INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

A comprehensive study of Convergent and Commutative Replicated Data Types

Marc Shapiro, INRIA & LIP6, Paris, France
Nuno Preguiça, CITI, Universidade Nova de Lisboa, Portugal
Carlos Baquero, Universidade do Minho, Portugal

Marek Zawirski, INRIA & UPMC, Paris, France

N° 7506

Janvier 2011

Thème COM

Rapport de recherche

ISSN 0249-6399 ISRN INRIA/RR--7506--FR+ENG

A comprehensive study of Convergent and Commutative Replicated Data Types *

Marc Shapiro, INRIA & LIP6, Paris, France
Nuno Preguiça, CITI, Universidade Nova de Lisboa, Portugal
Carlos Baquero, Universidade do Minho, Portugal
Marek Zawirski, INRIA & UPMC, Paris, France

Thème COM — Systèmes communicants
Projet Regal

Rapport de recherche n° 7506 — Janvier 2011 — 47 pages

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, montonic DAGs, and sequences. It discusses some properties needed to implement non-trivial CRDTs.

Key-words: Data replication, optimistic replication, commutative operations

* This research was supported in part by ANR project ConcoRDanT (ANR-10-BLAN 0208), and a Google Research Award 2009. Marek Zawirski is a recipient of the Google Europe Fellowship in Distributed Computing, and this research is supported in part by this Google Fellowship. Carlos Baquero is partially supported by FCT project Castor (PTDC/EIA-EIA/104022/2008).

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 Le Chesnay Cedex (France)
Téléphone : +33 1 39 63 55 11 — Télécopie : +33 1 39 63 53 30

Étude approfondie des types de données répliqués convergents et commutatifs

Résumé : La cohérence à terme vise à assurer que les répliques d'un objet partagé modifiable convergent sans synchronisation à priori. Les approches antérieures du problème sont *ad-hoc* et sujettes à erreur. Nous proposons une approche basée sur des principes formels : baser la conception des types de données sur des propriétés mathématiques simples, suffisantes pour garantir la cohérence à terme. Nous appelons ces types de données des CRDT *(Convergent/Commutative Replicated Data Types)*. Ce papier fournit formalise la réplication asynchrone, qu'elle soit basée sur l'état ou sur les opérations, et fournit une condition suffisante adaptée à chacun de ces cas. Il décrit plusieurs CRDT utiles, dont des contenants permettant les opérations *add* et *remove* avec une sémantique propre, et des types de données plus complexes comme les graphes, les graphes acycliques monotones, et les séquences. Il contient une discussion de propriétés dont on a besoin pour mettre en œuvre des CRDT non triviaux.

Mots-clés : Réplication des données, réplication optimiste, opérations commutatives

1 Introduction

Replication is a fundamental concept of distributed systems, well studied by the distributed algorithms community. Much work focuses on maintaining a global total order of operations [24] even in the presence of faults [8]. However, the associated serialisation bottleneck negatively impacts performance and scalability, while the CAP theorem [13] imposes a trade-off between consistency and partition-tolerance.

An alternative approach, *eventual consistency* or *optimistic replication*, is attractive to practioners [37, 41]. A replica may execute an operation without synchronising *a priori* with other replicas. The operation is sent asynchronously to other replicas; every replica eventually applies all updates, possibly in different orders. A background consensus algorithm reconciles any conflicting updates [4, 40]. This approach ensures that data remains available despite network partitions. It performs well (as the consensus bottleneck has been moved off the critical path), and the weaker consistency is considered acceptable for some classes of applications. However, reconciliation is generally complex. There is little theoretical guidance on how to design a correct optimistic system, and ad-hoc approaches have proven brittle and error-prone.[1]

In this paper, we study a simple, theoretically sound approach to eventual consistency. We propose the concept of a *convergent* or *commutative replicated data type* (CRDT), for which some simple mathematical properties ensure eventual consistency. A trivial example of a CRDT is a replicated counter, which converges because the increment and decrement operations commute (assuming no overflow). Provably, replicas of any CRDT converge to a common state that is equivalent to some correct sequential execution. As a CRDT requires no synchronisation, an update executes immediately, unaffected by network latency, faults, or disconnection. It is extremely scalable and is fault-tolerant, and does not require much mechanism. Application areas may include computation in delay-tolerant networks, latency tolerance in wide-area networks, disconnected operation, churn-tolerant peer-to-peer computing, data aggregation, and partition-tolerant cloud computing.

Since, by design, a CRDT does not use consensus, the approach has strong limitations; nonetheless, some interesting and non-trivial CRDTs are known to exist. For instance, we previously published Treedoc, a sequence CRDT designed for co-operative text editing [32].

Previously, only a handful of CRDTs were known. The objective of this paper is to push the envelope, studying the principles of CRDTs, and presenting a comprehensive portfolio of useful CRDT designs, including variations on registers, counters, sets, graphs, and sequences. We expect them to be of interest to practitioners and theoreticians alike.

Some of our designs suffer from unbounded growth; collecting the garbage requires a weak form of synchronisation [25]. However, its liveness is not essential, as it is an optimisation, off the critical path, and not in the public interface. In the future, we plan to extend the approach to data types where common-case, time-critical operations are commutative,

[1] The anomalies of the Amazon Shopping Cart are a well-known example [10].

and rare operations require synchronisation but can be delayed to periods when the network is well connected. This concurs with Brewer's suggestion for side-stepping the CAP impossibility [6]. It is also similar to the shopping cart design of Alvaro et al. [1], where updates commute, but check-out requires coordination. However, this extension is out of the scope of the present study.

In the literature, the preferred consistency criterion is linearisability [18]. However, linearisability requires consensus in general. Therefore, we settle for the much weaker *quiescent consistency* [17, Section 3.3]. One challenge is to minimise "anomalies," i.e., states that would not be observed in a sequential execution. Note also that CRDTs are weaker than non-blocking constructs, which are generally based on a hardware consensus primitive [17].

Some of the ideas presented here paper are already known in the folklore. The contributions of this paper include:

- In Section 2: *(i)* An specification language suited to asynchronous replication. *(ii)* A formalisation of state-based and operation-based replication. *(iii)* Two sufficient conditions for eventual consistency.
- In Section 3, an comprehensive collection of useful data type designs, starting with counters and registers. We focus on container types (sets and maps) supporting both *add* and *remove* operations with clean semantics, and more complex derived types, such as graphs, monotonic DAGs, and sequence.
- In Section 4, a study of the problem of garbage-collecting meta-data.
- In Section 5, exercising some of our CRDTs in a practical example, the shopping cart.
- A comparison with previous work, in Section 6.

Section 7 concludes with a summary of lessons learned, and perspectives for future work.

2 Background and system model

We consider a distributed system consisting of processes interconnected by an asynchronous network. The network can partition and recover, and nodes can operate in disconnected mode for some time. A process may crash and recover; its memory survives crashes. We assume non-byzantine behaviour.

2.1 Atoms and objects

A process may store *atoms* and *objects*. An atom is a base immutable data type, 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,

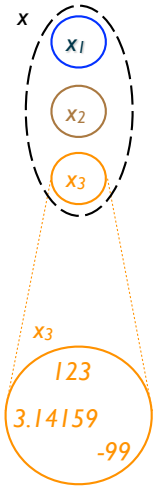

Figure 1: Object

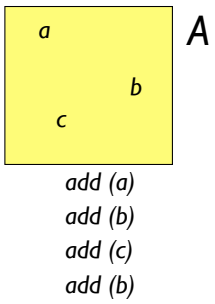

Figure 2: Grow-only Set: G-Set

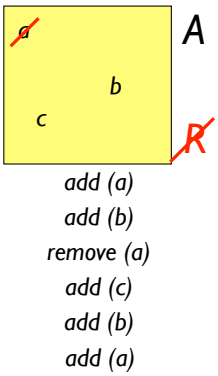

Figure 3: 2P-Set

etc., with their usual non-mutating operations. Atom types are written in lower case, e.g., "set."

An object is a mutable, replicated data type. Object types are capitalised, e.g., "Set." An object has an identity, a content (called its *payload*), which may be any number of atoms or objects, an initial state, and an interface consisting of *operations*. Two objects having the same identity but located in different processes are called *replicas* of one another. As an example, Figure 1 depicts a logical object x, its replicas at processes $1, 2$ and 3, and the current state of the payload of replica 3.

We assume that objects are independent and do not consider transactions. Therefore, without loss of generality, we focus on a single object at a time, and use the words process and replica interchangeably.

2.2 Operations

The environment consists of unspecified *clients* that query and modify object state by calling operations in its interface, against a replica of their choice called the *source* replica. A query executes locally, i.e., entirely at one replica. An update has two phases: first, the client calls the operation at the source, which may perform some initial processing. Then, the update is transmitted asynchronously to all replicas; this is the *downstream* part. The literature [37] distinguishes the *state-based* and *operation-based* (op-based for short) styles, explained next.

Specification 1 Outline of a state-based object specification. *Preconditions, arguments, return values and statements are optional.*

```
1:  payload Payload type; instantiated at all replicas
2:      initial Initial value
3:  query Query (arguments) : returns
4:      pre Precondition
5:      let Evaluate synchronously, no side effects
6:  update Source-local operation (arguments) : returns
7:      pre Precondition
8:      let Evaluate at source, synchronously
9:      Side-effects at source to execute synchronously
10: compare (value1, value2) : boolean b
11:     Is value1 ≤ value2 in semilattice?
12: merge (value1, value2) : payload mergedValue
13:     LUB merge of value1 and value2, at any replica
```

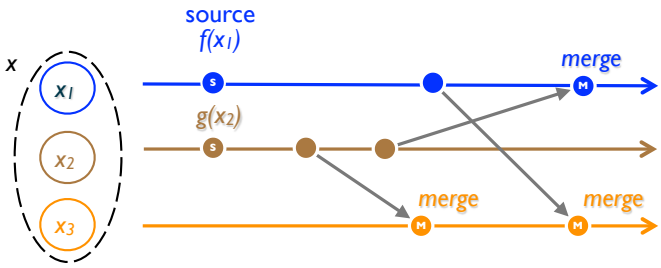

Figure 4: State-based replication

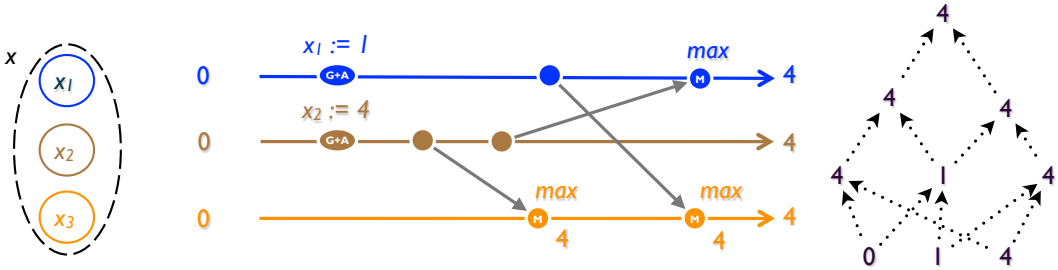

Figure 5: Example CvRDT: integer + max

2.2.1 State-based replication

In state-based (or passive) replication, an update occurs entirely at the source, then propagates by transmitting the modified payload between replicas, as illustrated in Figure 4.

We specify state-based object types as shown in Specification 1. Keyword payload indicates the payload type, and initial specifies its initial value at every replica. Keyword update indicates an update operation, and query a query. Both may have (optional) arguments and return values. Non-mutating statements are marked let, and payload is mutated by assignment :=. An operation executes atomically.

