HAL
archives-ouvertes.fr

Conflict-free Replicated Data Types

Marc Shapiro, Nuno Preguiça, Carlos Baquero, Marek Zawirski

▶ **To cite this version:**

Marc Shapiro, Nuno Preguiça, Carlos Baquero, Marek Zawirski. Conflict-free Replicated Data Types. Xavier Défago and Franck Petit and Vincent Villain. SSS 2011 - 13th International Symposium Stabilization, Safety, and Security of Distributed Systems, Oct 2011, Grenoble, France. Springer, 6976, pp.386-400, 2011, Lecture Notes in Computer Science. <10.1007/978-3-642-24550-3_29>. <hal-00932836>

HAL Id: hal-00932836
https://hal.inria.fr/hal-00932836

Submitted on 17 Jan 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Conflict-free Replicated Data Types *

Marc Shapiro[1,5], Nuno Preguiça[2,1], Carlos Baquero[3], and Marek Zawirski[1,4]

[1] INRIA, Paris, France
[2] CITI, Universidade Nova de Lisboa, Portugal
[3] Universidade do Minho, Portugal
[4] UPMC, Paris, France
[5] LIP6, Paris, France

Keywords: Eventual Consistency, Replicated Shared Objects, Large-Scale Distributed Systems

Abstract. Replicating data under Eventual Consistency (EC) allows any replica to accept updates without remote synchronisation. This ensures performance and scalability in large-scale distributed systems (e.g., clouds). However, published EC approaches are ad-hoc and error-prone. Under a formal Strong Eventual Consistency (SEC) model, we study sufficient conditions for convergence. A data type that satisfies these conditions is called a Conflict-free Replicated Data Type (CRDT). Replicas of any CRDT are guaranteed to converge in a self-stabilising manner, despite any number of failures. This paper formalises two popular approaches (state- and operation-based) and their relevant sufficient conditions. We study a number of useful CRDTs, such as sets with clean semantics, supporting both *add* and *remove* operations, and consider in depth the more complex Graph data type. CRDT types can be composed to develop large-scale distributed applications, and have interesting theoretical properties.

1 Introduction

Replication and consistency are essential features of any large distributed system, such as the WWW, peer-to-peer, or cloud computing platforms. The standard "strong consistency" approach serialises updates in a global total order [10]. This constitutes a performance and scalability bottleneck. Furthermore, strong consistency conflicts with availability and partition-tolerance [8].

When network delays are large or partitioning is an issue, as in delay-tolerant networks, disconnected operation, cloud computing, or P2P systems, *eventual consistency* promises better availability and performance [17,21]. An update executes at some replica, without synchronisation; later, it is sent to the other

* This research is supported in part by ANR project ConcoRDanT (ANR-10-BLAN 0208). Marek Zawirski is supported in part by his Google Europe Fellowship in Distributed Computing 2010. Nuno Preguiça is partially supported by CITI (PEst-OE/EEI/UI0527/2011). Carlos Baquero is partially supported by FCT project Castor (PTDC/EIA-EIA/104022/2008).

replicas. All updates eventually take effect at all replicas, asynchronously and possibly in different orders. Concurrent updates may conflict; conflict arbitration may require a consensus and a roll-back.[6]

This weaker consistency is considered acceptable for some classes of applications. However, conflict resolution is hard. The literature offers little guidance on designing a correct optimistic system. Ad-hoc approaches are brittle and error-prone; witness for instance the concurrency anomalies of the Amazon Shopping Cart [3].

We propose a simple, theoretically-sound approach to eventual consistency. Our system model, Strong Eventual Consistency or SEC, avoids the complexity of conflict resolution and of roll-back. *Conflict-freedom* ensures safety and liveness despite any number of failures. It leverages simple mathematical properties that ensure absence of conflict, i.e., monotonicity in a semi-lattice and/or commutativity. A trivial example is a replicated counter, which (assuming no overflow) converges because its increment and decrement operations commute. In our *conflict-free replicated data types* (CRDTs), an update does not require synchronisation, and CRDT replicas provably converge to a correct common state. CRDTs remain responsive, available and scalable despite high network latency, faults, or disconnection.

Non-trivial CRDTs are known to exist: for instance, we previously published Treedoc, a sequence CRDT for co-operative text editing [14]. Our aim here is to expand our knowledge of the principles and practice of CRDTs. We claim the following contributions for this paper:

- A solution to the CAP problem, Strong Eventual Consistency (SEC).
- Formal definitions of Strong Eventual Consistency (SEC) and of CRDTs.
- Two sufficient conditions for SEC.
- A strong equivalence between the two conditions.
- We show that SEC is incomparable to sequential consistency.
- Description of basic CRDTs, including integer vectors and counters.
- More advanced CRDTs, including sets and graphs.

We refer the interested reader to a separate technical report [18] for further detail and for a comprehensive portfolio of CRDT designs.

2 System model

We consider a system of processes interconnected by an asynchronous network. The network can partition and recover. We assume a finite set $\Pi = \{p_0, \ldots, p_{n-1}\}$ of non-byzantine processes. Processes in Π may crash silently; a crashed process may remain crashed forever, or may recover with its memory intact. A non-crashed process is said *correct*.

[6] A conflict is a combination of concurrent updates, which may be individually correct, but that, taken together, would violate some invariant.

