Moving Elements in List CRDTs

Martin Kleppmann
mk428@cl.cam.ac.uk
University of Cambridge
Cambridge, United Kingdom

Abstract

Conflict-free Replicated Data Types (CRDTs) for lists allow multiple users to concurrently insert and delete elements in a shared list object. However, existing algorithms behave poorly when users concurrently move list elements to a new position (i.e. reorder the elements in the list). We demonstrate the need for such a move operation, and describe an algorithm that extends a list CRDT with an explicit move operation. Our algorithm can be used in conjunction with any existing list CRDT algorithm. In addition to moving a single list element, we also discuss the open problem of moving ranges of elements.

CCS Concepts: • **Theory of computation** → **Distributed algorithms**; • **Software and its engineering** → **Consistency**; • **Human-centered computing** → *Computer supported cooperative work*; • **Information systems** → *Collaborative and social computing systems and tools*; • **Computer systems organization** → *Peer-to-peer architectures.*

Keywords: conflict-free replicated data types, distributed consistency, collaborative editing, optimistic replication

ACM Reference Format:
Martin Kleppmann. 2020. Moving Elements in List CRDTs. In *7th Workshop on Principles and Practice of Consistency for Distributed Data (PaPoC '20), April 27, 2020, Heraklion, Greece.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3380787.3393677

1 Introduction

Conflict-free Replicated Data Types (CRDTs) allow multiple replicas to concurrently modify some shared data object, while ensuring that all replicas eventually converge towards the same, consistent state [18]. CRDT algorithms have been developed for various different datatypes, and one of the most important datatypes is the *list* (also called *sequence* or *array* datatype). A list is a collection of elements in a total order, which is chosen by the user.

Replicated lists can be used to implement a variety of important applications, such as:

- collaborative text editors (text is a list of characters);
- collaborative graphics applications (a list of graphical objects, where the order determines visibility: objects "further down" may be occluded or filtered by the objects "higher up", as illustrated in Figure 1);
- to-do lists and task trackers (a list of tasks, where the order may reflect the relative priority of tasks as chosen by the user).

Many list CRDTs have been developed, such as WOOT [13], Treedoc [14], RGA [16], Causal Trees/Timestamped Insertion Trees [2, 4], Logoot [20, 21], and LSEQ [9, 10]. Moreover, most of the field of Operational Transformation algorithms is dedicated to algorithms for collaboratively editable text, i.e. lists [3, 11, 12, 15, 19].

All of the aforementioned algorithms allow replicas to *insert* or *delete* elements anywhere in the list. However, none of them have explicit support for *moving* elements from one position to another position in the list (*reordering*). This is a surprising omission because moving is a commonly required operation: in many to-do list applications, a user can drag and drop list elements to reorder them, and graphics software allows users to reorder the object list with commands such as "bring to front" (which moves an object to the top of the list, so that it occludes other objects) and "send to back" (which moves an object to the bottom, so that it is occluded by other objects). In the example of Figure 1, imagine the user first creates the pupils and then the white circles of the eyes; if the software by default places new objects in front of existing objects, the pupils would then not be visible. The user thus needs to reorder the object list to bring the pupil objects to the front.

In this paper we introduce an algorithm that allows an existing list CRDT to be extended with a move operation. The algorithm is generic in the sense that it can be implemented on top of any of the aforementioned list CRDTs.

2 Semantics of Concurrent Moves

The simplest way of moving a list element is to delete it from its existing position, and to re-insert it at the new position. However, if two replicas concurrently move the same element, the result is that the element is duplicated: the

