Ordering Operations for Generic Replicated Data Types using Version Trees

Nazmus Saquib
Univ. of California, Santa Barbara
USA
nazmus@cs.ucsb.edu

Chandra Krintz
Univ. of California, Santa Barbara
USA
ckrintz@cs.ucsb.edu

Rich Wolski
Univ. of California, Santa Barbara
USA
rich@cs.ucsb.edu

Abstract

Data replication facilitates availability and recovery in a distributed environment. However, concurrent updates to multiple replicas result in divergence of data. Conflict-Free Replicated Data Types (CRDTs) are abstract data types that provide a principled approach to asynchronously reconcile this divergence. We propose a different perspective on the divergence of data, whereby we treat data divergences as *versions* of the data. That is, instead of treating it only as a problem that needs to be solved, we consider it also to be a feature that provides a way to track versioning and evolution of data. Versioning information is helpful in multiple scenarios, such as provenance tracking and system debugging. Doing so allows us to leverage concepts such as the *version tree* found in the literature for persistent (versioned) data structures. We show that many techniques used in CRDTs to order elements can be derived from version trees, which predates CRDTs by more than 20 years. Using version trees for maintaining order and append-only logs for storage, we propose a method to ensure convergence of arbitrary data types, while maintaining information related to the evolution of data.

CCS Concepts: • **Computing methodologies → Distributed algorithms**.

Keywords: CRDT, data structure, replication, data versioning

ACM Reference Format:
Nazmus Saquib, Chandra Krintz, and Rich Wolski. 2022. Ordering Operations for Generic Replicated Data Types using Version Trees. In *9th Workshop on Principles and Practice of Consistency for Distributed Data (PaPoC'22), April 5–8, 2022, RENNES, France.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3517209.3524038

PaPoC'22, April 5–8, 2022, RENNES, France
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9256-3/22/04.
https://doi.org/10.1145/3517209.3524038

1 Introduction

Distributed systems often replicate data for low latency and high availability. The shared state among the replicas is maintained according to different system models. For example, strong consistency requires a replica to coordinate with other replicas to execute an operation. Coordination increases latency and an operation may fail as a result of a network partition, preventing the necessary communication between replicas. Weaker consistency models such as eventual consistency enable replicas to execute an operation locally and asynchronously propagate the operation to other replicas. This results in lower latency but with a temporary divergence in replica views that must be eventually reconciled.

Conflict-Free Replicated Data Types (CRDTs) [14, 20, 21] are abstract data types that provide a principled approach for this asynchronous reconciliation. CRDTs support a weaker model than strong consistency, namely, Strong Eventual Consistency (SEC) [21]. SEC guarantees that whenever two replicas receive the same set of updates, possibly in a different order, they reach the same state.

Broadly, there are two types of CRDTs: state-based and operation-based (or op-based) [21]. In state-based CRDTs, an operation is executed on the local replica state. A replica periodically propagates its state to other replicas to achieve consistency. A disadvantage of this approach is the communication overhead associated with shipping the full state, which at times can be large. Apart from this, state-based CRDTs require data type-specific *merge* function that provides a *join* for any pair of replica states, such that the set of all states forms a *semilattice*. In op-based CRDTs, an operation is executed on the local replica and the operation is asynchronously propagated to other replicas. Although op-based CRDTs do not communicate state, they require exactly-once causal broadcast. Moreover, op-based CRDTs require all operations on a data type to be commutative. Delta State Conflict-Free Replicated Data Types (δ-CRDTs) [1] combine the advantages of state-based and op-based CRDTs. Like the state-based, δ-CRDTs can tolerate unreliable networks and, in particular, do not require exactly-once causal broadcast as a communication network property. Moreover, like the op-based approach, they do not require full replica state to be communicated, but rather, they communicate only state changes or "deltas".

Due to the commutativity requirement of op-based CRDTs and the join semilattice requirement of the state-based CRDTs, often we must resort to using a restricted form of a data type rather than the conventional one. For example, the CRDT literature describes a number of variations for set, such as a grow-only set (G-set) where elements can only be added, two-phase set (2P-set) where a removed element cannot be added later, etc. In general, CRDTs for custom data types [3] have not been extensively studied in the literature, and the use of such data types often requires ad-hoc solutions.

