D.1.1 – Survey on Weak Consistency Approaches for Large-Scale Systems

Davide Frey, Achour Mostefaoui, Matthieu Perrin, François Taïani

D.1.1 – Survey on Weak Consistency Approaches for Large-Scale Systems

ANR-13-INFR-0003
socioplug.univ-nantes.fr

Davide Frey[1,2], Achour Mostéfaoui[3], Matthieu Perrin[3], François Taïani[1,2,4]

[1]*IRISA Rennes*
[2]*Inria Rennes - Bretagne Atlantique*
[3]*LINA, Université de Nantes*
[4]*Université de Rennes 1 - ESIR*
Email contact: {francois.taiani,davide.frey}@irisa.fr, matthieu.perrin@etu.univ-nantes.fr, achour.mostefaoui@univ-nantes.fr

Keywords: Infrastucture, Use cases, Complementarity of task

1 Introduction

Distributed systems are often viewed as more difficult to program than sequential systems because they require solving a number of communication issues. Shared objects that can be accessed concurrently by multiple parties can be used as a practical communication abstraction to let processes enjoy a more general view of the system. A precise specification of these objects is therefore essential to ensure their adoption as well as the reliability of distributed systems.

Many models have been proposed to specify shared memory, and several inventories [Mos93, AG96] can be found in the literature. In [Lam86], Lamport defines linearizable registers that ensure that everything appears as if all the operation where executed instantaneously at a point between the moment when the operation is called and the moment when it returns. Sequential consistency [Lam79] is a little weaker: it only guarantees that all the operations appear as totally ordered, and that this order be compatible with the *program order*, the order in which each process performs its own operations.

These strong consistency criteria are very expensive to implement in message-passing systems. In terms of latency, it is necessary to wait for answers for the reads or the writes for sequential consistency [LS88] and for all kinds of operations in the case of linearizability [AW94]. In terms of fault tolerance, strong hypotheses must be respected by the system: it is impossible to resist to partitioning [Bre00, GL02]. Many weaker consistency criteria have been proposed to solve this problem. Among them, PRAM [LS88], causal memory [ANB+95] and eventual consistency [Vog08] are best documented. PRAM is a local consistency criterion: each process only sees its own reads and all the writes, and all these operations appear to it as totally ordered by an order that respects the program order. Causal consistency strengthens PRAM by imposing to these total orders to be compatible with a partial order that is common to all processes, the *causal order*, that not only contains the process order, but also a *writes-into* relation, that encodes data dependencies

between the write and the read operations. Eventual consistency expects that, if all the processes stop writing, they will eventually read the same values.

Memory is a good abstraction in sequential programming models because all kinds of objects can be implemented using variables. Things are more complicated for distributed computing because of race conditions: complex concurrent editing can often lead to inconsistent states. Critical sections offer a generic solution to this problem but at a high cost: they reduce parallelism and they are unable to tolerate faulty behaviors. A better solution is to design the shared objects directly, without using shared memory.

This deliverable surveys the main consistency criteria that exist in the literature, and expresses them in a consistent way, thereby making it easy to compare them. An object should be totally specified by two facets: a sequential specification given by an abstract data type, and a consistency criterion, that defines a link between distributed histories and sequential specifications. Sequential specifications are very easy to design because they are based on the well studied and understood notions of languages and automata. This makes it possible to apply all the tools developed for sequential systems, from their simple definition using structures and classes to the most advanced techniques like model checking and formal verification. Sequential objects cannot be used directly in distributed environments. A consistency criterion is necessary to adapt the sequential specification to the environment. Graphically, we can imagine a consistency criterion as a way to take a picture of the distributed histories so that they look sequential.

The remainder of this document is organized as follows. Section 2 presents the basic notions of sequential specification and consistency criteria and studies their general properties. Section 3 surveys the main consistency criteria existing in the literature. Section 4 provides a brief descriptions of recent systems that trade off some consistency for performance. Finally, Section 5 concludes the survey.

2 Specifying shared objects

In distributed systems, sharing objects is a way to abstract communications 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. As shared objects are implemented in a distributed system, typically using replication, the events in a distributed history are partially ordered. A consistency criterion is therefore necessary to make the link between the sequential specifications and the distributed executions that invoke them, by a characterization of the histories that are admissible for a program that uses the objects, depending on their type.

The notions defined in this section, namely, operation, history, and partial order on operations are usual in the definition of strong consistency criteria [AW04, HS08, Lyn96, Ray12].

2.1 Update-query abstract data types

We use transition systems to specify sequential abstract data types. Our modeling of update-query abstract data type is intermediate between Mealy machines [Mea55] and transition systems. We separate the input alphabet into two classes of operations: the updates and the queries[†]. On the one hand, the updates can have a side-effect that usually affects everyone, but do not return a value. They correspond to transitions between abstract states in the transition system. On the other hand, the queries can only read the state of the object but not modify it. Like in mealy machines, they produce an output that depends on the state on which they are executed.

Definition 1. An *update-query abstract data type* (UQ-ADT) is a 7-tuple $T = (U, Q_i, Q_o, S, s_0, \mu, \varphi)$ such that:

- U is a countable set of *update* operations;

[†] Some objects, such as Compare&Swap and Test&Set for example, have operations that are both an update and a query. Such objects are out of the scope of this paper.

- 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) is denoted q_i/q_o;
- S is a countable set of *states*;
- $s_0 \in S$ is the *initial state*;
- $\mu : S \times U \to S$ is the *transition function* that specifies the updates;
- $\varphi : S \times Q_i \to Q_o$ is the *output function* that specifies the queries.

We consider the UQ-ADTs up to isomorphism as the names of the states are never used in our work. The set of all the UQ-ADTs is denoted by $\mathscr{T}$.

An infinite sequence of operations $(l_i)_{i\in\mathbb{N}} \in (U \cup Q)^\omega$ is recognized by T if there exists an infinite sequence of states $(s_i)_{i\geq 1} \in S^\omega$ such that for all $i \in \mathbb{N}$, $\mu(s_i, l_i) = s_{i+1}$ if $l_i \in U$ or $s_i = s_{i+1}$ and $\varphi(s_i, q_i) = q_o$ if $l_i = q_i/q_o \in Q$. The set of all infinite sequences recognized by T and their finite prefixes is denoted by $L(T)$. Informally, $L(T)$ contains the sequential histories that are admissible for T.

Suppose one wants to use several objects together in their program. We can model the composition of these objects as another object on which it is possible to apply the operations of both objects. The composition of two UQ-ADTs T_1 and T_2 is a parallel, asynchronous product of transitions systems. A word is recognized by the composition if it is the interleaving of two words recognized by T_1 and T_2 respectively. This composition is associative and commutative, and there is one neutral element T_1 that contains one state and no transition and an absorbing element T_0 that contains no state and no transition.

Definition 2. We define the composition of two UQ-ADTs $(U, Q_i, Q_o, S, s_0, \mu, \varphi) \times (U', Q_i', Q_o', S', s_0', \mu', \varphi')$ as the UQ-ADT $(U \sqcup U', Q_i \sqcup Q_i', Q_o \cup Q_o', S \times S', s_0', \mu'', \varphi'')$ with, for all $s \in S$, $s' \in S'$, $u \in U$, $u' \in U'$, $q \in Q_i$ and $q' \in Q_i'$, $\mu''((s,s'),u) = T(s,u)$, $\mu''((s,s'),u') = \mu'(s',u')$, $\varphi''((s,s'),q) = \varphi(s,q)$ and $\varphi''((s,s'),q') = \varphi'(s',q')$. The symbol $\sqcup$ stands for the disjoint union of sets.

