Convergent and Commutative Replicated Data Types

Marc Shapiro, Nuno Preguiça, Carlos Baquero, Marek Zawirski

Bulletin

of the

European Association for

Theoretical Computer Science

EATCS
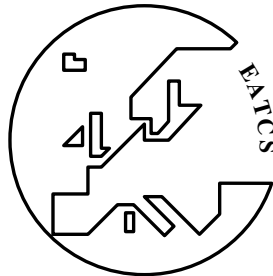
Number 104   June 2011

The Distributed Computing Column

by

Panagiota Fatourou

Department of Computer Science, University of Crete
P.O. Box 2208 GR-714 09 Heraklion, Crete, Greece
and
Institute of Computer Science (ICS)
Foundation for Research and Technology (FORTH)
N. Plastira 100. Vassilika Vouton
GR-700 13 Heraklion, Crete, Greece
`faturu@csd.uoc.gr`

Convergent and Commutative Replicated Data Types [*]

Marc Shapiro[†]   Nuno Preguiça[‡]

Carlos Baquero[§]   Marek Zawirski[¶]

[†]INRIA & LIP6, Paris, France
[‡]CITI, Universidade Nova de Lisboa, Portugal
[§]Universidade do Minho, Portugal
[¶]INRIA & UPMC, Paris, France

Abstract

Eventual consistency aims to ensure that replicas of some mutable shared object converge without foreground synchronisation. Previous approaches to eventual consistency are ad-hoc and error-prone. We study a principled approach: to base the design of shared data types on some simple formal conditions that are sufficient to guarantee eventual consistency. We call these types Convergent or Commutative Replicated Data Types (CRDTs). This paper formalises asynchronous object replication, either state based or operation based, and provides a sufficient condition appropriate for each case. It describes several useful CRDTs, including container data types supporting both *add* and *remove* operations with clean semantics, and more complex types such as graphs and monotonic DAGs. It discusses some properties needed to implement non-trivial CRDTs.

1   Introduction

Strong consistency protocols serialise updates in a global total order [5, 15]. This constitutes a performance and scalability bottleneck. Furthermore, strong consistency conflicts with availability and partition-tolerance [9].

When network delays are large or partitioning is an issue, as in delay-tolerant networks, disconnected operation, cloud computing, or P2P systems, *eventual consistency* has better availability and performance [23, 29]. An update happens at a replica, without synchronisation; then, it is sent to the other replicas. All updates eventually take effect at all replicas, asynchronously and possibly in different orders. Conflicts are resolved by a background consensus algorithm [3, 28]. This weaker consistency is considered acceptable for some classes of applications. However, conflict resolution is hard; there is little guidance on designing a correct optimistic system, and ad-hoc approaches are brittle and error-prone.[1]

We propose a simple, theoretically-sound approach to eventual consistency: to leverage simple mathematical properties that ensure absence of conflict: the values of state-based objects are monotonic in a semilattice, and the concurrent updates of operation-based objects commute. A trivial example is a replicated counter, which converges because its increment and decrement operations commute (assuming no overflow). Data types designed this way are called *convergent* or *commutative replicated data types* (CRDTs). CRDT updates do not require synchronisation, and its replicas provably converge to a common state that

---

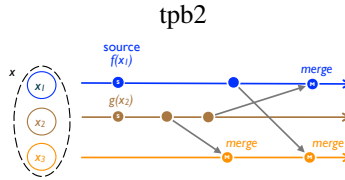[1] Consider for example the anomalies of the Amazon Shopping Cart [7].

Figure 1: State-based replication

is equivalent to some sequential execution. 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 [19]. Our aim here is to expand our knowledge of the principles and practice of CRDTs.

The contributions of this paper include the following: a specification language suited to asynchronous replication, a formalisation of state-based and operation-based replication, and two sufficient conditions for eventual consistency; example CRDTs, focusing on containers supporting both *add* and *remove* operations with clean semantics, and more complex types, such as graphs; a brief overview of the problem of garbage-collecting CRDTs; an exercise in applying CRDTs to a practical example, the bookstore shopping cart.

2   Background and system model

We consider a system of processes interconnected by an asynchronous network. The network can partition and recover, and nodes can operate in disconnected mode for some time. We assume non-byzantine processes that may crash and recover with their memory intact.

A process may store *atoms* and *objects*. An atom is an immutable datum identified by its literal content. Atoms can be copied between processes; atoms are equal if they have the same content. Atom types considered in this paper include integers, strings, sets, tuples, etc., with their usual non-mutating operations. Atom types are written in lower case, e.g., "set."

An object is a mutable, replicated datum. Object types are capitalised, e.g., "Set." An object has an identity, a content (its *payload*), which may be any number of atoms or objects, an initial state, and an interface consisting of *operations*. The replica of object x at process i is noted x_i. We assume that objects are independent and do not consider transactions; without loss of generality, we focus on a single object at a time, and use the words process and replica interchangeably.
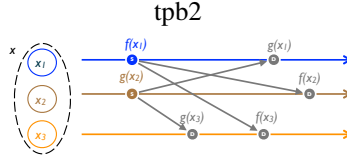
Figure 2: Operation-based replication

2.1 Asynchronous object replication

Unspecified *clients* access object state by calling operations against a replica of their choice, called the *source* replica. A query executes at a single replica. An update has two phases: in the *at-source phase*, the call executes at the source, assuming a *source precondition* is satisfied. We assume a communication subsystem that transmits updates from the source to all replicas; this enables the *downstream phase*.