One way to guarantee consistency among replicas of an arbitrary data type is to ensure all the replicas execute the same set of operations in the same order. In fact, this is the principle used in protocols such as Raft [12], which provides strong consistency. However, in a weaker consistency model where we allow the log of operations to diverge among the replicas, this might entail extra work. Specifically, we might need to rollback the log of operations up to a certain point, introduce new operation(s), and re-execute the previously rolled back operation(s). Figure 1 illustrates one such scenario. We represent a replica with ID s as X_s and its log of operations as $OpLog(X_s)$. Let us assume X_s executed op_x, op_y, and op_z in order (first column). Later on, X_s came to know of operation $op_{x'}$ which must be executed after op_x. Therefore, X_s first rolls back the log of operations upto op_x (second column), then executes the new operation $op_{x'}$ (third column), and finally re-executes op_y and op_z (fourth column). During the log rollback, the state changes resulting from the rolled back operations (in this case op_y and op_z) must be rolled back as well.

Merely applying a seemingly inverse operation might not have the intended outcome of rolling back operations. For example, roll back of an insertion into a binary search tree cannot be performed by a corresponding deletion, as this might result in a different tree structure than the one resulting from the insertion operation not being executed in the first place. Therefore, we need a mechanism to revert the underlying data structure to a state where the rolled back operations were not even applied, rather than executing operations that appear to negate the effect of previous operations. One generic way to achieve this is to represent the state of the underlying data type using append-only logs as well. Then to ensure that the state of the underlying data type is rolled back along with the log of operations, the replica need only roll back entries from the tails of the logs used to represent the state of the data type. A similar technique is used for the move operation in a replicated tree data model as well [8]. While representing states for simple data types such as registers using logs is simple; more involved data types such as linked lists and binary search trees require complex algorithms [18].

The CRDT literature contains multiple works on maintaining order in list or sequence such as a causal tree (CT) [5, 6], replicated growable array (RGA) [17, 20], LSEQ [11], Treedoc [15], and WOOT [13]. Most of these works were originally developed for collaborative text-editing environments. However, we can use these algorithms to maintain an ordered list of operations (i.e., $OpLog(X_s)$) for arbitrary data types as well. We show that RGA shares similarities with *version trees* [4]. Version trees are tree data structures used to record causal relationship in versioned data structures (also known as *persistent data structures*). This implies that it is possible to incorporate data versioning while maintaining the order of operations. Data versioning has multiple use cases, such as facilitating system debugging, efficiently answering temporal query, and tracking evolution of data [18].

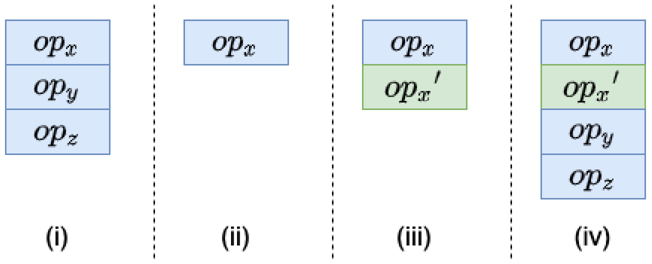

Figure 1. Log rollback to reach correct order of operations (from $op_x \rightarrow op_y \rightarrow op_z$ to $op_x \rightarrow op_{x'} \rightarrow op_y \rightarrow op_z$).

In this work, we first show how maintaining order of operations can lead to data versioning. At the same time, we explain the principle which allows us to track versions of arbitrary data types. Then, we show that version lists (text representations of version trees) for versioned data structures can use the same algorithm as RGA, a popular algorithm to maintain order in list or sequence CRDTs. This in turn implies any algorithm based on RGA has its root in version trees. In essence, the principle used for maintaining order in many list or sequence CRDTs lies in a 30 year old work on versioned data structures; which, interestingly, was formulated for a single-machine system with volatile memory. As versioning information can be captured while maintaining order of operations, this allows us to incorporate data versioning in our system without additional work; thus facilitating system debugging, data provenance, and efficient temporal query.

2 Persistent Data Structures

Conventional data structures are *ephemeral*, i.e., an update operation mutates the current state of the underlying data structure, resulting in a new state. However, many applications in computational geometry [19, 22] and text editing [16] can benefit from data versioning. Moreover, versioned data structures can facilitate system debugging and efficient temporal query [18].

In versioned data structures, an update operation results in a new version of the data structure while keeping records of all the previous versions. Versioned data structures are also known as persistent data structures [4] in the literature. A versioned data structure is *partially persistent* if all versions can be accessed but only the latest can be updated. A versioned data structure is *fully persistent* if all versions can be both accessed and updated. Although the literature contains detailed descriptions of both specific [7, 10] and generic [4] methods to make data structures persistent, in this work we are more concerned with how a *version stamp* is represented and ordered.