PaPoC '20, April 27, 2020, Heraklion, Greece

ACM ISBN 978-1-4503-7524-5/20/04…$15.00
https://doi.org/10.1145/3380787.3393677

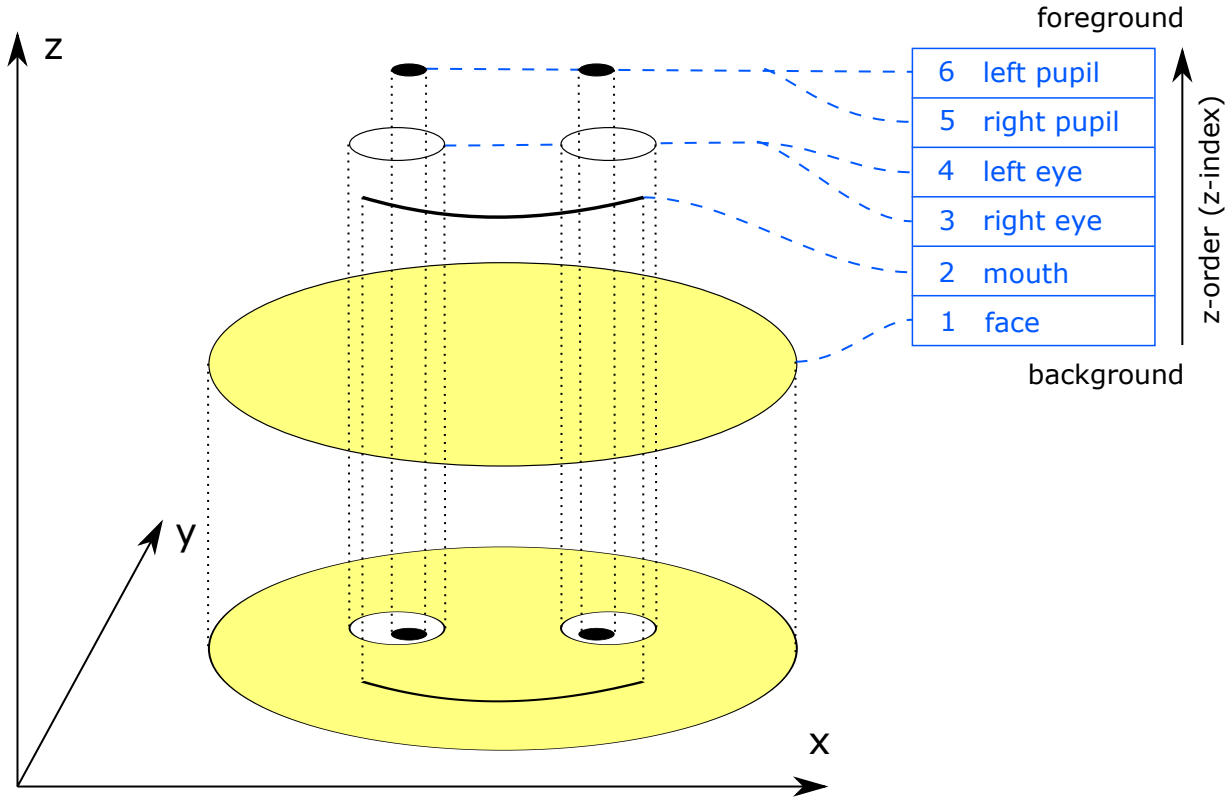

Figure 1. Graphics software often places objects in a list that determines the z-index; objects higher in the list occlude lower ones. Here, the eyeballs and mouth are placed above the yellow face, and the pupils in turn are placed above the eyeballs.

two deletions of the same element behave the same as one deletion, and the two insertions independently re-insert the element twice.

We argue that such duplication in the face of concurrency is undesirable, as it is generally not the behaviour expected by users. For example, say a user has replicas of their to-do list on their laptop and their smartphone. They move items on their laptop to match new priorities, and also move some items on their phone while the phone is offline, as illustrated in Figure 2. Later, when the phone is online again and synchronises with the laptop, any items that were moved on both devices will be duplicated.

This behaviour is especially confusing if the same item was moved to the same position (e.g. to the top of the list) on both devices, in which case the duplicated items are likely to be adjacent in the final list, as in Figure 2 (depending on any other insertions that might have taken place). However, even if the concurrent move operations had different destination positions, duplicating the moved element at both destinations is unlikely to be desirable behaviour in most applications.

Rather, we argue that the best semantics in this situation is for the system to pick one of the destination positions as the "winner"; as the replicas communicate, they all converge towards a list in which the moved element is located only at the winning position, and nowhere else. This approach is illustrated in Figure 3. Any deterministic method of picking the winner is suitable (e.g. based on a priority given to each replica, or based on the timestamp of the operations).