Here, the literature [23] distinguishes two approaches, illustrated in Figures 1 and 2 respectively.[2] In the *state-based* approach, the first phase updates the source replica; the subsystem transmits state; the update takes effect at downstream replicas by merging the delivered state into the local state. In the *operation-based* approach (op-based for short), the at-source phase has no side-effects; when it terminates, the subsystem sends the update operation and its parameters to all replicas (including the source). When the *downstream precondition* is satisfied at a replica, the update takes effect by executing at that replica.

Definition 2.1 (Correctness)**.** *A replicated object must satisfy the following conditions:*

Termination: *For any call whose source precondition is satisfied, its at-source phase terminates. At any replica where an operation takes effect, the downstream phase terminates.*

Eventual effect: *An update that takes effect at some replica eventually takes effect at all replicas. Referring to the causal history C defined later: $\forall i, j : f \in C(x_i) \Rightarrow \Diamond f \in C(x_j)$.*

Convergence: *Replicas where the same updates took effect have equivalent state. Formally: $\forall i, j : C(x_i) = C(x_j) \Rightarrow x_i \equiv x_j$, where $x_i \equiv x_j$ if all queries return the same values for x_i and x_j.*

Proof obligations are as follows. Assuming the preconditions are true, termination should be apparent from the object's specification. We assume the communication system sends and delivers updates. In state-based objects, as merge

---

[2] The two approaches are equivalent, in the sense that one can emulate the other [27, Section 2.4].

is always enabled, this implies eventual effect. For op-based objects, we must prove that the downstream precondition is eventually enabled at every replica. We give hereafter sufficient conditions for convergence, and must prove that the object satisfies such conditions.

2.2 State-based CRDT: Convergent Replicated Data Type (CvRDT)

We use a specification language adapted to asynchronous replication. In state-based specifications (e.g., Figure 5), keyword payload indicates the payload type; initial specifies its initial value at every replica. Operations are indicated by keyword query or update. Non-mutating statements are marked let, and payload is mutated by assignment :=. An operation executes atomically; pre indicates a source pre-condition that must hold in the source's current state.

The communication subsystem transmits state between arbitrary replicas at arbitrary times. This updates the payload with *merge (local-state, delivered-state)*; thus delivered updates *take effect*. Operation compare compares replica states, as will be explained shortly.

Definition 2.2 (Causal History (state-based)). *The causal history C of replica x_i of state-based object x is defined as follows [24]: (a) Initially, $C(x_i) = \emptyset$. (b) Atomically with executing update operation f, $C(x_i) := C(x_i) \cup \{f\}$. (c) Atomically with merging against x_j, $C(x_i) := C(x_i) \cup C(x_j)$.*

An update is said to take effect at some replica when it is in the causal history. The classical happened-before [15] relation can be defined as $f \rightarrow g \Leftrightarrow (\forall i : g \in C(x_i) \Rightarrow f \in C(x_i))$.

We now propose a sufficient condition for convergence in state-based objects. A join semilattice [6] (or just semilattice hereafter) is a partial order $\leq_v$ equipped with a *least upper bound* (LUB) $\sqcup_v$ for all pairs.

Definition 2.3 (Least Upper Bound (LUB)). *$m = x \sqcup_v y$ is a Least Upper Bound of $\{x, y\}$ under $\leq_v$ iff $x \leq_v m$ and $y \leq_v m$ and there is no $m' \leq_v m$ such that $x \leq_v m'$ and $y \leq_v m'$.*

It follows that $\sqcup_v$ is: commutative: $x \sqcup_v y =_v y \sqcup_v x$; idempotent: $x \sqcup_v x =_v x$; and associative: $(x \sqcup_v y) \sqcup_v z =_v x \sqcup_v (y \sqcup_v z)$.

Consider a state-based object whose payload takes its values in a semilattice, where merge(x, y) returns $x \sqcup_v y$, and where state is monotonically non-decreasing according to $\leq_v$ (i.e., after an update, the payload is greater or equal to the value before). Let us call this combination "monotonic semilattice." This is a sufficient condition for the object to converge towards the LUB of the most recent updates.

A type with these properties will be called a Convergent Replicated Data Type or CvRDT. In a CvRDT, we require that compare(x, y) return $x \leq_v y$, that $x \leq_v y \wedge y \leq_v x \Rightarrow x \equiv y$, and that merge be always enabled.

Proposition 2.1. *Two CvRDT replicas eventually converge, assuming the communication subsystem delivers payload infinitely often between them.*

We refer to a companion technical report for the proof, which basically formalises the above discussion [27].

The communication subsystem of CvRDTs may have very weak properties. Since merge is idempotent and commutative, messages may be lost, received out of order, or multiple times, as long as new state eventually reaches all replicas, either directly or indirectly.

2.3 Op-based CRDT: Commutative Replicated Data Type (CmRDT)

In op-based specifications (e.g., Figure 12), the payload, initial and query clauses have the same meaning as in the state-based case. The at-source phase is marked atSource. Its (optional) source pre-condition, marked pre, must be true in the source state. It executes atomically. It is not allowed to make side effects, but it may send additional arguments downstream. The downstream phase executes at some replica only if and when its *downstream precondition* is true: immediately at the source, and after the update is delivered, at all other replicas. It updates the downstream state atomically; thus the update *takes effect*.

