Achieving Strong Eventual Convergence for Real-time Collaborative Text Editing through Conflict-free Replicated Data Types

Aaron Diamond-Reivich
University of Pennsylvania
May, 2020

Abstract

This paper explores the evolving philosophy of distributed systems as an explanation of why the insistence on availability has led to important research into consistency models for distributed systems. It lays the foundation for why a tradeoff between availability and consistency exists and then explores several consistency models. It takes a deep dive into conflict free replicated data types as a tool to achieve eventual consistency. The paper explores both state-based and operation-based CRDTs, explaining the math behind each data structure, example implementations, and use cases. Once finished exploring CRDTs, the paper begins its exploration of collaborative text editors. It begins this section with an analysis of Google Docs and its implementation using operational transforms. The paper concludes with an implementation of a collaborative text editor using an operation-based CRDT.

Introduction

As distributed systems have become a part of everyday life and the backbone for important infrastructure, the desired properties of them have evolved.[1] It is no longer our desire for these systems to behave as if it were one computer – preferring the system to become unavailable than to have differences in the state of replicas. Instead, we now prefer that systems are always available.[2] The CAP Theorem, proposed by Brewer, tells us we must sacrifice strong consistency to have availability and partition tolerance.[3] As a result, the industry decided to tolerate temporary inconsistencies in the state of replicas in order to maximize availability.[4]

Eventual consistency dictates that once an operation is applied to one replica in the distributed system, it will eventually be applied to all replicas and that eventually the state of all replicas will converge. [5] Many eventual consistency models have complex conflict resolution processes which requires replicas to roll back state.[6] Strong eventual consistency is a subset of eventual consistency with the additional specification that any replicas that received the same updates have identical state.[7] In strong eventual consistent systems, replicas are immediately consistent "as soon as they have received all of the same transactions." [8]

One way to achieve strong eventual consistency is through conflict free replicated data types (CRDTs).[9] CRDT's are a specification of distributed datatypes that are designed to support the divergence of replica state, while guaranteeing that they will eventually converge by resolving conflicting updates in a deterministic manner. [10] Deterministic conflict resolution is possible due to metadata stored in the structure of the datatype.[11] There are two primary types of CRDT's, state-based (convergent) replicated data types and operation-based (commutative) replicated data types. These types differ in how they store their metadata. State-based data types encapsulate the metadata within the state itself whereas operation-based data types rely on the replication protocol. [12]

[1] Werner Vogels, "Eventually Consistent," *ACM Queue* 6, no. 6 (December 4, 2008), https://queue.acm.org/detail.cfm?id=1466448.
[2] Vogels.
[3] Seth Gilbert and Nancy Lynch, "Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services" (Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139, 2002), https://users.ece.cmu.edu/~adrian/731-sp04/readings/GL-cap.pdf. page 3.
[4] Vogels, "Eventually Consistent."
[5] Vogels.
[6] Marc Shapiro et al., "Conflict-Free Replicated Data Types," *Inria Reocquencourt* 25 (January 19, 2011), https://pages.lip6.fr/Marc.Shapiro/papers/RR-7687.pdf. page 5.
[7] Shapiro et al. page 5.
[8] Shapiro et al. page 5.
[9] Marc Shapiro et al., "A Comprehensive Study of Convergent and Commutative Replicated Data Types," *Hal-Inria*, January 13, 2011, https://doi.org/inria-00555588.
[10] Shapiro et al.
[11] Bartosz Sypytkowski, "An Introduction to State-Based CRDTs," *Bartosz Sypytkowski - Software Dev Blog* (blog), December 18, 2017, https://bartoszsypytkowski.com/the-state-of-a-state-based-crdts/.
[12] Sypytkowski.

Strong eventual convergence is a property applied to real-time collaborative editing systems.[13] Clients update their state locally to ensure an immediate response to the user's input, and once other clients receive that input, the state of the receiving client should be updated as well.[14] Google Docs is implemented using operational transforms, a system of adapting the received operation to ensure that the intent of the operation is not affected during its application due to concurrent updates to the state by other clients.[15] In theory, operational transforms are intuitive, but in practice they are extremely difficult to implement.[16]

This paper proposes an operation based CRDT implementation of a real-time collaborative text editor.

An Outdated Philosophy of Distributed Systems

Before we begin our analysis of Conflict-free replicable data types, let's start with a discussion of distributed systems. Formally, "A distributed system is a composition of a set of processes/participants invoking methods on shared objects (registers, queues, etc.). An object implements a programming interface (API) defined by a set of methods, M, with input and output from a data domain D." [17] IBM's 1979 *Note on Distributed Databases*, puts it more simply – a distributed database is a database with "multiple sites each of which stores data. These sites communicate over a slow, unreliable communication network. Such a network can lose messages, duplicate messages, and deliver messages out of order."[18] Fundamentally, replicating data in multiple locations across the network is done to maximize data availability. Consider the following example presented in the IBM paper, if each datastore is available with a probability p, then if each piece of data exists in only one location, each piece of data is accessible with probability p. If, on the other hand, each piece of data is replicated N times, then each piece of data is available with probability $1-(1-p)^N$. If we assume that $p = .95$ and $N = 4$ then the result of replicating the data changes the probability of availability from .95 to 0.00000625. Although .95 is still a relatively high probability of uptime, in large scale distributed systems that handle trillions of transactions (think AWS), improbable events like server downtime become almost guaranteed.[19] As a result, systems are designed with

[13] Google, "What's Different about the New Google Docs: Working Together, Even Apart," *Google Drive Blog* (blog), September 21, 2010, https://drive.googleblog.com/2010/09/whats-different-about-new-google-docs_21.html.
[14] Anton Zagorskii, "Operational Transformations as an Algorithm for Automatic Conflict Resolution," *Coinmonks* (blog), n.d., https://medium.com/coinmonks/operational-transformations-as-an-algorithm-for-automatic-conflict-resolution-3bf8920ea447.
[15] Google, "What's Different about the New Google Docs: Conflict Resolution," *Google Drive Blog* (blog), September 22, 2010, https://drive.googleblog.com/2010/09/whats-different-about-new-google-docs_22.html.
[16] Zagorskii, "Operational Transformations as an Algorithm for Automatic Conflict Resolution."
[17] Gaoang Liu and Xiuying Liu, "The Complexity of Weak Consitency," *Zenodo*, January 29, 2018, https://doi.org/10.5281/zenodo.1161960 page 3.
[18] Bruce Lindsay et al., "Note on Distributed Databases" (IBM Research Laboratory San Jose, California 95193: IBM, July 14, 1979), https://domino.research.ibm.com/library/cyberdig.nsf/papers/A776EC17FC2FCE73852579F100578964/$File/RJ2571.pdf. page 1
[19] Vogels, "Eventually Consistent."

replication of datastores to guarantee consistent availability.[20] This solution helps systems maintain their availability in exchange for creating complexity in making sure that data is consistent across the replicas.[21]

Ideally, the consistency model would say that when one update is made to one replica, that update is automatically reflected in real time on all other replicas.[22] Of course, it is not possible for an update to automatically update every replica without communication between replicas. [23] However, there is a consistency model that mimics this desired behavior, the unanimous agreement update strategy.[24] This strategy dictates that unless every replica accepts the update, the update is rejected. Thus, with a replica availability probability of p and N replicas, each update is only accepted with probability p^N. Again, if $p = .95$ and $N = 4$, each update is only accepted about 81% of the time.[25] Note, that as the number of replicas grows large, the probability of a successful write operation goes to 0. So, unless the database is used dramatically more for reading than writing, and data consistency is of the absolute most importance, the unanimous agreement update strategy prevents write transactions too frequently to be a suitable solution.[26]