In persistent data structures, each version is tagged with a monotonically increasing version stamp (an integer value). The first version of a data structure has the version stamp 1. Whenever an update operation is applied to the data structure resulting in a new version, the version stamp is incremented by 1 and the new version is tagged with this incremented value. Throughout this paper, we use version stamp to refer to both the version of the underlying data structure and the operation that resulted in that version. The intended use will be clear from the context.

Although for partially persistent data structures this versioning scheme results in a natural linear order, fully persistent data structures only have a partial order over the version stamps. This partial ordering is defined by a rooted *version tree*. Each node in a version tree contains a version stamp. A directed edge from node u to node v denotes that version v was obtained by updating version u. The *youngest* child, i.e., the latest version among the children of a parent is always placed at the leftmost position (cf. Figure 2). One way to impose a total order on the partial order represented through a version tree is to perform a preorder traversal and aggregate the traversed nodes in a list, known as the *version list*. In fact, we do not need to maintain the version tree explicitly to create the version list. If we simply insert a new version v immediately after its parent u in the list, it maintains the preorder traversal, which can be proved using mathematical induction [4]. This insertion scheme also implies that the version list has a special property: for any version u, the descendants of u in the version tree occur consecutively in the version list, starting with u.

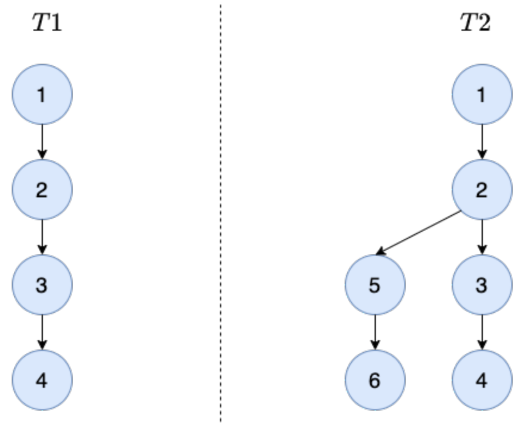

Figure 2. Example of two version trees. In the linear tree $T1$, a new version is obtained by always updating the latest version. Tree $T2$ illustrates a scenario where a new version can be obtained by updating some previous version. Although version 3 in $T2$ was obtained by updating the then latest version 2, version 5 was also created by updating version 2. Note that as 5 is greater (younger) than 3, 5 is placed to the left of 3. In general, the children of a node in a version tree are arranged in descending order of their version stamps from left to right. The corresponding version list of $T2$ is $[1, 2, 5, 6, 3, 4]$.

3 Version Trees as Replica States

The versioning scheme described in Section 2 can be used to model the divergence (using version trees) and the subsequent convergence (using version lists) among replicas in a distributed system. We consider a distributed system with N replicas (nodes). Each replica is assigned a replica ID from a set S where the elements in set can be sorted according to some criteria, e.g., lexicographically. As we will see in Section 3.1, it is crucial to have this sorting ability for achieving consistency.

We assume that each replica $X_s, s \in S$ maintains a log of operations $OpLog(X_s)$ (cf. Section 1). $OpLog(X_s)$ implicitly records a linear version tree ($VT(X_s)$). Note that even if the underlying data structure is not versioned (i.e., it is mutable), we can still map the divergence and convergence using version tree and version list respectively. In essence, we are using the *causal* or *happens-before* relationship embedded within the version tree/list to achieve consistency. This relationship will be present among the operations of any data structure, irrespective of the data structure being ephemeral or persistent.

We further assume that to achieve consistency, each replica reads the OpLog of another replica at a regular interval and performs a *merge* step. A merge step is always between two replicas, one known as the *source* and the other known as the *reader*. During a merge step, the reader incorporates all the operations unknown to it but known to the source into the reader's OpLog. Note that a merge step is unidirectional, i.e., operations known to the reader but unknown to the source are incorporated into the OpLog of the source during some other merge step where the current source is the reader. Once all replicas observe the full set of operations and apply a consistent ordering scheme, the system achieves consistency. As described in Section 3.3, the ordering scheme involves implicitly creating a version tree from the OpLogs of the source and the reader, followed by creating a linear tree from a modified preorder traversal of the version tree, which we call *InterleavedPreorder* (cf. Section 3.2).

