. The history on Fig. 1b is not update consistent, as the last operation must be a deletion. However, if the updates made by a process are not viewed by the other process before it makes its own updates, the insertions will win and the OR-set will converge to $\{1, 2\}$. On the contrary, a strong update consistent implementation of a set can always be used instead of an Insert-wins set, as it only forbids more histories.

Proposition 3 (Comparison with Insert-wins set)**.** *Let $H = (U, Q, E, \Lambda, \mapsto)$ be a history that is strong update consistent for $\mathcal{S}_{Val}$. Then H is strong eventually consistent for the Insert-wins set.*

Proof: Suppose H is strong update consistent for $\mathcal{S}_{Val}$. We define the new relation $\xrightarrow{IW}$ such that for all $e, e' \in E$, $e \xrightarrow{IW} e'$ if one of the following conditions holds:

- $e \xrightarrow{vis} e'$;
- e and e' are two updates on the same element and $e \leq e'$;
- e' is a query, and there is an update e'' such that $e \xrightarrow{IW} e''$ and $e'' \xrightarrow{IW} e'$.

The relation $\xrightarrow{IW}$ is acyclic because it is included in $\leq$, its growth and eventual delivery properties are ensured by the fact that it contains $\xrightarrow{vis}$. Moreover, no two updates for the same element are concurrent according to $\xrightarrow{IW}$ and the last updates are also the last for the $\leq$ relation, consequently H is strong eventually consistent for the Insert-wins set. ■

This result implies that an OR-set can always be replaced by an update consistent set, because the guaranties it ensures are weaker than those of the update consistent set. It does not mean that the OR-set is worthless. It can be seen as a cache consistent set [21] that, in some cases may have a better space complexity than update consistency.

VII. Generic Construction of Strong Update Consistent Objects

In this section, we give a generic construction of strong update consistent objects in crash-prone asynchronous message-passing systems. This construction is not the most efficient ever as it is intended to work for any UQ-ADT object in order to prove the universality of update consistency. For a specific object an ad hoc implementation on a specific system may be more suitable.

A. System Model

We consider a message-passing system composed of finite set of sequential *processes* that may fail by halting. A faulty process simply stops operating. A process that does not crash during an execution is correct. We make no assumption on the number of failures that can occur during an execution. Processes communicate by exchanging *messages* using a communication network complete and reliable. A message sent by a correct process to another correct process is eventually received. The system is asynchronous; there is no bound on the relative speed of processes nor on the message transfer delays. In such a situation a process cannot wait for the participation of any a priori known number of processes as they can fail. Consequently, when an operation on a replicated object is invoked

Algorithm 1: a generic UQ-ADT (code for p_i)

```
object $(U, Q_i, Q_o, S, s_0, T, G)$
    var $\mathsf{clock}_i \in \mathbb{N} \leftarrow 0$;
    var $\mathsf{update}_i \subset (\mathbb{N} \times \mathbb{N} \times U) \leftarrow \emptyset$;
    fun update $(\mathsf{u} \in U)$
        $\mathsf{clock}_i \leftarrow \mathsf{clock}_i + 1$;
        broadcast message $(\mathsf{clock}_i, i, \mathsf{u})$;
    end
    on receive message $(\mathsf{cl} \in \mathbb{N}, \mathsf{j} \in \mathbb{N}, \mathsf{u} \in Q)$
        $\mathsf{clock}_i \leftarrow \max(\mathsf{clock}_i, \mathsf{cl})$;
        $\mathsf{update}_i \leftarrow \mathsf{update}_i \cup \{(\mathsf{cl}, \mathsf{j}, \mathsf{u})\}$;
    end
    fun query $(\mathsf{q} \in Q_i) \in Q_o$
        $\mathsf{clock}_i \leftarrow \mathsf{clock}_i + 1$;
        var $\mathsf{state}_i \in S \leftarrow s_0$;
        for $(\mathsf{cl}, \mathsf{j}, \mathsf{u}) \in \mathsf{update}_i$ sorted on $(\mathsf{cl}, \mathsf{j})$ do
            $\mathsf{state}_i \leftarrow T(\mathsf{state}_i, \mathsf{u})$;
        end
        return $G(\mathsf{state}_i, \mathsf{q})$;
    end
end
```

locally at some process, it needs to be completed based solely on the local knowledge of the process. We call this kind of systems wait-free asynchronous message-passing system.