3 A Generic Moving Algorithm

Intuitively, if we want to pick one winner from among several concurrent moves for the same element, we can use a long-established CRDT: a last-writer wins (LWW) register [5, 17]. We need one such register for each list element, containing the position of that element. Moving an element to a new location is then merely a matter of updating the value of that element's register.

In general, we could also use a multi-value (MV) register [17], which would allow an element to exist at multiple positions in the list simultaneously after having experienced conflicting moves. This is not the same as the duplication described in Section 2, since the multiple positions still belong to the same element, and a subsequent move would collapse them back into a single position again. The MV register option can be used if the LWW behaviour (discarding all but one of the concurrent moves of the same element) is unacceptable.

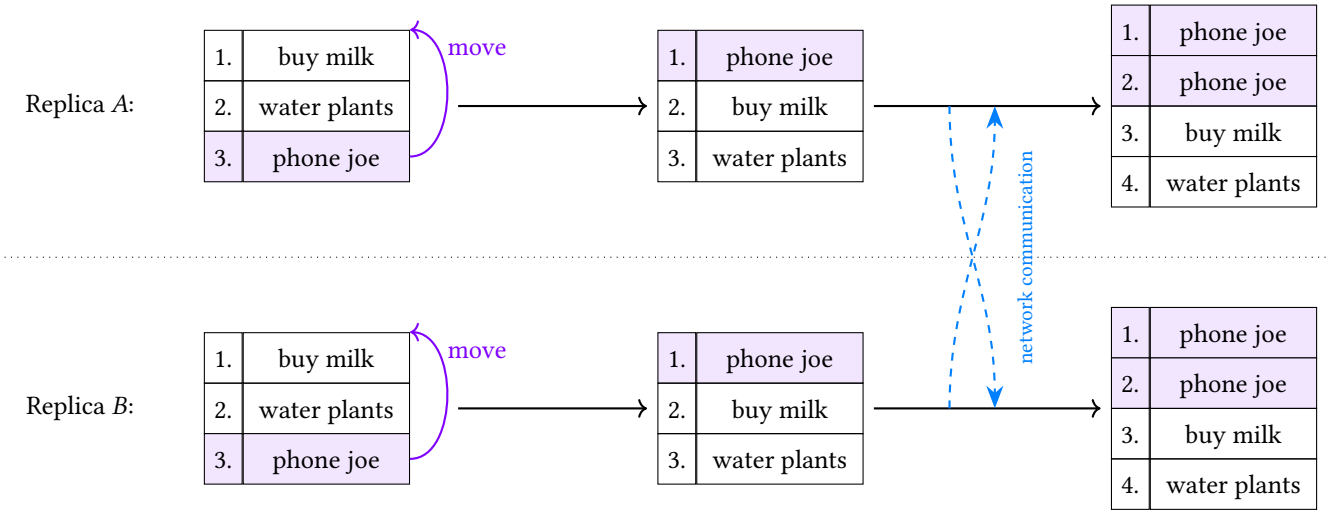

Figure 2. If moving is implemented by deleting and re-inserting, concurrent moves of the same element result in the anomaly shown here: the moved element is duplicated.