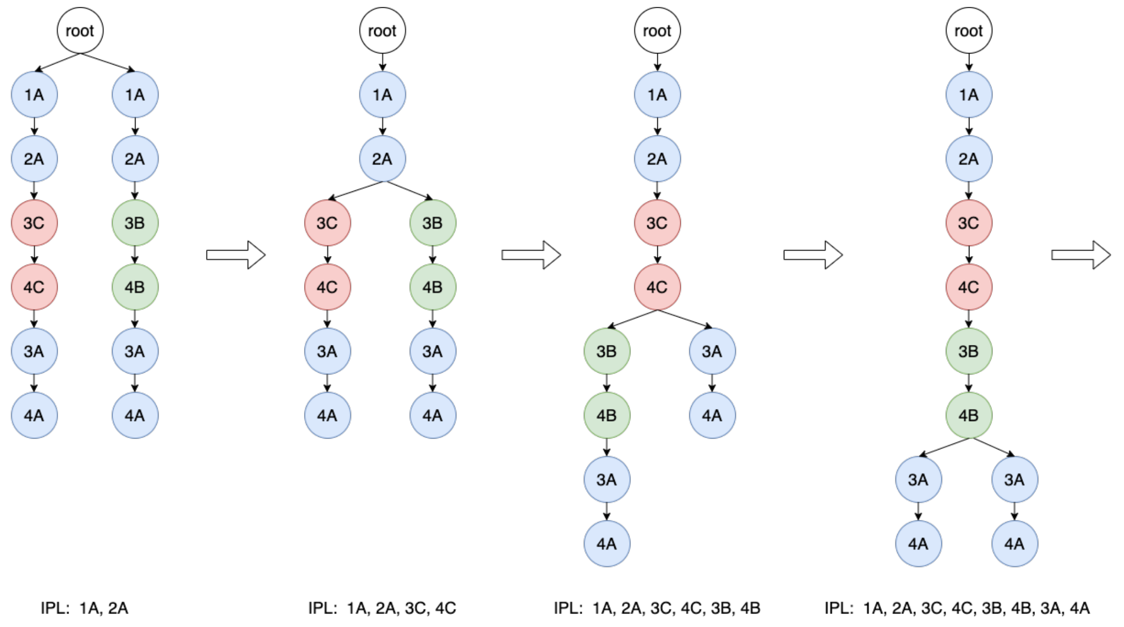
root
1A
2A
3C
4C
3A
4A
3B
4B
IPL: 1A, 2A
IPL: 1A, 2A, 3C, 4C
IPL: 1A, 2A, 3C, 4C, 3B, 4B
IPL: 1A, 2A, 3C, 4C, 3B, 4B, 3A, 4A

Figure 3. InterleavedPreorder traversal.

3.1 Version Stamps

The version stamps described in Section 2 are monotonically increasing. This is relatively simple to maintain in a single-machine system. However, in a distributed system with multiple machines, it requires either a special sequencer node or complex coordination to ensure that the next higher version stamp is allotted to only one replica. Hence, we propose using a concatenation of a monotonically increasing counter value (local to each replica) and the replica ID to represent the version stamp. We represent the counter and the replica ID of a version stamp vs by $vs.counter$ and $vs.replicaID$ respectively. We say version stamp vs_a is less than version stamp vs_b ($vs_a < vs_b$) if (i) the counter of vs_a is less than that of vs_b, or (ii) both the counters are the same but the replica ID of vs_a is less than that of vs_b.

When replica X_s executes a new operation in response to a client (i.e., a process that can send update/access request to any replica) request, X_s tags the operation with version stamp vs ($vs.replicaID = s$) which is greater than all other version stamps it has observed so far (operations that *happened before*). Thus if operation op_a happens before op_b, $vs_a < vs_b$ where vs_a and vs_b are the version stamps of operations op_a and op_b, respectively. We further assume that any version stamp is greater than the *null* version stamp, i.e., a version stamp having invalid/null values for counter and replica ID fields. This form of version stamp is essentially the Lamport timestamp [9]. This allows us to generate version stamps based only on local information without any complex coordination, all the while capturing the causal or happens-before relationship.

3.2 InterleavedPreorder Traversal

In this section, we define the interleavedPreorder traversal for trees with at most two branches. Trees with more branches can be generalized from this construction, by "flattening" branches in pairs (cf. Section 3.4). Each node η in a version tree contains three fields: (i) a version stamp vs, (ii) a left pointer, and (iii) a right pointer. We represent a field f of a node η as $\eta.f$. If a node has only one child, the left pointer points to the child whereas the right pointer points to the null node ϕ.