Definition 2.4 (Causal History (op-based)). *The causal history of replica x_i is defined as follows: (a) Initially, $C(x_i) = \varnothing$. (b) Atomically with executing the downstream phase of f at x_i, $C(x_i) := C(x_i) \cup \{f\}$.*

Again, happened-before is defined by $f \rightarrow g \Leftrightarrow (\forall i : g \in C(x_i) \Rightarrow f \in C(x_i))$. Operations are concurrent if not ordered by happened-before; formally: $f \parallel g \Leftrightarrow f \nrightarrow g \wedge g \nrightarrow f$.

Definition 2.5 (Commutativity). *Updates f and g commute, iff for any reachable replica state S where their downstream pre-condition is enabled, the downstream precondition of f (resp. g) remains enabled in state $S \cdot g$ (resp. $S \cdot f$), and $S \cdot f \cdot g \equiv S \cdot g \cdot f$.*

Causal delivery (defined as follows: at any replica, if $f \rightarrow g$ then f is delivered at any replica before g is delivered) is sufficient to ensure that the downstream precondition is true, for all objects in this paper, and operations take effect in that order. Thus, two operations that are causally related execute their downstream

phase in the same order at all replicas, and the final state is the same. Operations that are not related are concurrent; if they commute, the final states are equivalent.

Thus, 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).

Proposition 2.2. *Assuming a communication subsystem that reliably delivers updates in causal order, replicas of a CmRDT converge.*

Recall that reliable causal delivery does not require agreement. It is immune to partitioning, in the sense that replicas in a connected subset can deliver each other's updates, and that updates are eventually delivered to all replicas.

3 Example CRDTs

We now present a number of example CRDT designs: Registers, Sets, and Graphs. We refer the reader to a technical report for further examples, e.g., Counters, Maps, Monotonic DAGs, and Sequences [27].

Our specifications are written with clarity in mind, not efficiency. In many cases, there are clearly more efficient ways, but we preferred the more easily-understood version.

We write either state- or op-based specifications, as convenient. Proofs that objects fulfill the convergence conditions is generally trivial for the types hereafter.

3.1 Registers

A register is a memory cell storing an opaque atom or object (noted type *X* hereafter). It supports *assign* to update its value, and *value* to query it. Non-concurrent *assign*s preserve sequential semantics: the later one overwrites the earlier one. To make concurrent updates commute, two approaches are possible: either one takes precedence over the other (LWW-Register), or both are retained (MV-Register).

3.1.1 Last-Writer-Wins Register (LWW-Register)

A Last-Writer-Wins Register (LWW-Register) creates a total order of assignments by associating a timestamp with each update. Timestamps are assumed unique, totally ordered, and consistent with causality; i.e., if two assignments occur in happened-before order, the first one's timestamp is less than the second's [15].
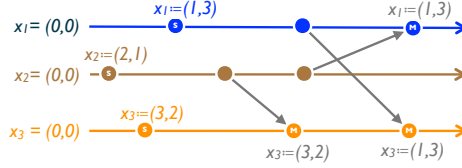
Figure 3: Integer LWW Register (state-based). Payload is a pair (value, timestamp)
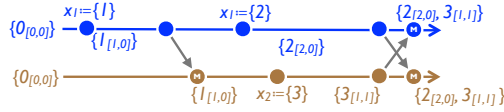
Figure 4: MV-Register (state-based). Notation "$1_{[1,0]}$" associates value 1 with version vector $[1, 0]$

This may be implemented as a per-replica counter concatenated with a unique replica identifier, such as its MAC address.

The state-based LWW-Register is presented in Figure 5, and the op-based specification in Figure 6. The value can be any data type X. Operation *value* returns the current value. Operation *assign* updates the payload with the new value, and generates a new timestamp. Values are ordered in the semilattice by their associated timestamp; merge selects the value with the highest timestamp. Figure 3 illustrates an integer LWW-Register.

LWW-Registers, first described by Thomas [13], are ubiquitous in distributed systems. For instance, in a replicated file system such as NFS, type X is a file (or even a block in a file).

```
payload X x, timestamp t                          -- X: some type
    initial ⊥, 0
update assign (X y)
    x, t := y, now()                    -- Timestamp, consistent with causality
query value () : X y
    let y = x
compare (R, S) : boolean b
    let b = (R.t ≤ S.t)
merge (R, S) : payload T
    if R.t ≤ S.t then T.x, T.t = S.x, S.t
    else T.x, T.t = R.x, R.t
```

Figure 5: State-based LWW-Register

```
    payload X x, timestamp t                          -- X: some type
       initial ⊥, 0
    query value () : X w
       let w = x
    update assign (X x′)
       atSource () :  t′
          let t′ = now()                              -- Timestamp
       downstream (x′, t′)
                        -- No downstream precondition: effect order is empty
          if t < t′ then x, t := x′, t′
```