We model executions as histories made up of the sequences of events generated by the different processes. As we focus on shared objects and their implementation, only two kinds of actions are considered: the operations on shared objects, that are seen as events in the distributed history, and message receptions.

B. A universal implementation

Now, we prove that strong update consistency is universal, in the sense that every UQ-ADT has a strong update consistent implementation in a wait-free asynchronous system. Algorithm 1 presents an implementation of a generic UQ-ADT. The principle is to build a total order on the updates on which all the participants agree, and then to rewrite the history *a posteriori* so that every replica of the object eventually reaches the state corresponding to the common sequential history. Any strategy to build the total order on the updates would work. In Algorithm 1, this order is built from a Lamport's clock [14] that contains the happened-before precedence relation. Process order is hence respected. A logical Lamport's clock is a pretotal order as some events may be associated with the same logical time. In order to have a total order, the events are timestamped with a pair composed of the logical time and the id of the process that produced it (process ids are assumed unique and totally ordered). The algorithm actions performed by a process p_i are atomic and totally ordered by an order $\mapsto_i$. The union of these orders for all processes is the program order $\mapsto$.

At the application level, a history is composed of update and query operations. In order to allow only strong update consistent histories, Algorithm 1 proposes a procedure $update()$ and a function $query()$. A history H is allowed by the algorithm if $update(u)$ is called each time a process performs an update u, and $query(q_i)$ is called and returns q_o when the event q_i/q_o appears in the history. The code of Algorithm 1 is given for process p_i. Each process p_i manages its view $clock_i$ of the logical clock and a list $updates_i$ of all timestamped update events process p_i is aware of. The list $updates_i$ contains triplets (cl, j, u) where u is an update event and (cl, j) the associated timestamp. This list is sorted according to the timestamps of the updates: $(cl, j) < (cl', j')$ if $(cl < cl')$ or $(cl = cl'$ and $j < j')$.

The algorithm timestamps all events (updates and queries). When an update is issued locally, process p_i informs all the other processes by reliably broadcasting a message to all other processes (including itself). Hence, all processes will eventually be aware of all updates. When a $message(cl, j, u)$ is received, p_i updates its clock and inserts the event to the list $updates_i$. When a query is issued, the function $query()$ replays locally the whole list of update events p_i is aware of starting from the initial state then it executes the query on the state it obtains.

Whenever an operation is issued, its is completed without waiting for any other process. This corresponds to wait-free executions in shared memory distributed systems and implies fault-tolerance.

Proposition 4 (Strong update consistency)**.** *All histories allowed by Algorithm 1 are strong update consistent.*

Proof: Let $H = (U, Q, E, \Lambda, \mapsto)$ be a distributed history allowed by Algorithm 1. Let $e, e' \in E_H$ be two operations invoked by processes p_i and $p_{i'}$, on the states $(\texttt{update}, \texttt{clock})$ and $(\texttt{update}', \texttt{clock}')$, respectively. We pose:

- $e \xrightarrow{vis} e'$ if $e \in U_H$ and $p_{i'}$ received the message sent during the execution of e before it starts executing e', or $e \in Q_H$ and $e \mapsto e'$. As the messages are received instantaneously by the sender, $\xrightarrow{vis}$ contains $\mapsto$. It is growing because the set of messages received by a process is growing with time.

- $e \leq e'$ if $c < c'$ or $c = c'$ and $i < i'$. This lexical order is total because two operations on the same process have a different clock. Moreover it contains $\xrightarrow{vis}$ because when $p_{i'}$ received the message sent by e, it executed line 9 and when it executed e', it executed line 5, so $c' \geq c + 1$. Moreover, the history of e contains at most $c \times n + i$ events, where n is the number of processes, so it is finite.

Let $q \in Q_H$ and $E_q = \{u \in U_H : u \xrightarrow{vis} q\}$. Lines 15 to 18 build an explicit sequential execution, that is in $\text{lin}\left(H^{\leq}_{E_q \cup \{q\}}\right)$ by definition of $\leq$ and in $L(O)$ by definition of O. ■

C. Complexity

Algorithm 1 is very efficient in terms of network communication. A unique message is broadcast for each update and each message only contains the information to identify the update and a timestamp composed of two integer values, that only grow logarithmically with the number of processes and the number of operations. Moreover, this algorithm is wait-free and its execution does not depend on the latency of the network.