The consistency model discussed above and written about by IBM in 1979, strive for distribution transparency – to the user, the distributed system behaves like it is one computer instead of a network of databases working together.[27] They believed in the philosophy that it was better if transactions failed than break the façade of distribution transparency.[28]

The CAP Theorem – formalizing tradeoffs

In 2002, researchers from MIT formalized the CAP theorem, which states, "it is impossible for a web service to provide the following three guarantees:" "consistency", "availability", and "partition-tolerance."[29] Consistency is the guarantee that there exists some ordering of all operations such that it appears as if each operation occurred at one singular instant. You can think of this as making the execution in a distributed environment look as if it were on a singular node.[30] Availability says that any request that reaches a node that does not fail must eventually terminate with some response.[31] And finally, a partition is the a division of the nodes in a network such that there are no successful communications between nodes in different

[20] Vogels.
[21] Vogels.
[22] Vogels.
[23] Vogels.
[24] Lindsay et al., "Note on Distributed Databases." page 2
[25] Lindsay et al. page 2
[26] Lindsay et al. page 2
[27] Vogels, "Eventually Consistent."
[28] Vogels.
[29] Gilbert and Lynch, "Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services." page 1.
[30] Gilbert and Lynch page 3.
[31] Gilbert and Lynch page 3.

partitions. Thus, partition-tolerance states that consistency and availability still occur even if the network is partitioned. [32]

Let us discuss the high-level impossibility proof that distributed databases cannot have strong consistency, availability and partition-tolerance. We will break this proof up into two claims.

First, in a distributed system it is impossible for a data object that supports both read and write transactions in an environment where messages may be lost to have availability and consistency.[33] The basis of the proof follows: assume a network contains two nodes n_1, n_2. Create a partition of the network such that n_1 and n_2 can no longer communicate with each other. Let function f_1 write data to n_1. Later, let f_2 read from n_2. The value returned from f_1 and $f_2$will be the same. Thus, this system is not consistent. [34]

Second, in a distributed system it is also impossible for r/w data object to be available in all executions and consistent in all executions in which no data is lost. Let us again discuss the high-level proof. First, note that the algorithm cannot determine if a message is lost or if its transmission through the network is facing some arbitrary delay. Thus, if the algorithm guarantees atomic consistency for all transactions in which no messages are lost, it must also guarantee atomic consistency for all transactions. However, we just showed that a network which might lose transactions cannot guarantee availability and consistency. Therefore, the network cannot guarantee availability in all transactions and atomic consistency only for transactions that are not lost.[35]

Shifting Philosophies

Let us remember that up until the mid-1990's the standard belief was that distributed systems should aim for distribution transparency, that is, it is better to fail than break consistency. [36]

However, as the internet grew and distributed systems became an increasingly popular and important tool to everyday life – sites like UseNet, a messaging board to exchange information on threaded topics [37] – , the idea of systems being unavailable became increasingly less tolerable. [38] As a result, the industry's mindset began to shift from one prioritizing consistency to one prioritizing availability. [39]

The CAP Theorem helped researchers understand the tradeoffs that could be made to maintain availability. It proved that when developing a distributed system, developers could only pull

[32] Gilbert and Lynch pages 3-4.
[33] Gilbert and Lynch. page 5.
[34] Gilbert and Lynch page 5.
[35] Gilbert and Lynch page 5.
[36] Lindsay et al., "Note on Distributed Databases." page 2
[37] "Usenet.Com," n.d., https://www.usenet.com/what-is-usenet/.
[38] Vogels, "Eventually Consistent."
[39] Vogels.

two of the following three properties out of their toolkit: system availability, strong data consistency, and partition tolerance. [40][41] However, "in large-scale distributed systems, network partitions are inevitable" so developers need to build systems that are partition tolerant. [42] If the thinking of the time is to prioritize availability, then it must be at the expense of consistency.[43][44]

Eventual Consistency

Because we know that partitions will occur, the CAP theorem dictates that because we have chosen to prioritize availability, we must settle for weak consistency. [45]

Let's informally define weak data consistency in the following way: "The system does not guarantee that subsequent accesses will return the updated value. A number of conditions need to be met before the value with be returned." [46] Those conditions are determined by the specific implementation of weak consistency. [47]

One such form of weak consistency is eventual consistency. Shapiro, et al. state that eventual consistency is combination of three properties.

- Eventual Delivery: An update that reaches one replica will eventually reach all replicas.
- Convergence: The state of any two replicas that have seen the same set of updates will eventually be the same.
- Termination: All transactions terminate. [48]

Eventual convergence guarantees that once no new updates are processed by a distributed system, eventually any query to any replica in the system will return the same value.[49]

Consider the following example inspired by Hackernoon.

As I am writing this paper, I want to take precautions to make sure that even if my laptop breaks, I will not lose my paper. To do this, I have bought a backup external hard drive and am also syncing my paper to Dropbox. With this setup, I can access my data in a few ways.

[40] Vogels.
[41] Gilbert and Lynch, "Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services." page 3
[42] Ariel Tseitlin, "The Antifragile Organization," *ACM Queue* 56, no. 8 (August 2013), https://doi.org/doi:10.1145/2492007.2492022 page 1.
[43] Vogels, "Eventually Consistent."
[44] Gilbert and Lynch, "Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services." page 3
[45] Gilbert and Lynch page 3.
[46] Vogels, "Eventually Consistent."
[47] Liu and Liu, "The Complexity of Weak Consitency. page 3"
[48] Marc Shapiro et al., "Conflict-Free Replicated Data Types," *Inria Reocquencourt* 25 (January 19, 2011) page 5.
[49] Vogels, "Eventually Consistent."

Scenario 1. Dropbox automatically syncs my paper to the Dropbox server every time I am connected to the internet and I manually back up my paper to my external hard drive every 20 days. If I want a friend to edit my paper in the middle of one of my twenty-day cycles, I hand them my hard drive even though it might contain a version of my paper that is not the most up to date. This allows my friend to get immediate access to my paper at the expense of having a slightly stale version. This is an eventually consistent model because I know that by the end of the next 20-day cycle, my data will once again be consistent across all three replicas.

Scenario 2. I use the same cadence for backing up my paper. On the twentieth day of my back-up cycle, I am editing my paper in a park and bring my hard drive with me. As I am uploading my newest version of the paper to my hard drive, I run into a friend, Jake. Jake is interested in reading my paper, so I share with him a link to my Dropbox paper. But because I have been making edits to my paper while in the park and not connected to wifi, I tell Jake to only access the link in an hour after I am able to return home, reconnect to wifi, and update the version of the paper stored on Dropbox. This strong consistent model allows Jake to have the most up to date version of my paper at the expense of immediate access to it. [50]

We can summarize the above example in the following sentences. In an eventually consistent model, data is easily accessed, but it may be stale. In a strong consistency model, data access may be delayed, but it will always be up to date.[51]

Strong Eventual Consistency

Many eventually consistent systems execute updates immediately upon receipt. This, however, creates the possibility that a future update conflicts with an update previously processed by a replica. In order to eventually achieve data consistency across replicas, each replica must arbitrate these discrepancies in the same manner using some consensus mechanism.[52] This arbitration process sometimes requires rolling back updates. It ends up being a large waste of resources that we would like to avoid. [53]

Enter strong eventual consistency. Strong eventual consistency is eventually consistent and also strong convergent. Strong convergence says that two replicas have the same state "as soon as they have received the same updates."[54] Instead of replicas which have seen the same updates being consistent *eventually,* they are now consistent *immediately.* [55]

[50] Saurabh V, "Eventual vs Strong Consitency in Distributed Databases," Hackernoon, July 16, 2017, https://hackernoon.com/eventual-vs-strong-consistency-in-distributed-databases-282fdad37cf7.
[51] V.
[52] Shapiro et al., "Conflict-Free Replicated Data Types." page 5
[53] Shapiro et al. page 5
[54] Shapiro et al., "Conflict-Free Replicated Data Types." page 5
[55] Vogels, "Eventually Consistent."

