Consistency without concurrency control in large, dynamic systems

Mihai Leția
École Normale Supérieure de Lyon and LIP6

Nuno Preguiça
CITI / DI-FCT, Universidade Nova de Lisboa

Marc Shapiro
INRIA Paris-Rocquencourt and LIP6

ABSTRACT

Replicas of a commutative replicated data type (CRDT) eventually converge without any complex concurrency control. We validate the design of a non-trivial CRDT, a replicated sequence, with performance measurements in the context of Wikipedia. Furthermore, we discuss how to eliminate a remaining scalability bottleneck: Whereas garbage collection previously required a system-wide consensus, here we propose a flexible two-tier architecture and a protocol for migrating between tiers. We also discuss how the CRDT concept can be generalised, and its limitations.

1. INTRODUCTION

Shared read-only data is easy to scale by using well-understood replication techniques. However, sharing *mutable* data at a large scale is a difficult problem, because of the CAP impossibility result [4]. Two approaches dominate in practice. One ensures scalability by giving up consistency guarantees, for instance using the Last-Writer-Wins (LWW) approach [6]. The other guarantees consistency by serialising all updates, centralising at a single database, or using state machine replication), which does not scale beyond a small cluster [11]. Another approach is optimistic replication, allowing replicas to diverge, eventually resolving conflicts either by LWW-like methods or by serialisation [10].

In some (limited) cases, a radical simplification is possible. If concurrent updates to some datum commute, and all of its replicas execute all updates in causal order, then the replicas converge.[1] We call this a Commutative Replicated Data Type (CRDT). The CRDT approach ensures that there are no conflicts, hence, no need for costly concurrency control. CRDTs are not a universal solution, but, perhaps surprisingly, we were able to design highly useful CRDTs. This new research direction is promising as it ensures consistency in the large scale at a low cost, at least for some applications. While the advantages of commutativity are well documented, we are the first (to our knowledge) to address the design of CRDTs.

A trivial example of a CRDT is a set with a single *add-element* operation. A *delete-element* operation can be emulated by adding "deleted" elements to a second set. This suffices, for instance, to implement a mailbox [1]. However, it is not practical, as the data structures grow without bound. Another example is WOOT, a CRDT for concurrent editing [8], but its metadata overhead is large, and it too grows without bound. Logoot [14] allows deleted items to be removed, at the cost of very large item identifiers and metadata. We conclude that efficiency and garbage collection are difficult issues for CRDTs.

As an existence proof of non-trivial, useful, practical and efficient CRDT, we previously published the design of Treedoc, which implements an ordered set with insert-at-position and delete operations [9]. Sequence elements are identified compactly using a naming tree. Metadata overhead is much better than in WOOT or Logoot. Concurrent user-level updates (i.e., edits) commute genuinely. To garbage-collect metadata requires (internal) rebalancing operations, which must be scheduled carefully, to avoid violating commutativity. In this paper, we address how to reconcile the system-wide consensus required for rebalancing with the requirements of a large and dynamic system.

After a brief summary of the CRDT concept and of the Treedoc design (Section 2), the contributions of this paper are as follows:

- We validate the design with performance measurements of a demanding Wikipedia benchmark (Section 3).
- For scalability, we propose a flexible two-tier architecture: A small, stable *core* supports both updates and consensus. It coexists with a unlimited, uncontrolled, dynamic *nebula* supporting only updates (Section 4).
- We present a novel protocol that allows a nebula site to catch up with the with the core's past consensuses, in order to send its updates to the core, and possibly to migrate into the core (Section 5).
- Section 6 discusses lessons learned and possible generalisations.

Finally, Section 7 concludes and outlines future work.

2. AN ORDERED-SET CRDT

We begin by a brief summary of a CRDT providing the abstraction of an ordered sequence of (opaque) *atoms*. Readers interested in more detail are referred to our previous publications [9, 12].

[1] Technically, LWW operations commute; however they achieve this by throwing away non-winning operations. We aim instead for *genuine* commutativity that does not lose work, i.e., the output should reflect the cumulative effect of the operations.