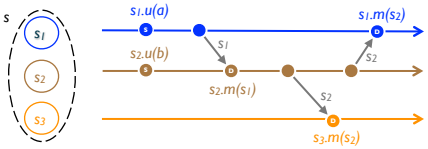

Fig. 1. State-based replication

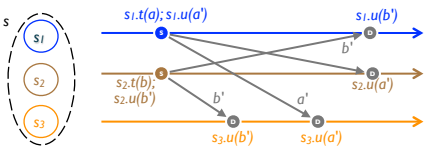

Fig. 2. Operation-based replication

2.1 State-based object

In this section we specify replicated objects in the so-called *state-based* style. The intuition is illustrated in Figure 1. Executing an update modifies the state of a single replica. Every replica occasionally sends its local state to some other replica, which *merges* the state thus received into its own state. In this way, every update eventually reaches every replica, either directly or indirectly.

With no loss of generality, we consider a single object with one replica at each process. An object is a tuple (S, s^0, q, u, m). The replica at process p_i has state $s_i \in S$, called its *payload*; the initial state is s^0. A client of the object may read the state of the object via *query* method q and modify it via *update* method u. Method m serves to *merge* the state from a remote replica. A method (whether q, u or m) executes at a single replica.

Systems that deliver every update to every replica eventually in a fault-tolerant manner are well-known in the literature, for instance gossip or anti-entropy approaches [5,13]. For simplicity, we will assume hereafter a fully connected communication graph, where every arc is a fair-lossy channel. Infinitely often, the replica at p_i sends (if it is correct) its current state to p_j; replica p_j (if it is correct) *merges* the received state into its local state by executing method m.

A method whose precondition is satisfied is said *enabled*. We assume that an enabled method executes as soon as it is invoked. Method executions at some replica are numbered sequentially from 1. The k^{th} method execution at replica i will be noted $f_i^k(a)$, where f is either q, u or m, and a denotes the arguments. We note $K_i(f)$ the ordinal of execution f at replica i, i.e., $K_i(f_j^k(a)) = k$ for $i = j$, and is undefined otherwise. (Abusing notation somewhat, we may drop subscripts, superscripts and/or arguments when there is no ambiguity.)

The states of a replica are numbered sequentially incrementing with each method execution. Thus, replica i has initial state $s_i^0 = s^0$. Before its k^{th} execution of a method it has state s_i^{k-1}, and s_i^k afterwards. We note the transition $s_i^{k-1} \bullet f_i^k(a) = s_i^k$.

We define state equivalence $s \equiv s'$ if all queries return the same result for s and s'. A query has no side-effects, i.e., $(s \bullet q) \equiv s$.

Definition 1 (Causal History (state-based)). *We define the object's* causal history $C = [c_1, \ldots, c_n]$ *(where c_i goes through a sequence of states $c_i^0, \ldots, c_i^k, \ldots$) as follows. Initially, $c_i^0 = \varnothing$, for all i. If the k^{th} method execution at i is:* (i) *a query q: the causal history does not change, i.e., $c_i^k = c_i^{k-1}$;* (ii) *an*

update (noted $u_i^k(a)$): it is added to the causal history, i.e., $c_i^k = c_i^{k-1} \cup \{u_i^k(a)\}$; (iii) *a merge $m_i^k(s_{i'}^{k'})$, then the local and remote histories are unioned together: $c_i^k = c_i^{k-1} \cup c_{i'}^{k'}$.*

We say that an update is *delivered* at some replica when it is included in the causal history at that replica. An update u *happened-before* u' iff u is delivered when u' executes: $u \to u' \stackrel{\text{def}}{=} u \in c_j^{k-1}$, where u' executes at replica p_j and $K_j(u') = k$. Updates are *concurrent* if neither happened-before the other: $u \parallel u' \stackrel{\text{def}}{=} u \not\to u' \wedge u' \not\to u$. Note that the causal history is a formal reasoning device, which is normally not needed in a concrete implementation.

Given our communication assumptions, we can conclude that, in a state-based object, every update is eventually delivered to all replicas. However, this is not sufficient to ensure that replicas converge. For instance, if the merge method m is a no-op, an update executed at some replica has no effect on other replicas, and they will never converge.

2.2 Strong Eventual Consistency

Informally, eventual consistency means that replicas eventually reach the same final value if clients stop submitting updates. We capture this intuition as follows:

Definition 2 (Eventual Consistency (EC)).
Eventual delivery: *An update delivered at some correct replica is eventually delivered to all correct replicas:* $\forall i, j : f \in c_i \Rightarrow \Diamond f \in c_j$.
Convergence: *Correct replicas that have delivered the same updates eventually reach equivalent state:* $\forall i, j : \Box c_i = c_j \Rightarrow \Diamond\Box s_i \equiv s_j$.
Termination: *All method executions terminate.*

Several EC systems will execute an update immediately, only to discover later that it conflicts with another, and to roll back to resolve this conflict [20]. This constitutes a waste of resources, and in general requires a consensus to ensure that all replicas arbitrate conflicts in the same way. To avoid this, we require a stronger condition:

Definition 3 (Strong eventual consistency (SEC)). *An object is Strongly Eventually Consistent if it is Eventually Consistent and:*
Strong Convergence: *Correct replicas that have delivered the same updates have equivalent state:* $\forall i, j : c_i = c_j \Rightarrow s_i \equiv s_j$.