Achieving Strong Eventual Consistency Through Conflict-free Replicated Data Types

A distributed system implemented with conflict-free replicated data type (CRDT) ensures that each node will converge to the same state regardless of the order in which they receive updates.[56][57][58] The conflict-free nomenclature is a nod to strong eventual consistency, which it achieves through "deterministic, automatic conflict resolution."[59] It's not that conflicts never occur, it's that the replica can deterministically resolve the conflict without external information and every replica will resolve the conflict in the same way.[60][61] Deterministic conflict resolution is possible due to metadata stored in the structure of the datatype. There are two categories of CRDTs which differ in how they store this extra metadata.[62] State-based CRDTs encapsulate this metadata as part of the data structure itself.[63] In this model, "state is changed locally and shipped and merged into other replicas."[64] Whereas operation-based data types rely more heavily on the replication protocol to transmit the extra information.[65] The fundamental difference is how an update to one replica is shared with others – is it incorporated into the replica's state and merged into the state of other replicas, or is it sent as an update transaction that each replica individually applies to its own state. [66]

As all good things do, this ability comes with a tradeoff – CRDT's can only service "simple, locally verifiable invariants." [67]

State-based or Convergent Replicated Data Type (CvRDT)

Baquero et al. eloquently describe how state-based CRDT's guarantee eventual convergence. They state, "In a state-based design, an operation is only executed on the local replica state. A replica propagates its local changes to other replicas through shipping its entire state. A received state is incorporated with the local state via a merge function that, deterministically, reconciles the merged states."[68] Simply put, state-based CRDTs ensure convergence by merging

[56] Shapiro et al., "A Comprehensive Study of Convergent and Commutative Replicated Data Types."
[57] Nitin Savant, Elise Olivares, and Sun-Li Beatteay, "Conclave - A Private and Secure Real-Time Collaborative Text Editor," Conclave, accessed May 12, 2020, https://conclave-team.github.io/conclave-site/.
[58] Paulo Almeida, Ali Shoker, and Carlos Baquero, "Delta State Replicated Data Types," *HASLab/INSEC TEC and Universidade Do Minho, Portugal*, March 4, 2016, https://arxiv.org/pdf/1603.01529.pdf page 4.
[59] Sypytkowski, "An Introduction to State-Based CRDTs."
[60] Sypytkowski.
[61] Murat Demirbas, "Conflict-Free Replicated Data Types," Blog, *Metadata* (blog), April 23, 2013, https://muratbuffalo.blogspot.com/2013/04/conflict-free-replicated-data-types.html.
[62] Sypytkowski, "An Introduction to State-Based CRDTs."
[63] Sypytkowski.
[64] Carlos Baquero, Almeida Paulo, and Ali Shoker, "Pure Operation-Based Replicated Data Types," *Cornell University*, October 13, 2017, https://doi.org/arXiv:1710.04469. page 1.
[65] Sypytkowski, "An Introduction to State-Based CRDTs."
[66] Baquero, Paulo, and Shoker. page 1.
[67] Demirbas, "Conflict-Free Replicated Data Types."
[68] Baquero, Paulo, and Shoker. page 1.

the states of replicas together. We will build intuition for why this works by exploring definitions presented by Marc Shapiro, the father of CRDTs. [69][70][71]

Definition 2.1 – Causal History: Given some state-based CRDT distributed system, R, composed of replicas r_i, for any r_i, the causal history, $\boldsymbol{C}$: [72]

- Initially $\boldsymbol{C}(r_i) = \emptyset$
- After executing an update, u, $\boldsymbol{C}(u(r_i)) = C(r_i)\ U\ \boldsymbol{C}\{u\}$
- After executing a merge between replicas r_i and r_j, written, $\boldsymbol{M}(r_i, r_j)$, $\boldsymbol{C}\left(\boldsymbol{M}(r_i, r_j)\right) = C(r_i)\ U\ C(r_j)$

For some event, *e,* the causal history, $\boldsymbol{C}(e)$, is a set of events which includes all of the events which causally preceded (aka: may have affected) *e*.

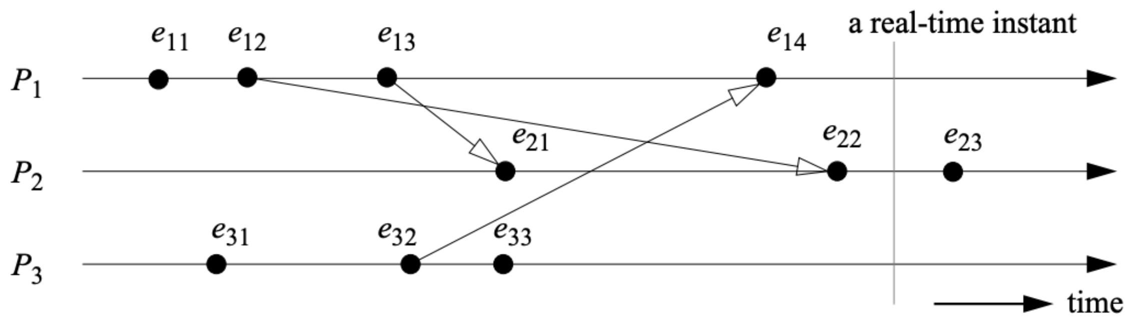


Figure 1. A time diagram of events across three replicas in a distributed system. source: Detecting Causal Relationships in Distributed Computations by Schwarz and Mattern

Figure 1 depicts a distributed system of three replicas where events are depicted as dots and messages between replicas are depicted as arrows. By applying the definition of causal history, we know that an event *e* can only be in the causal history of e′if there is a directed path from e to e'. For example, event e_{11} may affect local events e_{12}, e_{13} and remote events e_{21}, e_{22}, and e_{23}. However, e_{12} has no effect on e_{11} or e_{33}.[73]

We will use this formalization of causal history to reason about the convergence of a state-based CRDT.