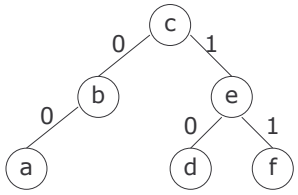

Figure 1: Example Treedoc. The TID for *"b"* is 0; the TID of *"c"* is the empty string; the TID of *"d"* is 10.

2.1 Model

We consider a collection of *sites* (i.e., networked computers), each carrying a *replica* of a shared ordered-set object, and connected by a reliable broadcast protocol (e.g., epidemic communication). We support a peer-to-peer, multi-master execution model: some arbitrary site *initiates* an operation and executes it against its local replica; each other site eventually receives the operation and *replays* it against its own replica. All sites eventually receive and execute all operations; causally-related operations execute in order, but concurrent operations may execute in different orders at different sites.

The update operations of the ordered-set abstraction are the following:

- *insert(ID,newatom)*, where *ID* is a fresh identifier. This operation adds atom *newatom* to the ordered-set.
- *delete(ID)*, deletes the atom identified *ID* from the ordered-set.

Two inserts or deletes that refer to different IDs commute. Furthermore, updates are idempotent, i.e., inserting or deleting the same ID any number of times has the same effect as once. To ensure commutativity of concurrent inserts, we only need to ensure that no two IDs are equal across sites. Our ID allocation mechanism will be described next.

2.2 Identifiers

Atom identifiers should have the following properties: *(i)* Two replicas of the same atom (in different replicas of the ordered-set) have the same identifier. *(ii)* No two atoms have the same identifier. *(iii)* An atom's identifier remains constant for the entire lifetime of the ordered-set.[2] *(iv)* There is a total order "<" over identifiers, which defines the ordering of the atoms in the ordered-set. *(v)* The identifier space is *dense*.

Property *(v)* means that between any two identifiers P and F, $P < F$, we can allocate a fresh identifier N, such that $P < N < F$. Thus, we are able to insert a new atom between any two existing ones.

The set of real numbers $\mathbb{R}$ is dense, but cannot be used for our purpose, because, as atoms are inserted, the precision required would grow without bound. We outline our alternative solution next.

[2]Later in this paper we will weaken this property.

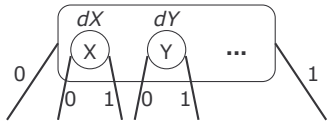

Figure 2: A treedoc major node

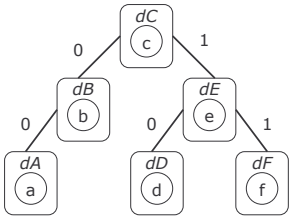

Figure 3: A treedoc node with disambiguators

2.3 The Treedoc CRDT

In Treedoc, an atom identifier, henceforth called a TID, represents a path in a tree. If the tree is balanced, the average TID size is logarithmic in the number of atoms. We experimented with both binary and 256-ary trees; for simplicity, we present only the binary version. The order "<" is infix traversal order (i.e., left to right). Figure 1 shows a binary Treedoc that contains the text *"abcdef"*. (The concrete implementation might store the ordered-set directly as a tree, or alternatively as a set of $(TID, atom)$ pairs.)

In a distributed environment, different sites might concurrently allocate the same TID. To avoid this, we extend the basic tree structure, allowing a node to contain a number of internal nodes, called *mini-nodes*. A node containing mini-nodes will be called a *major node*. Figure 2 shows an example major node. Inside a major node, mini-nodes are distinguished by a *disambiguator*, which is the identifier of the site that inserted the node. Disambiguators are unique and ordered, giving a total order between entries in the ordered-set.

Figure 3 shows a Treedoc structure with disambiguators represented at every node. Site A with disambiguator dA inserted atom a, site B inserted atom b, and so on. Mini-nodes are traversed in disambiguator order.

2.4 Treedoc insert and delete

We now describe the ordered-set update operations, insert and delete.