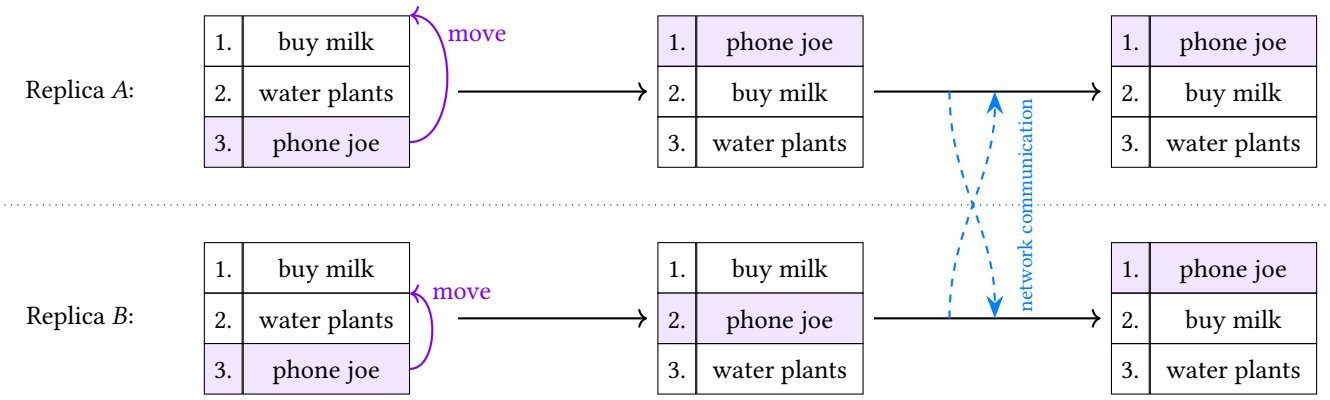

Figure 3. When the same element is concurrently moved to different positions on different replicas, one of those positions should become the "winning" position as the replicas converge. In this example, replica A's move operation wins.

Regardless of whether a LWW or MV register is used, this approach requires a stable way of referencing some position in the list (i.e. the value held by the register). An integer index is not suitable, because an insertion or deletion at any position would require changing all of the indices following that position. Fortunately, all known list CRDTs already incorporate schemes for stable position identifiers. The details vary: Treedoc uses a path through a binary tree [14], Logoot uses a list of (integer, replicaID) pairs [20], RGA uses a so-called s4vector [16], Causal Trees/Timestamped Insertion Trees use a timestamp [2, 4], and so on. But all of these approaches have in common that they can uniquely refer to a particular position in a list, in a way that is not affected by operations on other list positions.

Assume now that $p_1, p_2, \ldots$ are position identifiers according to one of the aforementioned schemes, and $v_1, v_2, \ldots$ are the values of list elements (e.g. to-do list items). We can now model the state of the list as a set of (*value*, *register*) pairs

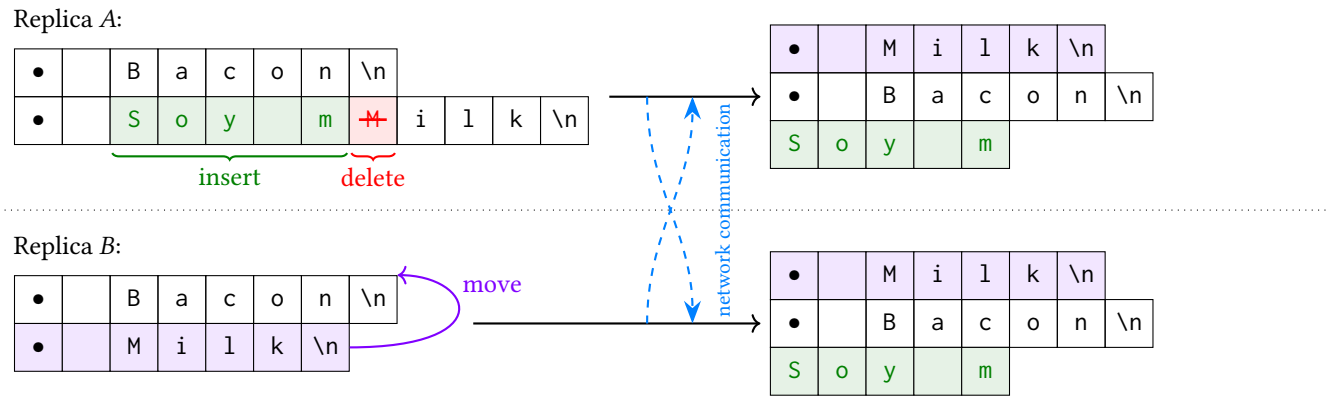

Figure 4. Replica *B* moves a range of characters (here, "• Milk"), while concurrently replica *A* makes an edit inside the same range. If the move is performed character-by-character, *A*'s edit remains at the old position, while its surrounding context has moved to a new position, leading to an anomalous outcome.

contained in an add-wins set CRDT:

$$state = \text{AWSet}(\{(v_1, \text{LWWRegister}(p_1)), \\ (v_2, \text{LWWRegister}(p_2)), \ \ldots\})$$

In some schemes (such as Treedoc, Logoot and LSEQ), the position identifiers themselves encode their relative ordering [7], and thus no further state is required. In others (such as RGA and CT), the list CRDT requires an additional data structure that determines the order in which position identifiers appear in the list.