[69] Sypytkowski, "An Introduction to State-Based CRDTs."
[70] Almeida, Shoker, and Baquero, "Delta State Replicated Data Types." page 4
[71] Shapiro et al., "A Comprehensive Study of Convergent and Commutative Replicated Data Types." page 7"
[72] Shapiro et al. page 7
[73] Reinhard Schwarz and Friedemann Mattern, "Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail" (Germany: University of Kaiserslautern, University of Saarland, n.d.), https://www.vs.inf.ethz.ch/publ/papers/holygrail.pdf. page 3

Definition 2.2 – Eventual Convergence: For any two replicas r_i and r_j of a distributed system R, r_i and r_j eventually converge when the following are satisfied: [74]

- Safety: $\forall i, j$: $\boldsymbol{C}(\boldsymbol{r_i}) = \boldsymbol{C}(\boldsymbol{r_j})$
- Liveness: $\forall i, j$: if $\boldsymbol{u} \in \boldsymbol{C}(\boldsymbol{r_i})$, then eventually $u \in \boldsymbol{C}(\boldsymbol{r_j})$

In practice, we can think of eventual convergence as query convergence. That for any query, **q**, $\forall i, j$: $\boldsymbol{q}(\boldsymbol{r_i}) = \boldsymbol{q}(\boldsymbol{r_j})$

This pairwise definition of eventual convergence allows us to conclude that because any arbitrary two replicas converge, it must be the case the all replicas in R converge.[75]

We will again turn to Shapiro for our definition of a Join Semilattice and its associated Least Upper Bound relationship.

Definition 2.3 – Least Upper Bound: $m = x \sqcup_v y$ is a Least Upper Bound of x, y under the partial order $\leq_v$ if and only if $x \leq_v m$ and $y \leq_v m$ and there is no $m' \leq_v \mathrm{m}$ such that if $x \leq_v m'$ and $y \leq_v m'$. [76]

Definition 2.4 – Join Semilattice: An ordered set (S, $\leq_v$) is a Join Semilattice if and only if $\forall x, y \in S,\ x \leq_v y$ exists. [77]

With these definitions in hand, let's formalize CvRDTs.

Definition 2.5 – State-Based or Convergent Conflict Free Replicated Data Types (CvRDT): A CvRDT is a distributed data structure composed of 1) local state and algorithms 2) an anti-entropy protocol.[78][79]

The local state and algorithms are: [80]

- *S,* a join semilattice
- *M,* a set of mutator operations that receive a state, $x \in S$, and return an updated state $x' = m(x)$ where $m \in M$ is an inflator such that $x \leq_v m(x)$
- *Q,* a set of read operations which return data by querying the state without altering it.

The anti-entropy algorithm is run by each of the replicas. When run by replica r_i, it:

- Sends the state of r_i to other replicas

---

[74] Shapiro et al. page 9.
[75] Shapiro et al. page 9
[76] Shapiro et al. page 10
[77] Shapiro et al. page 11
[78] Sypytkowski, "An Introduction to State-Based CRDTs."
[79] Almeida, Shoker, and Baquero, "Delta State Replicated Data Types." page 4
[80] Almeida, Shoker, and Baquero. page 4

- Receives the state of other replicas and performs a merge operation to merge the received state into its own. The merge operation is commutative, associative, and idempotent. [81]

Because query and mutator operations are performed on the local state of the replica and are executed without communication between replicas, concurrent mutations can cause replicas to diverge. [82] Convergence is eventually achieved through the anti-entropy algorithm, which, with our liveness assumptions, ensures that any causal event to the state of one replica is in the causal history of all replicas.

Before we prove that CvRDT's converge, let us walk through a State based CvRDT grow-only set with the following specification: [83]

```
Class GrowOnlySetReplica:

 /// The Set of values stored by this replica
 Set{} V;

 /// An Add element mutator which adds the element e to V
 Mutators:
   Add(element e): V <- V U {e}

 /// A Lookup query which returns true if e is in V
 Query:
   Lookup(element e): returns e ∈ V

 /// The Anti-Entropy Algorithms
 Anti-Entropy:

   /// Merges the state of a different replica into V
   Merge(ReplicaState V'):  V <- V U V'

   /// Sends V to another replica for merging
   SendState(Replica r): r.Merge(this.V)

```

Intuitively, it makes sense that a grow only set can be implemented as a CRDT. Because grow only sets are not ordered, the sequence of updates to the set has no effect on the final state of the set. Therefore, as long as each replica sees each update, the sets will eventually have the same state. Let us map this CRDT schema to our previously developed definition. The GrowOnlySetReplica above has local state, the Set V. It also has mutator and query functions, Add and Lookup. Finally, it has an anti-entropy system that allows the replicas to converge. The GrowOnlySetReplica uses a Merge function which takes another replica's state as an argument

[81] Sypytkowski, "An Introduction to State-Based CRDTs."
[82] Almeida, Shoker, and Baquero. page 4
[83] Shapiro et al., "A Comprehensive Study of Convergent and Commutative Replicated Data Types." page 22

and updates the current replica's state to be the union of the current replica's state and the argument state. It also has a SendState function which calls the Merge function on a different replica. You can see my implementation of this simple state-based CRDT here. Although this is a very trivial example, CRDT's get much more complicated.

Let's formally prove that CvRDT's converge. As Shapiro et al. do, we will prove the following claim:

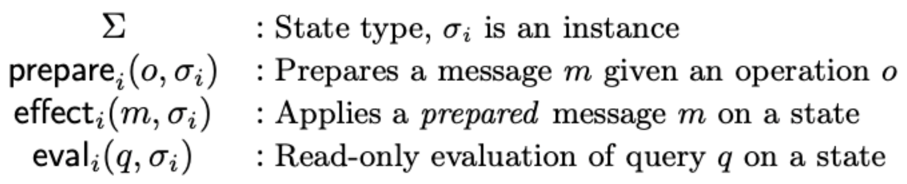
Proposition 1: If the network continues to run its anti-entropy protocol, then any two replicas of a CvRDT eventually converge.[84]

Consider any two replicas, r_1 and r_2. Given our liveness assumption, they will exchange states at some point either by exchanging them directly with each other or exchanging them indirectly, using other replicas as intermediaries. Because the CvRDT's state forms a monotonic semilattice, it is always possible for the replicas to merge states. Therefore, by the definition 2.1 of causal history, after r_1 merges the state of r_2 and r_2 merges the state of r_1, r_1 and r_2 will have the same causal history. Therefore, by the commutativity of the least upper bound, both r_1 and r_2 will have the same state. Therefore, if we refer back to definition 2.2 of eventual convergence, because we have satisfied both necessary properties, we have proven that this CvRDT will eventually converge. [85]

Operation-based or Commutative Replicated Data Type (CmRDT)

Operation-based replicated data types are another type of CRDT. Again, let us turn to the words of Baquero et al. to begin our analysis of operation-based replicated data types. They state, "In op-based designs, the execution of an operation is done in two phases: prepare and effect. The former is performed only on the local replica and looks at the operation and current state to produce a message that aims to represent the operation, which is then shipped to all replicas. Once received, the representation of the operation is applied remotely using effect. Different replicas are guaranteed to converge [as long as all messages are eventually propagated and received by all other replicas] and effect is designed to be commutative for concurrent operations." [86]

[84] Shapiro et al. page 10.
[85] Shapiro et al. page 10.
[86] Baquero, Paulo, and Shoker, "Pure Operation-Based Replicated Data Types." page 1



Σ	: State type, σ_i is an instance
$\mathsf{prepare}_i(o, \sigma_i)$	: Prepares a message m given an operation o
$\mathsf{effect}_i(m, \sigma_i)$	: Applies a *prepared* message m on a state
$\mathsf{eval}_i(q, \sigma_i)$	: Read-only evaluation of query q on a state

Figure 2: The general scheme of an operation-based conflict free replicated data type. Source: Pure Operation-Based Replicated Data Types by Baquero et al.