2.3 State-based Convergent Replicated Data Type (CvRDT)

We now propose a sufficient condition for strong convergence in state-based objects. A join semilattice (or just semilattice hereafter) is a partial order $\leq$ equipped with a *least upper bound* (LUB) $\sqcup$ for all pairs: $m = x \sqcup y$ is a Least Upper Bound of $\{x, y\}$ under $\leq$ iff $\forall m', x \leq m' \wedge y \leq m' \Rightarrow x \leq m \wedge y \leq m \wedge m \leq m'$. It follows that $\sqcup$ is: commutative: $x \sqcup y = y \sqcup x$; idempotent: $x \sqcup x = x$; and associative: $(x \sqcup y) \sqcup z = x \sqcup (y \sqcup z)$.

Definition 4 (Monotonic semilattice object). *A state-based object, equipped with partial order* $\leq$*, noted* $(S, \leq, s^0, q, u, m)$*, that has the following properties, is called a monotonic semi-lattice:* (i) *Set* S *of payload values forms a semilattice ordered by* $\leq$*.* (ii) *Merging state* s *with remote state* s' *computes the LUB of the two states, i.e.,* $s \bullet m(s') = s \sqcup s'$*.* (iii) *State is monotonically non-decreasing across updates, i.e.,* $s \leq s \bullet u$*.*

Theorem 1 (Convergent Replicated Data Type (CvRDT)). *Assuming eventual delivery and termination, any state-based object that satisfies the monotonic semilattice property is SEC.*

For lack of space, we omit the proof that is presented in a companion technical report [19]. A CvRDT converges towards the LUB of the most recent updates. We require that $x \leq y \wedge y \leq x \Rightarrow x \equiv y$.

2.4 Op-based Commutative Replicated Data Type (CmRDT)

Alternatively to the state-based style, a replicated object may be specified in the *operation-based* (or op-based) style. An op-based object is a tuple (S, s^0, q, t, u, P), where S, s^0 and q have the same meaning as above (respectively state domain, initial state and query method). An op-based object has no merge method; instead an update is split into a pair (t, u), where t is a side-effect-free *prepare-update* method and u is an *effect-update* method. The prepare-update executes at the single replica where the operation is invoked (its *source).* At the source, prepare-update method t is followed immediately by effect-update method u, i.e., $f_i^{k-1} = t \Rightarrow f_i^k = u$. (If this were not true, there would be no causality between successive updates.)

The effect-update method executes at all replicas (said *downstream*). The source replica delivers the effect-update to downstream replicas using a communication protocol specified by the delivery relation P, explained below.

We use the same notations for states and causal history as above, except that now f can refer to any of q, t or u. Both queries and prepare-update methods are side-effect-free, i.e., $s \bullet q \equiv s \bullet t \equiv s$.

Definition 5 (Causal History (op-based)). *An object's* causal history $C = \{c_1, \ldots, c_n\}$ *is defined as follows. Initially,* $c_i^0 = \varnothing$*, for all* i*. If the* k^{th} *method execution at* i *is:* (i) *a query* q *or a prepare-update* t*, the causal history does not change, i.e.,* $c_i^k = c_i^{k-1}$*;* (ii) *an effect-update* $u_i^k(a)$*, then* $c_i^k = c_i^{k-1} \cup \{u_i^k(a)\}$*.*

An update is said delivered at a replica when the update is included in the replica's causal history. Update (t, u) happened-before (t', u') iff the former is delivered when the latter executes: $(t, u) \rightarrow (t', u') \Leftrightarrow u \in c_j^{k-1}$, where t' executes at p_j and $k = K_j(t')$. The definition of concurrent updates remains as above.

We assume an underlying reliable causally-ordered broadcast communication protocol, i.e., one that delivers every message to every recipient exactly once and in an order consistent with happened-before. Such protocols are a standard feature of distributed systems; they do not require consensus and they deliver

to all correct processes as long as any network partition eventually recovers (as we assumed earlier). It follows that two updates that are related by happened-before execute at all replicas in the same sequential order: $(t, u) \rightarrow (t', u') \Rightarrow \forall i, K_i(u) < K_i(u')$. However, concurrent updates may be delivered in any order.

Definition 6 (Commutativity). *Updates (t, u) and (t', u') commute, iff for any reachable replica state s where both u and u' are enabled, u (resp. u') remains enabled in state $s \bullet u'$ (resp. $s \bullet u$), and $s \bullet u \bullet u' \equiv s \bullet u' \bullet u$.*

Clearly, a sufficient condition for convergence of an op-based object is that all its concurrent operations commute. An object satisfying this condition is called a Commutative Replicated Data Type (CmRDT).

P is a delivery precondition, i.e., effect-update method u is enabled only if the precondition is satisfied. We interpret this temporally, i.e., delivery of u at replica i may delayed, until $P(s_i, u)$ is true. Therefore, for liveness, we now have the added obligation to prove that delivery is eventually enabled. Therefore we restrict our scope to preconditions for which causally-ordered broadcast is sufficient to ensure P.