We illustrate UQ-ADTs by two examples. We first give the full sequential specification of the register and the memory. Then, we define the graph as a class, like in sequential object-oriented programming languages.

The integer *register* $\mathcal{M}_x$ can be accessed by a *write* update operation $\mathrm{w}_x(n)$, where $n \in \mathbb{N}$ and a *read* query operation r_x that returns the last value written, if there is one, or the default value 0 otherwise. The integer *memory* $\mathcal{M}_X$ is the collection of the integer registers of X. More formally, they correspond to the UQ-ADTs given in Example 1.

Example 1. Let x be any symbolic *register name*. We define the integer *register* on x by the UQ-ADT:

$$\mathcal{M}_x = (\{\mathrm{w}_x(n) : n \in \mathbb{N}\}, \{\mathrm{r}_x\}, \mathbb{N}, \mathbb{N}, 0, \mu, \varphi)$$

with, for all $n, p \in \mathbb{N}$, $\mu(n, \mathrm{w}_x(p)) = p$ and $\varphi(n, \mathrm{r}_x) = n$.

Let X be a countable set of register names, we define the integer *memory* on X by the UQ-ADT:

$$\mathcal{M}_X = \prod_{x\in X} \mathcal{M}_x.$$

Figure 1 presents another more usual way to define sequential specifications. Here, the graph type $\mathcal{G}$ is specified by a class. A graph is constituted of a set of vertices, here represented by integers, and a set of edges, that are pairs of vertices. It provides four update operations, to insert and delete edges and vertices, and two query operation that check if a vertex or an edge is present in the graph and return a boolean value. A graph must remain consistent: it cannot contain edges between vertices that do not exist in the graph. To ensure this, vertices are inserted or edges are deleted to avoid inconsistencies. An UQ-ADT can easily been deduced from this specification. The states and the operations are defined by the member variables and methods of the class, the output alphabet contains all the values that can be returned by the queries, here the

```
class 𝒢
  var vertices ⊂ ℕ ← ∅;
  var edges ⊂ ℕ × ℕ ← ∅;
  update I_v (v ∈ ℕ)                                   /* insert vertex */
    vertices ← vertices ∪ {v};
  end
  update D_v (v ∈ ℕ)                                   /* delete vertex */
    edges ← edges \ (V × {v} ∪ {v} × V);
    vertices ← vertices \ {v};
  end
  query Q_v (v ∈ ℕ) ∈ {⊥, ⊤}                            /* get vertex */
    return v ∈ vertices;
  end
  update I_e (v1 ∈ ℕ, v2 ∈ ℕ)                          /* insert edge */
    vertices ← vertices ∪ {v1, v2};
    edges ← edges ∪ {(v1, v2)};
  end
  update D_e (v1 ∈ ℕ, v2 ∈ ℕ)                          /* delete edge */
    edges ← edges \ {(v1, v2)};
  end
  query Q_e (v1 ∈ ℕ, v2 ∈ ℕ) ∈ {⊥, ⊤}                  /* get edge */
    return (v1, v2) ∈ edges;
  end
end
```

Figure 1: Sequential specification of the graph

booleans values true ($\top$) and false ($\bot$), the transition function is defined by the implementation of the update methods and the output function by the query methods. We consider the pruned transition system, in which all the states are reachable.

2.2 Distributed histories

During an execution, the participants call the operations of an object, an instance of the abstract data type, which produces a set of events labelled by the operations of the abstract data type. In general, the computing entities are sequential, which imposes a strict ordering between their own operations, so the events are partially ordered. 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. In this case, the processes correspond to the maximal chains of the history. This representation of distributed histories is generic enough to model a large number of distributed systems, such as peer-to-peer systems where peers join and leave permanently, or more complex modern systems in which new threads are created and destroyed dynamically. Note that the discreteness of the space of the events does not mean that the operations must return immediately, as our model does not introduce any notion of real time.

Definition 3. A distributed history (or simply history) is a 5-tuple $H = (U, Q, E, \Lambda, \mapsto)$ such that:

- U and Q are disjoint countable sets of *update* and *query* operations, and all queries are in the form 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, i.e. all event has a finite past.

The set of all the distributed histories is denoted by $\mathscr{H}$.

Let $H = (U, Q, E, \Lambda, \mapsto)$ be a distributed history. We now introduce a few notations.

- The update and query events of H are denoted by $U_H = \{e \in E : \Lambda(e) \in U\}$, $Q_H = \{e \in E : \Lambda(e) \in Q\}$.
- Two events e and e' are concurrent (denoted $e||e'$) if they are not comparable for $\mapsto$, i.e. $e \not\mapsto e'$ and $e' \not\mapsto e$.
- $\mathscr{P}_H$ denotes the set of the maximal chains of H:

$$\mathscr{P}_H = \left\{ p \subset E : \begin{array}{l} \forall e, e' \in p, (e \mapsto e' \vee e' \mapsto e) \\ \wedge \forall e'' \in E \setminus p, (e||e'' \vee e'||e'') \end{array} \right\}.$$

  In the case of sequential processes, each $p \in \mathscr{P}_H$ corresponds to the events produced by a process. In the remainder of this article, we use the term "process" to designate such a chain, even in models that are not based on a collection of communicating sequential processes.

- 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 replace the program order by another order $\rightarrow$: if $\rightarrow \cap (E \times E)$ respects the definition of a program order, $H^{\rightarrow} = (U, Q, E, \Lambda, \rightarrow \cap (E \times E))$ is the history in which the events are ordered by $\rightarrow$. Note that the projections commute, which allows the notation $H_F^{\rightarrow}$.
- A history in which a composition of two objects is used can also be seen as the composition of two sub-histories that only contain the events of one of the objects. Let $H = (U, Q, E, \Lambda, \mapsto)$ be a distributed history and $\{E_1, E_2\}$ be a partition of E. We say that H is a composition of H_{E_1} and H_{E_2}. There is more than one way to compose histories, so the composition of two histories is a set of histories. The set of all the compositions of H_1 and H_2 is denoted by $H_1 \times H_2$.
- Finally, a linearization of H corresponds to a sequential history that contains the events of H in an order consistent with the program order. More precisely, it is a word $\Lambda(e_0) \ldots \Lambda(e_n) \ldots$ such that $\{e_0, \ldots, e_n, \ldots\} = E$ and for all i and j, if $i < j$, then $e_j \not\mapsto e_i$. We denote by $\mathrm{lin}(H)$ the set of all linearizations of H.

2.3 Consistency criteria

A consistency criterion characterizes which histories are admissible for a given data type. More formally, it is a function C that associates a set of consistent histories $C(T)$ to all UQ-ADTs $T = (U, Q_i, Q_o, S, s_0, \mu, \varphi)$ such that, for all $H = (U', Q', E, \Lambda, \mapsto)$ in $C(T)$, $U' \subset U$ and $Q' \subset Q_i \times Q_o$. The set of all consistency criteria is denoted by $\mathscr{C}$. A shared object is C-consistent for a consistency criterion C and a UQ-ADT T if all the histories it admits are in $C(T)$.

We say that a criterion $C1$ is *stronger* than a criterion $C2$, denoted $C_2 \leq C_1$, if for all $T \in \mathcal{T}$, $C1(T) \subset C2(T)$. A strong consistency criterion guaranties stronger properties on the histories it admits, so a C_1-consistent implementation can always be used instead of a C_2-consistent implementation of the same abstract data type if $C_2 \leq C_1$.