To insert an atom, the initiator site chooses a fresh TID that positions it as desired relative to the other atoms. For instance, to insert an atom R to the right of atom L: If L does not have a right child, the TID of R is the TID of L concatenated with 1 (R becomes the right child of L). Otherwise, if L has a right child Q, then allocate the TID of the leftmost position of the subtree rooted at Q.

A *delete*(TID) simply discards the atom associated with TID. We retain the corresponding tree node and mark it as a *tombstone*, to avoid inserting a new node with the same identifier.

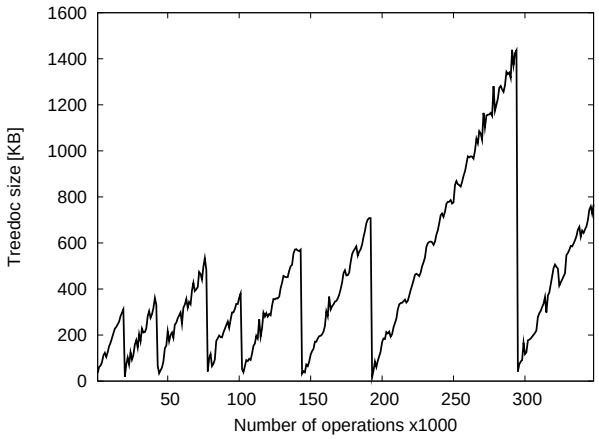

Figure 4: Treedoc size over time (GWB page)

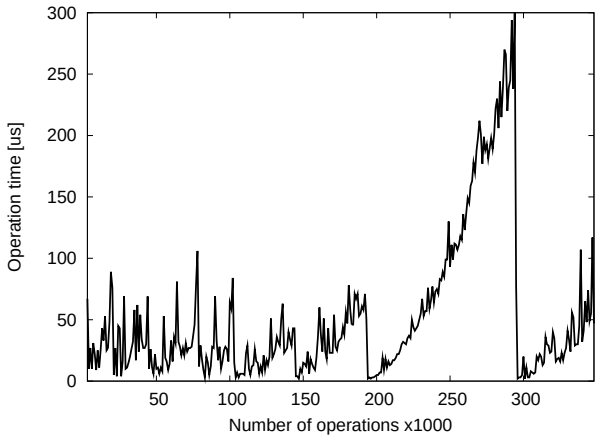

Figure 5: Execution time per update (GWB page)

Since a TID includes a disambiguator that identifies the site that initiated its insert, it follows that a tombstone is necessary on that site only. At the sites that replayed the original insert, a later delete may discard the node immediately, if, for instance, it is a leaf. (The measurements later in this paper do not include this optimisation, and assume that every site retains tombstones.)

2.5 Rebalancing the tree

In the approach described so far, depending on the pattern of inserts and deletes, the tree may become badly unbalanced or riddled with tombstones. To alleviate this problem, internal operation *rebalance* balances the tree. Since a balanced tree is equivalent to an array, this eliminates all memory overhead. Rebalancing is a radical form of garbage collection.

As rebalancing changes the TIDs, we modify Property *(iii)* of Section 2.2 to allow non-ambiguous renaming.

However, rebalancing does not genuinely commute with updates. We solve this using an update-wins approach: if a rebalance occurs concurrently with an update, the update wins, and the rebalance aborts with no effect. We use a two-phase commit protocol for this purpose (or, better, a fault-tolerant variant [5, 13]). The site that initiates the rebalance acts as the coordinator and collects the votes of all other sites. Any site that detects an update concurrent to the rebalance votes "no", otherwise it votes "yes." The coordinator aborts the rebalance if any site voted "no" or if some site is crashed. Commitment protocols are problematic in large-scale and dynamic systems; we explain how we address this issue in Section 4.

3. LARGE-SCALE BENCHMARKS

We ran a series of experiments based on cooperative editing traces. A number of Wikipedia pages were stored as Treedocs, interpreting differences between successive versions of a page as a series of inserts and deletes. In some experiments our atoms were words; in the ones reported here an atom is a whole paragraph. We also ran similar experiments based on traces of SVN repositories containing LaTeX Java source code. Somewhat to our surprise, we observe a large frequency of deletes in all the traces. Consequently, the garbage collection issue cannot be ignored.