In order to obtain the elements in their list order we need to query the above set by the position identifiers stored within the registers. We can do this efficiently by treating the set of pairs as a table in a relational database and using a database index on the position-register column. The index lets us efficiently find all the list elements with a particular position ID (or range of position IDs).

Note that when the position of a list element is changed, the old position identifier no longer has any value associated with it, and thus it should be regarded as nonexistent (like a tombstone in some CRDTs). It is also possible for multiple list elements to have the same position identifier, and thus for several list elements to appear at the same position. This is similar to having a multi-value register of values at each list position. (Note this situation is different from our earlier use of a MV register, which was about one element having multiple positions, not one position having multiple elements.)

We can avoid having multiple list elements with the same position identifier by always creating a fresh position identifier as the destination of a move operation. All of the aforementioned list CRDTs have methods of creating new, globally unique position identifiers for any location within a list.

This algorithm is a CRDT because its state is a straightforward composition of existing CRDTs.

4 Moving Ranges of Elements

The algorithm of Section 3 allows a single list element to be moved at a time. In some applications, a further challenge arises: we may want to move not just one element, but move an entire consecutive range of elements in one operation. For example, in a text document, a user may want to move a section of text (e.g. a paragraph, or the text of a bullet point) to a different position in the text [8].

We might try to move a range of elements by moving each element individually. However, this approach has a problem, which is illustrated in Figure 4. In this example, replica *B* wants to swap the order of two bullet points in the text of a shopping list by moving the character range "• Milk" (including the bullet point character •), while concurrently replica *A* changes the text "Milk" to "Soy milk". While the move is successful, the character insertions and deletion of *A*'s edit remain attached to the old position of "Milk" (near the end of the document, after "Bacon\n"). This is the case because CRDTs for lists and text perform insertions and deletions at particular position IDs, which remain unchanged even as elements are moved to new position IDs.

In contrast, the desired outcome of this scenario is shown in Figure 5: we want *A*'s edit to apply in the context in which it was originally made, even if that context has since moved to a new position in the document. In order to enable this semantics, it is not sufficient to move each element individually: we need to capture the fact that a certain range of elements is moved, so that any concurrent edits that fall within that range can be redirected to the new position of that range.

At present, there is no known algorithm for performing such moves of ranges of elements. Various tricky edge-cases would need to be handled by such an algorithm, such as concurrent moves of partially overlapping ranges, and moves whose destination position falls within another range that is

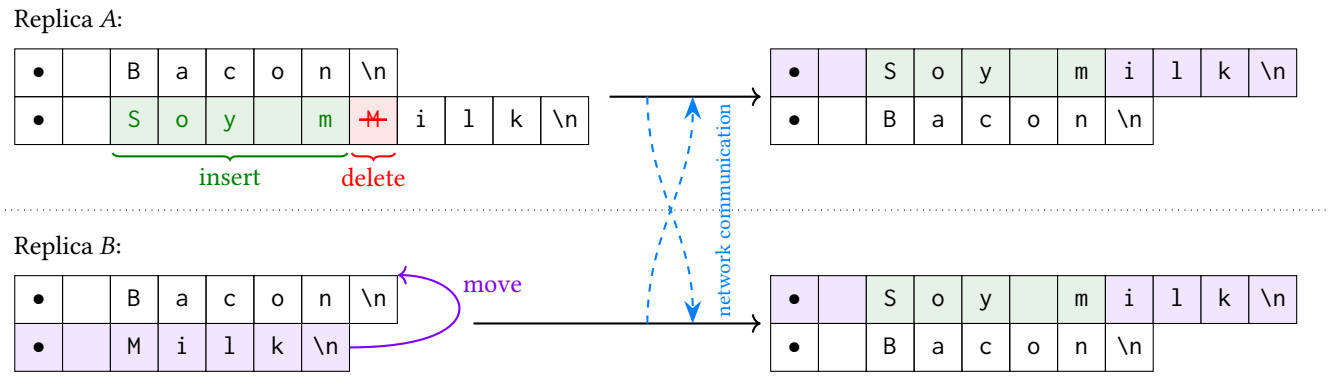

Figure 5. The desired outcome of the scenario in Figure 4.

itself being moved. Part of the solution may be to give each list element a unique identity that is independent from its position in the document, so that the position of a list element can be changed without changing its identity; insertions and deletions can then reference the *identities* of their context, rather than their position IDs.