Theorem 2 (Commutative Replicated Data Type (CmRDT)). *Assuming causal delivery of updates and method termination, any op-based object that satisfies the commutativity property for all concurrent updates, and whose delivery precondition is satisfied by causal delivery, is SEC.*

The proof is presented in [19].

3 Some results

3.1 Fault-tolerance and the CAP theorem

The CAP theorem states that it is impossible to simultaneously ensure strong consistency (C), availability (A) and tolerate network partition (P) [8]. As, network faults unavoidably occur in a large-scale environment, a real system must sacrifice either consistency or availability. Availability is often the top priority in practice [3]: does this mean giving up all consistency guarantees?

No: SEC provides a solution. A SEC replica is always available for both reads and writes, independently of network conditions. Any communicating subset of replicas of a SEC object eventually converges, even if partitioned from the rest of the network. SEC is weaker than strong consistency but nonetheless provides the well-defined guarantee of strong eventual convergence.

SEC provides an extreme form of fault tolerance, as a SEC object tolerates up to $n - 1$ simultaneous crashes. Remarkably, SEC does not require to solve consensus.

3.2 CvRDTs and CmRDTs are equivalent

Operation-based emulation of a state-based object

Theorem 3 (CmRDT emulation). *Any SEC state-based object can be emulated by a SEC op-based object of a corresponding interface.*

Proof. Given a CvRDT represented by tuple $(S, \leq, s^0, q, u, m)$, we emulate it by a CmRDT object (S, s^0, q, t, u', P), which we specify hereby.

State and query of CvRDT can be directly stored and processed by emulating CmRDT using the same definitions. A prepare-update $t(a)$ has the same interface (accepts the same domain of arguments and returns the same domain of value) as an update $u(a)$. It records the result of applying update $u(a)$ on a copy of current replica state s: $s' = s \bullet u(a)$; return value of $u(a)$ is passed to the client. Recorded state s' is used as an argument of an actual effect-update $u'(s')$, which is delivered to all replicas by the underlying protocol of CmRDT. Precondition P is unrestricted and enables delivery at any time. Effect-update $u'(s')$ merges received state using original CvRDT method: $s \bullet u'(s') \stackrel{\text{def}}{=} s \bullet m(s')$.

Since merge always commutes, then updates $u'(s')$ commute and since the communication is reliable, we have a CmRDT with strong eventual consistency, which propagates all updates of emulated CvRDT.

State-based emulation of an operation-based object State-based emulation of an operation-based object essentially formalises the mechanics of an epidemic reliable causal broadcast.

Theorem 4 (CvRDT emulation). *Any SEC op-based object can be emulated by a SEC state-based object of a corresponding interface.*

Proof. Given a CmRDT represented by tuple (S, s^0, q, t, u, P), we emulate it by a CvRDT object $((S \times U \times U), \leq, (s^0, \varnothing, \varnothing), q', u', m)$, which we specify hereby.

Without loss of generality, we assume that each invocation u_i^k is unique across replicas and set U denotes all possible updates. CvRDT's state is then defined as a triple (s_m, M, D), where s_m is a state of emulated CmRDT, M and D are two add-only sets of, respectively, known and delivered updates. A relation $\leq$ is defined as following: $(s_m, M, D) \leq (s'_m, M', D') \stackrel{\text{def}}{=} M \subseteq M' \wedge D \subseteq D'$.

A query $q'(a)$ has the same interface as $q(a)$; we define it as a trivial delegation to $q(a)$ on the CmRDT, $s_m \bullet q(a)$. An update $u'(a)$ has the same interface as prepare-update $t(a)$. It first delegates the invocation to prepare-update $t(a)$ of the CmRDT that in turn triggers effect-update $u(a)$, which becomes a locally known update. Finally, $u'(a)$ uses a recursive function d to process updates:

$$d(s_m, M, D) \stackrel{\text{def}}{=} \begin{cases} d(s_m \bullet u(a), M, D \cup \{u(a)\}) & \text{if } \exists u(a) \in M \setminus D : P(s_m, u(a)) \\ (s_m, M, D) & \text{otherwise} \end{cases}$$

Hence, $u'(a)$ is defined as: $(s_m, M, D) \bullet u'(a) \stackrel{\text{def}}{=} d(s_m \bullet t(a), M \cup \{u(a)\}, D)$.

Finally, merge m takes a union of known messages and processes available updates: $(s_m, M, D) \bullet m(s'_m, M', D') \stackrel{\text{def}}{=} d(s_m, M \cup M', D)$.

Since the emulation ensures that messages are delivered exactly once to each replica's embedded object, in the appropriate order, and since the CvRDT conforms to SEC criteria, the embedded CmRDT instance is also SEC.

Note that the emulating object forms a monotonic semilattice over domain $S \times U \times U$. Calling or delivering an operation adds it to the relevant message set, and therefore advances the state in the partial order. The merge method m is defined to take the union of the M sets and (possibly) updating D, and is thus a LUB operation. This construction is similar to Wuu and Bernstein's log covered in Section 4.2.

3.3 SEC is incomparable to sequential consistency

A state-based replica executes a sequence of query, update, and merge methods. In addition to its sequential behaviour, a CRDT specifies concurrent behaviours that must satisfy the strong convergence property. As we show now, this permits executions that would be impossible in a sequentially-consistent system.