To generate the InterleavedPreorder traversal list IPL, we first add all nodes up to the branching point in IPL. We maintain two pointers, one for the top of the left branch and another for the top of the right branch, advancing them when the corresponding nodes are incorporated in the IPL. Next, we start adding nodes from the left branch until we reach a node x, such that the version stamp of x is smaller than or equal to that of the topmost node of the right branch. In case the version stamps of the top of the branches are equal, we advance both the pointers. Once the top of the right branch has a greater version stamp than that of x, we create an intermediate tree where each node in the IPL appears one after another in a linear structure. The branch pointed to by the right pointer becomes the new left branch and the portion of the previous left branch starting from x becomes the new right branch. We then perform InterleavedPreorder traversal on this intermediate tree. This continues recursively until we end up with a tree having only a single branch. A linear traversal of this tree gives us the final value of IPL. Algorithm 1 presents the recursive procedure to populate IPL. The initial call to the procedure is made with a dummy root η connecting two linear trees and an empty list λ where the IPL is populated. Figure 3 illustrates InterleavedPreorder traversal on a tree with two branches.

Algorithm 1 InterleavedPreorder Traversal

```
Require: VT node $\eta$, list $\lambda$
Ensure: IPL is populated in $\lambda$
 1: procedure POPULATEIPL($\eta, \lambda$)
 2:     if $\eta = \phi$ then
 3:         return
 4:     end if
 5:     while $\eta.left \neq \phi \wedge \eta.right = \phi$ do    ▷ traverse linear part
 6:         $\eta \leftarrow \eta.left$
 7:         $\lambda.append(\eta.vs)$
 8:     end while
 9:     if $\eta.left = \phi \wedge \eta.right = \phi$ then    ▷ all nodes traversed
10:         return
11:     end if
12:     $\eta_l \leftarrow \eta.left$
13:     $\eta_r \leftarrow \eta.right$
14:     $\eta.right = \phi$
15:     while $\eta_l \neq \phi \wedge \eta_r \neq \phi \wedge \eta_l.vs \geq \eta_r.vs$ do    ▷ add from the
        left branch until a version stamp is observed that is less than
        that of the version stamp at the top of the right branch
16:         $\eta.left = \eta_l$
17:         $\eta = \eta.left$
18:         $\lambda.append(\eta.vs)$
19:         if $\eta_l.vs = \eta_r.vs$ then
20:             $\eta_r = \eta_r.left$
21:         end if
22:         $\eta_l = \eta_l.l$
23:     end while
24:     $\eta.left = \eta_r$    ▷ swap left and right branch for the next
        recursive call
25:     $\eta.right = \eta_l$
26:     POPULATEIPL($\eta, \lambda$)
27: end procedure
```

3.3 Mapping Divergence/Convergence between Replicas using Version Trees

An OpLog implicitly records a linear version tree. During a merge step, the reader scans the OpLogs of both the source and the reader from the top and skips over all the common elements until it finds a mismatch. Note that although this can be optimized so that the reader does not have to scan the logs from the top, we leave it out from our discussion as it is not the primary focus of our current exposition. Once it finds a mismatch, this indicates the occurrence of *concurrent* operations, i.e., two operations that cannot be ordered according to a happens-before relationship or causality. We can consider the paths after this point of mismatch as two branches of the version tree. In accordance with how version trees are created, we put the branch containing the greater version stamp at the top to the left of the branch containing the smaller version stamp. After that, we perform an InterleavedPreorder traversal of this intermediate tree which results in the final merged version tree.

Figure 4 illustrates how we can model the divergence between two replicas as a version tree. In column (i), X_A executes two operations $1A$ and $2A$. In column (ii), X_B (reader) performs a merge step with X_A (source). In this case, as the version tree of the reader, i.e., $VT(X_B)$ is empty, it trivially merges to that of the source. In column (iii), both X_A and X_B independently execute two operations. Note how the version stamps for these operations can be derived from local information alone. In column (iv), X_A (reader) performs a merge step with X_B (source). X_A skips over the first two elements in $VT(X_A)$ and $VT(X_B)$ as the corresponding elements are the same. However, after that $VT(X_A)$ has $3A$ and $VT(X_B)$ has $3B$. As $3B > 3A$, the branch containing $3B$ is placed to the left of the branch containing $3A$ to create the intermediate tree $VT'(X_A)$. Finally, an InterleavedPreorder "flattening" of $VT'(X_A)$ results in $VT(X_A)$. In column (v), X_B (reader) performs a merge step with X_A (source). All the current elements in $VT(X_B)$ are present in the same order from the top in $VT(X_A)$. Therefore, X_B skips over all these elements and simply adds the rest of the elements from $VT(X_A)$ to the tail of $VT(X_B)$. At this point, both X_A and X_B have observed the same set of operations (order of observance was different) and have incorporated all these operations in the same order (the final order in OpLogs is the same), i.e., the system is in a consistent state. Note that even if the order of merge steps was altered in columns (iv) and (v), we arrive at the same final state as shown in column (v).