We will briefly discuss one more approach for moving ranges, and explain why it is not sufficient. In the example of Figures 4 and 5, if the application needs to support reordering of bullet points, we might try to simplify the problem by using a two-level hierarchy instead of a flat sequence of characters. That is, we can create a top-level list object in which each list element contains the entire content of a single bullet point, and use a separate text CRDT object for the content of each bullet point. In this design, moving bullet points requires only a single-element move operation.

However, this approach has a different problem: when the user presses the enter key in the middle of the text of a bullet point, the usual behaviour in most editors is to split it into two bullet points. Conversely, hitting the delete key while the cursor is positioned at the end of a bullet point's text usually joins it with the following bullet point's text. If each bullet point is represented as a separate text object, these splitting and joining operations require moving a range of characters from one text object to another.

Thus, representing each bullet point as a single list element for the sake of moving does not actually obviate the need for moving ranges of characters. As before, implementing move as delete-and-reinsert has the effect of duplicating text if two users concurrently split or join the same bullet point. Thus, it seems that support for moving ranges of characters is an important feature for collaborative document editors.

5 Related work

Our previous unpublished work [6] discusses a single-element move operation for lists in the context of a larger specification framework for CRDTs. However, this work does not provide an efficient implementation of this operation.

Apart from this work, to our knowledge Ahmed-Nacer et al. [1] provide the only published list CRDT that includes a single-element move operation. However, it is designed to duplicate an element when moved concurrently – semantics we consider undesirable. Moreover, we believe the algorithm to be more complicated than necessary.

All other list CRDTs [2, 4, 9, 10, 13, 14, 16, 20, 21] and Operational Transformation algorithms [3, 11, 12, 15, 19] provide only insertion and deletion.

Lord [8] discusses the problem of moving ranges of elements, but does not present a solution.

6 Conclusions

This short paper has made the case for list CRDTs to support an explicit move operation, allowing list elements to be reordered without duplicating them in the case of concurrent moves. We have outlined an algorithm for single-element moves that can be retrofitted to any existing list CRDT that uses stable position IDs to refer to locations within the list (which is the case for all currently known list CRDTs). We have also demonstrated the need for moving a contiguous range of list elements to a new position; finding an algorithm that adds this feature to a CRDT is at present an open problem. We hope that future research will succeed in developing an algorithm for moving ranges of elements.

Acknowledgments

Martin Kleppmann is supported by a Leverhulme Trust Early Career Fellowship and by the Isaac Newton Trust.

References

[1] Mehdi Ahmed-Nacer, Pascal Urso, Valter Balegas, and Nuno Preguiça. 2013. Concurrency Control and Awareness Support for Multi-synchronous Collaborative Editing. In *9th IEEE International Conference on Collaborative Computing (CollaborateCom)*. ICST. https://doi.org/10.4108/icst.collaboratecom.2013.254113

[2] Hagit Attiya, Sebastian Burckhardt, Alexey Gotsman, Adam Morrison, Hongseok Yang, and Marek Zawirski. 2016. Specification and Complexity of Collaborative Text Editing. In *ACM Symposium*

on Principles of Distributed Computing (PODC). 259–268. https://doi.org/10.1145/2933057.2933090

[3] Clarence Ellis and S J Gibbs. 1989. Concurrency Control in Groupware Systems. In *ACM International Conference on Management of Data (SIGMOD)*. 399–407. https://doi.org/10.1145/67544.66963

[4] Victor Grishchenko. 2014. Citrea and Swarm: Partially ordered op logs in the browser. In *1st Workshop on Principles and Practice of Eventual Consistency (PaPEC)*. https://doi.org/10.1145/2596631.2596641

[5] Paul R Johnson and Robert H Thomas. 1975. RFC 677: The Maintenance of Duplicate Databases. https://tools.ietf.org/html/rfc677

[6] Martin Kleppmann, Victor B. F. Gomes, Dominic P. Mulligan, and Alastair R. Beresford. 2018. OpSets: Sequential Specifications for Replicated Datatypes (Extended Version). https://arxiv.org/abs/1805.04263