To capture safety, an operation is enabled only if a given *source pre-condition* (marked pre in a specification) holds in the source's current state. The source pre-condition is omitted if always enabled, e.g., incrementing or decrementing a Counter. Conversely, non-null pre-conditions may be necessary, for instance an element can be removed from a Set only if it is in the Set at the source.

The system transmits state between arbitrary pairs of replicas, in order to propagate changes. This updates the payload of the receiver with the output of operation *merge*, invoked with two arguments, the local payload state and the received state. Operation compare compares replica states, as will be explained shortly.

We define the causal history [38] $\mathcal{C}$ of replicas of some object x as follows:[2]
Definition 2.1 (Causal History — state-based)**.** *For any replica x_i of x:*

- *Initially, $\mathcal{C}(x_i) = \varnothing$.*
- *After executing update operation f, $\mathcal{C}(f(x_i)) = \mathcal{C}(x_i) \cup \{f\}$.*
- *After executing merge against states x_i, x_j, $\mathcal{C}(merge(x_i, x_j)) = \mathcal{C}(x_i) \cup \mathcal{C}(x_j)$.*

The classical happens-before [24] relation between operations can be defined as $f \rightarrow g \Leftrightarrow \mathcal{C}(f) \subset \mathcal{C}(g)$.

Liveness requires that any update eventually reaches the causal history of every replica. To this effect, we assume an underlying system that transmits states between pairs of replicas at unspecified times, infinitely often, and that replica communication forms a connected graph.

2.2.2 Operation-based (op-based) objects

In operation-based (or active) replication, the system transmits operations, as illustrated in Figure 6. This style is specified as outlined in Spec. 2. The payload and initial clauses are identical to the state-based specifications. An operation that does not mutate the state is marked query and executes entirely at a single replica.

An update is specified by keyword update. Its first phase, marked atSource, is local to the source replica. It is enabled only if its (optional) source pre-condition, marked pre, is

[2] $\mathcal{C}$ is a logical function, it is not part of the object.

Specification 2 Outline of operation-based object specification. *Preconditions, return values and statements are optional.*

```
1:  payload Payload type; instantiated at all replicas
2:      initial Initial value
3:  query Source-local operation (arguments) : returns
4:      pre Precondition
5:      let Execute at source, synchronously, no side effects
6:  update Global update (arguments) : returns
7:      atSource (arguments) : returns
8:          pre Precondition at source
9:          let 1st phase: synchronous, at source, no side effects
10:     downstream (arguments passed downstream)
11:         pre Precondition against downstream state
12:         2nd phase, asynchronous, side-effects to downstream state
```

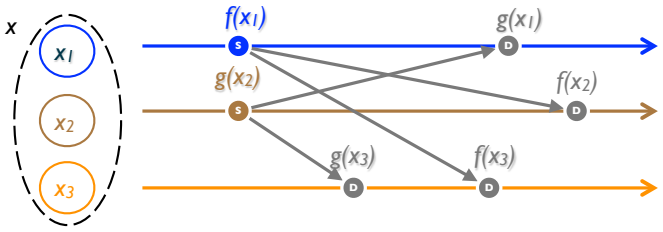

Figure 6: Operation-Based Replication

true in the source state; it executes atomically. It takes its arguments from the operation invocation; it is not allowed to make side effects; it may compute results, returned to the caller, and/or prepare arguments for the second phase.

The second phase, marked downstream, executes after the source-local phase; immediately at the source, and asynchronously, at all other replicas; it can not return results. It executes only if its *downstream precondition* is true. It updates the downstream state; its arguments are those prepared by the source-local phase. It executes atomically.

As above, we define the causal history of a replica $\mathcal{C}(x_i)$.

Definition 2.2 (Causal History — op-based)**.** *The causal history of a replica x_i is defined as follows.*

- *Initially, $\mathcal{C}(x_i) = \varnothing$.*
- *After executing the downstream phase of operation f at replica x_i, $\mathcal{C}(f(x_i)) = \mathcal{C}(x_i) \cup \{f\}$.*

Liveness requires that every update eventually reaches the causal history of every replica. To this effect, we assume an underlying system reliable broadcast that delivers every update to every replica in an order $<_d$ (called *delivery order*) where the downstream precondition is true.

As in the state-based case, the happens-before relation between operations is defined by $f \rightarrow g \Leftrightarrow \mathcal{C}(f) \subset \mathcal{C}(g)$. We define *causal delivery* $<_{\rightarrow}$ as follows: if $f \rightarrow g$ then f is delivered before g is delivered. We note that all downstream preconditions in this paper are satisfied by causal delivery, i.e., delivery order is the same or weaker as causal order: $f <_d g \Rightarrow f <_{\rightarrow} g$.

2.3 Convergence

We now formalise convergence.

Definition 2.3 (Eventual Convergence)**.** *Two replicas x_i and x_j of an object x* converge eventually *if the following conditions are met:*

- Safety*: $\forall i, j : \mathcal{C}(x_i) = \mathcal{C}(x_j)$ implies that the* abstract states *of i and j are equivalent.*
- Liveness*: $\forall i, j : f \in \mathcal{C}(x_i)$ implies that, eventually, $f \in \mathcal{C}(x_j)$.*

Furthermore, we define state equivalence as follows: x_i and x_j have equivalent abstract state if all query operations return the same values.

Pairwise eventual convergence implies that any non-empty subset of replicas of the object converge, as long as all replicas receive all updates.

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

A join semilattice [9] (or just semilattice hereafter) is a partial order $\leq_v$ equipped with a *least upper bound* (LUB) $\sqcup_v$, defined as follows:
Definition 2.4 (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 from the definition that $\sqcup_v$ is: commutative: $x \sqcup_v y =_v y \sqcup_v x$; idempotent: $x \sqcup_v x =_v x$; and associative: $(x \sqcup_v y) \sqcup_v z =_v x \sqcup_v (y \sqcup_v z)$.

Definition 2.5 (Join Semilattice)**.** *An ordered set $(S, \leq_v)$ is a Join Semilattice iff $\forall x, y \in S, x \sqcup_v y$ exists.*

A state-based object whose payload takes its values in a semilattice, and where $\mathsf{merge}(x, y) \stackrel{\text{def}}{=} x \sqcup_v y$, converges towards the LUB of the initial and updated values. If, furthermore, updates monotonically advance upwards according to $\leq_v$ (i.e., the payload value after an update is greater than or equal to the one before), then it converges towards the LUB of the most recent values. Let us call this combination "monotonic semilattice."

A type with these properties will be called a Convergent Replicated Data Type or CvRDT. We require that, in a CvRDT, $\mathsf{compare}(x, y)$ to return $x \leq_v y$, that abstract states be equivalent if $x \leq_v y \wedge y \leq_v x$, and merge be always enabled. As an example, Figure 5 illustrates a CvRDT with integer payload, where $\leq_v$ is integer order, and where $merge() \stackrel{\text{def}}{=} \max()$.

Eventual convergence requires that all replicas receive all updates. The communication channels of a CvRDT may have very weak properties. Since merge is idempotent and commutative (by the properties of $\sqcup_v$), messages may be lost, received out of order, or multiple times, as long as new state eventually reaches all replicas, either directly or indirectly via successive merges. Updates are propagated reliably even if the network partitions, as long as eventually connectivity is restored.
Proposition 2.1. *Any two object replicas of a CvRDT eventually converge, assuming the system transmits payload infinitely often between pairs of replicas over eventually-reliable point-to-point channels.*

Proof. Any two replicas x_i, x_j will converge, as long as they can exchange states by some (direct or indirect) channel that eventually delivers, by merging their states. Since CvRDT values form a monotonic semilattice, *merge* is always enabled, and one can make $x'_i := \mathsf{merge}(x_i, x_j)$ and $x'_j := \mathsf{merge}(x_j, x_i)$. By Definition 2.1, we have the same causal history in x'_i and x'_j, since $\mathcal{C}(x_i) \cup \mathcal{C}(x_j) = \mathcal{C}(x_j) \cup \mathcal{C}(x_i)$. Finally we have equivalent abstract states $x'_i =_v x'_j$ since, by commutativity of LUB, $x_i \sqcup_v x_j =_v x_j \sqcup_v x_i$. □

2.3.2 Operation-based CRDT: Commutative Replicated Data Type (CmRDT)

In an op-based object, a reliable broadcast channel guarantees that all updates are delivered at every replica, in the delivery order $<_d$ specified by the data type. Operations not ordered by $<_d$ are said *concurrent*; formally $f \parallel_d g \Leftrightarrow f \not<_d g \wedge g \not<_d f$. If all concurrent operations *commute*, then all execution orders consistent with delivery order are equivalent, and all replicas converge to the same state. Such an object is called a Commutative Replicated Data Type (CmRDT).

As noted earlier, for all data types studied here, causal delivery $<_\rightarrow$ (which is readily implementable in static distributed systems and does not require consensus) satisfies delivery order $<_d$. For some data types, a weaker ordering suffices, but then more pairs of operations need to be proved commutative.

Definition 2.6 (Commutativity)**.** *Operations f and g commute, iff for any reachable replica state S where their source pre-condition is enabled, the source precondition of f (resp. g) remains enabled in state $S \cdot g$ (resp. $S \cdot f$), and $S \cdot f \cdot g$ and $S \cdot f \cdot g$ are equivalent abstract states.*

Proposition 2.2. *Any two replicas of a CmRDT eventually converge under reliable broadcast channels that deliver operations in delivery order $<_d$.*

Proof. Consider object replicas x_i, x_j. Under the channel assumptions, eventually the two replicas will deliver the same operations (if no new operations are generated), and we have $\mathcal{C}(x_i) = \mathcal{C}(x_j)$. For any two operations f, g in $\mathcal{C}(x_i)$: (1) if they are not causally related then they are concurrent under $<_d$ (that is never stronger than causality) and must commute; (2) if they are causally related $a \rightarrow b$ but are not ordered in delivery order $<_d$ they must also commute; (3) if they are causaly related $a \rightarrow b$ and delivered in that order $a <_d b$ then they are applied in that same order everyware. In all cases an equivalent abstract state is reached in all replicas. □

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. As delivery order is never stricter than causal delivery, *a fortiori* this is true of all CmRDTs.

2.4 Relation between the two approaches

We have shows two approaches to eventual convergence, CvRDTs and CmRDTS, which together we call CRDTs. There are similarities and differences between the two.

State-based mechanisms (CvRDTs) are simple to reason about, since all necessary information is captured by the state. They require weak channel assumptions, allowing for unknown numbers of replicas. However, sending state may be inefficient for large objects; this can be tackled by shipping deltas, but this requires mechanisms similar to the op-based

Specification 3 Operation-based emulation of state-based object

```
1: payload State-based S                                  ▷ S: Emulated state-based object
2:     initial  Initial payload
3: update State-based-update (operation f, args a) : state s
4:     atSource (f, a) : s
5:         pre S.f.precondition(a)
6:         let s = S.f(a)                                 ▷ Compute state applying f to S
7:     downstream (s)
8:         S := merge(S, s)
```

approach. Historically, the state-based approach is used in file systems such as NFS, AFS [19], Coda [22], and in key-value stores such as Dynamo [10] and Riak.

Specifying operation-based objects (CmRDTs) can be more complex since it requires reasoning about history, but conversely they have greater expressive power. The payload can be simpler since some state is effectively offloaded to the channel. Op-based replication is more demanding of the channel, since it requires reliable broadcast, which in general requires tracking group membership. Historically, op-based approaches have been used in cooperative systems such as Bayou [31], Rover [21] IceCube [33], Telex [4].

2.4.1 Operation-based emulation of a state-based object

Interestingly, it is always possible to emulate a state-based object using the operation-based approach, and vice-versa.[3]

In Spec. 3 we show operation-based emulation of a state-based object (taking some liberties with notation). Ignoring queries (which pose no problems), the emulating operation-based object has a single update that computes some state-based update (after checking for its precondition) and performs merge downstream. The downstream precondition is empty because merge must be enabled in any reachable state. The emulation does not make use of compare.