Although at first sight, it seems we could have used a simple preorder traversal instead of the complex InterleavedPreorder traversal, this is not the general case. As an exception, we consider the version tree in Figure 3. A possible sequence of actions that lead to the initial tree of Figure 3 is: (i) X_A executes $1A, 2A$. (ii) X_B and X_C (both separately as the reader) merges with X_A (source). (iii) X_A, X_B and X_C execute $3A, 4A$; $3B, 4B$; and $3C, 4C$ respectively. (iv) X_B (reader) merges with X_A (source). (v) X_A (reader) merges with X_C (source). (vi) Finally, X_A (reader) attempts to merge with X_B (source). If we perform a simple preorder traversal, this results in duplicated operations for $3A$ and $4A$. Hence, we need InterleavedPreorder traversal.

3.4 Convergence among More than Two Replicas

In a distributed system with more than two replicas, it might appear that we can end up with a version tree with more than two branches. However, from our discussion in Section 3, we know that a merge step is performed among two replicas at a time. Hence, while we can represent the state of multiple replicas using a single version tree with two or more branches, we can also represent incrementally updated states as a sequence of version trees having at most two

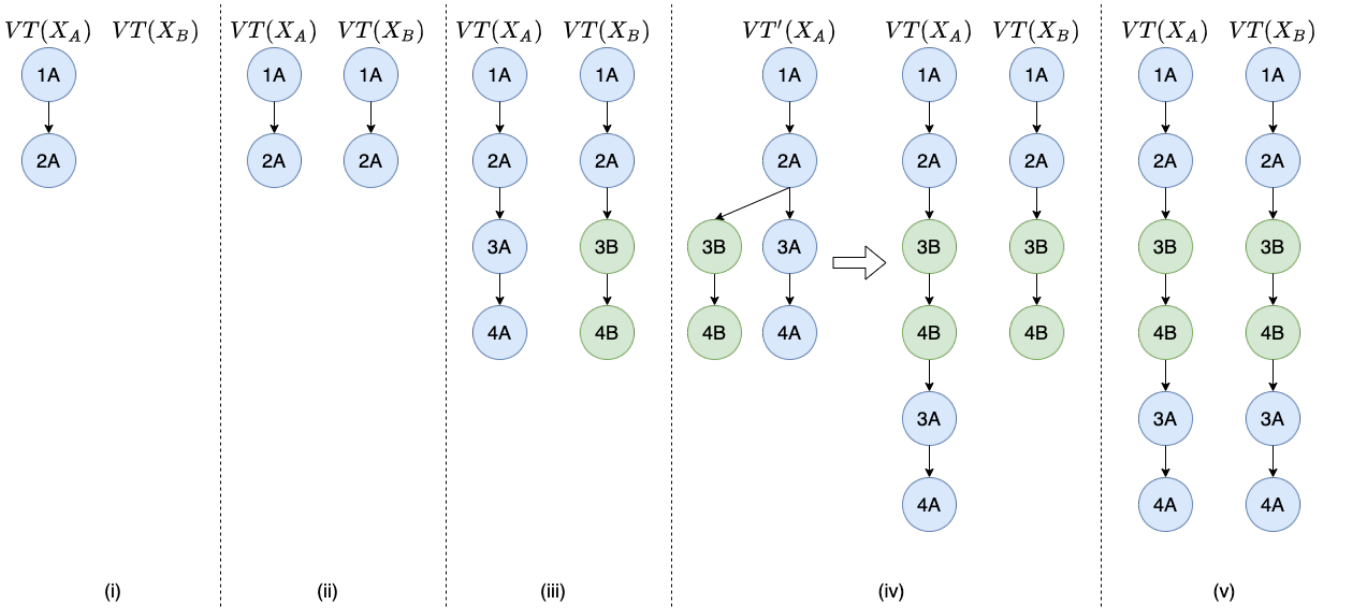

Figure 4. Merging version trees. (i) X_A executes operations. (ii) X_B merges with X_A. (iii) X_A and X_B executes operations concurrently but independently. (iv) X_A merges with X_B. (v) X_B merges with X_A.

branches. In this case, the reader merges with replicas one-by-one following some strategy (random, round-robin, etc.) until it observes all the operations. At the start of each merge step, the reader creates a version tree with two branches, one branch being its own OpLog and the other being the remote (i.e. source) OpLog.

