Designing a commutative replicated data type

Marc Shapiro, Nuno Preguiça

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Designing a commutative replicated data type

Marc Shapiro — Nuno Preguiça

N° ????

Octobre 2007

Thème COM

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

Designing a commutative replicated data type*

Marc Shapiro , Nuno Preguiça

Thème COM — Systèmes communicants
Projet Regal

Rapport de recherche n° ???? — Octobre 2007 — 23 pages

Abstract: Commuting operations greatly simplify consistency in distributed systems. This paper focuses on designing for commutativity, a topic neglected previously. We show that the replicas of *any* data type for which concurrent operations commute converges to a correct value, under some simple and standard assumptions. We also show that such a data type supports transactions with very low cost. We identify a number of approaches and techniques to ensure commutativity. We re-use some existing ideas (non-destructive updates coupled with invariant identification), but propose a much more efficient implementation. Furthermore, we propose a new technique, background consensus. We illustrate these ideas with a shared edit buffer data type.

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

* This work is supported in part by the EU FP6 project Grid4All, the French ARA project Respire, the French ARC project Recall, and the Portuguese FCT/MCTES project POSC/59064/2004, with FEDER funding.

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

Conception d'un type de données répliqué commutatif

Résumé : La commutativité des opérations simplifie grandement la cohérence dans les systèmes répartis. Ce papier aborde la conception visant la commutativité, qui est un sujet négligé. Nous démontrons que les réplicats tout *tout* type de données, dont les opérations concurrent commutent, convergent vers une valeur correcte, sous des hypothèses simples et courantes. Nous montrons aussi qu'un tel type de données peut exécuter des transactions à un coût très faible. Nous identifions quelques approches et quelques techniques qui assurent la commutativité. Nous réutilisons quelques idées existantes (les mises à jour non destructives couplées à une identification invariante) mais nous en proposons une réalisation beaucoup plus efficace qu'auparavant. De plus nous proposons une nouvelle technique, celle du consensus en tâche de fond. Nous illustrons ces idées sur un example de tampon d'édition partagé.

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

1 Introduction

To share information, users located at several sites may concurrently update a common object, e.g., a text document. Each user operates on a separate *replica* (i.e., local copy) of the document. A well-studied example is co-operatively editing a shared text.

As users make modifications, replicas diverge from one another. Operations initiated on some site propagate to other sites and are *replayed* there. Eventually every site executes every action. Even so, if sites execute them in different orders, their replicas might still not converge. Various solutions are available in the literature; for instance, serialising the actions [7] or operational transformation [22]. Such designs are usually complex and non-scalable; thus, despite an extensive literature, there is still no satisfactory solution to the shared text editing problem.

We suggest a different approach: design replicated data types such that operations commute with one another. Let us call such a type a *commutative replicated data type* or CRDT. CRDT replicas provably converge. Furthermore, CRDTs support transactions "for free." However, designing a non-trivial CRDT is difficult.

Although the advantages of commutativity are well known, the problem of designing data types for commutativity has been neglected. Recently, Oster et al. proposed a replicated character buffer CRDT called WOOT [14]. WOOT operations commute, because updates are non-destructive, and because the the identity of a character does not change with concurrent edits. However, WOOT has some drawbacks: it wastes a lot of space, and it does not support block operations such as cut-and-paste.

This paper presents the design of a non-trivial CRDT for concurrent editing, called *treedoc*. Since it is a CRDT, convergence is guaranteed. It supports block operations. Space overhead is kept to a minimum: there is no to little internal meta-data; deleted information can be forgotten; and identifiers are kept short. Common edit operations respond locally and suffer no network latency. Treedoc is fault-tolerant and supports disconnected operation.

As in WOOT, ordinary editing operations are non-destructive and identification does not change with concurrent edits, but our implementation is very different from WOOT. The basic treedoc structure is a binary tree of atoms. The path to a node is a bitstring. For efficiency, structural operations switch between a flat buffer and a tree. These operations are potentially non-commutative; to avoid this problem, structure changes rely either on common knowledge or on consensus. To avoid the latency associated with consensus, it occurs in the background (not in the critical path of editing operations) and aborts if it conflicts with an edit.

In summary, the contributions of this paper are the following:

- A design principle: concurrent operations should commute. We prove that *any* Commutative Replicated Data Type (CRDT) converge, under some simple and standard assumptions.

- We identify two alternative approaches to commutativity: operation coalescing vs. precedence. Coalescing is better, but is not always possible.
- The design of treedoc, a non-trivial, space-efficient, responsive, coalescing CRDT for distributed editing.
- We identify some previously-published techniques for coalescing, such as non-destructive update and invariant identity. We propose a novel implementation of these techniques.
- We propose a novel technique for coalescing: where a consensus is necessary, it is restricted to non-essential operations, occurs in the background, and aborts if it conflicts with an essential operation.
- We show that a CRDT readily supports transactions at a very low implementation cost.

The paper proceeds as follows. Section 1 is this introduction. We describe our system model in Section 2. Section 3 describes the shared buffer abstract data type. We suggest a simple implementation of this data type in Section 4. In Section 5, we examine how to convert to a more efficient representation and back. Section 6 explains the full treedoc implementation, combining the advantages of the two preceding sections. We build transactions on top of a CRDT in Section 7. Section 8 compares with previous work. Section 9 concludes. We provide a proof of convergence in Appendix A.

2 System model

2.1 Replicated execution and eventual consistency

We consider an asynchronous distributed system, consisting of N *sites* (computers) connected by a network. Communication between connected sites is reliable. A site may disconnect but eventually reconnects. We assume an epidemic style of communication, i.e., a site connects at arbitrary intervals with arbitrary other sites, sending both local updates and those previously received from other sites. Eventually, every update reaches every site, either directly or indirectly.

With no loss of generality, we consider a single object replicated at any number of sites. A user accesses the object through his local replica, *initiating* operations at the current site.[1] The operations execute locally and are logged. Eventually the log is transmitted and the operations it contains are *replayed* at other sites. Eventually all sites execute the same operations (either by local submission or by remote replay), in some sequential order, but not necessarily in the same order.

[1] We assume that a given operation is initiated at a unique site.