Note that if the base object is a CvRDT, then merge operations commute, and the emulated object is a CmRDT.

2.4.2 State-based emulation of an operation-based object

State-based emulation of an operation-based object essentially formalises the mechanics of an epidemic reliable broadcast, as shown in Spec. 4 (taking some liberties with notation). Again, we ignore queries, which pose no problems. Calling an operation-based update adds it to a set of M messages to be delivered; *merge* takes the union of the two message sets.

[3] Contrary to what Helland says [16], because he only considers read-write state, not a merge operation.

Specification 4 State-based emulation of operation-based object

```
1: payload Operation-based P, set M, set D    ▷ Payload of emulated object, messages, delivered
2:     initial Initial state of payload, ∅, ∅
3: update op-based-update (update f, args a) : returns
4:     pre P.f.atSource.pre(a)                          ▷ Check at-source precondition
5:     let returns = P.f.atSource(a)                    ▷ Perform at-source computation
6:     let u = unique()
7:     M := M ∪ {(f, a, u)}                             ▷ Send unique operation
8:     deliver()                                        ▷ Deliver to local op-based object
9: update deliver ()
10:    for (f, a, u) ∈ (M \ D) : f.downstream.pre(a) do
11:        P := P.f.downstream(a)                       ▷ Apply downstream update to replica
12:        D := D ∪ {(f, a, u)}                         ▷ Remember delivery
13: compare (R, R′) : boolean b
14:    let b = R.M ≤ R′.M ∨ R.D ≤ R′.D
15: merge (R, R′) : payload R″
16:    let R″.M = R.M ∪ R′.M
17:    R″.deliver()                                     ▷ Deliver pending enabled updates
```

Specification 5 op-based Counter

```
1: payload integer i
2:     initial 0
3: query value () : integer j
4:     let j = i
5: update increment ()
6:     downstream ()                                    ▷ No precond: delivery order is empty
7:         i := i + 1
8: update decrement ()
9:     downstream ()                                    ▷ No precond: delivery order is empty
10:        i := i − 1
```

When an update's downstream precondition is true, the corresponding message is delivered by executing the downstream part of the update. In order to avoid duplicate deliveries, delivered messages are stored in a set D.

Note that the states of the emulating object form a monotonic semilattice. Calling or delivering an operation adds it to the relevant message set, and therefore advances the state in the partial order. *merge* is defined to take the union of the M sets, and is thus a LUB operation. Remark that M is identical to the causal history of the replica; non-concurrent updates appear in M in causal order. If the emulated op-based object is a CmRDT, then delivery order is satisfied. Concurrent operations appear in M in any order; if the emulated object is a CmRDT, they commute. Therefore, after two replicas *merge* mutually, their D sets are identical and their P payloads have equivalent state.

3 Portfolio of basic CRDTs

To show the usefulness of the CRDT concept, we now present a number of CRDT designs. They are interesting to understand the challenges, possibilities and limitations of CRDTs. They also constitute a library of types that can be re-used and combined to build distributed systems. We start with some simple types (counters and registers), then move on to collection types (sets), and finally some types with more complex requirements (graphs, DAGs and sequences).

Our specifications are written with clarity in mind, not efficiency. In many cases, there are clearly equivalent approaches that conserve space, but we systematically preferred the more easily-understood version.

We write either state- or op-based specifications, as convenient. For each state-based example, we have the obligation to prove that its states form a monotonic semilattice and that merge computes a LUB. For each op-based example, we must demonstrate that a delivery order exists and that concurrent updates commute.

3.1 Counters

A Counter is a replicated integer supporting operations *increment* and *decrement* to update it, and *value* to query it. The semantics should be is that the value converge towards the global number of *increment*s minus the number of *decrement*s. (Extension to operations for adding and subtracting an argument is straightforward.) A Counter CRDT is useful in many peer-to-peer applications, for instance counting the number of currently logged-in users.

In this section we discuss different designs for implementing a counter CRDT. Despite its simplicity, the Counter exposes some of the design issues of CRDTs.

3.1.1 Op-based counter

An op-based counter is presented in Specification 5. Its payload is an integer. Its empty atSource clause is omitted; the downstream phase just adds or subtracts locally. It is well-known that addition and subtraction commute, assuming no overflow. Therefore, this data type is a CmRDT.

3.1.2 State-based increment-only Counter (G-Counter)

A state-based counter is not as straightforward as one would expect. To simplify the problem, we start with a Counter that only increments.

Suppose the payload was a single integer and *merge* computes max. This data type is a CvRDT as its states form a monotonic semilattice. Consider two replicas, with the same

Specification 6 State-based increment-only counter (vector version)

```
1: payload integer[n] P                                   ▷ One entry per replica
2:     initial [0, 0, ..., 0]
3: update increment ()
4:     let g = myID()                                     ▷ g: source replica
5:     P[g] := P[g] + 1
6: query value () : integer v
7:     let v = Σ_i P[i]
8: compare (X, Y) : boolean b
9:     let b = (∀i ∈ [0, n − 1] : X.P[i] ≤ Y.P[i])
10: merge (X, Y) : payload Z
11:     let ∀i ∈ [0, n − 1] : Z.P[i] = max(X.P[i], Y.P[i])
```

Specification 7 State-based PN-Counter

```
1: payload integer[n] P, integer[n] N                     ▷ One entry per replica
2:     initial [0, 0, ..., 0], [0, 0, ..., 0]
3: update increment ()
4:     let g = myID()                                     ▷ g: source replica
5:     P[g] := P[g] + 1
6: update decrement ()
7:     let g = myID()
8:     N[g] := N[g] + 1
9: query value () : integer v
10:    let v = Σ_i P[i] − Σ_i N[i]
11: compare (X, Y) : boolean b
12:    let b = (∀i ∈ [0, n − 1] : X.P[i] ≤ Y.P[i] ∧ ∀i ∈ [0, n − 1] : X.N[i] ≤ Y.N[i])
13: merge (X, Y) : payload Z
14:    let ∀i ∈ [0, n − 1] : Z.P[i] = max(X.P[i], Y.P[i])
15:    let ∀i ∈ [0, n − 1] : Z.N[i] = max(X.N[i], Y.N[i])
```

initial state of 0; at each one, a client originates *increment*. They converge to 1 instead of the expected 2.

Suppose instead the payload is an integer and merge adds the two values. This is not a CvRDT, as *merge* is not idempotent.

We propose instead the construct of Specification 6 (inspired by vector clocks). The payload is vector of integers; each source replica is assigned an entry. To increment, add 1 to the entry of the source replica. The value is the sum of all entries. We define the partial order over two states X and Y by $X \leq Y \Leftrightarrow \forall i \in [0, n-1] : X.P[i] \leq Y.P[i]$, where n is the number of replicas. Merge takes the maximum of each entry. This data type is a CvRDT, as its states form a monotonic semilattice, and *merge* produces the LUB.

This version makes two important assumptions: the payload does not overflow, and the set of replicas is well-known. Note however that the op-based version implicitly makes the same two assumptions.

Alternatively, G-Set (described later, Section 3.3.1) can serve as an *increment*-only counter. G-Set works even when the set of replicas is not known.

The *increment*-only counter is useful, for instance to count the number of clicks on a link in a P2P-replicated web page, or a P2P "I Like It/I Don't Like It" poll, as is common in social networks.

3.1.3 State-based PN-Counter

It is not straightforward to support *decrement* with the previous representation, because this operation would violate monotonicity of the semilattice. Furthermore, since *merge* is a max operation, decrement would have no effect.

Our solution, PN-Counter (Specification 7) basically combines two G-Counters. Its payload consists of two vectors: P to register *increment*s, and N for *decrement*s. Its value is the difference between the two corresponding G-Counters, its partial order is the conjunction of the corresponding partial orders, and *merge* merges the two vectors. Proving that this is a CRDT is left to the reader.

Such a counter might be useful, for instance, to count the number of users logged in to a P2P application such as Skype. To avoid excessively large vectors, only super-peers would replicate the counter. Due to asynchrony, the count may diverge temporarily from its true value, but it will eventually be exact.

3.1.4 Non-negative Counter

Some applications require a counter that is non-negative; for instance, to count the remaining credit of an avatar in a P2P game.

Specification 8 State-based Last-Writer-Wins Register (LWW-Register)

```
1: payload X x, timestamp t                          ▷ X: some type
2:     initial ⊥, 0
3: update assign (X w)
4:     x, t := w, now()                              ▷ Timestamp, consistent with causality
5: query value () : X w
6:     let w = x
7: compare (R, R') : boolean b
8:     let b = (R.t ≤ R'.t)
9: merge (R, R') : payload R''
10:    if R.t ≤ R'.t then R''.x, R''.t = R'.x, R'.t
11:    else R''.x, R''.t = R.x, R.t
```

However, this is quite difficult to do while preserving the CRDT properties; indeed, this is a global invariant, which cannot be evaluated based on local information only. For instance, it is not sufficient for each replica to refrain from decrementing when its local value is 0: for instance, two replicas at value 1 might still concurrently decrement, and the value converges to -1.

One possible approach would be to maintain any value internally, but to externalize negative ones as 0. However this is flawed, since incrementing from an internal value of, say, -1, has no effect; this violates the semantics required in Section 3.1.

A correct approach is to enforce a local invariant that implies the global invariant: e.g., rule that a client may not originate more *decrement*s than it originated *increment*s (i.e., $\forall g : P[g] - N[g] \geq 0$). However, this may be too strong.

Note that one of the Set constructs (described later, Section 3.3) might serve as a non-negative counter, using *add* to increment and *remove* to decrement. However this does not have the expected semantics: if two replicas concurrently remove the same element, the result is equivalent to a single *decrement*.

Sadly, the remaining alternative is to synchronise. This might be only occasionally, e.g., by reserving in advance the right to originate a given number of *decrement*s, as in escrow transactions [28].

3.2 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. Unless safeguards are taken, concurrent updates do not commute; two major approaches are that one takes precedence over the other (LWW-Register, Section 3.2.1), or that both are retained (MV-Register, Section 3.2.2).