We studied medium-sized Wikipedia pages such as "Distributed Computing," reaching 20 KB of text in 800 revisions, or "PowerPC" reaching 25 KB in 400 revisions. Applying all the revisions for these pages required less than 1 second when using paragraphs as atoms, and 2 seconds using words. We also studied some frequently-edited pages, e.g., "George W. Bush" (GWB) reaching 150 KB in 40,000 revisions. Because of vandalism, the GWB page contains an even higher proportion of deletes than in the other traces (in the absence of rebalancing, 95% of nodes would be tombstones).

Hereafter we report only on the most stressful benchmark, i.e., the GWB traces, with a 256-ary tree, and full paragraphs as atoms. We rebalance arbitrarily every 1,000 revisions (different heuristics could be used to decide when to rebalance; this is out of scope). Note that 1,000 revisions include a much larger number of individual updates; for instance, in Figures 4 and 5, there are approximately 100,000 update operations between the last two rebalances.

Figure 4 shows the size of the GWB Treedoc structure over the first 350,000 updates of the GWB page. Size increases with successive updates, then falls dramatically at each periodic rebalancing. The decrease is attributable mostly to discarding tombstones, but also to shorter TIDs: the average TID size shrinks from 60 bytes just before rebalancing, to only 2 bytes.

Figure 5 shows execution time per update. Again, rebalancing has dramatic effect. Without rebalancing, the per-update time would grow up to $3\,ms$. Periodic rebalancing decreases the depth of the tree to 2-3 levels, and the slowest update takes only $0.3\,ms$.

From this, we can estimate the scalability of Treedoc. Assume that a user at his replica continuously initiates one update every 3 seconds. Then the system can sustain 1,000

simultaneous users without rebalancing, and 10,000 when rebalancing at 1,000-revision intervals.

4. SUPPORTING LARGE AND DYNAMIC NUMBERS OF REPLICAS

The CRDT approach guarantees that replicas converge. However, we saw that metadata accumulates over time, with a big impact on performance, and must be garbage-collected from time to time. The attendant commitment or consensus, albeit being infrequent and in the background, is still an issue for scalability. In this section, we explain how Treedoc addresses this issue.

Consensus requires the participation of a well-identified set of sites. Even worse, commitment requires their unanimous agreement. This is problematic in a large-scale system, where sites fluctuate dynamically. For instance, in collaborative editing scenarios, new participants may enter at any time, leave the system definitely, or disconnect for an unpredictable period of time, while continuing to initiate updates.

To address this problem, we partition the sites in two disjoint tiers. The *core* consists of a small group of sites that are well known and well connected. Joining or leaving the core follows a membership protocol [2]. In the limit, the core could reduce to a single server. Only core sites participate in rebalancing. Core sites may freely initiate updates, and may replay each others' updates.

Sites that are not in the core are part of the *nebula*. They are assumed to be uniquely identified (for disambiguators), but are otherwise unrestricted. The nebula may contain any number of sites, which are connected to the network, or may be currently disconnected. Nebula sites may freely initiate updates, but do not participate in commitment.

Let us call an interval between successful rebalances an *epoch*. Each rebalance – each change of epoch – changes the frame of reference for TIDs: TIDs from some epoch are invalid in other epochs. Two sites may send updates to each other, and replay each others' updates, if and only if they are in the same epoch. Core sites are all in the most recent epoch by construction, but a nebula site may remain in a prior epoch.

5. NEBULA CATCH-UP PROTOCOL

A site can leave the core at any time, simply by invoking the membership protocol.

The converse is not true. As the core moves to new epochs, nebula sites remain in prior epochs. For a nebula site to send updates to the core, or in order for it to join the core, a protocol is needed, allowing it to *catch up* with past core rebalances.

We now describe the catch-up protocol at a high level. To simplify the description, assume that the core and the nebula sites started from the same initial state, and that the core executed a single rebalance since then: If the core is in epoch n (the "new" epoch), the nebula is in epoch $n-1$ ("old" epoch). Updates in the old epoch use "old" TIDs, whereas those in the new epoch use "new" TIDs.