This algorithm re-executes all past updates each time a new query is issued. In an effective implementation, a process can keep intermediate states. These intermediate states are re-computed only if very late message arrive. The algorithm does not look space efficient also as the whole history must be kept in order to rebuild a sequential history. Because data space is cheap and fast nowadays, compared to bandwidth, many applications can afford this complexity and would keep this information anyway. For example, banks keep track of all the operations made on an account for years for legal reasons. In databases systems, it is usual to record all the events in log files. Moreover, asynchrony is used as a convenient abstraction for systems in which transmission delays are actually bounded, but the bound is too large to be used in practice. This means that after some time old messages can be garbage collected.

The proposed algorithm is a theoretical work whose goal is to prove that any update-query object has a strong update consistent implementation. This genericity prevents an effective implementation that may take benefit from the nature and the specificity of the actual object. The best example of this are pure CRDTs like the counter and the grow-only set. If all the update operations commute in the sequential specification, all linearizations would lead to the same state so a naive implementation, that applies the updates on a replica as soon as the notification is received, achieves update consistency. In [22], Karsenty and Beaudouin-Lafon propose an algorithm to implement objects such that each update operation u contains an *undo* u^{-1} such that for all s, $T(T(s,u),u^{-1}) = s$. This algorithm is very close to ours as it builds the convergent state from a linearization of the updates stored by each replica. They use the undo operations to position newly known updates at their correct place, which saves computation time. As it is a very frequent example in distributed systems, we now focus on the shared memory object.

Algorithm 2: the shared memory (code for p_i)

```
object UC_mem($X, V, v_0$)
    var clock$_i \in \mathbb{N} \leftarrow 0$;
    var mem$_i \in \text{mem}(X, (\mathbb{N}^2 \times V), (0, 0, v_0))$;
    fun write (x $\in X$, v $\in V$)
        clock$_i \leftarrow$ clock$_i + 1$;
        broadcast msg (clock$_i$, $i$, x, v);
    end
    on receive msg (cl $\in \mathbb{N}$, j $\in \mathbb{N}$, x $\in X$, v $\in V$)
        clock$_i \leftarrow \max($clock$_i$, cl$)$;
        var (cl$'$, j$'$, v$'$) $\in \mathbb{N}^2 \times V \leftarrow$ mem$_i$.read(x);
        if (cl$'$, j$'$) $<$ (cl, j) then
            mem$_i$.write(x, (cl, j, v))
        end
    end
    fun read (x $\in X$) $\in V$
        var (cl, j, v) $\in \mathbb{N}^2 \times V \leftarrow$ mem$_i$.read(x);
        return v;
    end
end
```

Algorithm 2 shows an update consistent implementation of the shared memory object. A shared memory offers a set X of registers that contain values taken from a set V. The query operation read(x), where $x \in X$, returns the last value $v \in V$ written by the update operation write(x, v), or the initial value $v_0 \in V$ if x was never written. Algorithm 2 orders the updates exactly like Algorithm 1. As the old values can never be read again, it is not necessary to store them forever, so the algorithm only keeps in memory the last known value of each register and its timestamp in a local memory mem_i, implemented with an associative array. When a process receives a notification for a write, it updates its local state if the value is newer that the current one, and the read operations just return the current value. This implementation only needs constant computation time for both the reads and the writes, and the complexity in memory only grows logarithmically with time and the number of participants.

VIII. Conclusion

This paper proposes a new consistency criterion, update consistency, that is stronger than eventual consistency and weaker than sequential consistency. Our approach formalizes the intuitive notions of sequential specification for an abstract data type and distributed history. This formalization first allowed to prove that eventual consistency when associated with causal consistency or PRAM consistency can no more be implemented in an asynchronous distributed system where all but one process may crash.

This paper formalizes the new consistency criterion and proves that (1) it is strictly stronger than eventual consistency and (2) that it is universal in the sense that allowed any update consistent object can be implemented in wait-free systems. The latter has been proved through a generic construction that implement all considered data types.