Operation-based conflict free replicated data types have four key components, its state, and three functions – prepare, effect, and evaluate.[87] The prepare function packages an operation with the appropriate metadata for it to be properly executed by another replica.[88] The effect function applies a prepared message to a specific state.[89] And lastly, the evaluation operation takes as arguments a specific query and state and returns the result of running the query on the given the state. [90]

Let us again formalize.

Definition 3.1 – happened-before: The happened-before relationship, denoted →, orders two transactions, f and g, where f → g *iff* for all replicas r_i: $g \in \boldsymbol{C}(r_i) \implies f \in \boldsymbol{C}(r_i)$. [91]

Definition 3.2 – Concurrent Operation: If operations, f and g, are not ordered by the happened-before relation, then they are concurrent. Symbolically, $f \,||\, g \iff f$ not → g and g not → *f.* [92]

Definition 3.3 – Commutative Data Types: A concurrent data type is commutative *iff:*

- For any operations f and g, $f(g(\sigma)) = g(f(\sigma))$
- All concurrent operations can be equivalently processed as some linear application of the operations [93]

In plain English, operations commute if the order in which they are executed does not matter – all sequences of application will have the same result. [94]

[87] Baquero, Paulo, and Shoker. page 4.
[88] Savant, Olivares, and Beatteay, "Conclave - A Private and Secure Real-Time Collaborative Text Editor."
[89] Savant, Olivares, and Beatteay.
[90] Baquero, Paulo, and Shoker. page 4.
[91] Marc Shapiro et al., "The Distributed Computing Column - Convergent and Commutative Replicated Data Types," *Bulletin of the EATCS; European Association for Theoretical Computer Science*, no. 104 (n.d.), file:///Users/aarondiamond-reivich/Downloads/120-477-1-PB.pdf. page 72.
[92] Shapiro et al page 72.
[93] Baquero, Paulo, and Shoker. page 6.
[94] Savant, Olivares, and Beatteay, "Conclave - A Private and Secure Real-Time Collaborative Text Editor."

Definition 3.4 – Operation-Based or Commutative Replicated Data Types (CmRDT): CmRDT is a distributed data structure composed of 1) local state and algorithms 2) an anti-entropy protocol. [95]

The local state and algorithms are:

- S, the state of the client
- *M,* a set of mutator operations that receive a state, $x \in S$, and return an updated state $x' = m(x)$ and also prepares the operation to be executes by other clients
- *Q,* a set of read operations which return data by querying the state without altering it.

The anti-entropy protocol is the *effect* function which takes as an argument prepared messages and a replica. *Effect(m,* r_j*),* called by r_i, applies the prepared messages to r_j. [96]

Let's take a look at an example operation-based increment-only counter.

$$\begin{aligned} \Sigma = \mathbb{N} & \qquad \sigma_i^0 = 0 \\ \mathsf{prepare}_i(\mathsf{inc}, n) &= \mathsf{inc} \\ \mathsf{effect}_i(\mathsf{inc}, n) &= n + 1 \\ \mathsf{eval}_i(\mathsf{value}, n) &= n \end{aligned}$$

Figure 3: An op-based increment-only counter. source: Pure Operation-Based Replicated Data Types by Bauero et al.

The above op-based conflict free replicated data type is an increment only counter. [97] The instantiation above starts all replicas with a state value of zero. Recall from above that the prepare function takes as its first argument an operation and takes a state as its second argument.[98] The prepare function creates a message, in this case just the increment command. [99] When the message is received by the execution of the effect function, it increments the state of the replica. [100] And finally, the evaluate operation only takes in a single query, value, which returns the value of the specified state. [101]

You can see my implementation of a simple grow-only set operation-based CRDT here.

[95] Baquero, Paulo, and Shoker. page 2.
[96] Baquero, Paulo, and Shoker. page 2.
[97] Baquero, Paulo, and Shoker. page 6.
[98] Baquero, Paulo, and Shoker. page 4.
[99] Baquero, Paulo, and Shoker. page 5.
[100] Baquero, Paulo, and Shoker. page 5.
[101] Baquero, Paulo, and Shoker. page 5.

Proposition 2: Given a CmRDT distributed system, R, comprised of replicas r_{i}, any arbitrary $r_{\mathrm{i}}, r_{\mathrm{j}}$, eventually converge assuming that each replica eventually receives each message and that they are delivered in the delivery order, $<_d$. [102]

Consider any two replicas, r_1 and r_2. With our liveness assumption, eventually both replicas will have applied all operations to its own state. Therefore, the casual history of r_1 and r_2 are equivalent. Therefore, for any two operations f and g in the causal history of r_1, $\boldsymbol{C}(r_1)$, they fall into one of three cases: "(1) they are not causally related then they are concurrent under $<_d$ and must commute; (2) if they are causally related $f \rightarrow g$ but are not ordered in delivery order $<_d$ they must also commute; (3) if they are causally related $f \rightarrow g$ and delivered in that order $f <_d g$, then they are applied in the same order everywhere." [103] All three cases lead to every replica having an equivalent state. [104]

Comparing State-based vs Operation-based CRDTS

We have proven that both CvRDTs and CmRDTs accomplish the same goal – eventual convergence. However, there are important differences between these two mechanisms.

State-based CRDTs are generally simpler to reason about because the entire state of the replica is transported and merged in one step.[105] This merging concept is one that computer scientists are familiar with – thanks Git! However, as the size of the state grows, sending the entire state of a replica becomes inefficient. [106]

Operation-based CRDTs, on the other hand, often have a smaller per transaction payload because it does not require sending the entire state each time.[107] However, op-based CRDTs are generally more difficult to reason about because they require understanding the causal change of operations and sometimes complicated prepare and effect functions. [108] See the below schema for an op-based Add-Wins Set as an example of the potential complexity of operation-based CRDTs. [109] The Add-Wins set is a normal set implementation with the updating operations add and remove, and the query operation contains. The uniqueness is that if the system receives add and remove operations with the same argument, *x*, concurrently, the Add-Wins set keeps *x* in the set.[110]

[102] Shapiro et al., "A Comprehensive Study of Convergent and Commutative Replicated Data Types." page 11
[103] Shapiro et al. page 11.
[104] Shapiro et al. page 11
[105] Shapiro et al. page 11.
[106] Shapiro et al. page 11.
[107] Shapiro et al. page 12.
[108] Shapiro et al. page 12.
[109] Baquero, Paulo, and Shoker, "Pure Operation-Based Replicated Data Types." page 6"
[110] Ranadeep Biswas, Michael Emmi, and Constantin Enea, "On the Complexity of Checking Consistency for Replicated Data Types" (University de Paris, SRI International, n.d.), https://www.irif.fr/~cenea/papers/crdts-cav19.pdf. page 5.

$$
\begin{array}{rcl}
\Sigma = \mathbb{N} \times \mathcal{P}(I \times \mathbb{N} \times V) & & \sigma_i^0 = (0, \{\}) \\
\mathsf{prepare}_i([\mathsf{add}, v], (n, s)) & = & [\mathsf{add}, v, i, n+1] \\
\mathsf{effect}_i([\mathsf{add}, v, i', n'], (n, s)) & = & (n' \textbf{ if } i = i' \textbf{ otherwise } n \\
 & & , s \cup \{(v, i', n')\}) \\
\mathsf{prepare}_i[\mathsf{remove}, v], (n, s)) & = & [\mathsf{remove}, \{(v', i', n') \in s \mid v' = v\}] \\
\mathsf{effect}_i([\mathsf{remove}, r], (n, s)) & = & (n, s \setminus r) \\
\mathsf{eval}_i(\mathsf{elems}, (n, s)) & = & \{v \mid (v, i', n') \in s\}
\end{array}
$$

Figure 4: Schema for an op-based Add-Wins Set CRDT. Source: Pure Operation-Based Replicated Data Types by Baquero et al..