Sometimes, one wants an object to respect several consistency criteria simultaneously (e.g. a causally consistent and eventually consistent memory). We define a join operator $C_1 + C_2 : T \mapsto C_1(T) \cap C_2(T)$. $(\mathscr{C}, \leq, +)$ is a join-semilattice. It has a minimal element, $C_\perp$, that accepts all the histories for all the objects and a maximal element, $C_\top$, that accepts none of them.

We can now define the composition of two consistency criteria. If C_1 and C_2 are two consistency criteria, $C_1 \times C_2$ denotes the set of the histories that are the composition of a C_1 consistent history and a C_2 consistent history.

Definition 4. Let C_1 and C_2 be two consistency criteria. We define $C_1 \times C_2$ as the strongest consistency criterion such that, for all $T_1, T_2 \in \mathscr{T}$ and for all histories $H_1 \in C_1(T_1)$ and $H_2 \in C_2(T_2)$, $H_1 \times H_2 \subset (C_1 \times C_2)(T_1 \times T_2)$. This strongest criterion exists since the property that we require on it is conserved by $+$.

$(\mathscr{C}, \times)$ is a commutative monoid (i.e. $\times$ is commutative and associative and $C^\top$ is neutral), $C_\perp$ is absorbing, $\times$ is distributive for $+$ (i.e. $C_1 \times (C_2 + C_3) = C_1 \times C_2 + C_1 \times C_3$), $\leq$ is compatible with $\times$ (i.e. $C_1 \leq C_2 \Rightarrow C_1 \times C_3 \leq C_2 \times C_3$) and $C_1 \times C_2 \leq C_1$.

We use the classical power notation C^n for the composition of n C-consistent objects, and the compositional closure $C^\star : T \mapsto \bigcup_{n \in \mathbb{N}} C^n(T)$ stands for the composition of any number of C-consistent objects. Intuitively, $C^\star$ is the limit of C^n when n grows to infinity.

One could expect that a program that uses two C-consistent objects together will remain C-consistent with respect to the composition of the objects. This property, called composability, is an important property because it allows to program in a modular way, but only $C^2 \leq C$ is true in general.

Definition 5. A consistency criterion C is *composable* if it is idempotent for composition, i.e. $C^2 = C$.

As we will see in Section 3, composability is difficult to achieve. The reciprocal, however, is a natural request to all consistency criteria: if a history is globally consistent, it should also be consistent for all the objects that are involved in it. Decomposability means that, for all $T_1, T_2 \in \mathscr{T}$ and all history $H \in C(T_1 \times T_2)$, there exists histories $H_1 \in C(T_1)$, $H_2 \in C(T_2)$ such that $H \in H_1 \times H_2$. All the consistency criteria defined thereafter are decomposable.

3 Common consistency criteria

We now illustrate the concept of consistency criteria with common examples. We first illustrate the formalism with sequential consistency [Lam79] and its derivatives, cache consistency [Goo91] and linearizability [HW90]. We then formalize pipelined consistency [LS88], that is very close in its definition to local consistency. Causal consistency [ANB+95] is more complicated to extend because its definition is closely related to the semantics of the operations on the registers. We finish this presentation with eventual consistency [Vog08] and strong eventual consistency [SPB+11]. All these criteria are illustrated on small examples on the memory and graph data types.

3.1 Sequential consistency

Sequential consistency was originally defined by Lamport in [Lam79] as: "the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program". In our formalism, such a sequence is a word of update and query operations that has two properties: it is correct with respect to the sequential specification of the object (i.e. it belongs to $L(T)$) and the total order is compatible with the program order (i.e. it belongs to $\mathrm{lin}(H)$).

Definition 6. A history H is *sequentially consistent* (SC) with an UQ-ADT T if $\mathrm{lin}(H) \cap L(T) \neq \emptyset$.

Figure 2a shows a sequentially consistent history. It can be viewed as two processes sharing a graph of $\mathcal{G}$. The first process first inserts the edge $(1,2)$ and then reads that the vertex 3 was inserted, while the second process inserts the edge $(2,3)$ and then reads that the vertex 1 was not inserted yet. The word $\mathrm{I_e}(2,3).\mathrm{Q_v}(1)/\perp.\mathrm{I_e}(1,2).\mathrm{Q_v}(3)/\top$ is in both $\mathrm{lin}(H)$ and $L(\mathcal{G})$, so this history is sequentially consistent.

The history of Figure 2b is very close, but the shared object is a memory, and each query returns the initial value of the register. This history is not sequentially consistent because the first event

(a) $SC(\mathcal{G})$ (b) $SC^\star$, not SC

Figure 2: A sequential history for an instance of $\mathcal{G}$ and a cache consistent history for an instance of $\mathcal{M}_{\{x,y\}}$ that is not sequentially consistent

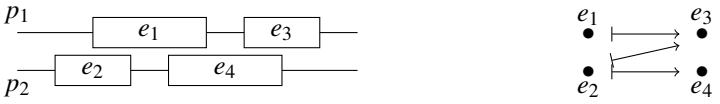


Figure 3: The program order is defined considering real time dependencies. We cannot have both $e_1||e_4$ and $e_2||e_3$

of any linearization must be a write which precedes the read of the same variable, that should return a 1. To be sequentially consistent, at least one read should return a 1.

Cache consistency The compositional closure seems to be an easy way to define composable consistency criteria, as $C^\star$ is the strongest composable consistency criterion weaker than C. However, this criterion can be very weak. For example, $SC^\star$, known as *cache consistency* [Goo91] or simply *coherence* [GLL+90] for memory in the literature, does not allow any guaranty of synchronization between the variables.

As $r_x/0.w_x(1)$ is a possible linearization for the events on $\mathcal{M}_x$ and $r_y/0.w_y(1)$ is a possible linearization for the events on $\mathcal{M}_y$, the history of Figure 2b is cache consistent for $\mathcal{M}_{\{x,y\}} = \mathcal{M}_x \times \mathcal{M}_y$.

Linearizability: the case of real time Linearizability is very close to sequential consistency, as it also imposes the existence of a total order on all the events in the history. Real time must be respected by this total order in linearizability: if an event e_1 finishes before another event e_2 starts, then e_1 must precede e_2 in the total order.

We did not introduce real time in our model yet because it is not relevant for most consistency criteria, and no global clock can be implemented in asynchronous distributed systems. However, it is possible to model real-time dependencies between events directly in the histories, by only considering interval orders [Fis85] for the program orders. In this paragraph, we change the modeling of the executions: an event e_1 precedes an event e_2 in the program order if and only if e_1 returns before e_2 starts. Let us consider Figure 3. A process p_1 produces the events e_1 and e_3 while a process p_2 produces the events e_2 and e_4. It is impossible that e_3 starts before e_2 finishes and e_4 starts before e_1 finishes at the same time, which implies $e_1 \mapsto e_4$ or $e_2 \mapsto e_3$ if the program order models real time.

Definition 7. A history $(U, Q, E, \Lambda, \mapsto)$ is *real time consistent* (RT) if $(e_1 \mapsto e_3 \wedge e_2 \mapsto e_4) \Rightarrow (e_1 \mapsto e_4 \vee e_2 \mapsto e_3)$ for all $e_1, e_2, e_3, e_4 \in E$.

Real time consistency is not composable. However, if real time is respected during the composition of sequentially consistent histories, the composed histories are also sequentially consistent.

Proposition 1. $SC^2 + RT = SC + RT$.