Figure 6: Op-based LWW-Register

```
    payload set S                          -- (x, V) pairs; x ∈ X; V its version vector
       initial {(⊥, [0, . . . , 0])}
    query incVV () : integer[n] V′
                                        -- Compute dominating version vector
       let g = myID()                              -- Index of source replica
       let 𝒱 = {V|∃x : (x, V) ∈ S}
       let V′ = [ maxV∈𝒱(V[j]) ]j≠g
       let V′[g] = maxV∈𝒱(V[g]) + 1
    update assign (set R)                       -- set of elements of type X
       let V = incVV()
       S := R × {V}                             -- Assign S  with (x,vv) pairs
    query value () : set S′
       let S′ = {x|∃V : (x, V) ∈ S }
    compare (A, B) : boolean b
       let b = (∀(x, V) ∈ A, (x′, V′) ∈ B : V ≤ V′)
    merge (A, B) : payload C
                        -- Union of values not dominated by other replica
       let A′ = {(x, V) ∈ A|∀(y, W) ∈ B : V || W ∨ V ≥ W}
       let B′ = {(y, W) ∈ B|∀(x, V) ∈ A : W || V ∨ W ≥ V}
       let C = A′ ∪ B′
```

Figure 7: State-based Multi-Value Register (MV-Register)

```
    payload set A, set R                     -- A: added; R: removed
       initial ∅, ∅
    query lookup (element e) : boolean b
       let b = (e ∈ A ∧ e ∉ R)
    update add (element e)
       A := A ∪ {e}
    update remove (element e)
       pre lookup(e)
       R := R ∪ {e}
    compare (S, T) : boolean b
       let b = (S.A ⊆ T.A ∧ S.R ⊆ T.R)
    merge (S, T) : payload U
       let U.A = S.A ∪ T.A
       let U.R = S.R ∪ T.R
```

Figure 8: State-based 2P-Set

3.1.2   Multi-Value Register (MV-Register)

An alternative kind of register takes the union of concurrent assignments, as in file systems such as Coda [12] or in Amazon's shopping cart (clients can later reduce multiple values to a single one with a new assignment) [7], or more generally merges them, as in Ficus [20].

This Multi-Value Register (MV-Register) is specified in Figure 7, and illustrated in Figure 4. In order to detect concurrency, the payload is a set of *(X, versionVector)* pairs. A *value* operation returns a copy of the payload. To overwrite, *assign* stores a version vector that dominates all previous ones.[3] Operation merge takes the union elements not dominated by another one.

As noted in the Dynamo article [7], in Amazon's shopping cart, a removed book may re-appear. This is illustrated in the example of Figure 9. The problem is that the MV-Register does not behave like a set.
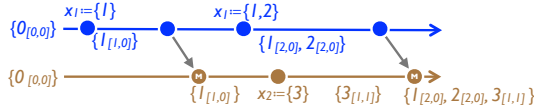
Figure 9: MV-Register counter-example
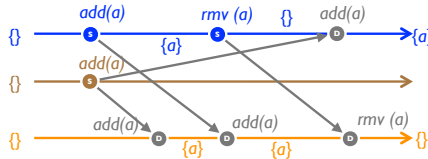
Figure 10: Counter-example: Set with concurrent *add* and *remove* (op-based)

3.2   Sets

We now present clean specifications of Sets. Sets constitute one of the most basic data structures. Containers, Maps, Graphs and Sequences are all based on Sets.

We consider mutating operations to *add* or *remove* an element. Unfortunately, the underlying union and set-minus do not commute with each other. Therefore,

---

[3] By symmetry with *value*, *assign* takes a set of values.
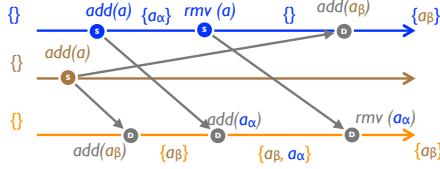
Figure 11: Observed-Remove Set (op-based)

a Set-like CRDT can only approximate the intuitive sequential specification. Figure 10 illustrates the issue with a naïve set implementation. Two replicas concurrently *add* and *remove* the same element, but the result depends on the order of delivery.

We now examine a few Set variants, which differ mainly in the result of concurrent *add*(*e*) with *remove*(*e*). The 2P-Set gives precedence to *remove*, OR-Set to *add*.

3.2.1 2P-Set

The simplest approach is the Add-Only-Set (G-Set), which avoids the problematic *remove* altogether [27, Section 3.3.1]. G-Set is useful as a building block for more complex constructions.

In a Two-Phase Set (2P-Set), an element may be added, then removed, but not added again, as specified in Figure 8. It combines a G-Set for adding with another for removing; the latter is colloquially known as the *tombstone set*.

```
payload set S                          -- Unique + causal delivery ⇒ no tombstones
    initial ∅
query lookup (element e) : boolean b
    let b = (e ∈ S )
update add (element e)
    atSource (e)
        pre e is unique
    downstream (e)
        S := S ∪ {e}
update remove (element e)
    atSource (e)
        pre lookup(e)                          -- 2P-Set precondition
    downstream (e)
        pre add(e) has been delivered          -- Causal order suffices
        S := S \ {e}
```

Figure 12: U-Set: Op-based 2P-Set with unique elements

77

```
        payload set S                        -- set of pairs { (element e, unique-tag u), …}
            initial ∅
        query lookup (element e) : boolean b
            let b = (∃u : (e, u) ∈ S )
        update add (element e)
            atSource (e)
                let u = unique()                          -- unique() returns a unique value
            downstream (e, u)
                S := S ∪ {(e, u)}                              -- e + unique tag
        update remove (element e)
            atSource (e)
                pre lookup(e)
                let R = {(e, u)|∃u : (e, u) ∈ S }
                                            -- Collect all unique pairs containing e
            downstream (R)
                pre ∀(e, u) ∈ R : add(e, u) has been delivered
                                       -- U-Set precondition; causal delivery suffices
                S := S \ R
                                   -- Downstream: remove pairs observed at source
```