We say operation o happens before o' (noted $o \rightarrow o'$) if some site initiates o' after the same site has executed o.[2] We require that if $o \rightarrow o'$, then all sites execute o before o' (not necessarily immediately before). Common epidemic protocols, such as Bayou's anti-entropy [16], ensure this so-called "causal ordering" property. To implement this property, it suffices to delay the execution of some operation o, until all operations that happen before o have been executed. Well-known techniques such as vector clocks or version vectors [10] can be used to track happens-before dependencies.

Operations are *concurrent* if neither happens before the other: $o \parallel o' \stackrel{\text{def}}{=} \neg(o \rightarrow o') \wedge \neg(o' \rightarrow o)$.

Two operations o and o' *commute* iff, whatever the current state of the object, such that execution of either o or o' succeeds, executing o immediately followed by o' also succeeds, and leads to the same state as executing o' immediately followed by o.

A *Commutative Replicated Data Type* (CRDT) is a data type where all concurrent operations commute with one another. We prove (in Appendix A) that CRDTs guarantee *eventual consistency*: provided that every site executes every operation in an order consistent with happens-before, the final state of replicas is identical at all sites.

Furthermore, CRDTs support serialisable transactions with virtually no overhead. If all operations commute, so do arbitrary sets of operations. If every site executes transactions sequentially, in an order consistent with happens-before, the local orders are all equivalent. This ensures serialisability. Furthermore, transactions never abort. Hence, very little mechanism is needed. We return to transactions in Section 7.

2.2 Ensuring that operations commute

Two operations α and β commute if, for any state T, execution sequences $\langle T \cdot \alpha \cdot \beta \rangle$ and $\langle T \cdot \beta \cdot \alpha \rangle$ are both correct states and are equivalent. There are two basic approaches for ensuring commutativity, which we call coalescing and precedence.

Intuitively, coalescing means that α preserves the effect of β and vice-versa; i.e., the post-condition of both operations is satisfied, whatever their relative execution order.[3] This is the standard meaning in mathematics, for instance when we say that addition and subtraction of integers commute.

The alternative is to define an order of precedence, say β takes precedence over α: when both operations execute in either order, β takes effect, but not necesariy α. A typical implementation is that in the order $\langle \alpha \cdot \beta \rangle$, the latter overwrites the results of the former, and in the order $\langle \beta \cdot \alpha \rangle$, the latter is replaced by a no-op. For instance, most replicated file systems follow the "Last Writer Wins" rule [19]: when two users write to the same file, the

[2] The site executes o either because it was initiated locally, or because it was initiated at another site and delivered here in a message. Thus the $\rightarrow$ relation is identical to Lamport's happens-before [7].

[3] This is sometimes called "intention preservation."

write with the highest timestamp takes precedence. The write with the lowest timestamp may be lost. (In contrast, writes to different files coalesce.)

Clearly, the coalescing approach is preferable to precedence; but precedence is much easier to achieve.

3 Shared buffer replicated data type

We consider a shared, replicated document, consisting of a linear sequence of atoms. An atom may be a character or some other immutable payload, e.g., a graphical illustration inserted inside the document. We designed treedoc it to be as unrestricted and flexible as possible, to enable a variety of applications to use it.

Each user has a copy of the document. Each user can modify his replica independently by executing two types of *edit operations*:

- *insert*(*insertpos*, *newatom*, S) visibly inserts atom *newatom* in the document. (In the underlying data structure, there may be other data before or after *insertpos*, but it is not visible to the user.) All atoms at positions strictly less than *insertpos* lie to the left of *newatom*; all those strictly greater than *insertpos* lie to its right. The S argument is the initiating site, as justified later.
- *delete*(*delpos*, S) visibly removes the atom existing at position *delpos*. (The atom may still be in the data structure but is not visible to the user any more.) The S argument is the initiating site.

We defer to Section 7 the description of a transaction construct, enabling atomic bulk operations such as cutting and pasting a block of text, or searching and replacing all instances of a pattern.

Our treedoc design ensures commutativity by coalescence, i.e., the effect of *insert* and *delete* is the same at all sites.

At a higher level, the application might have stronger requirements. For instance, it might disallow inserting characters inside deleted text; or it might ensure a proper hierarchy of chapters, sections and paragraphs; etc. Enforcing such semantic conflicts requires higher-level conflict detection and resolution mechanisms, which are non-commutative, but are out of our focus.

Similarly, we emphasise that the stringtree structure is completely decoupled from any higher-level document hierarchy (e.g., XML tree structure).

3.1 Unique identifiers for positions

Let us assume the existence of unique position identifiers, with the following properties:

- Each postion in the atom buffer has an identifier that is unique with respect to all other positions, and that remains constant, for the whole lifetime of the document.[4]
- There is a total order of position identifiers, noted $<$.
- Given any two identifiers L and R such that $L < R$, it is possible to generate a fresh unique identifier N such that $L < N < R$.

We will call these identifiers UIDs (unique identifiers). Real numbers have the properties required for UIDs, but the third property requires infinite precision, which is not realistic. In Section 4 we will present a practical alternative, based on trees.

3.2 Abstract atom buffer CRDT

Consider an abstract data type whose state T is a set of $(uid, atom)$ couples, where uids are unique. The content of state T is the sequence of all $atom$s in T ordered by their uid. Operation $insert(u, a, S)$ adds the pair (u, a) to the set. If a pair (u, a) exists in the set, operation $delete(u, S)$ removes the pair, whatever a. We now prove that concurrent operations of this data type commute.

Lemma 1. *Insert operations commute. For any data state T, any fresh unique identifiers u_1 and u_2, any atoms a_1 and a_2, and any originating sites S_1 and S_2:* $\langle T \cdot insert(u_1, a_1, S_1) \cdot insert(u_2, a_2, S_2)\rangle \equiv \langle T \cdot insert(u_2, a_2, S_2) \cdot insert(u_1, a_1, S_1)\rangle$.

Proof. After executing the two insert operations, the resulting state includes the two new atoms. Furthermore, atoms are ordered by unique identifiers. Therefore, the final state is the same. □

Lemma 2. *An insert operation commutes with a delete operation when they refer to different unique identifiers. For any state T, any fresh unique identifier u_1, any unique identifier $u_2 \neq u_1$, any atom a_1, and any originating sites S_1 and S_2:* $\langle T \cdot insert(u_1, a_1, S_1) \cdot delete(u_2, S_2)\rangle \equiv \langle T \cdot delete(u_2, S_2) \cdot insert(u_1, a_1, S_1)\rangle$.