Proof. It is clear that $SC^2 + RT \leq SC + RT$ and that $RT \leq SC^2 + RT$. We prove that $SC \leq SC^2 + RT$. Let $T_1, T_2 \in \mathscr{T}$, $H_1 = (U_1, Q_1, E_1, \Lambda_1, \mapsto_1) \in SC(T_1)$ and $H_2 = (U_2, Q_2, E_2, \Lambda_2, \mapsto_2) \in SC(T_2)$. We suppose that $H = (U, Q, E, \Lambda, \mapsto) \in H_1 \times H_2 \cap RT(T_1 \times T_2)$. We prove that $H \in SC(T_1 \times T_2)$.

By definition of sequential consistency, for $i \in \{1, 2\}$ there is a total order $\leq_i$ on E_i and a unique word l_i such that $\text{lin}\left(H_i^{\leq_i}\right) \cap L(T_i) = \{l_i\}$ and $\mapsto_i \subset \leq_i$. For sake of simplicity, we extend $\leq_i$ on E by

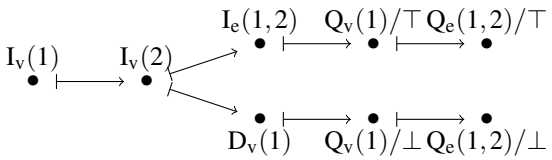


Figure 4: A locally consistent history on $\mathcal{G}$

taking its reflexive closure. We pose $\circ = \leq_1 \cup \leq_2 \cup \mapsto$ and $\bullet$ as the transitive closure of $\circ$. We first prove that $\bullet$ is antisymmetric.

Let $e_1 \circ e_2 \circ ... \circ e_{n-1} \circ e_n \in E$, with $e_1 \in E_1$. We prove by induction on n that there are $e, e' \in E$ such that $e_1 \leq_1 e \mapsto e' \leq_2 e_n$ (and by symmetry, $e_1 \leq_2 e \mapsto e' \leq_1 e_n$ if $e_1 \in E_2$). If $n = 1$, we have $e_1 \leq_1 e_1 \mapsto e_1 \leq_2 e_1$. Suppose the result is true for n and let us examine it for $n+1$. We have $e_1 \leq_1 e \mapsto e' \leq_2 e_n \circ e_{n+1}$. There are three cases for $\circ$.

- If $e_n \leq_2 e_{n+1}$, by transitivity, $e_1 \leq_1 e \mapsto e' \leq_2 e_{n+1}$.
- If $e_n \leq_1 e_{n+1}$ and $e_n \neq e_{n+1}$, then $e_n \in E_1$ so $e' = e_n$. Moreover, $e \in E_1$ because $e_1 \leq_1 e$, so $e \leq_1 e_n$. We have $e_1 \leq_1 e_{n+1} \mapsto e_{n+1} \leq_2 e_{n+1}$.
- If $e_n \mapsto e_{n+1}$, by real-time, either $e \mapsto e_{n+1}$ or $e_n \mapsto e'$ holds. In the first case, $e_1 \leq_1 e \mapsto e_{n+1} \leq_2 e_{n+1}$. In the second case, $e_n \leq_2 e' \leq_2 e_n$, so $e' = e_n$. By transitivity of $\mapsto$, $e_1 \leq_1 e \mapsto e_{n+1} \leq_2 e_{n+1}$.

Let $e_1, e_2 \in E$ such that $e_1 \bullet e_2$ and $e_2 \bullet e_1$. Let us prove that $e_1 = e_2$.

- Suppose $e_1, e_2 \in E_1$ (the case $e_1, e_2 \in E_1$ is symmetric). We have $e_1 \leq_1 e \mapsto e' \leq_2 e_2 \leq_1 e'' \mapsto e''' \leq_2 e_1$, so $e_1 \leq_1 e \leq_1 e' = e_2 \leq_1 e'' \leq_1 e''' = e_1$ and $e_1 = e_2$.
- Suppose now that $e_1 \in E_1$ and $e_2 \in E_2$ (which proves the case $e_1 \in E_2$ and $e_2 \in E_1$ by symmetry). We have $e_1 \leq_1 e \mapsto e' \leq_2 e_2 \leq_2 e'' \mapsto e''' \leq_1 e_1$. By real-time, we have either $e \mapsto e'''$ or $e'' \mapsto e'$. Suppose that $e \mapsto e'''$ (the other case is symmetric). We have $e_1 \leq_1 e \mapsto_1 e''' \leq_1 e_1$, so $e_1 = e = e'''$. Moreover, $e'' \mapsto e''' = e \mapsto e'$, which implies $e_2 = e'' = e'$. Finally, $e_1 = e \mapsto e' = e_2 = e'' \mapsto e''' = e_1$, so $e_1 = e_2$.

As $\bullet$ is a partial order, we can extend it in a total order $\leq$. $\text{lin}\left(H^{\leq}\right)$ contains exactly one word l, that is an interleaving of l_1 and l_2. As $l_1 \in L(T_1)$ and $l_2 \in L(T_2)$, $l \in L(T_1 \times T_2)$. Finally, $H \in SC(T_1 \times T_2)$. □

3.2 Pipelined consistency

Pipelined consistency is an extension of pipelined random access memory (PRAM) [LS88] for other data types. It allows the processes to be aware of some, but not all, of the events. The definition of pipelined consistency is very close to those of local consistency, a very easily understood consistency criterion that we introduce first to illustrate the formalism needed for pipelined consistency.

Local consistency Local consistency is the criterion respected by local variables. A local variable contains an object of type T that is not shared on the network. All the events on it are done by a sequential process, so they form a maximal chain p in the history. We recall that the maximal chains of the history H are contained into $\mathcal{P}_H$. This means that $\text{lin}(H_p)$ is a singleton $\{l\}$ that only contains the sequential history seen by p. Local consistency requires that this history is correct with respect to the sequential specification of the object, i.e. $l \in L(T)$. More formally, Definition 8 requires that $\text{lin}(H_p) \cap L(T)$ is not empty, as it must contain l.

Definition 8. A history H is *locally consistent* (LC) with an UQ-ADT T if $\forall p \in \mathcal{P}_H, \text{lin}(H_p) \cap L(T) \neq \emptyset$.

(a) $SC^\star$ but not PC (b) PC but not $SC^\star$

Figure 5: These histories on $\mathcal{M}_{x,y}$ show that PC is not comparable with $SC^\star$

The history on Figure 4 is locally consistent. It represents a graph edited by a thread. At one point, this thread forks and the graph is edited separately by the father thread and its son. There are two maximal chains in this history, and the first events are part of both. The operations made by one thread are ignored by the other thread. Local consistency does not make sense when threads are allowed to join, because one value must be discarded. If no join is allowed for the histories, for example because the computation model is based on parallel sequential processes, local consistency is composable, in the manner of sequential consistency with real time.

Pipelined consistency Pipelined consistency is close in its definition to local consistency, but the processes are also aware of the updates made by the other processes, and of the order in which they are made. Each process must be able to explain the history individually by a linearization of their own knowledge. The consistency is local to each process, as different processes can see concurrent updates in a different order.

Definition 9. A history H is *pipelined consistent* (PC) with an UQ-ADT T if $\forall p \in \mathcal{P}_H, \mathrm{lin}(H_{U_H \cup p}) \cap L(T) \neq \emptyset$.

Pipelined consistency is weaker than sequential consistency, for which the linearizations seen by different processes must be identical. It is not comparable with cache consistency, as illustrated on Figure 5. On the graph, that cannot be decomposed into simpler data types, cache consistency is equal to sequential consistency so we illustrate this on a shared memory.

For memory, cache consistency is very weak as finite locally consistent histories are also cache consistent. The history of Figure 5a is also cache consistent since $r_x/0.w_x(1)$ and $w_y(1).r_y/1$ are correct linearizations of the sub-histories that only consider one register at a time. It is not pipelined consistent because there is no linearization of all the events, that is required for the second process.