Consider a Set CRDT S with operations $add(e)$ and $remove(e)$. Immediately after $add(e)$, the state will satisfy $e \in S$; after $remove(e)$ the state satisfies $e \notin S$. In a sequential execution, the last update wins, e.g., after $remove(e) \rightarrow add(e)$ the state satisfies $e \in S$. Concurrent adds or removes of different elements are independent, e.g., after $add(e) \parallel remove(e')$ the state satisfies $e \in S \wedge e' \notin S$.

There is a choice of alternative semantics for concurrent updates of the same element. When concurrently adding and removing the same element, the add could win, or the remove could win, or the update of the replica with the highest IP address could win, or the state might be reset to a distinguished state $\perp$, and so on. All these alternatives satisfy the strong convergence condition, and any of them may be reasonable for some application.

Let us consider the add-wins alternative: after $add(e) \parallel remove(e)$ the state satisfies $e \in S$. Now consider the following scenario. Replica p_0 executes the sequence $add(e); remove(e')$. Concurrently, replica p_1 executes $add(e'); remove(e)$. Then, replica p_3 merges the state from p_0 and p_1. According to the concurrent specification, the final state at p_3 satisfies $e \in S \wedge e' \in S$. Such a state would never occur in a sequentially-consistent execution, in which either $remove(e)$ or $remove(e')$ must be last. Thus, there is a SEC object that is not sequentially consistent.

Now consider the converse. In the absence of crashes, a sequentially-consistent object is SEC. Indeed, sequential consistency is defined by a single order of operations, after which all replicas must terminate with the same state. However, in the general case, sequential consistency requires consensus, which cannot be solved in the presence of $n - 1$ crashes. Therefore, SEC is incomparable with sequential consistency.

4 Example CRDTs

We now recall some basic CRDTs that are known in the existing literature, which we will later compose to build higher-level objects. We will use state- or op-based specifications as most convenient. Generally, we find the state-based style more compact and easier to reason about formally, whereas the op-based style is often convenient for implementation.

4.1 Integer vectors and counters

Consider the state-oriented specification of a vector-of-integers object: $(\mathbb{N}^n, [0,\ldots,0], \leq^n, [0,\ldots,0], \mathit{value}, \mathit{inc}, \max^n)$. Vectors $v, v' \in \mathbb{N}^n$ are (partially) ordered by $v \leq^n v' \Leftrightarrow \forall j \in [0..n-1], v[i] \leq v'[i]$. A query invocation $\mathit{value}()$ returns a copy of the local payload. An update $\mathit{inc}(i)$ increments the payload entry at index i, that is, $s \bullet \mathit{inc}(i) = [s'[0], \ldots, s'[n-1]]$ where $s'[j] = s[j] + 1$ if $i = j$ and $s'[j] = s[j]$ otherwise. Merging two vectors takes the per-index maximum, i.e., $s \bullet \max^n(s') = [\max(s[0], s'[0]), \ldots, \max(s[n-1], s'[n-1])]$. We omit the proof that it is a CRDT.

If each process p_i is restricted to incrementing its own index $\mathit{inc}(i)$, this is the well-known vector clock [11].

An increment-only integer counter is very similar; the only difference being that query invocation $\mathit{value}()$ of a vector in state v returns $|v| \stackrel{\text{def}}{=} \sum_j v[j]$. We construct an integer counter that can be both incremented and decremented, by basically associating two increment-only counters I and D, where incrementing increments I and decrementing increments D, whereas $\mathit{value}()$ returns $|I| - |D|$. The ordering method $\leq$ is defined as $(I, D) \leq (I', D') \stackrel{\text{def}}{=} I \leq^n I' \wedge D \leq^n D'$.

4.2 U-Set, map and log

Another simple CRDT construct is an add-only set object $(S, \subseteq, \varnothing, \mathit{value}, \mathit{add}(e), \cup)$. The payload is any set; sets are ordered by inclusion. A query $\mathit{value}()$ returns a copy of the local payload. Update $\mathit{add}(e)$ adds element e to the set, i.e., $s \bullet \mathit{add}(e) = s \cup \{e\}$. It is well-known that sets ordered by $\subseteq$ form a semi-lattice with $\cup$ as the LUB operator. It is clearly monotonic by the definition of *add*. Therefore, the add-only set is a CRDT.

Wuu and Bernstein build further CRDTs by combination of these basic components [23]. They propose a set with both *add* and *remove* operations by associating two add-only sets A and R; adding an element adds it to A, removing it adds it to R; query $\mathit{value}()$ returns the set difference $A \setminus R$. (R is often called the tombstone set. A client is allowed to remove only an element that is currently in A). Note that they assume that every element is unique and added only once; we call their construct U-Set [18]. Wuu and Bernstein derive their Dictionary data type from U-Set in the obvious way.

A Log is a replicated object, whose payload contains a set (initially empty) of (event, timestamp) pairs. It assumes that each process maintains a vector clock

in the usual manner [11]. When an event e occurs at process i, the process invokes update $add(e)$; the update method updates the vector clock (say, to state v) and adds the pair (e, v) to the set. The timestamp ensures that each entry is unique. The merge method takes the union of the local and a remote set.