Figure 13: Op-based Observed-Remove Set (OR-Set)

Figure 8 specifies a state-based 2P-Set. The payload is composed of sets A for adding, and R for removing. Operation *lookup*(e) checks that e has been added and not removed. Adding and removing a same element are idempotent; adding a removed element has no effect. The tombs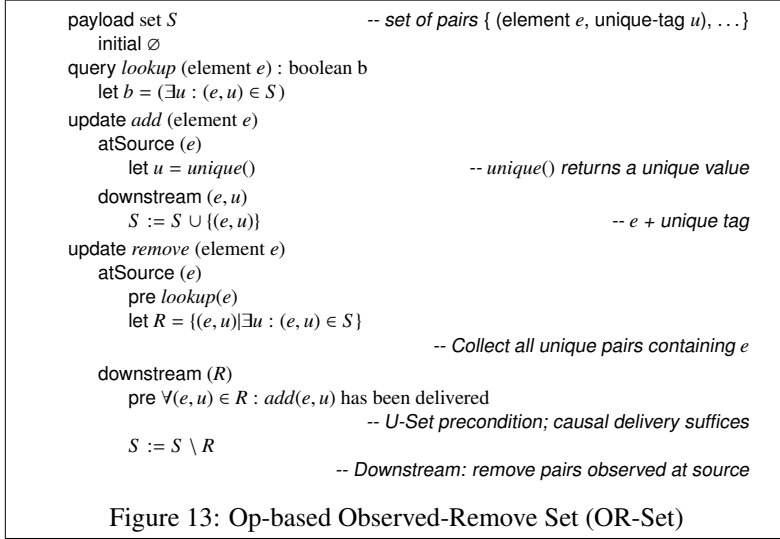tone ensures that *remove*(e) takes precedence over a concurrent *add*(e). Procedure merge takes the union of the individual added- and removed-sets, which is a LUB. Therefore, this is indeed a CRDT.

Consider now an op-based 2P-Set, under two simplifying (but standard) assumptions. If elements are unique, a removed element will never be added again. If *add*(e) is always delivered before *remove*(e), there is no need to record removed elements, and the remove-set is redundant. (Causal delivery is sufficient to ensure this precondition.) The specification in Figure 12 captures this data type, which we call U-Set.

3.2.2 Observed-Remove Set (OR-Set)

The preceding Set constructs are somewhat counter-intuitive. We present here the Observed-Removed Set (OR-Set), which does not limit adds and removes, and where the outcome depends only on its causal history and conforms to the sequential specification of a set.

The strategy is to tag each added element uniquely, without exposing the unique tags in the interface. When removing an element, the unique tags observed at the source replica are removed.

Figure 13 presents an op-based specification. The payload is a set of pairs *(element, unique-tag)*. Operation *add*(*e*) generates a unique identifier (*unique* is assumed to return a unique value, e.g., a Lamport clock) in the source replica; this is propagated to downstream replicas, which insert the pair into their payload. Adding the same element *e* twice generates two unique pairs, but *lookup*(*e*) masks the duplicates, extracting the element from the pairs.

payload set *VA*, *VR*, *EA*, *ER*
                                -- *V: vertices; E: edges; A: added; R: removed*
   initial $\emptyset, \emptyset, \emptyset, \emptyset$
query *lookup* (vertex *v*) : boolean b
   let $b = (v \in (VA \setminus VR))$
query *lookup* (edge (u, v)) : boolean b
   let $b = (lookup(u) \wedge lookup(v) \wedge (u, v) \in (EA \setminus ER))$
update *addVertex* (vertex *w*)
   downstream (*w*)
      $VA := VA \cup \{w\}$
update *addEdge* (vertex *u*, vertex *v*)
   atSource (u, v)
      pre $lookup(u) \wedge lookup(v)$
                        -- *Graph precondition: $E \subseteq V \times V$*
   downstream (u, v)
      $EA := EA \cup \{(u, v)\}$
update *removeVertex* (vertex *w*)
   atSource (*w*)
      pre $lookup(w)$ -- *2P-Set precondition*
      pre $\forall (u, v) \in (EA \setminus ER) : u \neq w \wedge v \neq w$
                        -- *Graph precondition: $E \subseteq V \times V$*
   downstream (*w*)
      pre *addVertex*(*w*) delivered   -- *2P-Set precondition*
      $VR := VR \cup \{w\}$
update *removeEdge* (edge (u, v))
   atSource $((u, v))$
      pre $lookup((u, v))$ -- *2P-Set precondition*
   downstream (u, v)
      pre *addEdge*(*u*, *v*) delivered   -- *2P-Set precondition*
      $ER := ER \cup \{(u, v)\}$

Figure 14: 2P2P-Graph (op-based)

When a client calls *remove*(*e*), the set of unique tags associated with *e* at the source is recorded. All such pairs are removed from the downstream payload. Thus, when *remove*(*e*) happens-after any number of *add*(*e*), all the corresponding pairs are removed, and the element is not in the set any more, as expected intuitively. When *add*(*e*) is concurrent with *remove*(*e*), the *add* takes precedence, as the unique tag generated by *add* cannot be observed by *remove*.