As there is only one register in the history of Figure 5b, all the events must be considered in the same linearization for cache consistency, which is not possible. However, it is pipelined consistent: $w_x(1).w_x(2).r_x/2$ is a linearization for the first process and $w_x(2).w_x(1).r_x/1$ is a linearization for the second process.

We now prove a negative result on composability: there exists no composable consistency criterion between pipelined consistency and sequential consistency.

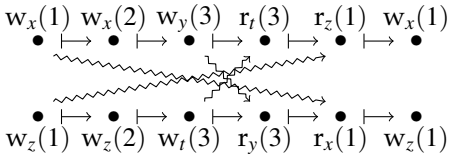
Proposition 2. $\forall C, PC \leq C \leq SC \Rightarrow C^2 \neq C$.

Proof. If there existed a C such that $PC \leq C \leq SC$ and $C = C^2$, we would have $PC \leq C = C^2 = C^\star \leq SC^\star$. The example of Figure 5a proves that $PC \not\leq SC^\star$, so such a C does not exist. □

3.3 Causal Consistency

Causal Consistency [ANB+95] is an exception as it cannot be written easily in our model. Indeed, causal consistency was only defined for memory, and its definition uses the semantics of read and write operations, which makes it difficult to extend. We recall here the definition of causal memory.

Causal memory Causal Consistency is an intermediate criterion between pipelined consistency and sequential consistency. Pipelined consistency is only local; each process has a consistent vision of the events it is aware of, but they can disagree on the order in which updates happen.



$l_1 = w_x(1)w_x(2)w_y(3)w_z(1)w_z(2)w_t(3)r_t(3)w_z(1)r_z(1)w_x(1)$

$l_2 = w_x(1)w_x(2)w_y(3)w_z(1)w_z(2)w_t(3)r_y(3)w_x(1)r_x(1)w_z(1)$

Figure 6: This writes-into order is not a correct explanation of the data dependencies in this history

On the other hand, sequential consistency imposes a total order on all the operations of the history. Causal consistency supposes the existence of a logical time, the causal order, composed of two kinds of dependences. On the one hand, the program order must be respected: like in pipelined consistency, if two operations happen on the same process, all the processes that see both operations must see them in the same order. On the other hand, two events related by data dependencies must be ordered in the causal order. More precisely, if a read returns the value written by a write, these events are related by a *writes-into* order that can affect the linearizations of all processes. We now recall the formal definition of causal memory.

Definition 10. Let $\mathcal{M}_X$ be a memory update-query abstract data type. A relation $\rightsquigarrow$ is a writes-into order if:

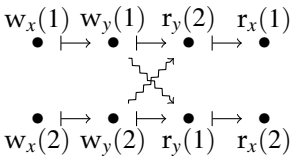
- for all $e, e' \in E$ such that $e \rightsquigarrow e'$, there are $x \in X$ and $n \in \mathbb{N}$ such that $\Lambda(e) = w_x(n)$ and $\Lambda(e') = r_x/n$,
- for all $e \in E$, $|\{e' \in E : e' \rightsquigarrow e\}| \leq 1$,
- for all $e \in E$ such that $\Lambda(e) = r_x/n$ and there is no $e' \in E$ such that $e' \rightsquigarrow e$, then $n = 0$.

A history H is $\mathcal{M}_X$*-causal* if there exists a writes-into order $\rightsquigarrow$ such that:

- the transitive closure $\rightarrow$ of $\rightsquigarrow \cup \mapsto$ is a partial order,
- $\forall p \in \mathcal{P}_H, \mathrm{lin}\left(H^{\rightarrow}_{U_H \cup p}\right) \cap L(\mathcal{M}_X) \neq \emptyset$.

Limits of causal memory The first limit of causal memory is that it cannot be easily extended to other data types. Actually, the definition of writes-into orders is deeply related to the semantics of registers. For other abstract data types, e.g. graphs, counters or stacks, the value returned by a query does not depend on one particular update, but on all the updates that happened before. Moreover, in the case of the stack, these updates must be sorted to take into account the order of the elements in the stack. It might be possible to define a writes-into order for each of these objects individually, but this approach cannot be used in a data type-independent definition of causal consistency.

The second limit comes from the fact that the writes-into order is not unique. This weakens the role of the logical time, as the intuition that a read must be bound to its corresponding write is not always captured by the definition. Let us illustrate that point with the history on Figure 6. In this history, we consider the writes-into order in which the reads on x and z are related to the first write of the other process. This writes-into order is correct, as each read is related to exactly one write, and the variables and the values are the same. Moreover, l_1 and l_2 are correct linearizations, so this history is causally consistent. However, in these linearizations, the value read by the two last reads was not written by their predecessors in the writes-into relation, but if we change this relation to restore the real data dependencies, we obtain a cycle in the causal order. This example shows that the approach of Definition 10, that uses the semantics of the operations, is not well suited to define the consistency criteria. This issue is solved in [Mis86] by the hypothesis that all written values are distinct. Even if this can be achieved by the addition of unique timestamps



$l_3 = w_x(2)w_x(1)w_y(1)w_y(2)r_y(2)r_x(1)$

$l_4 = w_x(1)w_x(2)w_y(2)w_y(1)r_y(1)r_x(2)$

Figure 7: In the case of concurrent writes, at least one process should eventually see the value of the other process

on the values stored by the memory, this solution is not acceptable because it changes the way the final object can be used. Who could accept to sacrifice the ease of use only for the need of specification?

Another example that cannot be solved by stamping is illustrated by Figure 7. The causal order from Definition 10 only considers write-read dependencies, while concurrent writes are not considered at all. In the example, there is no ambiguity on the writes-into order, as all the values written in each variable are different. The linearizations l_3 and l_4 show that this history is consistent for causal memory. Nevertheless, many people, thinking of causal reception, would reject it for causal consistency because it is impossible that the first process receives the notification message for $w_x(2)$ before it sends the one for $w_x(1)$, and the second process receives the notification message for $w_x(1)$ before it sends the one for $w_x(2)$: a message cannot be received before it is sent.

3.4 Eventual consistency

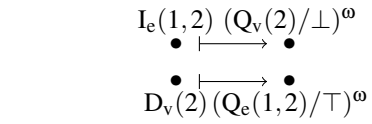
Eventual consistency [Vog08] is one of the few consistency criteria that was not originally designed for memory. It requires that, if all the participants stop updating, all the replicas eventually converge to the same state. In other word, H is eventually consistent if it contains an infinite number of updates (i.e. the participants never stop to write) or if there exists a state compatible with all but a finite number of queries (the consistent state).

In our formalism, a consistency criterion must impose to all the events to be labelled by correct operations of the data type. It is not clear in the literature whether it is the case for eventual consistency. A lot of work has been done to fully specify CRDTs [SPBZ11a], a kind of objects especially designed to achieve eventual consistency. In [BZP+12], it is explicitly mentioned that if an insertion and a deletion of the same element are done concurrently on the set, then any state can be specified as consistent state, including, for example, an error state. Moreover, no assumption is made on the queries made before convergence, so we can imagine that data inconsistencies are acceptable for a short amount of time. For example, a query operation that returns a local copy of the graph could return an edge starting from a vertex that does not exists. These exotic behaviors may cause issues with data integrity, and our definition does not allow them. It is necessary to explicitly modify the sequential specification, for example by adding unreachable states, to take them into account.