Specification 9 Op-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 precond: delivery order is empty
        if t < t' then x, t := x', t'
```

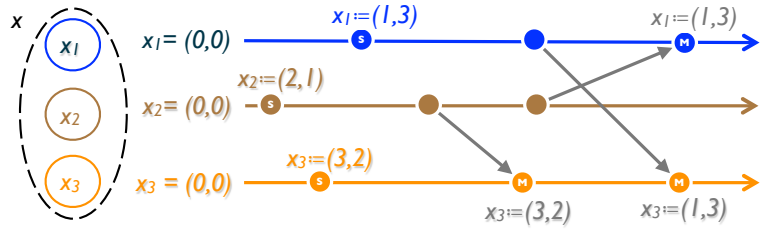

Figure 7: Integer LWW Register (state-based). Payload is a pair (value, timestamp)

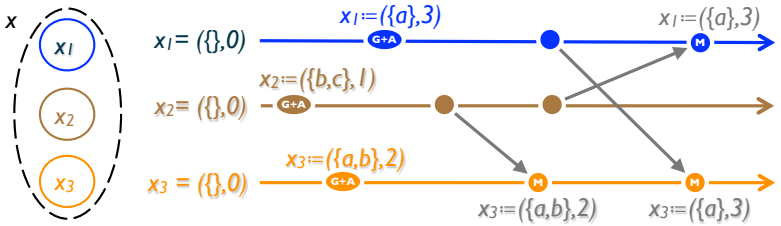

Figure 8: LWW-Set (state-based). Payload is a pair (set, timestamp)

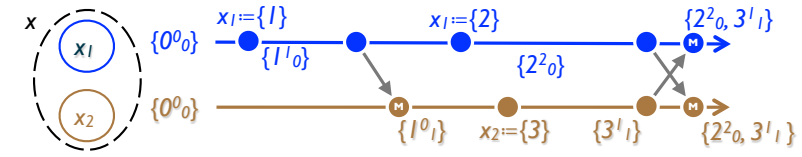

Figure 9: MV-Register (state-based)

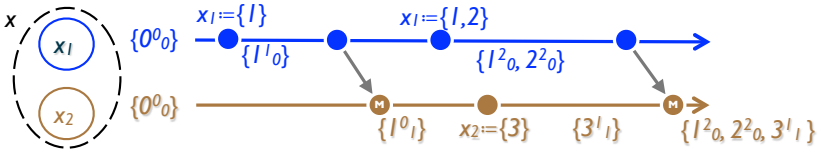

Figure 10: MV-Register counter-example

3.2.1 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 causal order; i.e., if assignment 1 happened-before assignment 2, the former's timestamp is less than the latter's [24]. This may be implemented as a per-replica counter concatenated with a unique replica identifier, such as its MAC address [24].

The state-based LWW-Register is presented in Specification 8. The type of the value can be any (local) data type X. The *value* operation returns the current value. The *assign* operation updates the payload with the new assigned value, and generates a new timestamp. The monotonic semilattice orders two values by their associated timestamp; *merge* procedure selects the value with the maximal timestamp. Clearly, this data type is a CvRDT. Figure 7 illustrates an integer LWW-Register.

Specification 9 presents the op-based LWW-Register. Operation *assign* generates a new timestamp at the source. Downstream, the update takes effect only if the new timestamp is greater than the current one. Because of the way timestamps are generated, this preserves the sequential semantics; concurrent assignments commute since, whatever the order of execution, only the one with the highest timestamp takes effect.

LWW-Registers, first described by Thomas [20], 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). Many other uses are possible; for instance, in Figure 8, X is a set (LWW-Set).

Specification 10 State-based Multi-Value Register (MV-Register)

payload set S ▷ set of (x, V) pairs; $x \in X$; V its version vector
  initial $\{(\bot, [0, \ldots, 0])\}$
query $incVV$ () : integer[n] V'
  let $g = myID()$
  let $\mathcal{V} = \{V | \exists x : (x, V) \in S\}$
  let $V' = [\ \max_{V \in \mathcal{V}}(V[j])\ ]_{j \neq g}$
  let $V'[g] = \max_{V \in \mathcal{V}}(V[g]) + 1$
update assign (set R) ▷ set of elements of type X
  let $V = incVV()$
  $S := R \times \{V\}$
query $value$ () : set S'
  let $S' = S$
compare (A, B) : boolean b
  let $b = (\forall (x, V) \in A, (x', V') \in B : V \leq V')$
merge (A, B) : payload C
  let $A' = \{(x, V) \in A | \forall (y, W) \in B : V \parallel W \vee V \geq W\}$
  let $B' = \{(y, W) \in B | \forall (x, V) \in A : W \parallel V \vee W \geq V\}$
  let $C = A' \cup B'$

3.2.2 Multi-Value Register (MV-Register)

An alternative semantics is to define a LUB operation that merges concurrent assignments, for instance taking their union, as in file systems such as Coda [19] or in Amazon's shopping cart [10]. Clients can later reduce multiple values to a single one, by a new assignment. Alternatively, in Ficus [34] *merge* is an application-specific resolver procedure.

To detect concurrency, a scalar timestamp (as above) is insufficient. Therefore the state-based payload is a set of *(X, versionVector)* pairs, as shown in Spec. 10, and illustrated in Figure 9 (the op-based specification is left as an exercise to the reader). A *value* operation returns a copy of the payload. As usual, *assign* overwrites; to this effect, it computes a version vector that dominates all the previous ones. Operation *merge* takes the union of every element in each input set that is not dominated by an element in the other input set.

As noted in the Dynamo article [10], Amazon's shopping cart presents an anomaly, whereby a removed book may re-appear. This is illustrated in the example of Figure 10. The problem is that, MV-Register does not behave like a set, contrary to what one might expect since its payload is a set. We will present clean specifications of Sets in Section 3.3.

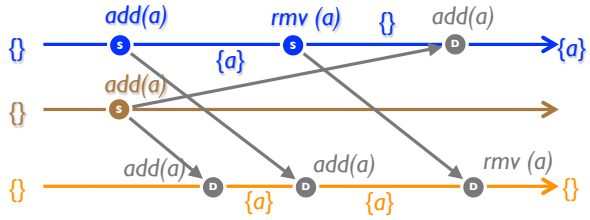

Figure 11: Counter-example: Set with concurrent *add* and *remove* (op-based)

3.3 Sets

Sets constitute one of the most basic data structures. Containers, Maps, and Graphs are all based on Sets.

We consider mutating operations *add* (takes its union with an element) and *remove* (performs a set-minus). Unfortunately, these operations do not commute. Therefore, a Set cannot both be a CRDT and conform to the sequential specification of a set.

To illustrate, consider the naïve replicated op-based set of Figure 11. Operations *add* and *remove* are applied sequentially as they arrive. Initially, the set is empty. Replica 1 adds element a, then removes a; its state is again empty. Replica 2 adds the same element a; when Replica 1 applies this operation, its (final) state becomes $\{a\}$. Replica 3 receives the two *add* operations; the second one has no effect since a is already in the set. Then it receives the *remove*, which makes its state empty. Both Replica 1 and Replica 3 have applied all operations in causal order, yet they diverge.

Thus, a CRDT can only approximate the sequential set. Hereafter, we will examine a few different approximations that differ mainly by the result of concurrent $add(e) \parallel_d remove(e)$. The 2P-Set hereafter (Section 3.3.2) gives precedence to *remove*, OR-Set (Section 3.3.5) to *add*.[4]

3.3.1 Grow-Only Set (G-Set)

The simplest solution is to avoid *remove* altogether. A Grow-Only Set (G-Set), illustrated in Figure 2, supports operations *add* and *lookup* only. The G-Set is useful as a building block for more complex constructions.

In both the state- and op-based approaches, the payload is a set. Since *add* is based on union, and union is commutative, the op-based implementation converges; G-Set is a

[4] Note that two clients may concurrently *remove* the same element. Despite a superficial similarity, our Sets are different from Tuple Spaces [7], in which *remove*s are totally ordered.

Specification 11 State-based grow-only Set (G-Set)

```
1: payload set A
2:     initial ∅
3: update add (element e)
4:     A := A ∪ {e}
5: query lookup (element e) : boolean b
6:     let b = (e ∈ A)
7: compare (S, T) : boolean b
8:     let b = (S.A ⊆ T.A)
9: merge (S, T) : payload U
10:    let U.A = S.A ∪ T.A
```

Specification 12 State-based 2P-Set

```
1: payload set A, set R                          ▷ A: added; R: removed
2:     initial ∅, ∅
3: query lookup (element e) : boolean b
4:     let b = (e ∈ A ∧ e ∉ R)
5: update add (element e)
6:     A := A ∪ {e}
7: update remove (element e)
8:     pre lookup(e)
9:     R := R ∪ {e}
10: compare (S, T) : boolean b
11:    let b = (S.A ⊆ T.A ∨ S.R ⊆ T.R)
12: merge (S, T) : payload U
13:    let U.A = S.A ∪ T.A
14:    let U.R = S.R ∪ T.R
```

CmRDT. The precondition to *add* is *true*, therefore delivery order is empty (operations can execute in any order).

In the state-based approach, *add* modifies the local state as shown in Specification 11. We define a partial order on some states S and T as $S \leq T \Leftrightarrow S \subseteq T$ and the *merge* operation as $merge(S, T) = S \cup T$. Thus defined, states form a monotonic semilattice and *merge* is a LUB operation; G-Set is a CvRDT.

3.3.2 2P-Set

Our second variant is a Set where an element may be added and removed, but never added again thereafter. This Two-Phase Set (2P-Set) is specified in Specification 12 and illustrated in Figure 3. It combines a G-Set for adding with another for removing; the latter is colloquially known as the *tombstone set*. To avoid anomalies, removing an element is allowed only if the source observes that the element is in the set.

Specification 13 U-Set: Op-based 2P-Set with unique elements

```
1:  payload set S
2:      initial ∅
3:  query lookup (element e) : boolean b
4:      let b = (e ∈ S)
5:  update add (element e)
6:      atSource (e)
7:          pre e is unique
8:      downstream (e)
9:          S := S ∪ {e}
10: update remove (element e)
11:     atSource (e)
12:         pre lookup(e)                          ▷ 2P-Set precondition
13:     downstream (e)
14:         pre add(e) has been delivered          ▷ Causal order suffices
15:         S := S \ {e}
```

State-based 2P-Set The state-based variant is in Specification 12. The payload is composed of local set A for adding, and local set R for removing. The *lookup* operation checks that the element has been added but not yet removed. Adding or removing a same element twice has no effect, nor does adding an element that has already been removed. The *merge* procedure computes a LUB by taking the union of the individual added- and removed-sets. Therefore, this is indeed a CRDT.

Note that a tombstone is required to ensure that, if a removed element is received by a downstream replica before its added counterpart, the effect of the *remove* still takes precedence.

Op-based 2P-Set Consider now the op-based variant of 2P-Set. Concurrent adds of the same element commute, as do concurrent removes. Concurrent operations on different elements commute. Operation pairs on the same element $add(e)/add(e)$ and $remove(e) \parallel_d remove(e)$ commute by definition; and $remove(e)$ can occur only after $add(e)$. It follows that this data type is indeed a CRDT.

U-Set 2P-Set can be simplified under two standard assumptions, as in Specification 13. If elements are unique, a removed element will never be added again.[5] If, furthermore, a downstream precondition ensures that $add(e)$ is 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.) Spec. 13 captures this data type, which we call U-Set.

[5] Function *unique* returns a unique value. It could be a Lamport clock, as in Section 3.2.1; alternatively, it might select a number randomly from a large space, ensuring uniqueness with high probability.

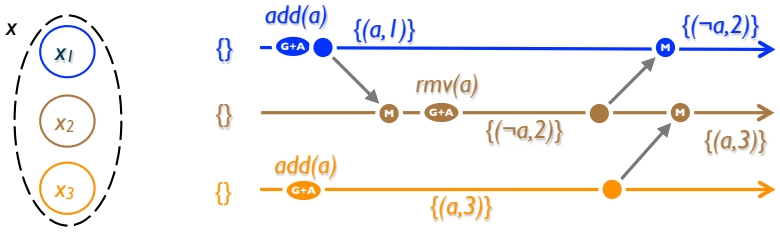

Figure 12: LWW-element-Set; elements masked by one with a higher timestamp are elided (state-based)

If we assume (as seems to be the practice) that every element in a shopping cart is unique, then U-Set satisfies the intuitive properties requested of a shopping cart, without the Dynamo anomalies described in Section 3.2.2.

U-Set is a CRDT. As every element is assumed unique, *add*s are independent. A *remove* operation must be causally after the corresponding *add*. Accordingly, there can be no concurrent *add* and *remove* of the same element.

3.3.3 LWW-element-Set

An alternative LWW-based approach,[6] which we call LWW-element-Set (see Figure 12), attaches a timestamp to each element (rather than to the whole set, as in Figure 8). Consider add-set A and remove-set R, each containing (element, timestamp) pairs. To add (resp. remove) an element e, add the pair $(e, now())$, where *now* was specified earlier, to A (resp. to R). Merging two replicas takes the union of their add-sets and remove-sets. An element e is in the set if it is in A, and it is not in R with a higher timestamp: $lookup(e) = \exists t, \forall t' > t : (e,t) \in A \wedge (e,t') \notin R)$. Since it is based on LWW, this data type is convergent.

3.3.4 PN-Set

Yet another variation is to associate a counter to each element, initially 0. Adding an element increments the associated counter, and removing an element decrements it. The element is considered in the set if its counter is strictly positive. An actual use-case is Logoot-Undo [43], a (totally-ordered) set of elements for text editing.

However, as noted earlier (Section 3.1.3), a CRDT counter can go positive or negative; adding an element whose counter is already negative has no effect. Consider the following example, illustrated in Figure 13. Initially, our PN-Set is empty. Replica 1 performs $add(e)$;

[6] Due to Hyun-Gul Roh [private communication].

Specification 14 Molli, Weiss, Skaf Set