Proof. Two cases must be considered. First, when T includes the atom with identifier u_2. By executing both operations in any order, the final state of T will include an additional atom identified by u_1 and it will not include the atom identified by u_2. As atoms are ordered by their unique identifier, the final state is the same. Second, when T does not include the atom with identifier u_2. By executing both operations in any order, the final state of T will include an additional atom identified by u_1 (the atom identified by u_2 was not present in the original state. As atoms are ordered by their unique identifier, the final state is the same. □

[4] However, an unused identifier can be garbage-collected and re-used. We do not attempt to formalise this property.

Lemma 3. *If an insert operation and a delete operation refer to the same unique identifier, then the insert happens-before the delete.*

Proof. According to the specification of Section 3, a user may initiate operation $delete(u, S)$ at site S only if a pair (u, a) exists (for some a) in the current state at site S. This pair must have been inserted by an *insert* operation executed previously at site S. □

Lemma 4. *Delete operations commute. For any state T, any unique identifiers u_1 and u_2 and any originating sites S_1 and S_2: $\langle T \cdot delete(u_1, S_1) \cdot delete(u_2, S_2)\rangle \equiv \langle T \cdot delete(u_2, S_2) \cdot delete(u_1, S_1)\rangle$.*

Proof. For any original state T, the final state will not include the atoms identified by u_1 and u_2, but it will include all other atoms, as no other atom will ever has the same unique identifier. Thus, the final state will include the same set of atoms and, as atoms are ordered by their unique identifier, the final state is exactly the same. □

Theorem 1. *The data type described in this section is a CRDT.*

Proof. By the above lemmas, all concurrent operation pairs (insert-insert, delete-delete, insert-delete) commute. □

4 Treedoc abstract data type

We start with a very simple design, which satisfies the coalescence requirement, but has some limitations. In later sections, we will improve the design.

4.1 Paths

We manage the document as a binary tree. A tree node contains either a single atom, or nil. The identifier of an atom is its *path* in the tree. The path to the root is the empty bitstring ϵ; the path concatenation operator is noted $\odot$. The left child of a node is 0, its right child is 1. We walk the tree in infix order, skipping nil nodes (but not their descendants).

For example, Figure 1 represents the document state `"abcdef"`, with the following identifiers: $id(\texttt{a}) = [00]$; $id(\texttt{b}) = [0]$; $id(\texttt{c}) = []$; $id(\texttt{d}) = [10]$; $id(\texttt{e}) = [1]$; $id(\texttt{f}) = [11]$.

We define the following partial order over identifiers. Node id_1 is to the left of id_2 (or, equivalently, id_2 is to the right of id_1), noted $id_1 < id_2$, iff:

- $id_1 = [c_1...c_n]$ is a prefix of $id_2 = [c_1...c_n j_1...j_m]$ and $j_1 = 1$, or
- $id_2 = [c_1...c_n]$ is a prefix of $id_1 = [c_1...c_n i_1...i_m]$ and $i_1 = 0$, or
- $id_1 = [c_1...c_n i_1...i_n]$ has a common prefix with $id_2 = [c_1...c_n j_1...j_m]$ and $i_1 = 0$. The prefix may be empty.

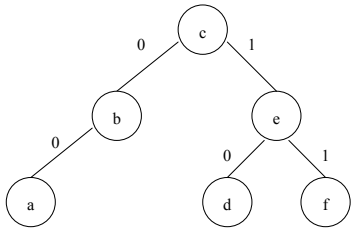

Figure 1: Identifiers in a shared text buffer

We also define the ancestry of a node. Node u is the (direct) parent of node v, noted u/v, iff $id(v) = id(u) \odot 0 \vee id(v) = id(u) \odot 1$; equivalently, v is a (direct) child of u. Node u is an ancestor of v (or, equivalently, v is a descendant of u), noted $u/^{+}v$, if u is a parent, or grand-parent, or great-grand-parent, etc., of v.

4.2 Deleting

We start with the simplest procedure, deleting an atom: simply replace the content of the node with nil. Since the identification of the deleted node is unique, it is clear that the initiator and replay executions will all delete the same node. Sometimes, during replay, the node to be deleted may not exist, but this can only be because it was already deleted previously.

We will say that the delete is *stable* once it has been executed at all nodes. No operation that happens-after the delete is stable will ever refer to the node identifier; therefore, if the node is a leaf, it can be completely forgotten (and so on recursively). Thus a subtree that contains only stably deleted nodes can be completely removed and forgotten.

To this effect we introduce a $gc(N)$ procedure that removes leaf N if it is stably deleted. A node may call $gc(N)$ at any time after N is deleted. Operation gc is local only, it does not have a replay version.

Procedure $stabledel(N)$ tests for stability. Conceptually, $stabledel(N)$ waits for acknowledgments from all sites that have executed *delete*N. We refer to Golding [4] for an efficient implementation of stability that compacts acknowledgments for all past operations into a single vector clock or matrix clock.

4.3 Implementing inserts

To insert *newatom* between $atom_p$ and $atom_f$, we must grow the tree in a way that satisfies the relation $id(atom_p) < id(newatom) < id(atom_f)$. In this section, we present a very simple algorithm that does not attempt to balance the tree; later, we will resolve this issue.

Algorithm 1 New unique identifier for insert

1: **function** *newUID* $((atom_p, uid_p), (atom_f, uid_f))$ // $(atom_p, uid_p)$*: previous atom;* $(atom_f, uid_f)$*: following atom;*
2: **Require:** $uid_p < uid_f$
3: **if** $\exists (atom_m, uid_m) : uid_p < uid_m < uid_f$ **then return** $newUID((atom_p, uid_p), (atom_m, uid_m))$
4: **else if** $(atom_p, uid_p) /^+ (atom_f, uid_f)$ **then return** $uid_f \odot 0$
5: **else if** $(atom_f, uid_f) /^+ (atom_p, uid_p)$ **then return** $uid_p \odot 1$
6: **else return** $uid_p \odot 1$

Algorithm 1 starts by checking whether there is a node between $atom_p$ and $atom_f$. If so, it recursively looks for the leftmost predecessor of uid_f that remains to the right of uid_p. When there is no node between $atom_p$ and $atom_f$, three cases may occur:

- Node uid_p is an ancestor of uid_f, i.e., uid_f is a right descendant of uid_p. In this case, uid_f has no left child, so we create a new left child of node uid_f. The new identifier is $uid_f \odot 0$.
- Or, symmetrically, node uid_f is an ancestor of uid_p. The new identifier us $uid_p \odot 1$.
- Or, neither is an ancestor of the other. In this case, uid_p has no right child, so we create a new right child of node identified $uid_p \odot 1$.