To avoid unbounded growth, Wuu and Bernstein propose a distributed garbage collection algorithm that discards unneeded entries. In order to tolerate $n-1$ crashes, only an entry that has been delivered to all processes may be discarded. If vector clock entry $v_i[j] = k$, this implies that process i has delivered all k first events of process p_j. Each replica maintains in its payload a copy of all remote vector clocks; for each remote site, the merge procedure keeps the largest version. Then, a replica may discard a log entry as soon as its timestamp is less than all the remote vector clocks. This algorithm does not require a consensus, but it is live only if no process is crashed. However, this may be acceptable, since the liveness of garbage collection does not impact the correctness of the main algorithm.

This algorithm may be adapted to other data types, for instance to discarding the A and R entries of a removed element in the U-Set.

5 Directed Graph CRDT

Now let us examine how one would design a more complex data type: a Directed Graph CRDT. Graphs are an important general-purpose data structure. Some important applications and algorithms work on graphs, e.g., shortest-path or web page-rank.

5.1 Thought experiment

To motivate our graph design, consider the "thought experiment" of designing a web search engine. The search engine uses a directed graph representing the web structure. This graph may be used, among other things, to compute page rank. Such an application processes large amounts of data and performs many updates. For efficiency and scalability, processing should be asynchronous; for responsiveness, processing should be incremental, as fast as each page is crawled. Processing should not require any synchronisation, e.g., transactions. A CRDT could be ideal.

We start with a Set CRDT containing some initial URLs to be crawled. A number of crawler processes run in parallel; each one removes some URL from the set and downloads it. (It might happen that the same page is downloaded twice but this does not impact correctness.)

When a crawler finds a new page, it executes the corresponding *addVertex*. For every page, it parses the links that it contains, comparing it with the page's previous version, if any, and executes the corresponding *addArc* and *removeArc* invocations. Finally, the URLs of the linked pages are added to the set to be crawled. Note that *addArc* must work even if the page at the tail of the arc has not yet been found (it might not even exist), but such an arc is not functional; a

payload set V, A -- *sets of pairs* { (element e, unique-tag w), ... }
  initial $\varnothing, \varnothing$ -- *V: vertices; A: arcs*
query *lookup* (vertex v) : boolean b
  let $b = (\exists w : (v, w) \in V)$
query *lookup* (arc (v', v'')) : boolean b
  let $b = (lookup(v') \wedge lookup(v'') \wedge (\exists w : ((v', v''), w) \in A)$
update *addVertex* (vertex v)
  prepare $(v) : w$
    let $w = unique()$ -- *$unique()$ returns a unique value*
  effect (v, w)
    $V := V \cup \{(v, w)\}$ -- *v + unique tag*
update *removeVertex* (vertex v)
  prepare $(v) : R$
    pre $lookup(v)$ -- *precondition*
    pre $\nexists v' : lookup((v, v'))$ -- *v is not the head of an existing arc*
    let $R = \{(v, w) | \exists w : (v, w) \in V\}$ -- *Collect all unique pairs in V containing v*
  effect (R)
    $V := V \setminus R$
update *addArc* (vertex v', vertex v'')
  prepare $(v', v'') : w$
    pre $lookup(v')$ -- *head node must exist*
    let $w = unique()$ -- *$unique()$ returns a unique value*
  effect (v', v'', w)
    $A := A \cup \{((v', v''), w)\}$ -- *(v', v'') + unique tag*
update *removeArc* (vertex v', vertex v'')
  prepare $(v', v'') : R$
    pre $lookup((v', v''))$ -- *$arc(v', v'')$ exists*
    let $R = \{((v', v''), w) | \exists w : ((v', v''), w) \in A\}$
  effect (R) -- *Collect all unique pairs in A containing arc (v', v'')*
    $A := A \setminus R$

Fig. 3. Directed Graph Specification (op-based)

lookup of the corresponding arc should fail. Similarly if a node has been removed, all arcs incident to the node disappear. In this way, the behaviour of our CRDT will be consistent with that of web pages, which are allowed to contain non-functional URLs. Once the linked page is created, the link become relevant, e.g., for navigation and for page-rank computation.

In the web application, the graph is very large; sending the state between replicas and merging would be very costly. Therefore, we choose an op-based approach.

5.2 Design alternatives for arc removal

A directed graph is a pair of sets (V, A), called vertices and arcs respectively, such that $A \subseteq V \times V$. Updates must maintain the invariant that the head and

tail vertices of an arc both exist. Therefore, adding an arc to A has the precondition that its two vertices are in V; conversely, a vertex may be removed only if it supports no arc; these are preconditions to prepare-update. Furthermore, the system must ensure that concurrent $addArc(v', v'') \parallel removeVertex(v')$ do not violate the invariant. Several alternatives may be considered: *(i)* Give precedence to $removeVertex(v')$: all edges to or from v' are removed as a side effect. This is easy to implement, by hiding any arc that includes a removed vertex. *(ii)* Give precedence to $addArc(v', v'')$: if either v' or v'' has been removed, it is restored. This requires recreating nodes that have being explicitly deleted. *(iii)* $removeVertex(v')$ is delayed until all concurrent *addArc* operations have executed. This requires synchronisation which violates the goals of asynchrony and fault tolerance. There is no perfect choice. Hereafter, we choose Option *(i)* because it is adequate in our application scenario.