```
1:  payload set S = {(element, count), ...}                    ▷ set of pairs
2:      initial  E × {0}                                       ▷ Initialise all counts to 0
3:  query lookup (element e) : boolean b
4:      let b = ((e, k) ∈ S ∧ k > 0)
5:  update add (element e)
6:      atSource (e) : integer j                               ▷ j: increment
7:          if ∃(e, k) ∈ S : k ≤ 0 then
8:              let j = |k| + 1
9:          else
10:             let j = 1
11:     downstream (e, j)
12:         let k' : (e, k') ∈ S
13:         S := S \ {(e, k')} ∪ {(e, k' + j)}
14: update remove (element e)
15:     atSource (e)
16:         pre lookup(e)
17:     downstream (e)
18:         S := S \ {(e, k')} ∪ {(E, k' − 1)}
```

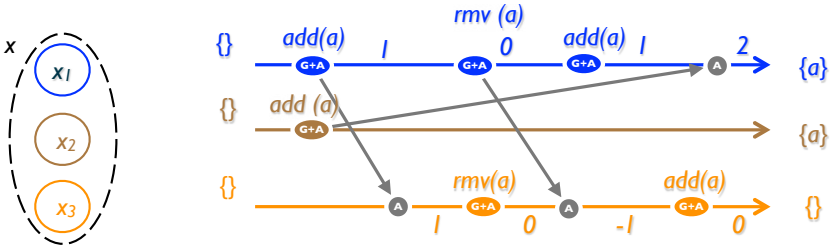

Figure 13: PN-Set (op-based)

Specification 15 Op-based Observed-Remove Set (OR-Set)

```
1:  payload set S                                   ▷ set of pairs { (element e, unique-tag u), ... }
2:      initial ∅
3:  query lookup (element e) : boolean b
4:      let b = (∃u : (e, u) ∈ S)
5:  update add (element e)
6:      atSource (e)
7:          let α = unique()                        ▷ unique() returns a unique value
8:      downstream (e, α)
9:          S := S ∪ {(e, α)}
10: update remove (element e)
11:     atSource (e)
12:         pre lookup(e)
13:         let R = {(e, u)|∃u : (e, u) ∈ S}
14:     downstream (R)
15:         pre ∀(e, u) ∈ R : add(e, u) has been delivered ▷ U-Set precondition; causal order suffices
16:         S := S \ R                              ▷ Downstream: remove pairs observed at source
```

element e has a count of 1. The operation propagates to Replica 3. Now Replicas 1 and 3 both concurrently execute *remove*(e); after Replica 3 applies both operations, e has a count of -1. A subsequent *add*(e) has no effect: thus, after adding an element to an empty "set" it remains empty! For some applications, this may be the intended semantics. for instance, in an inventory, a negative count may account for goods in transit. In others, this may be considered a bug.

Although the semantics are strange, PN-Set converges; thus if Replica 2 concurrent executes *add*(e) all replicas converge to state $\{e\}$.

An alternative construction due to Molli, Weiss and Skaf [private communication] is presented in Specification 14. To avoid the above *add* anomaly, *add* increments a negative count of k by $|k| + 1$; however this presents other anomalies, for instance where *remove* has no effect.

Both these constructs are CRDTs because they combine two CRDTS, a Set and a Counter.

3.3.5 Observed-Remove Set (OR-Set)

The preceding Set constructs have practical applications, but are somewhat counter-intuitive. In 2P-Set (Section 3.3.2), a removed element can never be added again; in LWW-Set (Figure 8) the outcome of concurrent updates depends on opaque details of how timestamps are allocated.

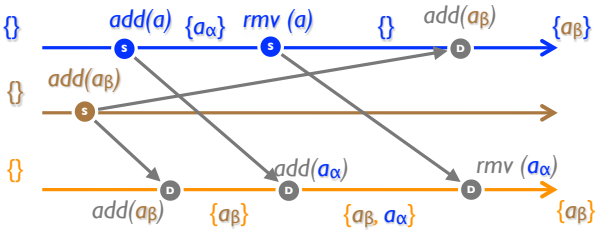

Figure 14: Observed-Remove Set (op-based)

We present here the Observed-Removed Set (OR-Set), which supports adding and removing elements and is easily understandable. The outcome of a sequence of adds and removes depends only on its causal history and conforms to the sequential specification of a set. In the case of concurrent add and remove of the same element, add has precedence (in contrast to 2P-Set).

The intuition is to tag each added element uniquely, without exposing the unique tags in the interface. When removing an element, all associated unique tags observed at the source replica are removed, and only those.

Spec. 15 is op-based. The payload consists of a set of pairs *(element, unique-identifier)*. A *lookup*(e) extracts element e from the pairs. Operation *add*(e) generates a unique identifier in the source replica, which is then propagated to downstream replicas, which insert the pair into their payload. Two *add*(e) generate two unique pairs, but *lookup* masks the duplicates.

When a client calls *remove*(e) at some source, the set of unique tags associated with e at the source is recorded. Downstream, all such pairs are removed from the local payload. Thus, when *remove*(e) happens-after any number of *add*(e), all duplicate 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 14. The two *add*(a) operations generate unique tags α and β. 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.

OR-Set is a CRDT. Concurrent *add*s commute since each one is unique. Concurrent *remove*s commute because any common pairs have the same effect, and any disjoint pairs have independent effects. Concurrent *add*(e) and *remove*(f) also commute: if $e \neq f$ they are independent, and if $e = f$ the *remove* has no effect.

We leave the corresponding state-based specification as an exercise for the reader. Since every *add* is effectively unique, a state-based implementation could be based on U-Set.

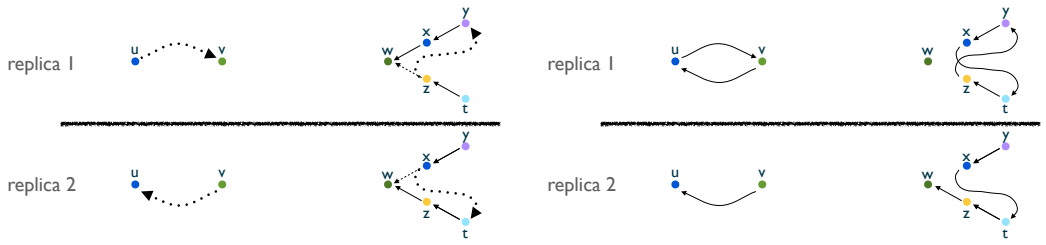

Figure 15: Maintaining strong properties in a graph (counter-example). Left: initial state and update (dashed edges removed, dotted edges added); right: final state.

Specification 16 2P2P-Graph (op-based)