Note that the convergence does not depend on the order in which two replicas execute the merge steps. To understand how InterleavedPreorder traversal results in a consistent state of the underlying data structure once every operation has been observed by all the replicas, we note a couple of points. First, upon branching the version stamps in a version tree are ordered arbitrarily but deterministically. This follows from the construction of the version tree, where the branch containing the greater version stamp at the top is placed to the left of the branch containing the smaller version stamp at the top. Second, InterleavedPreorder traversal does not change any existing relative order of two version stamps in a version tree (and thus the OpLog), it only imposes a new order among version stamps that were not previously ordered (i.e., version stamps in two different branches). This holds as the IPL includes all the nodes in the version tree in order up to the branching point, upon which it includes unique version stamps deterministically. Hence once all the replicas have observed the same set of operations, the system achieves consistency.

3.5 Supporting Generic Data Structures

Although InterleavedPreorder traversal facilitates the consistent ordering of operations, we must rollback OpLog if unknown operations have to be placed before the existing ones (cf. Fig 1). This means that the underlying data structure also needs to rollback its states which were obtained through the application of the rolled back operations. Depending on the exact data structure, rolling back state can be complex. For example, an increment operation on a counter can be rolled back by performing a corresponding decrement operation and vice versa. However, a binary search tree insertion cannot always be rolled back by performing a corresponding deletion, because the resulting tree structure might differ from the one that would have existed had the insert not taken place at all.

One way to ensure the underlying data structure states can also be rolled back is to represent the states using append-only logs as well. While representing data types such as registers and counters using logs can be relatively simple, other data structures such as linked lists and binary search trees require complex algorithms [18]. Once a data structure is represented using logs, we must maintain a map from the operations to the range of sequence numbers of the log(s) used to represent the state of the data structure. Then during rollback of the OpLog, we can use this map to perform rollback on other log(s) as well.

As an example, let us assume that apart from appending to the $OpLog(X_s)$ for each operation, the underlying data structure needs to perform one additional append to an auxiliary log, $AuxLog(X_s)$. Then to rollback the last n operations, we must rollback the last n entries from both $OpLog(X_s)$ and $AuxLog(X_s)$. In practice, an operation might append to more than one log and different operations might append to different sets of logs [18]. Hence this information must be recorded in a map, where the key is the unique version stamp vs of the operation and the value is a list of tuples. Each tuple contains the name of an auxiliary log and the range of sequence numbers corresponding to the entries in that log appended during the execution of operation vs.

4 Replicated Growable Arrays (RGA)

As an example of a CRDT used to maintain order in a list or sequence, we review Replicated Growable Array (RGA) [17]. RGA uses the same algorithm of Causal Trees (CT) [5, 6] as shown in [2]. We also explore the similarity between the version tree approach and RGA (and thus other approaches including CT and any method based on RGA). This reveals that the principles used in these algorithms are the same ones used in version trees, which predates these approaches by more than 20 years.

RGA implements a sequence as a linked list. It supports operations $addRight(x, a)$ to add element a immediately after element x [17, 20]. Elements can be uniquely identified using timestamps which are ordered consistently with causality. If a client invokes addRight operations twice at the same place one after another, e.g., $addRight(x, a)$ followed by $addRight(x, b)$; the latter insert occurs to the left of the former and has a higher timestamp.

This is the exact same construction of IPL from version trees during a merge step as described in Section 3. To see this, let us consider the OpLog of the reader as a linked list. The OpLog of the source then implicitly provides a list of addRight operations that need to be incorporated into the OpLog of the reader. Specifically, if element v immediately follows u in the OpLog of the reader, this translates to the operation $addRight(u, v)$. If u and v are already present in the OpLog of the reader one after another, nothing is changed. This is equivalent to the case in IPL formation where the two branches have the same elements (i.e., version stamps), in which case values from only one branch is added to IPL but a pointer for the other branch is advanced to avoid duplicates. If u is the last element of OpLog of the source, v can be simply appended, e.g., transition of $VT(X_B)$ from column (iv) to (v) in Figure 4. However, if an element w already exists after u ($u \neq v$) in the OpLog of the reader, we need to order v and w based on the value of the version stamps. As we place the greater version stamp to the left of the smaller version stamp at the branching point, the InterleavedPreorder traversal places the greater version stamp to the left of the smaller one in IPL; giving us the exact same order as in RGA. Hence, the underlying algorithm of both RGA and IPL are the same.

5 Conclusion and Future Work

In this work, we showed how to order operations for generic replicated data types using version trees. The key to supporting generic data types is to represent them using append-only logs, which facilitates rollback of operations required while achieving a consistent order. We also explore the similarities between our approach and an existing approach to maintain order in sequence or list CRDT. This implies that we can potentially use versioning schemes while ordering operations, which can provide multiple benefits to the system including facilitating debugging, efficiently executing temporal query, and supporting data lineage. In the future, we plan to extend this work by creating a portfolio of versioned data types that can be represented using append-only logs.