A core site maintains a buffer of update messages that it needs to send to the nebula, some of the old epoch, some of the new one. Conversely, a nebula site maintains a buffer of update messages to be sent to the core; they are all in the old epoch.

Old messages buffered in the core can be sent to the nebula site (operating in the old epoch) and replayed there. However, the converse is not true: since the core is in the new epoch, it cannot replay old updates from the nebula. The nebula must first bring them into the new epoch. To this effect, and once it has applied all old core updates, the nebula site rebalances its local replica of the tree, using the tree itself to keep track of the mapping between old and new TIDs. Then it translates old TIDs in buffered messages into the corresponding new TIDs. At this point, the nebula site is in the new epoch. (It may now either join the core, or remain in the nebula.) Finally, it sends its buffered messages to the core, which can now replay them.

Since epochs are totally ordered, every nebula site will go through the same catch-up protocol. Concurrent updates remain commutative, even if initiated in different epochs.

5.1 TID translation algorithm

We now describe in more detail how a nebula site translates TIDs from the old to the new epoch. It needs to distinguish updates that were received from the core and are serialised before the rebalance, from those initiated locally or received from other nebula sites, which must be serialised after the rebalance. For this purpose we colour the corresponding nodes either Cyan (C stands for Core) or Black (= *Noir* in French, where N stands for Nebula).

Thus we distinguish cyan nodes, cyan tombstones, black nodes and black tombstones. A node can be both a cyan node and a black tombstone; the converse is not possible.

We will now describe the steps that a nebula site needs to take in order to execute a rebalance operation. We will assume that all the updates from the core issued prior to the rebalance have been executed as well as some black updates, some local and some from other nebula sites. Once the rebalance is performed, the site will be able to send the black updates to the core. The rebalance will construct a list of subtrees, each having as root a cyan node.

The first step is to go through the tree and examine only cyan nodes and tombstones. Assume a sentinel node n_b marks the beginning of the ordered-set and ensures the tree is not empty. We identify the following cases:

- **Cyan Node** (can also be a black tombstone): add to the list along with any black children it has.
- **Cyan Tombstone**: add any black children to the subtree of the last node in the rebalanced list. We preserve the correct order by adding at the end of the subtree. If no cyan nodes have been seen so far, we add the black children to n_b.

The second step is to create the new balanced tree from the roots of the subtrees stored in the linear list. The nodes that have black children will be transformed into major nodes if

both a cyan child and a black child should be placed on the same position.

The last step is to go though the new tree and generate the updates to be sent to the core. We examine only black nodes and tombstones:

- **Black Node** - send *insert* operation with this TID and atom
- **Black Tombstone** - send *delete* operation with this TID

When a nebula site connects to the core, it sends not only black updates generated locally, but also updates received from other nebula sites. It may happen that a site receives the same update multiple times, but this causes no harm since updates are idempotent.

6. DISCUSSION

Massive distributed computing environments, such as Zookeeper or Dynamo [3], replicate data to achieve high availability, performance and durability. Unless such systems ensure replicas are consistent, application programmers will be faced with overwhelming complexity. However, strong consistency does not scale to these large-scale environments. For some applications, eventual consistency is sufficient [3], but in the general case it requires conflict detection and resolution, which is just consensus in a different guise.

In this paper, we propose to use CRDTs, because they ensure eventual consistency without requiring consensus. Although they do require garbage collection, which generally speaking requires consensus, this does not impact application performance and scalability, as it is off the critical path and remains hidden inside the abstraction boundary.

Not all abstractions can be converted into a CRDT: for instance a queue or a stack rely on a strong invariant (a total order) that inherently requires consensus. Treedoc on the other hand maintains a local, partial order, and the outcome of its operations need not be unique.

Even when an abstraction is not a CRDT, it is very useful to design it so that most pairs of operations commute when concurrent. Those pairs can benefit from cheap, high-performance protocols, resorting to consensus only for non-commuting pairs [7]. We expect that in many cases, non-commuting operations are infrequent and can be confined to a small core, whereas the vast majority of operations occur in the uncontrolled nebula.