```
1:  payload set VA, VR, EA, ER
2:                                   ▷ V: vertices; E: edges; A: added; R: removed
3:      initial ∅, ∅, ∅, ∅
4:  query lookup (vertex v) : boolean b
5:      let b = (v ∈ (VA \ VR))
6:  query lookup (edge (u, v)) : boolean b
7:      let b = (lookup(u) ∧ lookup(v) ∧ (u, v) ∈ (EA \ ER))
8:  update addVertex (vertex w)
9:      atSource (w)
10:     downstream (w)
11:         VA := VA ∪ {w}
12: update addEdge (vertex u, vertex v)
13:     atSource (u, v)
14:         pre lookup(u) ∧ lookup(v)          ▷ Graph precondition: E ⊆ V × V
15:     downstream (u, v)
16:         EA := EA ∪ {(u, v)}
17: update removeVertex (vertex w)
18:     atSource (w)
19:         pre lookup(w)                      ▷ 2P-Set precondition
20:         pre ∀(u, v) ∈ (EA \ ER) : u ≠ w ∧ v ≠ w   ▷ Graph precondition: E ⊆ V × V
21:     downstream (w)
22:         pre addVertex(w) delivered         ▷ 2P-Set precondition
23:         VR := VR ∪ {w}
24: update removeEdge (edge (u, v))
25:     atSource ((u, v))
26:         pre lookup((u, v))                 ▷ 2P-Set precondition
27:     downstream (u, v)
28:         pre addEdge(u, v) delivered        ▷ 2P-Set precondition
29:         ER := ER ∪ {(u, v)}
```

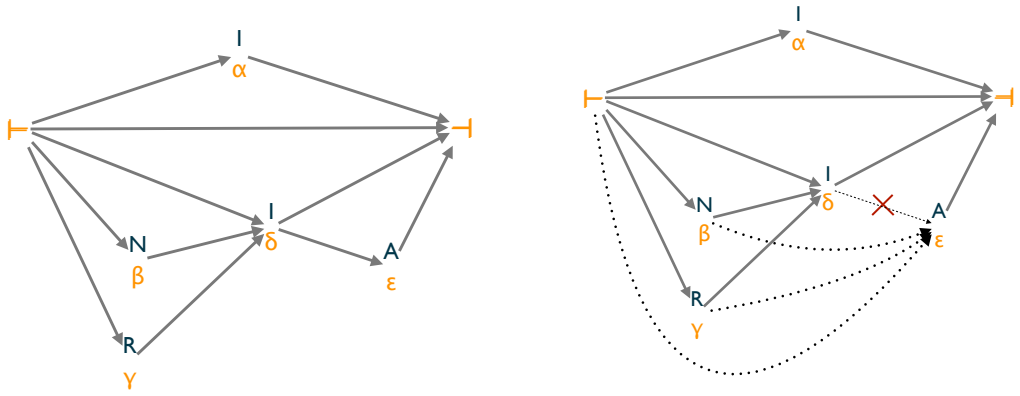

Figure 16: Monotonic DAG. Left: Greek letters indicate vertex identifiers; roman letters are characters in a text-editing application. Right: Remove OK only if paths are maintained. Dashed: removed; dotted: added.

3.4 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 to V and E.

Because of the invariant $E \subseteq V \times V$, operations on vertices and edges are not independent. An edge may be added only if the corresponding vertices exist; conversely, a vertex may be removed only if it supports no edge. What should happen upon concurrent $addEdge(u, v) \parallel_d removeVertex(u)$? We see three possibilities: *(i)* Give precedence to $removeVertex(u)$: all edges to or from u are removed as a side effect. This it is easy to implement, by using tombstones for removed vertices. *(ii)* Give precedence to $addEdge(u, v)$: if either u or v has been removed, it is restored. This semantics is more complex. *(iii)* $removeVertex(u)$ is delayed until all concurrent *addEdge* operations have executed. This requires synchronisation. Therefore, we choose Option *(i)*. Our Spec. 16 uses a 2P-Set for vertices (in order to have tombstones) an another for edges (since they are not unique).

A 2P2P-Graph is the combination of two 2P-Sets; as we showed, the dependencies between them are resolved by causal delivery. Dependencies between *addEdge* and *removeEdge*, and between *addVertex* and *removeVertex* are resolved as in 2P-Set. Therefore, this construct is a CRDT.

Specification 17 Add-only Monotonic DAG (op-based)

payload set V, set E ▷ V: vertices; E: edges
  initial $\{\vdash, \dashv\}, \{(\vdash, \dashv)\}$ ▷ Initialised with two sentinels and single edge.
query *lookup* (vertex v) : boolean b
  let $b = (v \in V)$
query *lookup* (edge (u, v)) : boolean b
  let $b = ((u, v) \in E)$
query *path* (edge (u, v)) : boolean b
  let $b = (\exists w_1, \ldots, w_m \in V : w_1 = u \wedge w_m = v \wedge (\forall j : (w_j, w_{j+1}) \in E))$
update *addEdge* (vertex u, vertex v)
  atSource (u, v)
    pre *lookup*$(u) \wedge$ *lookup*(v) ▷ Graph precondition
    pre *path*(u, v) ▷ Monotonic-DAG condition
  downstream (u, v)
    pre *lookup*$(u) \wedge$ *lookup*(v) ▷ Graph precondition
    $E := E \cup \{(u, v)\}$
update *addBetween* (vertex u, v, w)
  atSource (u, v, w)
    pre v is unique
    pre *lookup*$(u) \wedge$ *lookup*(w) ▷ Graph precondition
    pre *path*(u, w) ▷ Monotonic-DAG condition
  downstream (u, w, v)
    pre *lookup*$(u) \wedge$ *lookup*(w) ▷ Graph precondition
    $V := V \cup \{v\}$
    $E := E \cup \{(u, v), (v, w)\}$

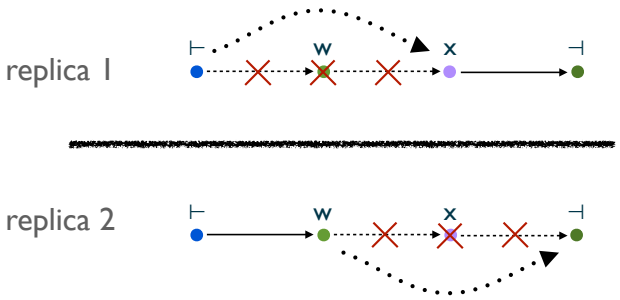

Figure 17: Monotonic DAG: remove is not live. Dashed: removed; dotted: added.

3.4.1 Add-only monotonic DAG

In general, maintaining a particular shape, such as a tree or a DAG, cannot be done by a CRDT.[7] Such a global invariant cannot be determined locally; maintaining it requires synchronisation.

Figure 15 presents two counter-examples. Replicated graph u, v contains no edge. A client adds edge (u, v) at Replica 1; concurrently another client adds (v, u) at Replica 2. Each of these maintains the DAG shape, but when the changes at Replica 2 propagate to Replica 1, the graph is cyclic. Similarly, initially the graph w, x, y, z, t form a replicated tree. Clients at Replicas 1 and 2 add and remove edges as indicated in the figure, maintaining the tree shape. However, after propagation, the graph is cyclic.

However, some stronger forms of acyclicity are implied by local properties, for instance a *monotonic DAG*, in which an edge may be added only if it oriented in the same direction as an existing path.[8] That is, the new edge can only strengthen the partial order defined by the DAG; it follows that the graph remains acyclic. Specification 17 specifies an Add-Only Monotonic DAG, illustrated in Figure 16 (left). The DAG is initialised with left and right sentinels $\vdash$ and $\dashv$ and edge $(\vdash, \dashv)$. The only operation for adding a vertex is *addBetween* in order to maintain the DAG property. The first operation must be $addBetween(\vdash, \dashv)$.

Add-only Monotonic DAG is a CRDT, because concurrent *addEdge* (resp. *addBetween*) either concern different edges (resp. vertices) in which case they are independent, or the same edge (resp. vertex), in which case the execution is idempotent.

Generalising monotonic DAG to removals proves problematic. It should be OK to remove an edge (expressed as a precondition on *removeEdge*) as long as this does not disrupt paths between distinct vertices. Namely, if there exists a path from u to v, and $w \neq u, v$, then a path should remain after removing (x, w) or (w, x), whatever $x \in V$. A client could satisfy it by creating an alternative path if necessary, e.g., by calling $addEdge(u, v)$ before removing (u, w), as illustrated in Figure 16 (right).

Unfortunately, this is not live, as illustrated by the scenario of Figure 17. Here, a client adds a vertex around w, removes the edges to and from w, and finally removes w. Concurrently, another client (at another source replica) does the same with x. When the former operations propagate, the downstream precondition of *addEdge* is false at Replica 2, and, consequently the downstream precondition of *removeVertex* can never be satisfied; and vice-versa.

3.4.2 Add-Remove Partial Order data type

The above issues with vertex removal do not occur if we consider a Partial Order data type rather than a DAG. Since a partial order is transitive, implicitly all alternate paths exist;

[7] Unless of course the graph has the required shape for some other reason. For instance, a 2P2P-Graph could record causal dependence between events in a distributed system, which is acyclic.

[8] It is inspired by WOOT, a CRDT for concurrent editing [30].

Specification 18 Add-Remove Partial Order

```
1:  payload set VA, VR, E                          ▷ V: vertices; E: edges; A: added, R: removed
2:      initial {⊢, ⊣}, ∅, {(⊢, ⊣)}                ▷ Edge between left and right sentinels
3:  query lookup (vertex v) : boolean b
4:      let b = (v ∈ VA \ VR)
5:  query before (vertex u,v) : boolean b
6:      pre lookup(u) ∧ lookup(v)
7:      let b = (∃w_1, ..., w_m ∈ VA : w_1 = u ∧ w_m = v ∧ (∀j : (w_j, w_{j+1}) ∈ E))
8:                                                 ▷ Removed vertices are considered too
9:  update addBetween (vertex u, v, w)
10:     atSource (u, v, w)
11:         pre w is unique
12:         pre before(u, w)                       ▷ Monotonic-DAG precondition
13:     downstream (u, v, w)
14:         pre u ∈ VA ∧ v ∈ VA
15:         VA := VA ∪ {v}
16:         E := E ∪ {(u, v), (v, w)}
17: update remove (vertex v)
18:     atSource (v)
19:         pre lookup(v)                          ▷ 2P-Set precondition
20:         pre v ≠ ⊢ ∧ v ≠ ⊣                      ▷ May not remove sentinels
21:     downstream (v)
22:         VR := VR ∪ {v}
```

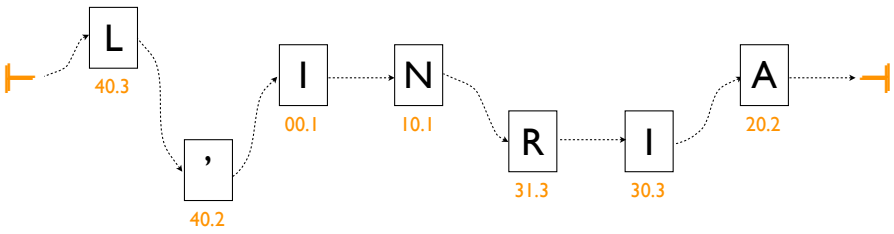

Figure 18: Replicated Growable Array (RGA)

thus the problematic precondition on vertex removal is not necessary. For the representation, we use a minimal DAG and compute transitive relations on the fly (operation *before*). To ensure transitivity, a removed vertex is retained as a tombstone.[9] Thus, Spec. 18 uses a 2P-Set for vertices, and a G-Set for edges.

We manage vertices as a 2P-Set. Concurrent *addBetween*s are either independent or idempotent. Any dependence between *addBetween* and *remove* is resolved by causal delivery. Thus this data type is a CRDT.

3.5 Co-operative text editing

Peer-to-peer co-operative text editing is a particularly interesting use case of an add-remove order. A text document is a sequence of text elements (characters, strings, XML tags, embedded graphics, etc.). Users sharing a document repeatedly insert a text element (*addBetween*) or remove one (*remove*). Using a CRDT for this ensures that concurrent edits never conflict and converge, even for users who remain disconnected from the network for long periods, as long as they eventually reconnect. Thus, the WOOT data structure for concurrent editing corresponds directly to the Add-Remove Partial Order of Specification 18.

A Partial Order presents a difficulty, as text is normally sequential, but two concurrent inserts at the same position remain unordered. A total order, or sequence, does not have this drawback, and in addition can be implemented much more efficiently.

A sequence for text editing (or just *sequence* hereafter) is a totally-ordered set of elements, each composed of a unique identifier and an atom (e.g., a character, a string, an XML tag, or an embedded graphic), supporting operations to add an element at some position, and to remove an element.[10] We now study two different sequence designs. Such a sequence is a CRDT because it a subclass of add-remove total order.

[9] We do not include operations *addEdge* or *removeEdge* because it is not clear what semantics would be reasonable.

[10] Note that despite the superficial similarity, a sequence cannot implement a queue or stack, as the latter support atomic *pop* operations.

Specification 19 Replicated Growable Array (RGA). Represented as a 2P-Set of vertices in a linked list. A vertex is a pair *(atom, timestamp)*. Timestamps are unique, positive, and increase consistently with causality.

```
1:  payload set VA, VR, E                                  ▷ VA, VR: 2P-set of vertices; E: edges
2:                                                         ▷ Vertex = (atom, timestamp)
3:      let ⊢ = (⊥, −1)
4:      let ⊣ = (⊥, 0)
5:      initial {⊢, ⊣}, ∅, {(⊢, ⊣)}                        ▷ Initially, a single edge (⊢, ⊣)
6:  query lookup (vertex v) : boolean b
7:      let b = (v ∈ VA \ VR)
8:  query before (vertex u, vertex v) : boolean b
9:      pre lookup(u) ∧ lookup(v)
10:     let b = (∃w_1, ..., w_m ∈ VA : w_1 = u ∧ w_m = v ∧ ∀j : (w_j, w_{j+1}) ∈ E)
11: query successor (vertex u) : vertex v
12:     pre lookup(u)
13:     let v ∈ VA : (u, v) ∈ E
14: query decompose (vertex u) : atom a, timestamp t
15:     let a, t : u = (a, t)                              ▷ Decompose u into atom, timestamp
16: update addRight (vertex u, atom a) : vertex w
17:     atSource (u, a) : w
18:         pre u ∈ VA \ (VR ∪ {⊣})                        ▷ Graph precondition
19:         let t = now()                                  ▷ Unique timestamp
20:         let w = (a, t)
21:     downstream (u, w)
22:         pre u ∈ VA                                     ▷ Graph precondition
23:         let a, t = decompose(w)                        ▷ p = u
24:         l, r := u, successor(u)
25:         b := true
26:         while b do                                     ▷ Find an edge (l, r) within which to splice w
27:             let a′, t′ = decompose(r)
28:             if t < t′ then                             ▷ Right position, wrong order
29:                 l, r := r, successor(r)                ▷ Iterate
30:             else                                       ▷ r = ⊣ ∨ t > t′
31:                 E := E \ (l, r) ∪ {(l, w), (w, r)}
32:                 b := false
33: update remove (vertex w)
34:     atSource (w)
35:         pre lookup(w)                                  ▷ 2P-Set precondition
36:     downstream (w)
37:         pre addRight(_, w) delivered                   ▷ 2P-Set precondition
38:         VR := VR ∪ {w}
```

3.5.1 Replicated Growable Array (RGA)

The Replicated Growing Array (RGA), due to Roh et al. [35] implements a sequence as a linked list (a linear graph), as illustrated in Figure 18. It supports operations $addRight(v, a)$, to add an element containing atom a immediately after element v. An element's identifier is a timestamp, assumed unique and ordered consistently with causality, i.e., if two calls to *now* return t and t', then if the former happened-before the latter, then $t < t'$ [24]. If a client inserts twice at the same position, as in "$addRight(v, a); addRight(v, b)$" the latter insert occurs to the *left* of the former, and has a higher timestamp. Accordingly, two downstream inserts at the same position are ordered in opposite order of their timestamps. As in Add-Remove Partial Order, removing a vertex leaves a tombstone, in order to accommodate a concurrent add operation.

For example, in Figure 18, timestamps are represented as a pair *(local-clock.client-UID)*. Client 3 added character I at time 30, then R at time 31, to the right of N. Clients 2 and 3 concurrently (at time 40) inserted an L and an apostrophe to the right of the beginning-of-text marker $\vdash$.

As noted above, RGA is a CRDT because it is a subclass of Add-Remove Partial Order.

3.5.2 Continuous sequence

An alternative approach to maintaining a mutable sequence is to place its elements in the continuum. Spec. 20 specifies a sequence based on identifying elements in a *dense* identifier space such as $\mathbb{R}$, i.e., where a unique identifier can always be allocated between any two given identifiers. Adding an element assigns it an appropriate identifier; identifiers are unique and totally ordered (and unrelated by causality).

As noted above, this data structure is a CRDT because it is a subclass of Add-Remove Partial Order. More directly, concurrent adds commute because they occur at different positions in the continuum. Adding and deleting different elements commute because they are independent operations. Adding an element precedes removing it, and they will be applied downstream in that order, by the U-Set assumption of causal delivery.

Its performance depends crucially on the implementation of identifiers and of *allocateIdentifierBetween*. Using real numbers would certainly be possible but costly.

Identifier tree Instead, we represent the continuum using a tree. The first element is allocated at the root. Thereafter, it is always possible to create a new leaf e between any two nodes n and m, either to the right of n or to the left of m. To allocate a node e to the right of a node n:

(i) If n has a right sibling $m' \leq m$ and there exists a free unique tag m'' such that $m < m'' < m'$, allocate e as m''.[11]

[11] As tags are integers, there is not an infinite supply of unique free tags between two given tags.

Specification 20 Mutable sequence based on the continuum