Although we can guarantee convergence of an arbitrary data structure by ordering operations, convergence itself does not mean the user expectations on the semantics are satisfied. As an example, the consistency of a conventional set can be achieved by consistent ordering of operations. However, the consistency for the more constrained OR-set following an *add*-wins strategy might violate the expected semantics, as a *delete* can supersede a concurrent *add* if the former is ordered after the latter following the strategy described in Section 3.2. In the future, we plan to devise mechanisms to respect the expectations on the semantics of the underlying data structure as well.

Acknowledgments

This work has been supported in part by NSF awards CNS-2107101, CNS-1703560, ACI-1541215.

References

[1] Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. 2018. Delta state replicated data types. *J. Parallel and Distrib. Comput.* 111 (2018), 162–173.
[2] Hagit Attiya, Sebastian Burckhardt, Alexey Gotsman, Adam Morrison, Hongseok Yang, and Marek Zawirski. 2016. Specification and complexity of collaborative text editing. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*. 259–268.
[3] Kevin De Porre, Florian Myter, Christophe De Troyer, Christophe Scholliers, Wolfgang De Meuter, and Elisa Gonzalez Boix. 2019. Putting order in strong eventual consistency. In *IFIP International Conference on Distributed Applications and Interoperable Systems*. Springer, 36–56.
[4] J. Driscoll, N. Sarnak, D. Sleator, and R. Tarjan. 1989. Making data structures persistent. *J. Comput. Syst. Sci.* 38, 1 (1989).
[5] Victor Grishchenko. [n. d.]. Causal trees: towards real-time read-write hypertext.
[6] Victor Grishchenko. 2010. Deep hypertext with embedded revision control implemented in regular expressions. In *Proceedings of the 6th International Symposium on Wikis and Open Collaboration*. 1–10.
[7] Robert T Hood and Robert C Melville. 1980. *Real time queue operations in pure Lisp.* Technical Report. Cornell University.
[8] Martin Kleppmann, Dominic P Mulligan, Victor BF Gomes, and Alastair R Beresford. 2020. A highly-available move operation for replicated trees and distributed filesystems.
[9] Leslie Lamport. 2019. Time, clocks, and the ordering of events in a distributed system. In *Concurrency: the Works of Leslie Lamport*. 179–196.
[10] Eugene W Myers. 1983. An applicative random-access stack. *Information processing letters* 17, 5 (1983), 241–248.

[11] Brice Nédelec, Pascal Molli, Achour Mostefaoui, and Emmanuel Desmontils. 2013. LSEQ: an adaptive structure for sequences in distributed collaborative editing. In *Proceedings of the 2013 ACM symposium on Document engineering*. 37–46.

[12] Diego Ongaro and John Ousterhout. 2014. In search of an understandable consensus algorithm. In *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*. 305–319.

[13] Gérald Oster, Pascal Urso, Pascal Molli, and Abdessamad Imine. 2006. Data consistency for P2P collaborative editing. In *Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work*. 259–268.

[14] Nuno Preguiça, Carlos Baquero, and Marc Shapiro. 2019. *Conflict-Free Replicated Data Types CRDTs*. Springer International Publishing, Cham, 491–500. https://doi.org/10.1007/978-3-319-77525-8_185

[15] Nuno Preguica, Joan Manuel Marques, Marc Shapiro, and Mihai Letia. 2009. A commutative replicated data type for cooperative editing. In *2009 29th IEEE International Conference on Distributed Computing Systems*. IEEE, 395–403.

[16] Thomas Reps, Tim Teitelbaum, and Alan Demers. 1983. Incremental context-dependent analysis for language-based editors. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 5, 3 (1983), 449–477.

[17] 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.

[18] Nazmus Saquib, Chandra Krintz, and Rich Wolski. 2021. PEDaLS: Persisting Versioned Data Structures. In *2021 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 179–190.

[19] Neil Sarnak and Robert E Tarjan. 1986. Planar point location using persistent search trees. *Commun. ACM* 29, 7 (1986), 669–679.

[20] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. *A comprehensive study of convergent and commutative replicated data types*. Technical Report RR-7506. Inria.

[21] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-free replicated data types. In *Symposium on Self-Stabilizing Systems*. Springer, 386–400.

[22] Garret Frederick Swart. 1986. EFFICIENT ALGORITHMS FOR COMPUTING GEOMETRIC INTERSECTIONS (DECISION TREE, HIDDEN LINE REMOVAL, GRAPHICS, COMPLEXITY). (1986).