This behaviour is illustrated in Figure 11, noting $\alpha, \beta, \ldots$ the unique tags. The *remove*(*a*) called at the top replica translates to removing (a, α) downstream.

The *add* called at the second replica is concurrent to the *remove* of the first one, therefore (a, β) remains in the final state.

3.3   Graphs

A graph is a pair of sets (V, E) (called vertices and edges respectively) such that $E \subseteq V \times V$. Any of the Set implementations described above can be used for V and E.

Because of the invariant $E \subseteq V \times V$, operations on vertices and edges are not independent. At source, an edge may be added only if the corresponding vertices exist; conversely, a vertex may be removed only if it supports no edge. The specification in Figure 14 uses one 2P-Set for vertices and another for edges. The dependencies between them are resolved by causal delivery. Even if vertices are unique, we do not use U-Set because tombstones are needed to guard *addEdge* against concurrent *removeVertex*. In case of a concurrent *addEdge* and *removeVertex*, the effect of *removeVertex* takes precedence, as an edge only exists if their vertices have not been removed (as defined in edge lookup).

```
payload set S                         -- triplets (isbn k, integer n, unique-tag u), . . .
    initial ∅
query get (isbn k) : integer n
    let N = {n′|(k′, n′, u′) ∈ S ∧ k′ = k}
    if N = ∅ then
        let n = 0
    else
        let n = max(N)
update add (isbn k, integer n)
    atSource (k, n)
        pre n > 0
        let u = unique()
        let R = {(k′, n′, u′) ∈ S|k′ = k}
    downstream (R, k, n, u)
        pre ∀(k′, n′, u′) ∈ R :
           add(k′, n′, u) has been delivered
                                              -- OR-Set remove precondition
        S := (S \ R) ∪ {(k, n, u)}
                                      -- Replace elements observed at source
update remove (isbn k)
    atSource (k)
        let R = {(k′, n′, u′) ∈ S|k′ = k}
    downstream (R)
        pre ∀(k, n, u) ∈ R : add(k, n, u) has been delivered
                                              -- OR-Set precondition
        S := S \ R              -- Remove elements observed at source
```

Figure 15: Op-based Observed-Remove Shopping Cart (OR-Cart)

In general, a CRDT cannot maintain a particular graph shape such as DAG or a tree, as this requires evaluating a precondition in a globally consistent state [4]. However, specific structures are possible, for instance a Monotonic DAG in which adding vertices and edges can only strengthen the existing order.

An even more specialised graph structure is a Sequence, as used for instance in collaborative text editing [19, 21, 31]. A text editing buffer is often viewed as an array, but array operations do not commute; whereas, when viewed as a specific Graph, commutativity is simple to achieve. In particular, the first two authors previously designed the Treedoc Sequence CRDT, based on the idea of approximating the continuum as a binary tree [19].

4   Garbage collection

Some CRDTs tend to become less efficient over time, as tombstones accumulate and internal data structures become unbalanced [16, 19]. *Garbage collection* (GC) alleviates these problems; it may require synchronisation, but its liveness is not essential. We investigate two classes of GC mechanisms, with different synchronisation requirements.

An update f sometimes adds information $r(f)$ to the payload in order to deal cleanly with concurrent operations, e.g. in Graph, *remove* leaves a tombstone to handle concurrent *addBetween*s. Our first class of GC discards such $r(f)$ when it does not serve any useful purpose any more:

Definition 4.1 (Stability). *Update f is stable at replica x_i (noted $\Phi_i(f)$) if all updates concurrent to f have taken effect at x_i. Formally, $\Phi_i(f) \Leftrightarrow \forall j : f \in C(x_j) \wedge (\nexists g \in C(x_j) \setminus C(x_i) : f \parallel g)$.*

Liveness of Φ requires that the set of replicas be known and that they do not crash permanently (undetectably). Under these assumptions, the algorithm of Wuu and Bernstein [32] can be adapted to detect stability of f and thus discard $r(f)$. We note that this information is generally available when using a reliable broadcast channel.[4] Importantly, GC based on Φ can be performed in the background, so its liveness is not critical for correctness.

A second class of GC problems resets the payload across all replicas. An example is removing tombstones from a 2P-Set (thus allowing to re-add deleted elements again), removing entries from a version vector, or rebalancing a replicated tree [16]. This requires a commitment protocol. To alleviate the strong

---

[4] Note furthermore that such a channel already does GC internally, often making a CmRDT simpler than the corresponding CvRDT.

requirements of commitment, and to collect asynchronously with updates, Leția et al. [16] propose to perform commitment within a small, stable subset of replicas only, the *core*; the other replicas reconcile their state with core replicas. This approach works well for Treedoc; we are working on generalising it to other CRDTs.

5   Putting CRDTs to work

As a concrete example, consider shopping carts in an e-commerce bookstore. For high throughput and availability, data is replicated at both data-centre and geographical scale [7]. Given these assumptions, strong consistency would be slow and would not tolerate network partitions. CRDTs provide a good solution.

5.1   Observed-remove Shopping Cart

A shopping cart maps a book number (ISBN) to the quantity that the user wants. Any of the Set CRDTs presented earlier extends readily to a Map; we choose to extend OR-Set (Section 3.2.2). This design is simple, and does not have the cost of the version vectors needed by Dynamo's MV-Register.