```
1:  payload set S                                   ▷ U-Set of (X, identifier) pairs; X: some type
2:      initial ∅
3:  query lookup (element e) : boolean b
4:      let b = (e ∈ S)
5:  query decompose (element e) : X x, identifier i
6:      let x, i : e = (x, i)
7:  query before (element e, element e′) : boolean b
8:      pre lookup(e) ∧ lookup(e′)
9:      let x, i = decompose(e)
10:     let x′, i′ = decompose(e′)
11:     let b = (i < i′)
12: query allocateIdentifierBetween (identifier i, j) : identifier k
13:     pre i < j
14:     let k : i < k < j and k unique
15: update addBetween (element e, X b, element e′) : element f
16:     atSource (e, b, e′) : f
17:         pre lookup(e) ∧ lookup(e′)
18:         pre before(e, e′)
19:         let x, i = decompose(e)
20:         let x′, i′ = decompose(e′)
21:         let f = (b, allocateIdentifierBetween(i, i′))
22:     downstream (f)
23:         S := S ∪ {f}
24: update remove (element e)
25:     atSource (e)
26:         pre e ∈ S                                   ▷ U-Set precondition
27:     downstream (e)
28:         pre add(e) delivered                        ▷ U-Set precondition
29:         S := S \ {e}
```

(ii) Otherwise, if n has no right child, allocate e as the right child of n.

(iii) Otherwise, let n' be the leftmost descendant of n's right child; clearly, $n < n'$. Recursively, allocate e to the left of n'.

Allocating to the left of m is symmetric, substituting left for right and vice-versa.

Identifiers A node identifier is a (possibly empty) sequence of pairs $(d_1, u_1)\bullet \ldots \bullet (d_m, u_m)$, one per level in the tree. At each level, d_j indicates the direction (0 for left child, 1 for right child), and u_j is a unique integer tag.

The root node has the empty identifier. A child of some node n has identifier $m = n \bullet (d, u)$. Siblings are ordered by their relative identifiers; thus siblings $m = n \bullet (d, u)$ and $m' = n \bullet (d', u')$ compare as $m < m' \Leftrightarrow d < d' \vee (d = d' \wedge u < u')$). As the tree is traversed in in-order, a parent n is greater than its left children and less than its right children; i.e., n compares with its child $m = n \bullet (d, u)$ thus: $n < m \Leftrightarrow d = 0$.

In summary, two identifiers n and n' compare as follows. Let $j \geq 0$ be the length of their longest common prefix: $n = (d_1, u_1) \bullet \ldots \bullet (d_j, u_j) \bullet (d_{j+1}, u_{j+1}) \bullet \ldots \bullet (d_{j+k}, u_{j+k})$ and $n' = (d_1, u_1) \bullet \ldots \bullet (d_j, u_j) \bullet (d'_{j+1}, u'_{j+1}), \ldots \bullet (d'_{j+k'}, u'_{j+k'})$. Then:

(i) If $k = 0$ and $k' = 0$, the two identifiers are identical.

(ii) If $k = 0$ and $k' > 0$, then n' is a descendant of n. It is a right descendant iff $d'_{j+1} = 1$, i.e., $n < n' \Leftrightarrow d'_{j+1} = 1$.

(iii) Symmetrically, if $k > 0$ and $k' = 0$ then $n < n' \Leftrightarrow d_{j+1} = 0$.

(iv) If $k > 0$ and $k' > 0$, then either n and n' are siblings, or they descend from siblings. In both cases, they are ordered by the siblings' relative identifiers: $n < n' \Leftrightarrow d_{j+1} < d'_{j+1} \vee (d_{j+1} = d'_{j+1} \wedge u_{j+1} < u'_{j+1})$.

Experience Two tree-based CRDTs designed for concurrent editing are Logoot and Treedoc, differing in the details. Logoot [43] always allocates to the right, thus does not require d. Treedoc [25, 32] groups sequential adds from the same source into a compact binary tree with tombstones (no u part), and uses a sparse, unique tag for concurrent adds only.

If the tree is well balanced, the identifier size adjusts to the size of the sequence, and operations have logarithmic complexity. Experiments with text editing show that over time the tree becomes unbalanced. Rebalancing the tree is a kind of garbage collection, which we discuss in the next section.

4 Garbage collection

Our practical experience with CRDTs shows that they tend to become inefficient over time, as tombstones accumulate and internal data structures become unbalanced [25, 32]. To avoid these issues, we investigate *garbage collection* (GC) mechanisms. Solving distributed GC would be difficult without synchronisation. We distinguish two kinds of GC problems, which differ by their liveness requirements.

When these requirements are not met, GC may block. We consider this to be acceptable, as GC does not impact correctness (only performance), and the normal operations in the object's interface remain live.

GC issues concern both state- and op-based CRDTs. However, as CmRDTs hide some complexity by requiring stronger channels, this also affects GC. Indeed, reliable broadcast channels often implement GC mechanisms of their own.

4.1 Stability problems

An update f will sometimes add some information $r(f)$ to the payload in order to deal cleanly with operations concurrent with f. As an example, in the Add-Remove Partial Order of Section 3.4.2, *remove* leaves a tombstone in order to allow *addBetween*s to proceed.

Once f is *stable*, i.e., all operations concurrent with f have been delivered, $r(f)$ serves no useful purpose. A GC opportunity exists to detect this condition and discard $r(f)$.

Definition 4.1 (Stability)**.** *Update f is stable at replica x_i (noted $\Phi_i(f)$) if all updates concurrent to f according to delivery order $<_d$ are already delivered at x_i. Formally,* $\Phi_i(f) \Leftrightarrow \forall j : f \in \mathcal{C}(x_j) \wedge \nexists g \in \mathcal{C}(x_j) \setminus \mathcal{C}(x_i) : f \parallel_d g$.

Liveness of Φ requires that the set of replicas be known and that they not crash permanently (undetectably). Under these assumptions, the stability algorithm of Wuu and Bernstein [44] can be adapted. The algorithm assumes causal delivery. An update g has an associated vector clock $v(g)$. Replica x_i maintains the last vector clock value received from every other replica x_j, noted $V_i^{\min}(j)$, which identifies all updates that x_i knows to have been delivered by x_j. Replica must periodically propagate its vector clock to update $V_i^{\min}$ values, possibly by sending empty messages. With this information, $(\forall j : V_i^{\min}(j) \geq v(f)) \Rightarrow \Phi_i(f)$. Importantly, the information required is typically already used by a reliable delivery mechanism, and GC can be performed in the background.

For instance, our Add-Remove Partial Order data type from Section 3.4.2 could use Φ to remove tombstones left by *remove* once all concurrent *addBetween* updates have been delivered. In the state-based emulation of Section 2.4.2, stable messages could be discarded (this is Wuu's original motivation). RGA also uses this approach (Section 3.5.1), as do Treedoc and Logoot [32, 35, 43].

Specification 21 Op-based Observed-Remove Shopping Cart (OR-Cart)