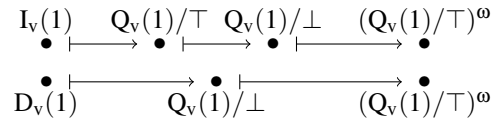
Definition 11. A history H is *eventually consistent* (EC) if it contains an infinite number of updates or there exists a state $s \in S$ such that the set $\{q_i/q_o \in Q_H : \varphi(s, q_i) \neq q_o\}$ of queries that cannot be issued while in the state s is finite.

The history of Figure 8a is not eventually consistent since there are only two updates and no valid state can contain the edge $(1,2)$ but not the vertex 2. The other histories of Figure 8 are eventually consistent since only one query is repeated infinitely often.

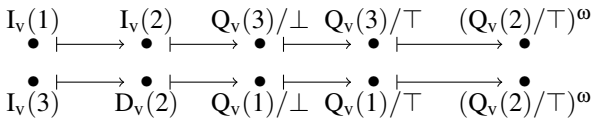
Strong eventual consistency Strong eventual consistency [SPB+11] requires that two replicas of the same object converge as soon as they have received the same updates. The problem with that definition is that the notions of replica and message reception are inherent to the implementation, and are hidden to the programmer that uses the object, so they should not be used in a



(a) CC but not EC



(b) EC but not SEC



(c) SEC but not PC

Figure 8: On these histories for an instance of $\mathcal{G}$, An event labelled ω is repeated an infinity of times. We can see that *SEC* $\neq$ *EC* and that *PC* and *SC* are not comparable with *EC* and *SEC*.

specification. To capture this, a visibility order is introduced to explain the history.

Definition 12. A history H is *strong eventually consistent* (SEC) if there exists an acyclic and reflexive relation $\rightarrow$ (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: $\forall u \in U_H, \{e \in E, u \not\rightarrow e\}$ is finite;
- Growth: if an event has been viewed once by a process, it will remain visible forever: $\forall e, e', e'' \in E, (e \rightarrow e' \wedge e' \mapsto e'') \Rightarrow (e \rightarrow e'')$;
- Strong convergence: if two query operations view the same past of updates V, they are issued in the same state s: $\forall V \subset U_H, \exists s \in S, \forall q_i/q_o \in Q_H$, $\{u \in U_H : u \rightarrow q_i/q_o\} = V \Rightarrow \varphi(s, q_i) = q_o$.

Strong eventual consistency is strictly stronger than eventual consistency. Figure 8b shows a history that is strong eventually consistent but not eventually consistent. The update $I_e(1)$ 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 ($\{I_e(1)\}$ and$\{I_e(1), D_e(1)\}$) for these events. By the growth property, the query event $Q_e(1)/\perp$ must have the same view as the previous event or the following event, which is impossible.

Eventual consistency and strong eventual consistency are not comparable with pipelined consistency and causal consistency. The history of Figure 8a is causally consistent but not eventually consistent. Conversely, the history of Figure 8c is strong eventually consistent but not pipelined consistent. To build the linearization of the first process, it is necessary to insert the $I_v(3)$ between the two $Q_v(3)$, but it is impossible to insert the $D_v(2)$ before any $Q_v(2)$. If these queries returned $\perp$, the history would be causally consistent but no more eventually consistent.

Eventual consistency can hardly be used directly to program reliable applications because it gives too few guaranties on the operations made before convergence. It can be used, however, for applications in which the object is controlled by humans. For example, it makes sense for a collaborative text editor like Logoot [WUM09] to ensure that all the collaborators will eventually see the same document.

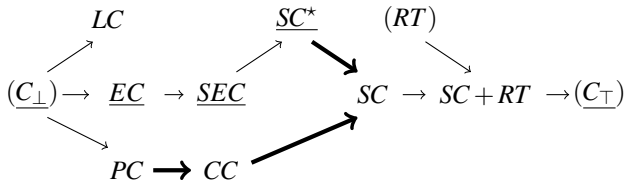


Figure 9: A summary of the criteria exposed in Section 3. The underlined criteria are composable. An arrow between C_1 and C_2 express the fact that $C_1 \leq C_2$. This arrow is thick if no composable consistency criteria exists along it

4 Survey of Weakly Consistent Systems

Figure 9 sums up all the consistency criteria evoked in Section 3. Some, like $C_\perp$, $C_\top$ and RT only have a theoretical interest, but others have real practical applications. For brevity, we cannot present this deep analysis for all the weak consistency conditions that have been proposed in the past. But we will now briefly review the most recent systems that offer various weak consistency guarantees. We first consider general purpose systems such as key-value stores, and then consider the special case of collaborative editing.

Consistency in distributed data stores. Amazon was one of the first companies to contribute to the success of key value stores. In 2007, they presented Dynamo [DHJ+07], the system they were using to manage their large-scale online-shopping infrastructure . The architecture of Dynamo resembles that of a distributed hash table (DHT) like Chord [SMK+01], although each Dynamo node maintains global membership information. Dynamo replicates data items over multiple nodes for availability and fault tolerance, but in order to scale, it adopts a quorum-based system and propagates write and read operations asynchronously to respectively W and R nodes, where $W + R > N$, N being the total number of replicas. This yields a form of eventual consistency: writes always go through, while reads may yield conflicting data versions that have to be reconciled. Specifically, Dynamo detects conflicting version by tagging each data version with a vector clock [Mat89]. If the vector clocks of two version on the same data item can be ordered, then Dynamo can safely discard the older. Otherwise, it leaves it to the application to reconcile the conflicting updates.

Cassandra [LM10] also provides a ring-based key-value data store, but unlike Dynamo it supports data with multiple columns. Cassandra uses quorums to provide different forms of consistency and fault tolerance that depend on the nature of the required quorums. More precisely, applications can specify for each operation a consistency guarantee that is dependent on the location of replicas: for instance the constraints LOCAL_QUORUM requires a quorum of replicas in the local data center, while EACH_QUORUM requires a quorum in each data center.

Subsequent work has proposed key-value stores that offer stronger semantic guarantees. Walter [SPAL11], for example, introduces a model called parallel snapshot isolation in the context of geo-distributed systems. Specifically, Walter provides snapshot isolation within each single site, but relaxes this property between transactions executed on different sites. Specifically, it allows them to exhibit different commit orderings on different sites, while at the same time guaranteeing the absence of write-write conflicts between committed transactions.

Others [CRS+08, TPK+13] have proposed data stores that allow programmers to choose different consistency guarantees in their code by offering an enriched API. In particular,Pileus [TPK+13] allows developers to establish service-level agreements in the trade-off between consistency and responsiveness. For example, a developer can specify that a request should be satisfied within a given delay and if possible with strong consistency guarantees. This allows the data store to dynamically offer the strongest consistency guarantee that satisfies the developer's request.

Salt [XSK+14] proposes a compromise between traditional ACID transactions and the BASE paradigm (Basically Available Soft-state Eventually consistent). Salt decouples atomicity from isolation by introducing BASE transactions, consisting of sequences of a new form of nested

transactions, *alkaline* transaction. Salt maintains atomicity and isolation between ACID transactions, and also between ACID and BASE transactions. However, it makes the committed state of alkaline transactions visible to other alkaline or BASE transactions. This yields an important performance improvements for those transactions that can get away without strong ACID semantics, while at the same time simplifying the implementation of transaction that require stronger guarantees.

Weak consistency guarantees in the face of distribution are linked to optimistic replication schemes [SS05], which can be understood as an early attempt to model and systematize eventual consistency. The idea of optimistic replication is to strive for (eventual) consistency across replicas, but still allow some level of inconsistency. These schemes, some of which dates back to early internet systems such as DNS or Usenet, typically include some form of delayed conflict detection and resolution, in a form that is reminiscent of eventually consistent objects.