Figure 15 presents an op-based OR-Cart. The payload is a set of triplets *(key, value, unique-identifier)*. Operation *remove* discards all existing mappings for the given ISBN: the source records the triplets associated with that key, to be removed, downstream, from the payload. Operation *add* overwrites by first discarding existing mappings as above, then inserting a unique triplet. As in OR-Set, causal delivery is sufficient to satisfy the downstream precondition.

We now show informally that concurrent updates commute. Two *remove*s commute, as the downstream set-minus operations are either independent or idempotent. The triplets created by concurrent *add*s cannot be in the removal set of the other, and (similarly to *remove*), their downstream set-minuses commute. Operation *add* is independent from, or idempotent with, a concurrent *remove*, as the triplet added by the former is disjoint from the triplets removed by the latter.

5.2   E-commerce bookstore

The bookstore maps user accounts to OR-Carts, using a U-Map (derived from U-Set in the obvious way). A shopping cart is added when the account is first created, and removed when it is deleted.

When the user chooses book b, the user interface calls $add(b, 1)$ against some replica. To change the quantity to $q > 0$, it calls $add(b, q)$. If the user cancels the book, or brings the quantity to zero, the interface calls $remove(b)$.

Non-concurrent updates have the expected semantics, i.e., later ones take precedence. Even though the user interface may address updates to different replicas (which may be out of sync with one another [7]), concurrent updates have clear, understandable semantics, i.e., it is the largest value that is chosen.

6 Comparison with previous work

Eventual consistency has been an active topic of research in highly-available, large-scale asynchronous systems [23]. Contrary to much previous work [7, for instance], we take a formal approach grounded in the theory of commutativity and semilattices. However, we are not the first to study commutativity in context of concurrency and replication. Commutativity has been studied to improve concurrency control and disconnected operation in transactional systems [10, 14, 30]. Closer to our motivation, Helland and Campbell leverage commutativity to improve availability [11].

6.1 Existing CRDTs

Although the concept itself was identified only recently, previous CRDT designs have been published. Johnson and Thomas invented the LWW-Register [13]. They propose a database of registers that can be created, updated and deleted, using the LWW rule to arbitrate between concurrent assignments and removes (i.e., a removed element can be recreated). LWW ensures a total order of operations, but it is an arbitrary extension of happened-before, so, inherently, some updates are lost.

Wuu and Bernstein [32] describe Dictionary and Log CRDTs. Their Dictionary is a Map CmRDT, similar to our U-Set. Their Log serves as a reliable broadcast channel for Dictionary. They study how to propagate the log effectively; to limit log growth, they propose the algorithm to detect when an entry is stable and can be collected, used in Section 4.

Concurrent editing has been the focus of CRDT and related research. WOOT is a Graph CRDT designed for collaborative editing [18]. The same authors designed the Logoot Sequence CRDT that supports an *undo* mechanism based on a CRDT Counter [31]. Preguiça and Shapiro propose Treedoc, a Sequence CRDT

for concurrent editing [19]. They later identified the GC issue, and studied how to move it into the background [16].

6.2   Related concepts

The CRDT concept was invented by Shapiro and Preguiça [26]. Other work has used similar ideas.

Ellis and Gibbs' [8] Operational Transformation (OT) studies op-based Sequences for shared editing. 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. Many OT algorithms have been proposed; Oster et al. show that most OT algorithms for a decentralised architecture are incorrect [17]. We believe that designing for commutativity from the start is both cleaner and simpler.

The foundations of CvRDTs were introduced by Baquero and Moura [1, 2]. We extend their work with a specification language, by considering CmRDTs, by studying more complex examples, and by considering GC.

Recently, Alvaro et al. proposed the Bloom programming language, which ensures eventual consistency by enforcing logical monotonicity. This is akin to the rule for CvRDTs, that every update or merge move forward in the monotonic semilattice. However, Bloom does not support *remove* without synchronization.

Roh et al. [21, 22] independently developed the Replicated Abstract Data Type concept, which is quite similar to CRDT. They generalise LWW to a partial order of updates, which they leverage to build several LWW-style classes; we allow any LUB merge function.

Serafini et al. suggest to leverage periods of good network conditions to achieve the stronger and more desirable linearizability property [25]. They distinguish strong (i.e., linearisable) operations from weak ones that need to be linearised eventually only. They show that, if all operations must terminate, the $\diamond S$ failure detector is insufficient for solving this problem. Our future work includes adding infrequent strong operations to CRDTs, e.g., to commit a result; we will study the impact of Serafini's results on such designs.

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, and merge should compute a least upper bound. In the op-based style, concurrent operations should commute. Assuming only that the communication subsystem eventually delivers, both styles of CRDTs are guaranteed to converge towards a common, correct state, without requiring any synchronisation.

We specified a number of interesting data types, in a high-level specification language based on simple logic. In particular, we focused on Set types with clean semantics for *add* and *remove* operations; Maps, Graphs, and Sequences can be built above Sets. Our bookstore example shows how CRDTs might be used practically.

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 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] Carlos Baquero and Francisco Moura. Specification of convergent abstract data types for autonomous mobile computing. Technical report, Departamento de Informática, Universidade do Minho, October 1997.

[2] Carlos Baquero and Francisco Moura. Using structural characteristics for autonomous operation. *Operating Systems Review*, 33(4):90–96, 1999.

[3] Lamia Benmouffok, Jean-Michel Busca, Joan Manuel Marquès, Marc Shapiro, Pierre Sutra, and Georgios Tsoukalas. Telex: A semantic platform for cooperative application development. In *Conf. Française sur les Systèmes d'Exploitation (CFSE)*, Toulouse, France, September 2009.