```
 1: payload set S                                  ▷ triplets { (isbn k, integer n, unique-tag u), ... }
 2:     initial ∅
 3: query get (isbn k) : integer n
 4:     let N = {n' | (k', n', u') ∈ S ∧ k' = k)}
 5:     if N = ∅ then
 6:         let n = 0
 7:     else
 8:         let n = ∑ N
 9: update add (isbn k, integer n)
10:     atSource (k, n)
11:         let α = unique()
12:         let R = {(k', n', u') ∈ S | k' = k}
13:     downstream (k, n, α, R)
14:         pre ∀(k, n, u) ∈ R : add(k, n, u) has been delivered      ▷ U-Set precondition
15:         S := (S \ R) ∪ {(k, n, α)}      ▷ Replace elements observed at source by new one
16: update remove (isbn k)
17:     atSource (k)
18:         let R = {(k', n', u') ∈ S | k' = k}
19:     downstream (R)
20:         pre ∀(k, n, u) ∈ R : add(k, n, u) has been delivered      ▷ U-Set precondition
21:         S := S \ R                  ▷ Downstream: remove elements observed at source
```

4.2 Commitment problems

Some GC problems require a stronger form of synchronisation. One example is resetting the payload across all replicas; for instance, safely removing an entry in a Counter (or in a vector clock), removing tombstones from a 2P-Set (thus allowing deleted elements to be added again) or rebalancing the tree in Treedoc [25]. In first approximation, this requires an atomic, unanimous agreement between all replicas, i.e., a commitment protocol such as 2-Phase Commit or Paxos Commit [15]. The set of replicas must be known, and liveness requires that they all be reachable and responsive.

To overcome these strong requirements, Leţia et al. [25] perform commitment only by a small, stable subset of replicas, called the core. The other replicas asynchronously reconcile their state with core replicas.

5 Putting CRDTs to work

We now turn to a concrete example, maintaining shopping carts in an e-commerce bookstore. A shopping cart must be always available for writes, despite failures or disconnection [10]. To ensure reliability, data is replicated across both within a data centre for throughput,

and across several geographically-distant servers for reliability. Given these assumptions, linearisability would incur long response times; CRDTs provide an the ideal solution.

5.1 Observed-remove Shopping Cart

We define a shopping cart data type as a map from an ISBN number (a unique number representing the book edition) to an integer representing the number of units of the book the user wants to buy. Any of the Set abstractions presented earlier extends readily to a Map; we choose to extend OR-set presented in Section 3.3.5, as it minimises anomalies. An element is a $(key, value)$ pair; concretely the key is a book ISBN (a unique product identifier), and the value is a number of copies.

An op-based OR-Cart is presented in Spec. 21. The payload is a set of triplets *(key, value, unique-identifier)*, all initially empty. Two update operations are defined. The *add* operation adds a new, unique, from ISBN to value, which co-exists with existing mappings.

The *remove* operation removes all existing mappings for a given key. The source replica computes the set of triplets with the given key. Downstream, the update removes the triplets computed by the source from the downstream payload. The downstream precondition is the same as in 2P-Set and U-Set, namely, that the corresponding *add*s have been delivered; causal delivery is sufficient.

To order a new book, or to increase the number of copies, the client should call *add*. To cancel an order, the client should call *remove*. Checking out also calls *remove*. Decreasing the number of copies requires to first cancel the existing order, then adding the number required.

We now prove that OR-Cart is a CRDT by showing that concurrent updates commute. Two *add*s commute, since each triplet is unique. Also, two *remove*s commute, as the downstream set-minus operations are either independent or idempotent. Operation *add* is independent of a concurrent *remove*, as its triplets are unique.

5.2 E-commerce bookstore

Our e-commerce bookstore maintains the following information. Each user account has a separate OR-Cart. Assuming accounts are uniquely identified, the mapping from user to OR-Cart can be maintained by a U-Map, derived from U-Set in the obvious way. The shopping cart is created when the account is first created, and removed when it is deleted from the system.

Let us assume a web interface to the shopping cart. When the user selects book b with quantity q, the interface calls $add(b, q)$. If the user increases the quantity to q', the interface calls $add(b, q' - q)$. To decrease the quantity to q', the interface calls $remove(b)$ followed by $add(b, q')$. If the user cancels the book, or brings the quantity to zero, the interface calls $remove(b)$.

Assume a user calls some operation based on the observed state of his shopping cart. Delivery order ensures that, for each product, the state of the shopping cart reflects the last operation that the user observed. However, updates might be received by replicas in different states, either because failures cause them to be out of synch, as reported by Amazon [10], or when two users (e.g., family members) share the same account. In this case, although the state observed by the user may be stale, our approach minimises anomalies. Concurrent *add*s are merged as expected; a *remove* concurrent with an *add* will cancel the products already in the cart, but not those just added, which we believe is the cleanest semantics in this case.

This design remains simple and does not incur the remove anomaly reported for Dynamo [10], and does not bear the cost of the version vector needed by Dynamo's MV-Register approach.

6 Comparison with previous work

Eventual consistency has been a topic of research in highly-available, large-scale asynchronous systems [37]. With the explosive growth of peer-to-peer, edge computing, grid and cloud systems, eventual consistency has become an urgent issue for the industry [5, 41]. Contrary to much previous work [10, for instance], we take a formal approach grounded in the theory of commutativity and monotonic semilattices. However, we are far from being the first to study commutativity as a way to increase performance, availability, responsiveness, and to provide consistency at low cost.

6.1 Commutativity in transactional systems

Gray et al. show that reconciliation rate is a critical scalability factor for highly available replicated database systems [14]. They find that transactions commutativity eases reconciliation in such a setup. They do not assume that all concurrent operations commute as we do in this work: we are simplifying the reconciliation problem, but also limiting the design space. Similarly, Helland and Campbell suggest to use associative, commutative and idempotent operations in order to tolerate transient faults, and to improve scalability and availability [16].

Weihl designs high-performance concurrency control algorithms for a transactional ADT, using commutativity to identify non-conflicting concurrent operations [42]. Weihl distinguishes between *forward* and *backward commutativity*, which differ by how return values and failures are handled. We believe this distinction is not relevant to our specifications, where downstream operations do not return values and are never allowed to fail.

Klingemann et al. [23] build upon Weihl's theory in a distributed cooperative application framework that minimises reconciliation. Forward commutativity relations identify conflicting operations, and backward commutativity identifies dependent operations.

6.2 Existing CRDTs

Previous work has designed commutative data types, without identifying the concept of a CRDT.

Johnson and Thomas invented what we called LWW-Register [20]. They compose multiple registers into a larger CRDT, 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 when necessary). LWW ensures a total order of operations (without consensus) but this order is arbitrary and some updates are inherently lost.

Wuu and Bernstein [44] describe two CRDTs that they call Dictionary and Log. Their Dictionary is a Map CmRDT, similar to our U-Set. It is built on top of a replicated Log of operations, which acts as a reliable epidemic broadcast channel; this inspired our state-based CmRDT emulation. The main focus of their article is ensuring effective log propagation and pruning (as described in Section 4.1), to alleviate unbounded growth of the log.

Collaborative editing is an area where commutativity has been used (often implicitly) to provide user with high responsiveness even in disconnected operation. Thus, Operational Transformation attempts to achieve commutativity after the fact [26]. WOOT is an early CRDT designed for collaborative editing [30], followed up with Logoot [43] The first two authors of this paper invented the CRDT concept when working on the Treedoc data structure for collaborative editing [25, 32].

This work exposed the issue of garbage collection in CRDTs. In order to cope with the lack of liveness of GC in the presence of faults, Leţia et al. suggest to move it into a subset of stable replicas and reconcile with other replicas asynchronously [25].

Weiss et al. designed the sequential buffer CRDT Logoot, extended with a general-purpose undo mechanism based on a PN-Counter [43]. This approach suffers from anomalies when the counter goes negative. Using our OR-Set can improve tracking causality of visibility-related operations. Martin et al. generalize Logoot to a CRDT maintaining an XML [27]. This is a notable real-world application of CRDT composition.

Dynamo is an example of a production key-value store built for availability [10]. Dynamo uses the CRDT technique that we call MV-Register in this paper. Used for Amazon's shopping cart service, Dynamo exposes the anomalies of MV-Register. We propose herein to use one of our Set types instead in order to ensure clean semantics.

6.3 Commutativity-oriented design

Some previous work already focused on commutativity or semilattices for eventual consistency.

The foundations of CvRDTs were introduced by Baquero and Moura [2, 3]. This paper extends their work with a specification language, by considering CmRDTs, by studying more complex examples, and by considering GC.

Roh et al. [35, 36] independently developed the Replicated Abstract Data Type concept, which is quite similar to CRDT. They generalise LWW to a generic partial order of operations, called precedence transitivity, which they leverage to build several LWW-style classes. They present the RGA replicated sequence for co-operative editing. The current work considers a larger design space, as we allow any merge function that computes a LUB. We formalise Roh's observation that causal delivery is not always strictly necessary with downstream preconditions. Roh addresses the GC issue with Wuu and Bernstein's stability-detection algorithm.

Ellis and Gibbs' [12] Operational Transformation (OT) studies sequences for shared editing designed as op-based objects. Operations are not commutative by design; however, a replica receiving an operation transforms it against previously-received concurrent updates. The concurrent editing community has studied OT intensively, and many OT algorithms have been proposed. However, Oster et al. demonstrate that most OT algorithms for a decentralized OT architecture are incorrect [29]. We believe that designing data types for commutativity is both cleaner and simpler.

Dennis et al. propose to verify commutativity using a declarative modeling language [11]. They were able to detect non-commutativity between operations on a particular ADT. However, lack of such counterexamples found by the tool does not guarantee commutativity. They appear to assume a synchronous system model.

Alvaro et al.'s so-called CALM approach ensures eventual consistency by enforcing a monotonic logic [1]. This is somewhat similar to our rule for CvRDTs, that every update or merge operation move forward in the monotonic semilattice. Their Bloom domain-specific language comes with a static analysis tool that analyses program flow and identifies non-monotonicity points, which require synchronization. This approach encourages programmers to write monotonic programs and makes them aware of synchronization requirements. Monotonic logic is more restrictive than our monotonic semilattice. Thus, Bloom does not support *remove* without synchronisation.

6.4 Exploiting good connectivity for stronger consistency

Although eventual consistency ensures availability when a system partitions, Serafini et al. suggest to leverage periods of good network conditions to achieve the stronger and more desirable linearisability property [39]. They define *weak operations*, ones that need only to be eventually linearized. They show that it is impossible to build such a shared object using the $\Diamond S$ failure detector, if one requires that all operations terminate even in the presence of failures. In future work, we plan to add small doses of synchronous operations, for instance to commit a result; it will be interesting to 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 replica *merge* compute a least upper bound. In the op-based style, concurrent operations should commute.

State-based objects require only eventual communication between pairs of replicas. Op-based replication requires reliable broadcast communication with delivery in a well-defined delivery order. Both styles of CRDTs are guaranteed to converge towards a common, correct state, without requiring any synchronisation.

We specified a number of interesting CRDTs, in a high-level specification language for asynchronous replication based on simple logic. In particular, we focused on container types with clean semantics for *add* and *remove* operations. The Set is the basic container, from which we derive Maps, Graphs, and Sequences. To alleviate unbounded growth and unbalance, garbage collection can be performed using a weak form of synchronisation, off of the critical path of client-level operations.

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 experimentally. Another direction is to study adding small doses of synchronisation to support infrequent, non-critical client 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] Peter Alvaro, Neil Conway, Joe Hellerstein, and William Marczak. Consistency analysis in Bloom: a CALM and collected approach. In *Biennial Conf. on Innovative DataSystems Research (CIDR)*, Asilomar, CA, USA, January 2011.

[2] 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.

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

[4] 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.

[5] Ken Birman, Gregory Chockler, and Robbert van Renesse. Toward a Cloud Computing research agenda. *ACM SIGACT News*, 40(2):68–80, June 2009.

[6] Eric Brewer. On a certain freedom: exploring the CAP space. Invited talk at PODC 2010, Zurich, Switzerland, July 2010.

[7] Nicholas Carriero and David Gelernter. Linda in context. *Communications of the ACM*, 32:444–458, April 1989.

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

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

[10] 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.

[11] Greg Dennis, Robert Seater, Derek Rayside, and Daniel Jackson. Automating commutativity analysis at the design level. In *Int. Symp. on Software Testing and Analysis (ISSTA)*, pages 165–174, Boston, MA, USA, 2004. Assoc. for Comp. Machinery.

[12] 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.

[13] 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.

[14] 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.

[15] Jim Gray and Leslie Lamport. Consensus on transaction commit. *Trans. on Database Systems*, 31(1):133–160, March 2006.

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

[17] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, March 2008.

[18] Maurice Herlihy and Jeannette Wing. Linearizability: a correcteness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[19] 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.

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

[21] Anthony D. Joseph, Alan F. deLespinasse, Joshua A. Tauber, David K. Gifford, and M. Frans Kaashoek. Rover: a toolkit for mobile information access. In *Symp. on Op. Sys. Principles (SOSP)*, pages 156–171, Copper Mountain, CO, USA, December 1995.

[22] James J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. *ACM Trans. on Comp. Sys. (TOCS)*, 10(5):3–25, February 1992.

[23] 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.

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

[25] 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.

[26] Rui Li and Du Li. Commutativity-based concurrency control in groupware. In *Int. Conf. on Collab. Comp.: Networking, Apps. and Worksharing (CollaborateCom)*, page 10, San Jose, CA, USA, December 2005.

[27] Stéphane Martin, Pascal Urso, and Stéphane Weiss. Scalable XML collaborative editing with undo (short paper). In *Int. Conf. on Coop. Info. Sys. (CoopIS)*, Crete, Greece, November 2010.

[28] Patrick E. O'Neil. The escrow transactional method. *Trans. on Database Systems*, 11:405–430, December 1986.

[29] 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.

[30] 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.

[31] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. Flexible update propagation for weakly consistent replication. In *Symp. on Op. Sys. Principles (SOSP)*, pages 288–301, Saint Malo, October 1997. ACM SIGOPS.

[32] 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.

[33] Nuno Preguiça, Marc Shapiro, and Caroline Matheson. Semantics-based reconciliation for collaborative and mobile environments. In *Int. Conf. on Coop. Info. Sys. (CoopIS)*, volume 2888 of *Lecture Notes in Comp. Sc.*, pages 38–55, Catania, Sicily, Italy, November 2003. Springer-Verlag GmbH.

[34] 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.

[35] 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.

[36] 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.

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

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

[39] 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.

[40] 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.

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

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

[43] 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.

[44] 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.

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399