In the example of Figure 1, for inserting atom `X` between `c` and `d`, a left child is created under `d` with identifier [100], as shown in Figure 2.

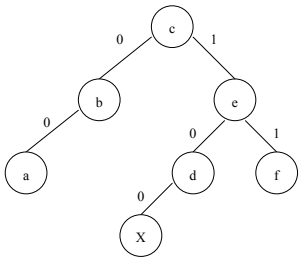

Figure 2: Identifiers after inserting a new character atom

4.4 Concurrent inserts

In case of concurrent updates, a binary tree becomes insufficient, because two users can concurrently insert an atom at the same position. We maintain the basic binary tree structure,

but we extend a node to contain any number of *side nodes* (and their descendence), disambiguated by a $(siteID, counter)$ pair, where *siteID* identifies the initiator site. We assume a total order of site identifiers, hence of disambiguators, hence of side nodes: $(s_1, c_1) < (s_2, c_2)$, iff $c_1 < c_2$ or $c_1 = c_2$ and $s_1 < s_2$.

Algorithm 1 generates new unique identifiers for insertion. Figure 3 shows an example of the situation. Assuming that characters `X` and `Y` were inserted with the associated disambiguator idX and idY respectively, we have $id(\texttt{X}) = [(1)(0)(0, idX)]$ and $id(\texttt{Y}) = [(1)(0)(0, idY)]$.

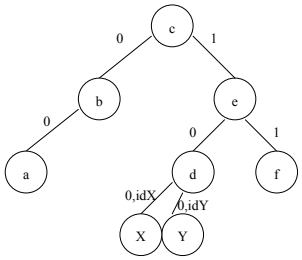

Figure 3: Side nodes and their identifiers, after concurrent inserts at position [100]

Since a concurrent update can occur at every level, conceptually, every node may include a disambiguator. Thus, in the example, we could have, for example, $id(\texttt{d}) = [(--, idC)(1, idE)(0, idD)]$. The full state is as in Figure 4.

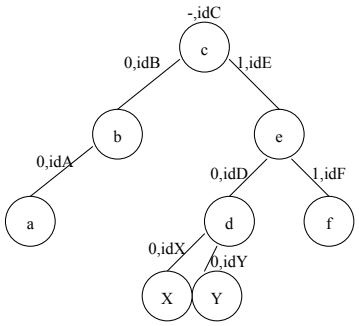

Figure 4: Fully expanded identifiers, after concurrent inserts at position [100]

When disambiguators are used, the total order among identifiers is defined as follows: Node id_1 is left of node id_2 (or, equivalently, id_2 right of id_1), noted $id_1 < id_2$, iff:

- $id_1 = [c_1...c_n]$ is a prefix of $id_2 = [c_1...c_n j_1...j_m]$ and $j_1 = (1, *)$, or
- $id_2 = [c_1...c_n]$ is a prefix of $id_1 = [c_1...c_n i_1...i_m]$ and $i_1 = (0, *)$, or

- $id_1 = [c_1...c_n i_1...i_n]$ has a common prefix with $id_2 = [c_1...c_n j_1...j_m]$ and $i_1 = (0, *) \wedge j_1 = (1, *)$ or $i1 = (k, d_1) \wedge j1 = (k, d_2) \wedge d_1 < d2$. The prefix may be empty.

To generate a new unique identifier, the algorithm 1 is used, with the returned identifier extended with a freshly generated disambiguator.

Once all concurrent inserts at the same location have executed at some site, redundant disambiguators can be removed. We will say that an insert is stable at some site, once that site has received from all other sites some operation that happens-after the insert. At that point, it is guaranteed not to receive another concurrent insert at the same node. (To ensure this happens quickly, sites that are not actively editing should send out occasional no-ops.) Procedure *cleanside* removes redundant disambiguators; it is a local procedure (it has no replay version).

4.5 Treedoc abstract data type

Algorithm 2 contains detailed specification of the simple treedoc data type. In addition to the operations given at the beginning of Section 3, we specify *gc* and *cleanside* as explained above.

The initiator versions of *insert* and *delete* have pre-conditions, to make sure that the user only addresses valid nodes, and to avoid wastage of space. However, there may be no restrictions on the replay version. Therefore, the replay version has no precondition, and simply re-creates any nodes that it may be missing.

This data type satisfies the coalescence requirement. Since every atom has an identifier that does not change with other operations, replaying a delete removes the intended atom. Inserting an operation with a path positions it with respect to its left and right neighbours, replaying an insert preserves the intended location. Since this data type is a CRDT, replicas are guaranteed to converge to the same (correct) value.

In Algorithm 2, the notation $N[siteID]$ stands for the side node of N identified by *siteID*. The notation $N[[siteID]]$ stands for $N[siteID]$, if it exists, and N otherwise. *IamInitiator* is true on the initiator site, and false on all replay sites.

5 Identifying a sequential buffer to a binary tree

The approach so far has a number of limitations. Paths are variable length and can become very inefficient if the tree is unbalanced (e.g., if users always append to the end of the buffer). Concurrent inserts complicate the structure and the paths. The tree metadata consumes memory; for instance, if atoms are one-byte characters the overhead can be several times the payload. Finally, deleted atoms that cannot be garbage-collected waste space.