Causal+ consistency [LFKA11] is a stronger version of causal consistency, that combines causal and eventual consistency criteria by insuring that replicated items are causally consistent, but also eventually converge to a state reflecting a total order of operations. The convergence is guaranteed by resolving conflicting updates to the same data item in an identical manner across all replicas. In the system proposed by Lloyd *et al.*, called COPS (*Clusters of Order-Preserving Servers*) Causality is implemented using version numbers, and eventual convergence using Lamport timestamps to impose an eventual global order on all write operations.

Almeida, Leitão, and Rodrigues have proposed ChainReaction [ALaR13], a geo-replicated data store that also implement causal+ consistency, but this time by building on a method known as *chain replication*. In each datacenter, ChainReaction organizes replicas in a ring DHT in which data items are replicated across k successive nodes. The approach then adapts a method of linearization called *chain replication* [VRS04]. *Chain replication* provides linearizability by serializing read and write requests respectively either to the head, or the tail of the chain of servers handling the corresponding data items. ChainReaction adapts this schemes by exploiting intermediate servers for read operations in a way that insures causal+ consistency within a datacenter. Causal+ consistency is then insures across datacenters by keeping track of the propagation of updates in different datacenters in a *chain index vector*. This vector is used to detect when a new version of an item has stabilized across all datacenters.

Convergent Replicated Datatypes [SPBZ11b] (or CRDT) are distributed abstract data types (i.e. abstract data types whose operations are invoked from distributed nodes) that provide guarantees of eventual convergence towards a well-defined distributed state. In survey presented by Shapiro, Preguiça, Baquero, and Zawirski distinguished between state-based and operation-based CDRDTs. The simpler form, state-based CRDTs are formally defined on the basis of a semi-lattice, i.e. a partial order $\leq$ in which a *least upper bound* operation (LIB, usually noted $\sqcup$) returns the smallest element that is higher than both operands. Said differently, for any x and y, the set $\{z|x \leq z \wedge y \leq z\}$ admits a smallest element, which is $x \sqcup y$. A semi-lattice induces a state-based CRDT that insure that all replicas of the CRDT converge to the LUB of its initial and updates values by propagating updates through the replicas and using the LUB operation (termed *merge* in this case [SPBZ11b]) to combine all values into a uniform, converged final result. State-based CRDTs realize a data types whose operations are commutative and associative. Operation-based CRDTs are data types that contain some operations that do not commute and must add further constrains to guarantee convergence. These additional constrains come in the form of a delivery order (typically an order that is weaker or equivalent to causal order) imposes on the propagation of operations that do not commute, but yet converge if this delivery order is respected. A number of data types can be implemented using the CDRTs frameworks, including counters, registers, sets, graphs, and text [SPBZ11b]. These datatypes often do not have the exact same properties of the traditional data type they take their name of (e.g. in the 2-phase set, an element can only be added and removed once), which allow them to provide the convergence property without recurring to delivery orders stronger than causality.

Bolt-on causal consistency [BGHS13] is a mechanisms proposed by Bailis, Ghodsi, Hellerstein,

and Stoica to add causal consistency as a generic mechanism on top of an eventually consistent data store (noted ECDS in the original publication). The idea is to extend an ECDS with a shim layer that keeps a local view of the system's $(key, values)$ pair while insuring that this local view always forms a *causal cut*. The notion of causal cut is extended from that of consistent cut often found in distributed checkpoint algorithm, and guarantees that the values read from the local view respect causality. The shim then proactively updates this view while tracking write causal dependencies, in practice delaying updates that are visible in the ECDS, but for which all causal dependencies have not been resolved yet. Conceptually, this treats the ECDS as an unreliable broadcast medium, with an additional property of eventual delivery.

In their work, Burckhardt *et al.* [BGYZ14], present a formal approach derived from abstract data types to specify formally the behavior of complex CRDTs. The approach uses the notion of *operations context* (the set of events perceived) to describe the axioms an single data-type must respect. It then introduce consistency axioms to describe the allowed behavior of a whole data store, in which multiple objects might co-exist. The author use this formalism to propose a method of verification of the implementation of CDRTs that abstracts away some the complexity induced by distribution and replication when trying to link an abstract specification and a concrete algorithmic description. Finally the same work offers lower bounds on the size of the meta-data required to implement different data types, and a method to obtain these bounds.

Consistency for collaborative editing. Molli et al. [OUMI06] proposed WOOT, a set of algorithms for maintaining consistency in the context of a distributed collaborative editing platform. The authors observe that while causal consistency represents a desirable and necessary condition for collaborative editing, it does not suffice by itself. Not only must participating nodes agree on the relative order of causally related operations (character insertions/deletions), but they must also agree on a correct character ordering for the final text sequence. The authors model this need as the combination of three properties: causality, convergence, and intention preservation. Causality and convergence could be provided by a total-ordering protocol such as Lamport's *logical clocks* [Lam78] model, but this would introduce spurious potential dependencies that would not satisfy intention preservation. WOOT, on the other hand, only relies on semantic dependencies and therefore makes it possible to reorder operations that do not lead to changes in the final output value.

Preguiça et al. [PMSL09] also propose a consistency framework for collaborative editing by exploiting the concept of commutative replicated data type. They propose a tree structure, Treedoc, that encodes a document in the form of an infix-ordered path in a binary tree. To support concurrent edits, they employ mini-nodes, which aggregate characters that are inserted concurrently at the same position. The framework associates each (mini) node with a disambiguator—for example, a globally unique identifier—which is uses to establish an ordering between mini-nodes inserted at the same position. Unlike WOOT, the use of Unique identifiers in the TreeDoc structure makes it possible to delete tree nodes associated with deleted characters, thereby limiting memory consumption for frequently modified documents.

Enforcing consistency criteria. Another important topic consists in understanding how to best enforce consistency guarantees during the software development process. To this end [ABCH13] provides a nice overview of levels at which consistency criteria can met within an application. The paper considers two extremes: I/O-level and application-level approaches. The former include traditional techniques such as state-machine replication [Lam84], group-communication [BJ87], and all the different consistency guarantees offered at the storage level [CDE+12]. The latter involves embedding consistency code directly into each application in a customized fashion.

The authors of [ABCH13] argue that both these approaches are unsatisfactory because they either underfit, or overfit application needs. To this end, they give the example of two applications that operate on a graph: a deadlock detector, and a garbage collector. The former only considers stable properties that persist once they are established; as a result, it can work with very weak consistency criteria that can reorder any operation. The latter, on the other hand, re-

quires that all the graph be explored before declaring an object as unreferenced. A developer that wished to use the same consistency-aware graph library for both application would obtain an overly constrained deadlock detector, while one that decided to have application-specific code would end up with error-prone code duplication. The authors explore intermediate solutions that include object level approaches like CRDTs [SPBZ11a, SPB+11], flow-level approaches like Blazes [ACHM13], and language-level approaches like Bloom [ACHM11].

5 Conclusion

Sharing objects is essential to abstract communication complexity in large scale distributed systems. A lot of work has been done until now to specify many kinds of shared memory, but very few for other data types. In this survey, we proposed a framework to easily specify shared objects. This framework is based on a clear separation between sequential specifications and consistency criteria. The interest of this approach is that sequential specifications are easy to understand as they are already widely used in sequential object-oriented programming.