Both types of CRDT's are productionized. For example, the League of Legends chat service is built using state-based CRDTs. The system supports 7.5 million concurrent players and over 11,000 messages per second.[111] Alternatively, SoundCloud uses operation-based CRDT's to propagate new events across users' timelines. [112]

Let's now shift gears to discuss collaborative editing before tying the two threads together with a discussion about how CRDT's can be used to implement collaborative editing software.

Collaborative Editing

As the name implies, a collaborative editing system allows multiple individuals to edit a file at the same time from their own computer. [113][114] In order to replicate the user experience of editing a document locally on a system like Microsoft Word, collaborative editing software requires that each client maintains a local copy of the document.[115] Therefore, the biggest issue that collaborative editing software faces is maintaining consistency between each of the clients. [116][117] Collaborative editing software must resolve conflicts that occur when two clients make conflicting changes to the same part of the text at the same time[118] with the ultimate goal that each client converges on the same (and accurate) state of the file. [119] Sound familiar?

[111] "How League of Legends Scaled Chat To 70 Million Players - It Takes Lots of Minions.," High Scalability, n.d., http://highscalability.com/blog/2014/10/13/how-league-of-legends-scaled-chat-to-70-million-players-it-t.html.
[112] Peter Bourgon, "Roshi: A CRDT System for Timestamped Events," *Backstage Blog* (blog), May 9, 2014, https://developers.soundcloud.com/blog/roshi-a-crdt-system-for-timestamped-events.
[113] Chengzheng Sun, *Issues and Experiences in Designing Real-Time Collaborative Editing Systems*, Lecture (YouTube, GoogleTechTalks, n.d.), https://www.youtube.com/watch?v=84zqbXUQIHc.
[114] Zagorskii, "Operational Transformations as an Algorithm for Automatic Conflict Resolution."
[115] Zagorskii.
[116] Google, "What's Different about the New Google Docs: Working Together, Even Apart."
[117] Zagorskii, "Operational Transformations as an Algorithm for Automatic Conflict Resolution."
[118] Zagorskii.
[119] Zagorskii.

Before we look at using CRDTs to build collaborative editing software, let's first look at an alternative implementation – operational transforms, which are used in the preeminent collaborative document editor, Google Docs. [120]

Google Docs and Operational Transforms

Let's motivate our discussion with the following example. [121] Imagine that Alice and Bob are working together to edit a document, using a server to propagate changes from one client to the other. In this example, both Alice and Bob start with a document that says, "The quick brown fox." The collaborative editing software keeps clients up to date by propagating changes between clients using a central server. Let's suppose that Alice and Bob make incompatible simultaneous changes – Alice bolds the words "brown fox" while bob replaced them with "brown dog." The server does not know how to reconcile these changes. From Bob's perspective, "The quick **brown fox** dog," "The quick **brown** dog," and "The quick **brown** dog **fox**" are all valid ways of merging these documents.[122] The software needs more information to correctly incorporate these incompatible concurrent changes.

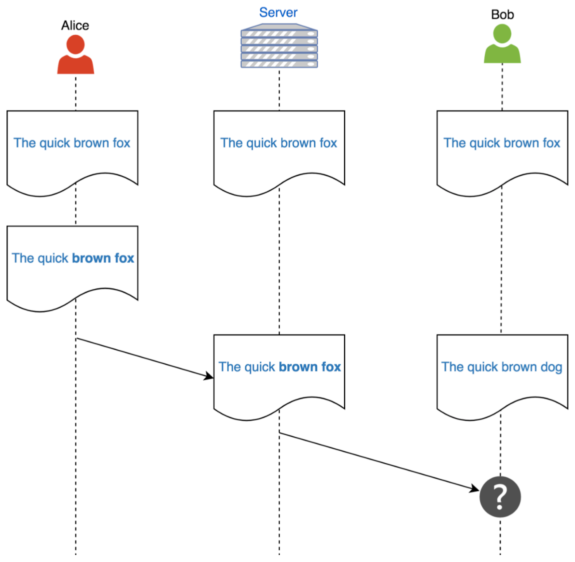


Figure 5: Display of the collaborative editing consistency problem. Source: Operational Transformations as an algorithm for automatic conflict resolution

To resolve the problem presented above, Google docs saves each document not as a document of text, but as a revision log composed of the operations insert text, delete text, and style range of text.[123] Editing the document is not editing the actual character of the Google Doc, but

[120] Google, "What's Different about the New Google Docs: Working Together, Even Apart."
[121] Zagorskii, "Operational Transformations as an Algorithm for Automatic Conflict Resolution."
[122] Google, "What's Different about the New Google Docs: Working Together, Even Apart."
[123] Google, "What's Different about the New Google Docs: Conflict Resolution."

instead, it is appending an operation onto the end of the revision log.[124] In order to display a document, a client must chronologically apply all of the revisions in the revision log.[125]

However, there is one more twist. Consider the following example presented by Google.[126] Imagine again that Alice and Bob are collaborating. This time, they start with the sentence, "Easy as 123." Again, imagine that Alice and Bob make concurrent changes – Alice changes the document to read "Easy as ABC" while Bob changes the document to read, "It's Easy as 123." Alice's edits would be represented by the following four operations: [Delete(9-11), Insert(A, 9), Insert(B, 10), Insert(C,11)]. While Bob's edits would be represented as: [Insert(I, 1), Insert(T, 2)]. If Bob applies Alice's changes as he receives them, then after applying the first operation, he will have deleted the string "S 1" because the operations that he first applied on his client shifted the indexes of the elements in the document. Bob deleted the wrong characters! [127]

Operational transforms help clients preserve the intent of each edit by applying updates in accordance with the happened-before relation and transforming operations accordingly.[128] Transformation functions take two forms.

1) Inclusion/forward transformation: The goal of inclusion transformations is for two operations, *f* happens-before *g,* to make sure that that the effect of *g* is included. For example, an inclusion transformation would occur when *f* and *g* are both insertion operations. [129]
2) Exclusion/backward transformations: The goal of exclusion transformations is for two operations, *f* happens-before *g,* to make sure that the effect of *g* is not included. For example, an exclusion transformation would occur when *f* is a deletion and *g* is an insertion. [130]

As a final example, consider Alice and Bob again. This time, they start with the document "LIFE 2018." Alice updates the sentence to read, "LIFE 2019," which is done by the following operations: [Delete(8), Insert(9,8)]. Concurrently, Bob updates the document to read, "CRAZY LIFE 2018" by applying: [Insert(C,0), Insert(R,1), Insert(A,2), Insert(Z,3), Insert(Y, 4), Insert(" ", 5)]. Now, when Bob receives Alice's operations, he transforms them locally using the information he has about the local operations that he has already applied to his client since last receiving updates from the server. Thus, Bob updates the operations Delete(8) to Delete(14) and Insert(9,8) to Insert(9,14). Now, when Bob applies the transformed operations, he computes the document, "CRAZY LIFE 2019." [131]

[124] Google.
[125] Google.
[126] Google.
[127] Google.
[128] Zagorskii, "Operational Transformations as an Algorithm for Automatic Conflict Resolution."
[129] Zagorskii.
[130] Zagorskii.
[131] Zagorskii.