Rather than attempt to fix each of these issues individually, we propose a more radical solution. In this section, we discuss structural operations that switch between the efficient

Algorithm 2 Simple treedoc

1: **procedure** *contents* (N) *// N: a treedoc node*
2: Walk subtree rooted at N in infix order
3: Return the atoms of the non-empty nodes

4: **procedure** *insert* (A, N, S)
5: *// A: atom to insert*
6: *// N: insertion position, chosen by newUID*
7: **Require:** S = the initiator site
8: **Require:** $A \neq \mathsf{nil}$
9: **Require:** *IamInitiator* $\Rightarrow$ all ancestors of node N exist
10: **Require:** *IamInitiator* $\Rightarrow N[[S]]$ does not exist
11: Create any missing ancestors of N
12: If necessary, create node N
13: Create side node $n = N[siteID]$ with $n.contains = A$

14: **procedure** *delete* (N, S) *// N : node to be deleted*
15: **Require:** *IamInitiator* $\Rightarrow N[[S]]$ exists and $N[[S]].contains \neq \mathsf{nil}$
16: **Require:** S = the initiator site
17: Create any missing ancestors of N
18: If necessary, create node N
19: $N[[S]].contains := \mathsf{nil}$
20: Send acknowledgment of $delete(N, S)$ to all sites

21: **procedure** *stabledel* (N) *// Await stable delete of node N*
22: **if** acknowledgment for $delete(N, *)$ received from all sites **then return** true
23: **else return** false

24: **procedure** *gc* (N) *// N: a treedoc leaf*
25: **Require:** $N.contents(=)\mathsf{nil}$
26: **Require:** *stabledel* (N)
27: **Require:** *IamInitiator*
28: Remove N

29: **procedure** *cleanside* (N) *// N: a treedoc node*
30: **Require:** *stableinsert* (N)
31: **Require:** *IamInitiator*
32: **if** $|\{N[S]|\forall S\}| = 1$ **then** *// There is a single side node*
33: $N := N[S]$ *// Remove redundant disambiguator*

34: **procedure** *stableinsert* (N)
35: **if** current site has received some operation that happens-after $insert(*, N, *)$ from every site **then**
36: **return** true
37: **else return** false

Algorithm 3 *explode* and *flatten*

```
1: procedure explode (atomstring)
2:     depth = ⌈log2(length(atomstring) + 1)⌉
3:     T = Allocate a complete binary tree of depth depth
4:     Populate T in infix order with the atoms of atomstring
5:     Remove any remaining nodes
6:     Return T

7: procedure flatten (N) // N: root of a subtree to be flattened
8:     Walk subtree in infix order
9:     Return a linear buffer containing the atoms of the non-empty nodes
```

flat buffer representation, and the edit-oriented tree representation. The specification of these operations is as follows.

- *explode*(*atomstring*). Returns a treedoc whose contents is identical to *atomstring*.
- *flatten*(*path*) Returns an atom string whose contents is identical to the sub-treedoc rooted at *path*.

The initiator and replay versions of these operations must have identical effect. In particular, *explode* must return exactly the same structure at all sites.

Different implementations of *explode* are possible, as long as it has the same effect at every site. Observing that the capacity of a complete binary tree with *depth* levels is $2^{depth} - 1$, we suggest the simple implementation in Algorithm 3.

With these two operations, we can choose the string representation or the treedoc representation. The former is compact and efficient, but it does not readily support concurrent edits. When a treedoc becomes unbalanced or contains many nil nodes, it suffices to *flatten* then *explode* it to fix the problem.

However, these structural operations do not commute with edit operations. We study the solution of this problem next.

6 Mixed tree

In this section, we study how to combine edit operations and structural operations, while still retaining the advantages of a CRDT.

A first observation is that the *explode* operation is not really necessary. Algorithm 3 can be interpreted as a mapping from a string to a canonical treedoc representation. Applying a path to a string implicitly converts the string to the canonical treedoc. Eliminating the explicit *explode* operation removes the need to make it commute with edits.

A second observation is that *flatten* is not an essential operation. Aborting a *flatten* (leaving no side-effects) causes no harm. Therefore, if *flatten* is concurrent with an edit operation in the same subtree, we abort it. More precisely, for *flatten* to take effect, it executes a distributed commitment procedure. When executing *flatten* at some site, if this site observes the execution of a concurrent *insert* or *delete*, that site votes "No" to commitment, otherwise it votes "Yes." The operation succeeds only if all sites vote "Yes," otherwise it has no effect.

Any distributed commitment protocol from the literature will do, for instance two-phase commit or Gray and Lamport's fault-tolerant protocol [5].

We may now envisage a mixed tree, where parts that are currently being edited are in treedoc representation, and parts that are currently quiescent are represented as strings.

6.1 Fault tolerance and disconnected operation

Ensuring fault tolerance and disconnected operation for disconnected edits is straightforward. Every site logs all its operations (whether locally initiated or remote) on persistent storage. When a site that was disconnected for some time reconnects with the rest of the system, it simply exchanges with other sites the missing information. If a site fails and recovers the situation is the same. If a site crashes, losing its memory, then when it restarts it behaves like a new site, and copies over the state of some other site; operations that it initiated before the crash and never sent to another site are lost.

The situation is more complex for *flatten*s, since they require a consensus. To ensure that consensus is solvable in the presence of crashes, we assume the existence of fault detectors [2].

To allow disconnected operation, fault detectors must be capable of distinguishing disconnection from a crash. During the commit phase of *flatten*, a disconnected site is assumed to be voting No, and *flatten* aborts. This is distinct from a crashed site, which does not participate in the commitment.

Similarly, *stabledel* should be modified to return true if all non-crashed sites have acknowledged, and *stableinsert* should return true if all non-crashed sites have sent an operation that happens-after the insert.

Note that if a disconnected site is falsely diagnosed as crashed, any operations that it initiated within a sub-tree that was *flatten*ed cannot be replayed, because they use now-forgotten node identities. Such operations are lost. Similarly, if this site initiated operations that depend on a node that was deleted and garbage-collected, then these operations are lost.

7 Block edits and transactions

In practice, single-character edit operations are insufficient for concurrent editing. Users working on the same portion of text may find it unpleasant to see their edits mixed together. Furthermore, common operations such as as cutting and pasting, or global replacement, are block operations. We need transactions, to allow a user to insert, delete, replace or move a block of text, without concurrent operations destroying the integrity or location of the block.

Fortunately, a CRDT is ideal for building complex transactions out of simple operations. Since individual operations commute when concurrent, concurrent groups of operations commute as well. To ensure serialisability, it is sufficient to ensure that transactions are executed sequentially (in any order compatible with happens-before), whether at the initiator site or during replay.[5]

A transaction executes atomically (all-or-nothing), indivisibly (its intermediate results cannot be observed) and durably (its results are observable by all later operations). We do not see any need for nested transactions.