Programming in a modular way is very important for reliability because it helps focusing on simpler pieces of codes. Composability is required to put the pieces together in the final program. However, this property seems very difficult to achieve, in particular for the strongest criteria. In this paper, we have shown that there exists no composable consistency criterion between pipelined consistency and sequential consistency. The example of linearizability shows that it might be possible to add constraints on the composition to make another consistency criterion composable. We could imagine an algorithm to compose C-consistent objects with respect to a criterion C' such that $C^2 + C' = C + C'$. Such a C' always exists, as $C^2 + C = C + C$ but a weaker criterion could lead to more efficient implementations. We leave all this as future work.

References

[ABCH13] Peter Alvaro, Peter Bailis, Neil Conway, and Joseph M. Hellerstein. Consistency without borders. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 23:1–23:10, New York, NY, USA, 2013. ACM.

[ACHM11] Peter Alvaro, Neil Conway, Joe Hellerstein, and William R. Marczak. Consistency analysis in bloom: a calm and collected approach. In *CIDR*, pages 249–260. www.cidrdb.org, 2011.

[ACHM13] Peter Alvaro, Neil Conway, Joseph M. Hellerstein, and David Maier. Blazes: Coordination analysis for distributed programs. *CoRR*, abs/1309.3324, 2013.

[AG96] Sarita V Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *computer*, 29(12):66–76, 1996.

[ALaR13] Sérgio Almeida, João Leitão, and Luís Rodrigues. Chainreaction: A causal+ consistent datastore based on chain replication. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 85–98, New York, NY, USA, 2013. ACM.

[ANB+95] Mustaque Ahamad, Gil Neiger, James E Burns, Prince Kohli, and Phillip W Hutto. Causal memory: Definitions, implementation, and programming. *Distributed Computing*, 9(1):37–49, 1995.

[AW94] Hagit Attiya and Jennifer L Welch. Sequential consistency versus linearizability. *ACM Transactions on Computer Systems (TOCS)*, 12(2):91–122, 1994.

[AW04] Hagit Attiya and Jennifer Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. John Wiley & Sons, 2004.

[BGHS13] Peter Bailis, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Bolt-on causal consistency. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 761–772, New York, NY, USA, 2013. ACM.

[BGYZ14] Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. Replicated data types: Specification, verification, optimality. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 271–284, New York, NY, USA, 2014. ACM.

[BJ87] K. Birman and T. Joseph. Exploiting virtual synchrony in distributed systems. *SIGOPS Oper. Syst. Rev.*, 21(5):123–138, November 1987.

[Bre00] Eric A Brewer. Towards robust distributed systems. In *PODC*, page 7, 2000.

[BZP+12] Annette Bieniusa, Marek Zawirski, Nuno Preguiça, Marc Shapiro, Carlos Baquero, Valter Balegas, and Sérgio Duarte. Brief announcement: Semantics of eventually consistent replicated sets. In *Distributed Computing*, pages 441–442. Springer, 2012.

[CDE+12] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally-distributed database. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 261–264, Hollywood, CA, October 2012. USENIX Association.

[CRS+08] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1(2):1277–1288, August 2008.

[DHJ+07] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, October 2007.

[Fis85] Peter C. Fishburn. Interval graphs and interval orders. *Discrete Mathematics*, 55(2):135 – 149, 1985.

[GL02] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 33(2):51–59, 2002.

[GLL+90] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. *Memory consistency and event ordering in scalable shared-memory multiprocessors*, volume 18. ACM, 1990.

[Goo91] James R Goodman. *Cache consistency and sequential consistency*. University of Wisconsin-Madison, Computer Sciences Department, 1991.

[HS08] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[HW90] Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[Lam79] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *Computers, IEEE Transactions on*, 100(9):690–691, 1979.

[Lam84] Leslie Lamport. Using time instead of timeout for fault-tolerant distributed systems. *ACM Trans. Program. Lang. Syst.*, 6(2):254–280, April 1984.

[Lam86] Leslie Lamport. On interprocess communication. *Distributed computing*, 1(2):86–101, 1986.

[LFKA11] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with cops. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 401–416, New York, NY, USA, 2011. ACM.

[LM10] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.

[LS88] Richard J Lipton and Jonathan S Sandberg. *PRAM: A scalable shared memory*. Princeton University, Department of Computer Science, 1988.

[Lyn96] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[Mat89] Friedemann Mattern. Virtual time and global states of distributed systems. In M. Cosnard et. al., editor, *Parallel and Distributed Algorithms: proceedings of the International Workshop on Parallel & Distributed Algorithms*, pages 215–226. Elsevier Science Publishers B. V., 1989.

[Mea55] George H Mealy. A method for synthesizing sequential circuits. *Bell System Technical Journal*, 34(5):1045–1079, 1955.

[Mis86] Jayadev Misra. Axioms for memory access in asynchronous hardware systems. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(1):142–153, 1986.

[Mos93] David Mosberger. Memory consistency models. *ACM SIGOPS Operating Systems Review*, 27(1):18–26, 1993.

[OUMI06] Gérald Oster, Pascal Urso, Pascal Molli, and Abdessamad Imine. Data consistency for p2p collaborative editing. In *Proceedings of the 2006 20th Anniversary Conference on Computer Supported Cooperative Work*, CSCW '06, pages 259–268, New York, NY, USA, 2006. ACM.

[PMSL09] Nuno Preguica, Joan Manuel Marques, Marc Shapiro, and Mihai Letia. A commutative replicated data type for cooperative editing. In *Proceedings of the 2009 29th IEEE International Conference on Distributed Computing Systems*, ICDCS '09, pages 395–403, Washington, DC, USA, 2009. IEEE Computer Society.

[Ray12] Michel Raynal. *Concurrent Programming: Algorithms, Principles, and Foundations*. Springer Publishing Company, Incorporated, 2012.

[SMK+01] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '01, pages 149–160, New York, NY, USA, 2001. ACM.

[SPAL11] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. Transactional storage for geo-replicated systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 385–400, New York, NY, USA, 2011. ACM.

[SPB+11] Marc Shapiro, Nuno Preguiça, Carlos Baquero, Marek Zawirski, et al. A comprehensive study of convergent and commutative replicated data types. Technical report, INRIA, 2011.

[SPBZ11a] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *Stabilization, Safety, and Security of Distributed Systems*, pages 386–400. Springer, 2011.

[SPBZ11b] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Convergent and commutative replicated data types. *Bulletin of the EATCS*, 104:67–88, 2011.

[SS05] Yasushi Saito and Marc Shapiro. Optimistic replication. *ACM Comput. Surv.*, 37(1):42–81, March 2005.

[TPK+13] Douglas Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos Aguilera, and Hussam Abu-Libdeh. Consistency-based service level agreements for cloud storage. In *Proceedings ACM Symposium on Operating Systems Principles*. ACM, November 2013.

[Vog08] Werner Vogels. Eventually consistent. *Queue*, 6(6):14–19, 2008.

[VRS04] Robbert Van Renesse and Fred B Schneider. Chain replication for supporting high throughput and availability. In *OSDI*, volume 4, pages 91–104, 2004.

[WUM09] Stephane Weiss, Pascal Urso, and Pascal Molli. Logoot: a scalable optimistic replication algorithm for collaborative editing on p2p networks. In *Distributed Computing Systems, 2009. ICDCS'09. 29th IEEE International Conference on*, pages 404–412. IEEE, 2009.

[XSK+14] Chao Xie, Chunzhi Su, Manos Kapritsos, Yang Wang, Navid Yaghmazadeh, Lorenzo Alvisi, and Prince Mahajan. Salt: Combining acid and base in a distributed database. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 495–509, Broomfield, CO, October 2014. USENIX Association.