The trouble with operational transforms is that they are actually quite difficult to implement because it requires so much indexes transformations.[132] If you have ever had to do a lot of index shifting, you know it is a painful, tedious, excruciating task. Take the word of Joseph Gentle, a former Google Wave engineer who wrote, "Unfortunately, implementing OT sucks. There's a million algorithms with different tradeoffs, mostly trapped in academic papers. The algorithms are really hard and time consuming to implement correctly. […] Wave took 2 years to write and if we rewrote it today, it would take almost as long to write a second time." [133]

Implementing Collaborative Editing Using CRDTs

In the operational transform model, we treat each absolute index of the document as having a specific value. We made sure that all documents converged by adjusting the indexes of each operation based on the application of concurrent operations to the document. [134] Instead, of computing these complicated transformations, we can implement a collaborative text editor using CRDTs. [135]

To my initial surprise, CRDT based collaborative text editors actually exist! We will explore the implementation of one of them - Conclave. [136]

As we previously discussed, in order to implement a collaborative text editor using CRDT's we need to keep track of some extra metadata.[137] In this case, the metadata must make each character globally unique globally ordered. [138]

Maintaining globally unique characters

Conclave ensures that characters are globally unique by assigning a Site ID to each character in the document.[139] Consider the following example adapted from Conclave which highlights the importance of globally unique characters.[140] Although in practice, Site IDs are often objects, in this case we will represent them as real numbers.[141]

[132] Zagorskii.
[133] Zagorskii.
[134] Savant, Olivares, and Beatteay, "Conclave - A Private and Secure Real-Time Collaborative Text Editor."
[135] Savant, Olivares, and Beatteay.
[136] Savant, Olivares, and Beatteay.
[137] Savant, Olivares, and Beatteay.
[138] Savant, Olivares, and Beatteay.
[139] Savant, Olivares, and Beatteay.
[140] Savant, Olivares, and Beatteay.
[141] Savant, Olivares, and Beatteay.

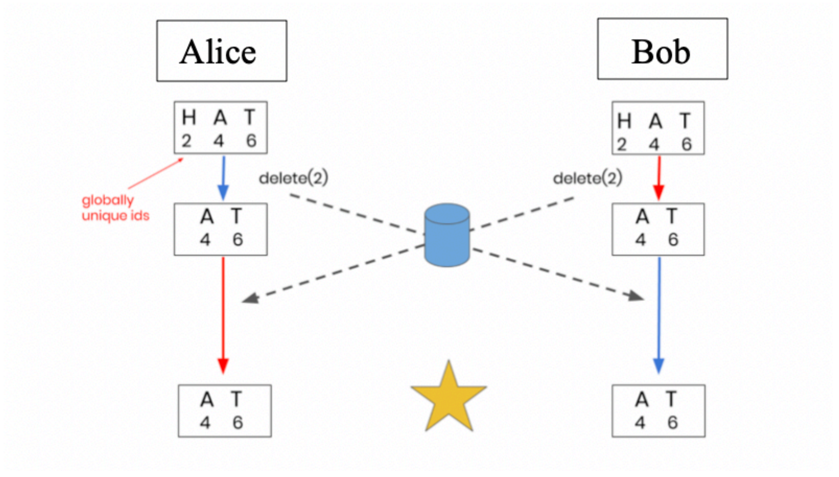


Figure 6: This figure highlights the importance of globally unique characters by illustrating how Conclave handles a simultaneous delete of the same character. Source: Conclave Case Study

In the example above, we again have Alice and Bob. They start with the document, “HAT” where each letter has its own unique identifier. (“H”, 2), (“A”, 4), (“T”, 6). Simultaneously, Alice and Bob delete the globally unique character, 2, leaving both Alice and Bob with the new document (“A”, 4), (“T”, 6). When Alice and Bob receive the operation, delete(2) from the other client, the execution of the operation discovers that no character universally identifiable as 2 exists, and thus because the operation is requesting to delete 2, the execution stops. Both Alice and Bob’s documents remain accurate, (“A”, 4), (“T”, 6). [142] If instead, received operations used index references, both Alice and Bob would have deleted “A”, shifted the index of “T” and then deleted “T” as well. It’s a problem that we already observed when discussing operational transforms.

Maintaining Globally Ordered Characters

The goal of globally ordered characters is to ensure that inserting a character on one client results in the character being inserted in the same location on all other clients.[143] Consider the following characters with their corresponding global order (“C”, 0), (“A”, 1), (“T”, 2). If we try to update “CAT” to “CHAT,” we will find ourselves in the unfortunate position of realizing that no integers fall between 0 and 1.[144] This poses a challenge to our globally ordered characters schema. Instead, Conclave uses fractional indices implemented as a list of integers. For example, the index 2.5 would be represented as [2, 5].[145] Fractional indices allow us to always provide newly inserted elements with a unique position identifier without needing to shift the location of any other elements.[146] Now consider updating the word “CHAT” to “CHEAT”. We

[142] Savant, Olivares, and Beatteay.
[143] Savant, Olivares, and Beatteay.
[144] Savant, Olivares, and Beatteay.
[145] Savant, Olivares, and Beatteay.
[146] Savant, Olivares, and Beatteay.

need to give the character "E" a global position identifier. To find one, we can consider traversing the following tree. [147]

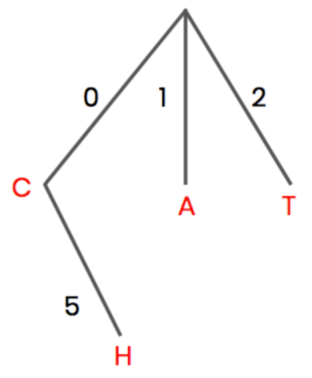


Figure 7: A tree of global position identifiers. Source: Conclave Case Study

We know that the character comes after "C", but before "A". Thus, we traverse down the "C" branch. Again, we know that "E" comes after "H", so we traverse down the "H" branch. Because there are no characters following "H" that come before "A", we can now assign a global position identifier to "E" – [0, 5, 5]. [148]