Since individual operations commute, a transaction never aborts, therefore transaction support can be very cheap. All that is needed is some book-keeping of the beginning and end of transactions, and buffering received operations to ensure sequential execution. While a site is executing a locally-initiated transaction, it buffers remote operations, delaying their replay until the end of the transaction. When a site receives a remote transaction, it buffers the operations it contains until the end of the transaction is received, and replays them all at once. Thus, *begin_transaction* and *end_transaction* are basically no-ops used only as place-holders in the log.

- *begin_transaction* opens a transaction. At the initiator site, the transaction will include all operations initiated at the same site, until the next *end_transaction*. It is illegal to initiate two successive *begin_transaction* operations without an intervening *end_transaction*.
- *end_transaction* closes the current transaction by the same initiator. It is illegal to initiate a *end_transaction* unless a transaction is open.

Considering any two transactions (or isolated operations), either one happens-before the other, or they are concurrent. In particular, if a transaction contains an operation that edits node N, and any operation of the same transaction is concurrent with $\mathit{flatten}(N')$, and $N'/^{+}N \vee N = N'$, then the *flatten* operation aborts.

Note that we could now define block operations such as a block move or a global search-and-replace. From the commutativity perspective, such new operation types are considered equivalent to a transaction of *insert* and *delete* operations, but they can be implemented much more efficiently.

[5] For the purpose of sequential execution, an operation that is not part of any transaction is considered as a separate transaction of itself.

8 Related work

A comparison of several approaches to the problem of collaboratively editing a shared text was written by Ignat et al. [6].

Operational transformation (OT) [22] considers collaborative editing based on non-commutative single-character operations. To this end, OT transforms the arguments of remote operations to take into account the effects of concurrent executions. OT requires two correctness conditions [22]: the transformation should enable concurrent operations to execute in either order, and furthermore, transformation functions themselves must commute. The former is relatively easy. The latter is more complex, and Oster et al. [13] prove that all existing transformations violate it.

OT attempts to make non-commuting operations commute after the fact. We believe that a better approach is to design operations to commute in the first place. This is more elegant, and avoids the complexities of OT.

A number of papers study the advantages of commutativity for concurrency and consistency control [1, 23, for instance]. Systems such as Psync [11], Generalized Paxos [9], Generic Broadcast [15] and IceCube [17] make use of commutativity information to relax consistency or scheduling requirements. However, these works do not address the issue of achieving commutativity.

Weihl [23] distinguishes between forward and backward commutativity. They differ only when operations fail their pre-condition. In this work, we consider only operations that succeed at the submission site, and ensure by design that they won't fail at replay sites.

Roh et al. [18] were the first to suggest the CRDT approach. They give the example of an array with a slot assignment operation. To make concurrent assignments commute, they propose a deterministic procedure (based on vector clocks) whereby one takes precedence over the other.

This is similar to the well-known Last-Writer Wins algorithm, used in shared file systems. Each file replica is timestamped with the time it was last written. Timestamps are consistent with happens-before [7]. When comparing two versions of the file, the one with the highest timestamp takes precedence. This is correct with respect to successive writes related by happens-before, and constitutes a simple precedence rule for concurrent writes.

In the precedence design of Roh et al., concurrent writes to the same location are lost. This is inherent to the destructive assignment operation that they consider. In ours, concurrent inserts are always coalesced, which is important in order to support co-operative work.

In Lamport's replicated state machine approach [7], every replica executes the same operations in the same order. This total order is computed either by a consensus algorithm such as Paxos [8] or, equivalently, by using an atomic broadcast mechanism [3]. Such algorithms can tolerate faults [**?**]. However they are complex and scale poorly; consensus occurs within the critical execution path, adding latency to every operation.

The precedence approach can be viewed as a poor-man's total order. It does not require an online consensus algorithm, but it loses work.

In the treedoc design, common edit operations execute optimistically, with no latency; it uses consensus in the background only. Previously, Golding relied on background consensus for garbage collection [4]. We are not aware of previous instances of background consensus for structural operations, nor of aborting consensus when it conflicts with essential operations.

9 Conclusion

It was known previously that commutativity simplifies consistency maintenance, but the issue of designing systems for commutativity was neglected. This paper suggested a new paradigm for replication: the Commutative Replicated Data Type or CRDT, designed such that concurrent operations commute. We prove that, under some simple and standard execution conditions, replicas of any CRDT eventually converge. This makes the implementation of replicated systems much simpler than before. Furthermore, CRDTs support transactions at very low cost.

However, designing a CRDT with the desirable property that no work is lost (coalescence) is not easy. We give a coalescing CRDT solution to the problem of a shared edit buffer, by implementing some known techniques in a novel way (using paths in a binary tree as invariant identifiers) and by some new techniques (abortable consensus in the background). This is possible only because updates are not destructive.

Our techniques are not limited to this particular problem, and are generaliseable non-destructive updates in other data structures, such as directories.

We purposely designed treedoc to support arbitrary mixtures of edit operations. We separate out the issue of semantic constraints and conflict detection, which we study elsewhere [12, 17, 20, 21]. We interpret conflicts as cases of irreducible non-commutativity. Any real system must support a mix of data types, some coalescing, some using precedence, and some not commutative.

Our next step in this research will be to enable peer-to-peer co-operative editing at a large scale, by implementing treedoc within an existing text editor or wiki system. This will enable a deeper investigation of pragmatic issues and performance studies.

A Proof of eventual consistency

We prove the following property. Assuming:

- That every operation, initiated at any site, eventually executes at all sites,
- That if $o \rightarrow o'$, then o executes before o' at every site,

- That all concurrent operations commute,

then the state of the object eventually converges at all sites.

Our proof is by recurrence over all legal execution schedules.

A.1 Notation

A multilog $M = (V, H)$ is a directed graph, consisting of a set of operations $V = \{\alpha, \beta, \dots\}$ connected by the happens-before relations $H = \{(x, y) \in V \times V : x \rightarrow y\}$.

A schedule $T = \langle \alpha \cdot \beta \cdot \dots \rangle$ of a multilog is a sequential enumeration of operations in some order consistent with happens-before. Formally, $T = (L, <_T) \in \mathit{sched}((V, H)) \Leftrightarrow \forall\, x, y \in L,\ (x, y \in V) \wedge (x <_T y \Rightarrow x \neq y) \wedge (x \rightarrow y \Rightarrow x <_T y)$. The sequence operator is noted $\cdot$.

A schedule $T = (L, <_T) \in \mathit{sched}((V, H))$ is said complete with respect to its multilog, iff it contains all operations, i.e., iff $L = V$.