[7] Martin Kleppmann, Victor B. F. Gomes, Dominic P. Mulligan, and Alastair R. Beresford. 2019. Interleaving anomalies in collaborative text editors. In *6th Workshop on Principles and Practice of Consistency for Distributed Data (PaPoC 2019)*. ACM. https://doi.org/10.1145/3301419.3323972

[8] Robert Lord. 2019. Notes on Splicing CRDTs for Structured Hypertext. https://lord.io/blog/2019/splicing-crdts/

[9] Brice Nédelec, Pascal Molli, and Achour Mostefaoui. 2016. CRATE: Writing Stories Together with our Browsers. In *25th International World Wide Web Conference (WWW)*. 231–234. https://doi.org/10.1145/2872518.2890539

[10] Brice Nédelec, Pascal Molli, Achour Mostefaoui, and Emmanuel Desmontils. 2013. LSEQ: an Adaptive Structure for Sequences in Distributed Collaborative Editing. In *13th ACM Symposium on Document Engineering (DocEng)*. 37–46. https://doi.org/10.1145/2494266.2494278

[11] David A Nichols, Pavel Curtis, Michael Dixon, and John Lamping. 1995. High-Latency, Low-Bandwidth Windowing in the Jupiter Collaboration System. In *8th Annual ACM Symposium on User Interface Software and Technology (UIST)*. 111–120. https://doi.org/10.1145/215585.215706

[12] Gérald Oster, Pascal Molli, Pascal Urso, and Abdessamad Imine. 2006. Tombstone Transformation Functions for Ensuring Consistency in Collaborative Editing Systems. In *2nd International Conference on Collaborative Computing (CollaborateCom)*. https://doi.org/10.1109/COLCOM.2006.361867

[13] Gérald Oster, Pascal Urso, Pascal Molli, and Abdessamad Imine. 2006. Data Consistency for P2P Collaborative Editing. In *ACM Conference on Computer Supported Cooperative Work (CSCW)*. https://doi.org/10.1145/1180875.1180916

[14] Nuno Preguiça, Joan Manuel Marquès, Marc Shapiro, and Mihai Letia. 2009. A commutative replicated data type for cooperative editing. In *29th IEEE International Conference on Distributed Computing Systems (ICDCS)*. https://doi.org/10.1109/ICDCS.2009.20

[15] Matthias Ressel, Doris Nitsche-Ruhland, and Rul Gunzenhäuer. 1996. An Integrating, Transformation-Oriented Approach to Concurrency Control and Undo in Group Editors. In *ACM Conference on Computer Supported Cooperative Work (CSCW)*. 288–297. https://doi.org/10.1145/240080.240305

[16] Hyun-Gul Roh, Myeongjae Jeon, Jin-Soo Kim, and Joonwon Lee. 2011. Replicated abstract data types: Building blocks for collaborative applications. *J. Parallel and Distrib. Comput.* 71, 3 (2011), 354–368. https://doi.org/10.1016/j.jpdc.2010.12.006

[17] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. *A comprehensive study of Convergent and Commutative Replicated Data Types*. Technical Report 7506. INRIA. http://hal.inria.fr/inria-00555588/

[18] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-free Replicated Data Types. In *13th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*. 386–400. https://doi.org/10.1007/978-3-642-24550-3_29

[19] Chengzheng Sun and Clarence Ellis. 1998. Operational Transformation in Real-Time Group Editors: Issues, Algorithms, and Achievements. In *ACM Conference on Computer Supported Cooperative Work (CSCW)*. 59–68. https://doi.org/10.1145/289444.289469

[20] Stéphane Weiss, Pascal Urso, and Pascal Molli. 2009. Logoot: A Scalable Optimistic Replication Algorithm for Collaborative Editing on P2P Networks. In *29th IEEE International Conference on Distributed Computing Systems (ICDCS)*. 404–412. https://doi.org/10.1109/ICDCS.2009.75

[21] Stéphane Weiss, Pascal Urso, and Pascal Molli. 2010. Logoot-Undo: Distributed Collaborative Editing System on P2P networks. *IEEE Transactions on Parallel and Distributed Systems* 21, 8 (Jan. 2010), 1162–1174. https://doi.org/10.1109/TPDS.2009.173