5.3 Graph specification

Figure 3 shows our specification for a Directed-Graph CRDT. We prove that this object is indeed a CmRDT in [19].

This CRDT maintains two sets internally, one for the vertices and one for the arcs. To add a vertex v, the prepare-update method creates a unique identifier, w, and the effect-update method adds the pair (v, w) to the set of vertices. With this approach, each vertex has an unique internal identifier. If the same vertex is added twice, the two additions will be distinguished by their two unique identifiers. A lookup will mask the duplicates.

To remove vertex v, the prepare-update computes the set R of pairs that contain v, i.e., all copies known in the source replica; the effect-update method removes this same set R from the set of vertices in all replicas. As operations are delivered in causal order, when the effect-update method executes in some replica, for each pair in R, the correspondent *addVertex* operations has already executed. Thus, unlike the state-based solution of Section 4.2, a set need not keep tombstones.

If the same vertex is removed and added concurrently, the *addVertex* wins, as the new unique identifier is not included in the set computed by the remove's prepare-update. This approach is consistent with a sequential execution, as the a vertex can removed only if it is observed. The same approach is used for arcs.

To remove a vertex, the source replica checks that the vertex is observed, and also that it is not the head of any existing arc. Conversely, to add an arc, its head node must exist, but there is no check for existence of the tail. The lookup method will mask the existence of such an arc. However, if the tail is added later, then the arc becomes visible. Similarly, concurrent updates may remove a vertex that is the head of an arc. However, the lookup method will mask such an arc.

6 Comparison with previous work

Eventual consistency has been an active topic of research in highly-available, large-scale asynchronous systems [17,21]. Contrary to much previous work [3, for instance], we take a formal approach grounded in the theory of commutativity and semilattices.

The state-based approach was invented for register-like objects, where the only update operation is assignment. It is in wide use in file systems such as NFS, AFS or Coda, and in key-value stores such as Dynamo [3] and Riak. Op-based approaches are used when the cost of transferring state is too high, e.g., databases, and when operation semantics are important, e.g., cooperative systems such as Bayou [13] or IceCube [15].

Although the CRDT concept was identified only recently, related designs have been published before. Johnson et. al. invented the LWW-Register [9]. They propose a database of registers that can be created, updated and deleted, using the last-writer-wins (LWW) rule to arbitrate between concurrent changes. LWW ensures a total order of operations, at the cost of losing concurrent updates.

Concurrent editing uses the related concept of Operational Transformation (OT), due to Ellis and Gibbs [7]. To ensure responsiveness, a local operation executes immediately. Operations are not designed to commute; however, a replica receiving an update transforms it against previously-executed concurrent updates to achieve a similar result. OT algorithms for a decentralised architecture have been proposed; Oster et al. show that most of them are incorrect [12]. We believe that designing for commutativity from the start is cleaner and simpler.

The foundations of CvRDTs were introduced by Baquero and Moura [1]. We extend their work with CmRDTs and with a number of new results. The CRDT concept was invented by Shapiro and Preguiça on their work on Treedoc, a Sequence CRDT for concurrent editing [14]. Logoot is another Sequence CRDT that supports an *undo* mechanism based on a CRDT Counter [22].

Roh et al. [16] independently developed the related concept of Replicated Abstract Data Type. They generalise LWW to a partial order of updates, which they leverage to build several LWW-style classes.

Burckhardt and Leijen propose the Concurrent Revisions programming model for shared abstract data types, in which a forked revision runs in isolation until it joins again. Join is based on a three-way merge function [2]. They show that simple sequential merge functions exist for ADTs built upon Abelian groups. We have also demonstrated the relation between CRDTs and sequential consistency in a similar, but more loosely-coupled, replication model.

Ducourthial et al. study algebraic structures with specific properties in order to solve self-stabilisation problems [6]. They propose the so-called r-operator for "silent" tasks [4]. Strong convergence can be seen as as a silent task, given a limited number of disturbing updates. However, there are differences between the two approaches. Whereas a self-stabilising system must tolerate arbitrary memory corruption, a shared mutable object should change state durably only by executing update operations. Furthermore, whereas CvRDT states constitute a monotonic semi-lattice, the r-operator requries a total order.

7 Conclusion

We presented the concept of a CRDT, a replicated data type for which some simple mathematical properties guarantee eventual consistency. In the state-based style, the successive states of an object should form a monotonic semilattice, with merge computing a least upper bound. In the op-based style, concurrent operations should commute. Assuming only that the communication subsystem ensures eventual delivery (in causal order for op-based objects), CRDTs are guaranteed to converge towards a common, correct state, without requiring any synchronisation.

We presented some simple CRDT examples, such as sets, and detailed how to create a directed Graph CRDT, which might be used in a large-scale web search engine. Our data types have a clean and deterministic semantics in the presence of concurrent updates.

Eventual consistency is a critical technique in many large-scale distributed systems, including delay-tolerant networks, sensor networks, peer-to-peer networks, collaborative computing, cloud computing, and so on. However, work on eventual consistency was mostly ad-hoc so far. Although some of our CRDTs were known before in the literature or in the folklore, this is the first work to engage in a systematic study. We believe this is required if eventual consistency is to gain a solid theoretical and practical foundation.