Operation v extends schedule $T \in \mathit{sched}(M)$ if $\langle T \cdot v \rangle$ is in $\mathit{sched}(M)$.

A state *quasi-state* is a schedule $T \in \mathit{sched}(M)$ whose initial element is distinguished operation INIT (denoting the common initial state). We assume that we can further distinguish (in some application-specific manner) between illegal and legal quasi-states. Any extension of an illegal quasi-state is itself illegal. A legal quasi-state will be called a *state* henceforth: $T \in \mathit{state}(M)$. We assume the existence of an equivalence relation between states $\equiv$. Like legality, equivalence is application dependent.

A.2 Recurrence proof

We require that all concurrent operations commute, i.e., given any state T and concurrent operations x and y that extend T, the sequences $\langle T \cdot x \cdot y \rangle$ and $\langle T \cdot y \cdot x \rangle$ are states and are equivalent. Formally: $\forall\, T \in \mathit{state}((V, H)),\ \forall\, x, y \in V : x \parallel y \wedge \langle T \cdot x \rangle \in \mathit{state}((V, H)) \wedge \langle T \cdot y \rangle \in \mathit{state}((V, H)),\ \langle T \cdot y \cdot x \rangle \in \mathit{state}((V, H)) \wedge \langle T \cdot x \cdot y \rangle \equiv \langle T \cdot y \cdot x \rangle$

Given a set of natural numbers $N = \{1, 2, \dots, n-1\}$, we note ρ some permutation of N, with elements $\rho(1), \rho(2), \dots, \rho(n-1)$.

Theorem 2 (All complete states of M are equivalent)**.** *Let $M = (V, H)$ be a multilog of size $|V| = n$. Let $T = \langle \text{INIT} \cdot \alpha_1 \cdot \dots \cdot \alpha_{n-1} \rangle$ be a complete state of M, i.e., $T \in \mathit{state}(M) \wedge \{\text{INIT}, \alpha_1, \dots, \alpha_{n-1}\} = V$. Let ρ be some arbitrary permutation of $\{1, 2, \dots, n-1\}$, and let T_ρ denote the sequence $\langle \text{INIT} \cdot \alpha_{\rho(1)} \cdot \dots \alpha_{\rho(n-1)} \rangle$. Then, if T_ρ is a state, it is equivalent to T: $T_\rho \in \mathit{state}(M) \Rightarrow T \equiv T_\rho$.*

The proof is by recurrence. The theorem is obviously true for $n = 1$ and $n = 2$. Assume it is true for abitrary n; we shall prove that it remains true for $n+1$. Let $T = \langle \text{INIT} \cdot \alpha_1 \cdot \ldots \cdot \alpha_{n-1} \rangle$ be a complete state of $M = (V, H)$ where $n = |V|$.

Consider $M' = (V', H')$ such that $M \subseteq M' \wedge |V'| = n + 1$. V and V' differ by a single element, β. With no loss of generality, we assume that β does not happen before any element of V, i.e., $\nexists\, x \in V,\ (\beta, x) \in H'$.[6] Note $T' = \langle T \cdot \beta \rangle$.

If $T' \notin \mathit{state}(M')$ the theorem is trivially true, because T' is not a state. Therefore, assume $T' \in \mathit{state}(M')$. It follows that T' is a complete state of M'.

For some permutation ρ, let $T'_\rho = \langle \text{INIT} \cdot \alpha_{\rho(1)} \cdot \ldots \cdot \alpha_{\rho(n-1)} \cdot \beta \rangle$. Consider now the set of sequences derived from T'_ρ by permuting the position of β:

$$
\begin{array}{l}
\text{INIT} \,;\, \alpha_{\rho(1)} \,;\, \alpha_{\rho(2)} \,;\, \ldots \,;\, \alpha_{\rho(n-2)} \,;\, \alpha_{\rho(n-1)} \,;\quad \beta \\
\text{INIT} \,;\, \alpha_{\rho(1)} \,;\, \alpha_{\rho(2)} \,;\, \ldots \,;\, \alpha_{\rho(n-2)} \,;\quad \beta \quad ;\, \alpha_{\rho(n-1)} \\
\text{INIT} \,;\, \alpha_{\rho(1)} \,;\, \alpha_{\rho(2)} \,;\, \ldots \,;\quad \beta \quad ;\, \alpha_{\rho(n-2)} \,;\, \alpha_{\rho(n-1)} \\
\qquad\qquad \ldots \\
\text{INIT} \,;\, \alpha_{\rho(1)} \,;\quad \beta \quad ;\, \ldots \,;\, \alpha_{\rho(n-3)} \,;\, \alpha_{\rho(n-2)} \,;\, \alpha_{\rho(n-1)} \\
\text{INIT} \,;\quad \beta \quad ;\, \alpha_{\rho(1)} \,;\, \ldots \,;\, \alpha_{\rho(n-3)} \,;\, \alpha_{\rho(n-2)} \,;\, \alpha_{\rho(n-1)}
\end{array}
$$

If the sequence on any line is not a state, then none of the following lines is a state either; for these, the theorem is trivially true.

The first line is precisely T'_ρ. If $T'_\rho \in \mathit{state}(M')$, then, by assumption, $T'_{rho} = \langle T_\rho \cdot \beta \rangle \equiv \langle T \cdot \beta \rangle = T'$.

Now examine the second line. Either $\alpha_{\rho(n-1)} \parallel \beta$, and they commute, and therefore it is a state equivalent to the first line, and hence to T'; or $\alpha_{\rho(n-1)} \rightarrow \beta$, and the second line is not a state. Similarly for all the following lines: each sequence is either not a state, or is a state equivalent to T'.

Thus we have proven the recurrence clause for all permutations of $\{1, 2, \ldots, n\}$ that are in the same order as ρ. Furthermore, since the recurrence clause is true for all permutations ρ of $\{1, 2, \ldots, n-1\}$, it is true for all permuations of $\{1, 2, \ldots, n\}$. QED.

A.3 Eventual consistency

The above proves that, if different sites execute schedules consisting of the same set of operations, the order of every schedule is consistent with happens-before, and concurrent operations commute, then their final states are equivalent.