Our experience teaches us a few interesting lessons about the requirements for CRDTs. To commute, operations must have identical pre-condition; in practice, all operations should have pre-condition "true." A central requirement is the use of unique, unchanging identifiers. To be practical, the data structure must remain compact. The Treedoc naming tree ensures that the metadata and identifier overhead is logarithmic in the size of the data.

7. CONCLUSION

It is well known that commutativity simplifies consistency maintenance, as it removes the need for complex concurrency control, allowing updates to execute in arbitrary orders while guaranteeing that replicas converge to the same result. However, the issue of designing shared data types for commutativity was neglected. We presented the Commutative Replicated Data Type or CRDT, designed to make concurrent updates commute. CRDTs enable increased performance and scalability compared to classical approaches.

We proposed a CRDT called Treedoc that maintains an ordered set of atoms while providing insert and delete operations. To overcome the challenges of practicality and scalability, we explored some innovative solutions. Each atom has a unique, system-wide, compact identification that does not change between rebalances. Garbage collection is a requirement in practice; it is disruptive and requires consensus, but it has lower precedence that updates, and it is not in the critical path of applications. We side-step the non-scalability of consensus by dividing sites into two tiers with different roles.

Our future work includes searching for other CRDTs as well as studying the interaction between CRDTs and classical data structures.

Acknowledgments

This work is supported in part by the Portuguese FCT/MCTES project PTDC/EIA/74325/2006, and under European Union FP7 contract Grid4All.

8. REFERENCES

[1] Baquero, C., and Moura, F. Using structural characteristics for autonomous operation. *Operating Systems Review 33*, 4 (1999), 90–96.

[2] Chockler, G. V., Keidar, I., and Vitenberg, R. Group communication specifications: a comprehensive study. *ACM Trans. Comput. Syst. 33*, 4 (Dec. 2001), 427–469.

[3] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. Dynamo: Amazon's highly available key-value store. In *21st Symp. on Op. Sys. Principles (SOSP)* (Stevenson, Washington, USA, Oct. 2007), vol. 41 of *Operating Systems Review*, ACM Sigops, ACM, pp. 205–220.

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

[5] Gray, J., and Lamport, L. Consensus on transaction commit. *Trans. on Database Systems 31*, 1 (Mar. 2006), 133–160.

[6] Johnson, P. R., and Thomas, R. H. The maintenance of duplicate databases. Internet Request for Comments RFC 677, Information Sciences Institute, Jan. 1976.

[7] Lamport, L. Generalized consensus and Paxos. Tech. Rep. MSR-TR-2005-33, Microsoft Research, Mar. 2005.

[8] Oster, G., Urso, P., Molli, P., and Imine, A. Data consistency for P2P collaborative editing. In *Int. Conf. on Computer-Supported Coop. Work (CSCW)*

(Banff, Alberta, Canada, Nov. 2006), ACM Press, pp. 259–268.

[9] Preguiça, N., Marquès, J. M., Shapiro, M., and Leția, M. A commutative replicated data type for cooperative editing. In *Int. Conf. on Distributed Comp. Sys. (ICDCS)* (Montréal, Canada, June 2009), pp. 395–403.

[10] Saito, Y., and Shapiro, M. Optimistic replication. *Computing Surveys 37*, 1 (Mar. 2005), 42–81.

[11] Schiper, N., and Pedone, F. Optimal atomic broadcast and multicast algorithms for wide area networks. In *Symp. on Principles of Dist. Comp. (PODC)* (Portland, OR, USA, 2007), Assoc. for Computing Machinery, pp. 384–385.

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

[13] Skeen, D., and Stonebraker, M. A formal model of crash recovery in a distributed system. *IEEE Trans. Softw. Eng. 9*, 3 (May 1983), 219–228.

[14] Weiss, S., Urso, P., and Molli, P. Logoot: a scalable optimistic replication algorithm for collaborative editing on P2P networks. In *Int. Conf. on Distributed Comp. Sys. (ICDCS)* (Montréal, Canada, June 2009).