Future work is both theoretical and practical. On the theory side, this will include understanding the class of computations that can be accomplished by CRDTs, the complexity classes of CRDTs, the classes of invariants that can be supported by a CRDT, the relations between CRDTs and concepts such as self-stabilisation and aggregation, and so on. On the practical side, we plan to implement the data types specified herein as a library, to use them in practical applications, and to evaluate their performance analytically and experimentally. Another direction is to support infrequent, non-critical synchronous operations, such as committing a state or performing a global reset. We will also look into stronger global invariants, possibly using probabilistic or heuristic techniques.

References

1. Baquero, C., Moura, F.: Specification of convergent abstract data types for autonomous mobile computing. Tech. rep., Departamento de Informática, Universidade do Minho (Oct 1997)
2. Burckhardt, S., Leijen, D.: Semantics of concurrent revisions. Programming Languages and Systems pp. 116–135 (2011)
3. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: Amazon's highly available key-value store. In: Symp. on Op. Sys. Principles (SOSP). Operating Systems Review, vol. 41, pp. 205–220 (Oct 2007)
4. Delaët, S., Ducourthial, B., Tixeuil, S.: Self-stabilization with R-operators revisited. Self-Stabilizing Systems pp. 68–80 (2005)

5. Demers, A.J., Greene, D.H., Hauser, C., Irish, W., Larson, J.: Epidemic algorithms for replicated database maintenance. In: Symp. on Principles of Dist. Comp. (PODC). pp. 1–12 (Aug 1987), also appears Op. Sys. Review 22(1): 8-32 (1988)
6. Ducourthial, B.: R-semi-groups: a generic approach for designing stabilizing silent tasks. In: Int. Conf. on Stabilization, Safety, and Security of Distributed Systems (SSS). pp. 281–295 (2007)
7. Ellis, C.A., Gibbs, S.J.: Concurrency control in groupware systems. In: Int. Conf. on the Mgt. of Data (SIGMOD). pp. 399–407 (1989)
8. Gilbert, S., Lynch, N.: Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. SIGACT News 33(2), 51–59 (2002)
9. Johnson, P.R., Thomas, R.H.: The maintenance of duplicate databases. Internet Request for Comments RFC 677, Information Sciences Institute (Jan 1976)
10. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Communications of the ACM 21(7), 558–565 (Jul 1978)
11. Mattern, F.: Virtual time and global states of distributed systems. In: Int. W. on Parallel and Distributed Algorithms. pp. 215–226 (1989)
12. Oster, G., Urso, P., Molli, P., Imine, A.: Proving correctness of transformation functions in collaborative editing systems. Rapport de recherche RR-5795, LORIA – INRIA Lorraine (Dec 2005)
13. Petersen, K., Spreitzer, M.J., Terry, D.B., Theimer, M.M., Demers, A.J.: Flexible update propagation for weakly consistent replication. In: Symp. on Op. Sys. Principles (SOSP). pp. 288–301 (Oct 1997)
14. Preguiça, N., Marquès, J.M., Shapiro, M., Leţia, M.: A commutative replicated data type for cooperative editing. In: Int. Conf. on Distributed Comp. Sys. (ICDCS). pp. 395–403 (Jun 2009)
15. Preguiça, N., Shapiro, M., Matheson, C.: Semantics-based reconciliation for collaborative and mobile environments. In: Int. Conf. on Coop. Info. Sys. (CoopIS). Lecture Notes in Comp. Sc., vol. 2888, pp. 38–55 (Nov 2003)
16. Roh, H.G., Jeon, M., Kim, J.S., Lee, J.: Replicated abstract data types: Building blocks for collaborative applications. Journal of Parallel and Dist. Comp. ((To appear) 2011)
17. Saito, Y., Shapiro, M.: Optimistic replication. ACM Comput. Surv. 37(1), 42–81 (Mar 2005)
18. Shapiro, M., Preguiça, N., Baquero, C., Zawirski, M.: A comprehensive study of Convergent and Commutative Replicated Data Types. Rapport de recherche 7506, Institut Nat. de la Recherche en Informatique et Automatique (INRIA), Rocquencourt, France (Jan 2011)
19. Shapiro, M., Preguiça, N., Baquero, C., Zawirski, M.: Conflict-free Replicated Data Types. Rapport de recherche 7686, Institut Nat. de la Recherche en Informatique et Automatique (INRIA), Rocquencourt, France (Jul 2011)
20. Terry, D.B., Theimer, M.M., Petersen, K., Demers, A.J., Spreitzer, M.J., Hauser, C.H.: Managing update conflicts in Bayou, a weakly connected replicated storage system. In: 15th Symp. on Op. Sys. Principles (SOSP). pp. 172–182 (Dec 1995)
21. Vogels, W.: Eventually consistent. ACM Queue 6(6), 14–19 (Oct 2008)
22. Weiss, S., Urso, P., Molli, P.: Logoot-undo: Distributed collaborative editing system on P2P networks. IEEE Trans. on Parallel and Dist. Sys. (TPDS) 21, 1162–1174 (2010)
23. Wuu, G.T.J., Bernstein, A.J.: Efficient solutions to the replicated log and dictionary problems. In: Symp. on Principles of Dist. Comp. (PODC). pp. 233–242 (Aug 1984)