If all clients stop initiating operations, and assuming that the system transmits and executes every operation at all sites, then all replicas converge to the same state. This is

[6] In other words, the operations are sorted in $\rightarrow$ order before constructing the successive multilogs.

the traditional definition of Eventual Consistency. Our result is actually stronger, since the final state is a correct state, and it includes all the submitted operations.

However, in a practical system, clients don't stop initiating operations. We can prove nonetheless that, for any time t, the state at every site eventually includes a an equivalent prefix containing all operations up to t. Assume that initiating an operation is atomic. Consider the set O of operations initiated at all sites up to time t, and the set O' of operations initiated after t. (As sites do not have access to a common clock, these sets cannot be computed, but they exist nonetheless.)

Any operation $o \in O$ is either concurrent or happens-before any operation $o' \in O'$. If $o \parallel o'$, then $\langle o \cdot \ldots \cdot o' \rangle \equiv \langle o' \cdot \ldots \cdot o \rangle$; we need consider only the former. If $o \rightarrow o'$, then the only legal order is $\langle o \cdot \ldots \cdot o' \rangle$. The operations in O are eventually executed at all sites, and the operations in O' execute after. Thus the state at all sites has as common prefix the operations in O.

References

[1] B. R. Badrinath and Krithi Ramamritham. Semantics-based concurrency control: beyond commutativity. *ACM Transactions on Database Systems*, 17(1):163–199, March 1992. `http://doi.acm.org/10.1145/128765.128771`.

[2] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722, 1996. `http://doi.acm.org/10.1145/234533.234549`.

[3] Xavier Défago, André Schiper, and Péter Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys*, 36(4):372–421, December 2004. `http://doi.acm.org/10.1145/1041680.1041682`.

[4] Richard A. Golding. *Weak-consistency group communication and membership*. PhD thesis, University of California Santa Cruz, Santa Cruz, CA, USA, December 1992. Tech. Report no. UCSC-CRL-92-52, `ftp://ftp.cse.ucsc.edu/pub/tr/ucsc-crl-92-52.ps.Z`.

[5] Jim Gray and Leslie Lamport. Consensus on transaction commit. *ACM Transactions on Database Systems*, 31(1):133–160, March 2006. `http://doi.acm.org/10.1145/1132863.1132867`.

[6] Claudia Ignat, Gérald Oster, Pascal Molli, Michèle Cart, Jean Ferrié, Anne-Marie Kermarrec, Pierre Sutra, Marc Shapiro, Lamia Benmouffok, Jean-Michel Busca, and Rachid Guerraoui. A comparison of optimistic approaches to collaborative editing of Wiki pages. Research Report RR-6278, Institut National de la Recherche en Informatique et Automatique (INRIA), September 2007. `https://hal.inria.fr/inria-00169395`.

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

[8] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998. http://doi.acm.org/10.1145/279227.279229.

[9] Leslie Lamport. Generalized consensus and Paxos. Technical Report MSR-TR-2005-33, Microsoft Research, March 2005. ftp://ftp.research.microsoft.com/pub/tr/TR-2005-33.pdf.

[10] Friedmann Mattern. Virtual time and global states of distributed systems. In *Int. W. on Parallel and Distributed Algorithms*, pages 215–226. Elsevier Science Publishers B.V. (North-Holland), 1989. http://www.informatik.tu-darmstadt.de/VS/Publikationen/.

[11] S. Mishra, L.L. Peterson, and R.D. Schlichting. Implementing fault-tolerant replicated objects using Psync. In *Symp. on Reliable Dist. Sys.*, pages 42–52, Seattle, WA, USA, October 1989. IEEE. http://ieeexplore.ieee.org/iel2/259/2469/00072747.pdf?tp=&isnumber=2469 &arnumber=72747.

[12] James O'Brien and Marc Shapiro. An application framework for nomadic, collaborative applications. In *Int. Conf. on Dist. App. and Interop. Sys. (DAIS)*, pages 48–63, Bologna, Italy, June 2006. IFIP WG 6.1. http://www-sor.inria.fr/~shapiro/papers/Joyce_DAIS-2006.pdf.

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

[14] Gérald Oster, Pascal Urso, Pascal Molli, and Abdessamad Imine. Data consistency for P2P collaborative editing. In *Int. Conf. on Computer-Supported Cooperative Work (CSCW)*, pages 259–268, Banff, Alberta, Canada, November 2006. ACM Press. http://doi.acm.org/10.1145/1180875.1180916.

[15] Fernando Pedone and André Schiper. Handling message semantics with generic broadcast protocols. *Distributed Computing Journal*, 15(2):97–107, 2002. http://www.inf.unisi.ch/faculty/pedone/papers/2002DC.pdf.

[16] 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. http://www.parc.xerox.com/csl/projects/bayou/.

[17] 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. http://www-sor.inria.fr/~shapiro/papers/coopis-2003.pdf.

[18] 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. http://kernel.kaist.ac.kr/~jinsoo/publication/csi06.pdf.

[19] Yasushi Saito and Marc Shapiro. Optimistic replication. *Computing Surveys*, 37(1):42–81, March 2005. http://doi.acm.org/10.1145/1057977.1057980.

[20] Marc Shapiro, Karthikeyan Bhargavan, and Nishith Krishna. A constraint-based formalism for consistency in replicated systems. In *Proc. 8th Int. Conf. on Principles of Dist. Sys. (OPODIS)*, number 3544 in Lecture Notes in Comp. Sc., pages 331–345, Grenoble, France, December 2004. http://www-sor.inria.fr/~shapiro/papers/opodis2004-final-2004-10-30.pdf .

[21] Marc Shapiro, Nuno Preguiça, and James O'Brien. Rufis: mobile data sharing using a generic constraint-oriented reconciler. In *Conf. on Mobile Data Management*, pages 146–151, Berkeley, CA, USA, January 2004. http://www-sor.inria.fr/~shapiro/papers/mdm-2004-final.ps.gz.

[22] Chengzheng Sun and Clarence Ellis. Operational transformation in real-time group editors: issues, algorithms, and achievements. In *Int. Conf. on Computer-Supported Cooperative Work (CSCW)*, page 59, Seattle WA, USA, November 1998. http://doi.acm.org/10.1145/289444.289469.

[23] W. E. Weihl. Commutativity-based concurrency control for abstract data types. *IEEE Transactions on Computers*, 37(12):1488–1505, December 1988. http://www.computer.org/tc/tc1988/t1488abs.htm.

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