Acknowledgment

This work has been partially supported by a French government support granted to the CominLabs excellence laboratory (Project *DeSceNt: Plug-based Decentralized Social Network*) and managed by the French National Agency for Research (ANR) in the "Investing for the Future" program under reference Nb. ANR-10-LABX-07-01. This work was partially funded by the French ANR project SocioPlug (ANR-13-INFR-0003).

References

[1] R. J. Lipton and J. S. Sandberg, *PRAM: A scalable shared memory*. Princeton University, Department of Computer Science, 1988.

[2] H. Attiya and J. L. Welch, "Sequential consistency versus linearizability," *ACM Transactions on Computer Systems (TOCS)*, vol. 12, no. 2, pp. 91–122, 1994.

[3] H. Attiya, A. Bar-Noy, and D. Dolev, "Sharing memory robustly in message-passing systems," *J. ACM*, vol. 42, no. 1, pp. 124–142, 1995.

[4] W. Vogels, "Eventually consistent," *Queue*, vol. 6, no. 6, pp. 14–19, 2008.

[5] M. Ahamad, G. Neiger, J. E. Burns, P. Kohli, and P. W. Hutto, "Causal memory: Definitions, implementation, and programming," *Distributed Computing*, vol. 9, no. 1, pp. 37–49, 1995.

[6] M. Herlihy, "Wait-free synchronization," in *ACM Transactions on Programming Languages and Systems*, 1991, pp. 124–149.

[7] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: amazon's highly available key-value store," in *ACM SIGOPS Operating Systems Review*, vol. 41. ACM, 2007, pp. 205–220.

[8] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "Conflict-free replicated data types," in *Stabilization, Safety, and Security of Distributed Systems*. Springer, 2011, pp. 386–400.

[9] M. Shapiro, N. Preguiça, C. Baquero, M. Zawirski *et al.*, "A comprehensive study of convergent and commutative replicated data types," INRIA, Tech. Rep., 2011.

[10] C. Sun, X. Jia, Y. Zhang, Y. Yang, and D. Chen, "Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems," *ACM Transactions on Computer-Human Interaction (TOCHI)*, vol. 5, no. 1, pp. 63–108, 1998.

[11] D. Li, L. Zhou, and R. R. Muntz, "A new paradigm of user intention preservation in realtime collaborative editing systems," in *International Conference on Parallel And Distributed Systems*. IEEE, 2000, pp. 401–408.

[12] A. Bieniusa, M. Zawirski, N. Preguiça, M. Shapiro, C. Baquero, V. Balegas, and S. Duarte, "An optimized conflict-free replicated set," *arXiv preprint arXiv:1210.3368*, 2012.

[13] S. Burckhardt, A. Gotsman, H. Yang, and M. Zawirski, "Replicated data types: specification, verification, optimality," in *Proceedings of the 41st symposium on Principles of programming languages*. ACM, 2014, pp. 271–284.

[14] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[15] M. Perrin, M. Petrolia, C. Jard, and A. Mostéfaoui, "Consistent shared data types: Beyond memory," LINA, Université de Nantes, Tech. Rep., 2014.

[16] G. H. Mealy, "A method for synthesizing sequential circuits," *Bell System Technical Journal*, vol. 34, no. 5, pp. 1045–1079, 1955.

[17] M. Perrin, A. Mostéfaoui, and C. Jard, "Brief announcement: Update consistency in partitionable systems," in *Proceedings of the 28th International Symposium on Distributed Computing*. Springer, 2014, p. 546.

[18] G. T. Wuu and A. J. Bernstein, "Efficient solutions to the replicated log and dictionary problems," *Operating systems review*, vol. 20, no. 1, pp. 57–66, 1986.

[19] K. Aslan, P. Molli, H. Skaf-Molli, S. Weiss *et al.*, "C-set: a commutative replicated data type for semantic stores," in *RED: Fourth International Workshop on REsource Discovery*, 2011.

[20] M. Mukund, G. Shenoy, and S. Suresh, "Optimized or-sets without ordering constraints," in *Distributed Computing and Networking*. Springer, 2014, pp. 227–241.

[21] J. R. Goodman, *Cache consistency and sequential consistency*. University of Wisconsin-Madison, Computer Sciences Department, 1991.

[22] A. Karsenty and M. Beaudouin-Lafon, "An algorithm for distributed groupware applications," in *ICDCS*, 1993.