[4] Nikolaj Bjørner. Models and software model checking of a distributed file replication system. In *Formal Methods and Hybrid Real-Time Systems*, pages 1–23, 2007.

[5] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722, 1996.

[6] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 1990.

[7] 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. In *Symp. on Op. Sys. Principles (SOSP)*, volume 41 of *Operating Systems Review*, pages 205–220, Stevenson, Washington, USA, October 2007. Assoc. for Computing Machinery.

[8] C. A. Ellis and S. J. Gibbs. Concurrency control in groupware systems. In *Int. Conf. on the Mgt. of Data (SIGMOD)*, pages 399–407, Portland, OR, USA, 1989. Assoc. for Computing Machinery.

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

[10] Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. The dangers of replication and a solution. In *Int. Conf. on the Mgt. of Data (SIGMOD)*, pages 173–182, Montréal, Canada, June 1996. ACM SIGMOD, ACM Press.

[11] Pat Helland and David Campbell. Building on quicksand. In *Biennial Conf. on Innovative DataSystems Research (CIDR)*, Asilomar, Pacific Grove CA, USA, June 2009.

[12] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and

performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

[13] Paul R. Johnson and Robert H. Thomas. The maintenance of duplicate databases. Internet Request for Comments RFC 677, Information Sciences Institute, January 1976.

[14] Justus Klingemann and Thomas Tesch. Semantics-based transaction management for cooperative applications. In *Int. W. on Advanced Trans. Models and Arch.*, pages 234–252, Goa, India, August 1996.

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

[16] Mihai Leţia, Nuno Preguiça, and Marc Shapiro. CRDTs: Consistency without concurrency control. In *SOSP W. on Large Scale Distributed Systems and Middleware (LADIS)*, volume 44 of *Operating Systems Review*, pages 29–34, Big Sky, MT, USA, October 2009. ACM SIG on Operating Systems (SIGOPS), Assoc. for Comp. Machinery.

[17] Gérald Oster, Pascal Urso, Pascal Molli, and Abdessamad Imine. Proving correctness of transformation functions in collaborative editing systems. Rapport de recherche RR-5795, LORIA – INRIA Lorraine, December 2005.

[18] Gérald Oster, Pascal Urso, Pascal Molli, and Abdessamad Imine. Data consistency for P2P collaborative editing. In *Int. Conf. on Computer-Supported Coop. Work (CSCW)*, pages 259–268, Banff, Alberta, Canada, November 2006. ACM Press.

[19] Nuno Preguiça, Joan Manuel Marquès, Marc Shapiro, and Mihai Leţia. A commutative replicated data type for cooperative editing. In *Int. Conf. on Distributed Comp. Sys. (ICDCS)*, pages 395–403, Montréal, Canada, June 2009.

[20] Peter Reiher, John S. Heidemann, David Ratner, Gregory Skinner, and Gerald J. Popek. Resolving file conflicts in the Ficus file system. In *Usenix Conf.* Usenix, June 1994.

[21] Hyun-Gul Roh, Myeongjae Jeon, Jin-Soo Kim, and Joonwon Lee. Replicated abstract data types: Building blocks for collaborative applications. *Journal of Parallel and Dist. Comp.*, (To appear) 2011.

[22] Hyun-Gul Roh, Jin-Soo Kim, and Joonwon Lee. How to design optimistic operations for peer-to-peer replication. In *Int. Conf. on Computer Sc. and Informatics (JCIS/CSI)*, Kaohsiung, Taiwan, October 2006.

[23] Yasushi Saito and Marc Shapiro. Optimistic replication. *ACM Computing Surveys*, 37(1):42–81, March 2005.

[24] R. Schwarz and F. Mattern. Detecting causal relationships in distributed computations: In search of the Holy Grail. *Distributed Computing*, 3(7):149–174, 1994.

[25] Marco Serafini, Dan Dobre, Matthias Majuntke, Péter Bokor, and Neeraj Suri. Eventually linearizable shared objects. In *Symp. on Principles of Dist. Comp. (PODC)*, pages 95–104, Zürich, Switzerland, 2010. Assoc. for Comp. Machinery.

[26] Marc Shapiro and Nuno Preguiça. Designing a commutative replicated data type. Rapport de recherche RR-6320, Institut Nat. de la Recherche en Informatique et Automatique (INRIA), Rocquencourt, France, October 2007.

[27] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 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, January 2011.

[28] Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, Mike J. Spreitzer, and Carl H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *15th Symp. on Op. Sys. Principles (SOSP)*, pages 172–182, Copper Mountain, CO, USA, December 1995. ACM SIGOPS, ACM Press.

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

[30] W. E. Weihl. Commutativity-based concurrency control for abstract data types. *IEEE Trans. on Computers*, 37(12):1488–1505, December 1988.

[31] Stephane Weiss, Pascal Urso, and Pascal Molli. Logoot-undo: Distributed collaborative editing system on P2P networks. *IEEE Trans. on Parallel and Dist. Sys. (TPDS)*, 21:1162–1174, 2010.

[32] Gene T. J. Wuu and Arthur J. Bernstein. Efficient solutions to the replicated log and dictionary problems. In *Symp. on Principles of Dist. Comp. (PODC)*, pages 233–242, Vancouver, BC, Canada, August 1984.