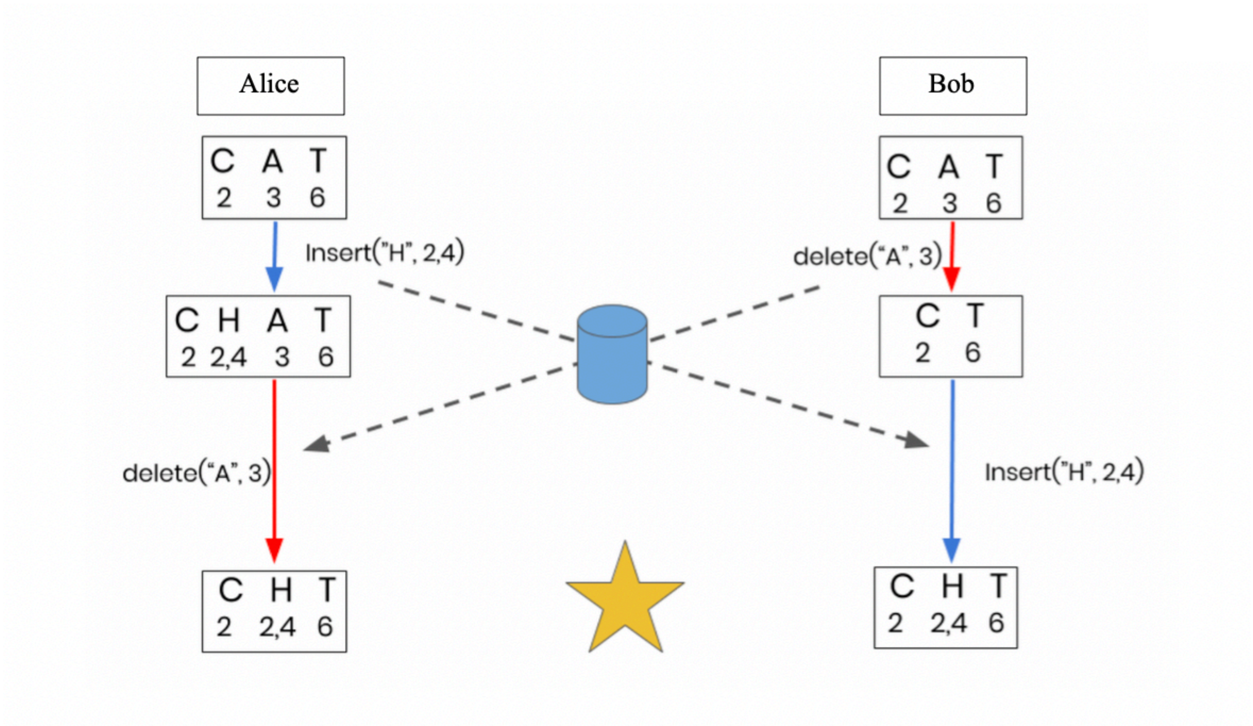


Figure 8: Inserting and Deleting with characters with globally unique identifiers to preserve correctness. Source: Conclave Case Study

[147] Savant, Olivares, and Beatteay.
[148] Savant, Olivares, and Beatteay.

Let's again turn to Conclave for an example of globally ordered characters in action. Alice and Bob again start with the document ("CAT"). Alice inserts "H" with the global order, 2.4 and Bob deletes the character "A" with the unique identifier 3. Thus, they are both left with the string "CHT." [149]

Because we use fractionally indexed positions, deleting a character does not effect inserting another character – the insert and delete operations commute.[150]

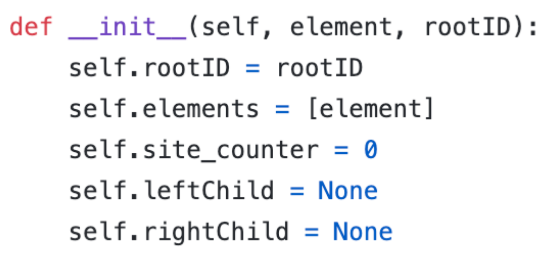
Although not included in the Conclave case study, let's understand what is going on underneath the hood. What type of CRDT is this? Referring back to definition 2.5 and 3.4 of state-based and operation-based conflict-free replicated data types, we quickly realize that Conclave has implemented its text editor using an operation-based approach. If you recall, an operation-based CRDT has prepare and effect operations. In this case, the prepare function received a command from the client in the form of a user input and transforms that command into an insert, delete, or style function which references characters by their globally unique identifiers. This preparation allows other clients to apply the update with the same effect.

Implementing a real time collaborative text editor using operation based CRDTs

You can check out the code for this section here or view it in the appendix. There are four components to this codebase: a client class implemented as an operation based CRDT, a universal identifier tree node, a set of utility functions for accessing the universal identifier tree and its interpretation as a document, and finally, code that simulates local inserts, local deletes and the execution of remote operations by the clients.

As we previously discussed, operation-based CRDT's have local state and algorithms as well as an anti-entropy protocol (see definition 2.5).

This implementation of an operation-based CRDT has uses a binary tree of universally unique identifiers to store the state of client.



```
def __init__(self, element, rootID):
    self.rootID = rootID
    self.elements = [element]
    self.site_counter = 0
    self.leftChild = None
    self.rightChild = None
```

[149] Savant, Olivares, and Beatteay.
[150] Savant, Olivares, and Beatteay.

Each tree node has a unique identifier that is a path of “0” and “1” from the root node to the node where the character is stored. A “0” is used to represent the left child and a “1” is used to represent the right child. To retrieve a document representation of the state, we perform an in-order traversal of the tree.

The client has two mutator operations: local_insert and local_delete.

The local_insert function is a class method of the client and takes as input an element to insert and the universal identifying id of the element previous to the newly inserted element.

```
# inserts the element directly after the previous_id on the client
def local_insert(self, element, previous_id):
```

The algorithm traverses the universal identifying tree representation of the document to select the appropriate universal identifying id for the newly inserted character. The algorithm then prepares this operation for remote execution by other clients. It creates a safe element by attaching the site id to the inserted character, making sure it is unique in the case that another client inserted the same element at the same location concurrently. Lastly, the operation packages the operation for remote execution by storing all of the necessary information.

```
self.unshared_operations.append(("insert", safe_element, inserted_id))
```

The local_delete function is also a class method of the client that takes as input an element to delete and the universal identifying id of the element.

```
# deletes the element from the node with id on the client
def local_delete(self, element, id):
```

Similar to the local_insert function, local_delete traverses the universal identifying tree representation of the document. Once the correct node is identified, it removes the element from
the node. It does not, however, delete the node completely in order to maintain its ability to reach any of the node’s children. Finally, the local_delete packages the operation for remote execution by storing all of the necessary information.

```
self.unshared_operations.append(("delete", element, id))
```

The last algorithm of note is the anti-entropy protocol, remote_operations, which receives a list of prepared messages from another client.

```
# execute received remote operations
def remote_operations(self, remote_operations):
```

This operation unpacks each prepared message one by one and applies the operation to its own state by executing the operations locally.

Conclusion

As distributed systems have become a more fundamental part of society, our expectations of them have evolved. Conflict free replicated data types can be used to build available systems with strong eventual convergence. Unlike traditional distributed systems, CRDT's do not require a central server, which are not always the most effective coordinators of communication between computers. Firstly, servers introduce sometimes unnecessary latency into the communication channels between computers.[151] Consider two roommates writing a paper together using a collaborative editor in the Philadelphia apartment. Although it would only take a few milliseconds for the computers to communicate directly with each other, if the server that the collaborative text editor is using is in Colorado, the round-trip communication time increases by an order of magnitude.[152] Secondly, central servers are costly to operate. And most importantly, central servers insist that users trust the server with their data. CRDT's allow for direct peer-to-peer communication between clients, allowing communities to control their own data.[153] They're a data structure that deserves ongoing research and mainstream adoption.

[151] Savant, Olivares, and Beatteay.
[152] Savant, Olivares, and Beatteay.
[153] Savant, Olivares, and Beatteay.

Works Cited

Almeida, Paulo, Ali Shoker, and Carlos Baquero. “Delta State Replicated Data Types.” *HASLab/INSEC TEC and Universidade Do Minho, Portugal*, March 4, 2016. https://arxiv.org/pdf/1603.01529.pdf.

Baquero, Carlos, Almeida Paulo, and Ali Shoker. “Pure Operation-Based Replicated Data Types.” *Cornell University*, October 13, 2017. https://doi.org/arXiv:1710.04469.

Biswas, Ranadeep, Michael Emmi, and Constantin Enea. “On the Complexity of Checking Consistency for Replicated Data Types.” University de Paris, SRI International, n.d. https://www.irif.fr/~cenea/papers/crdts-cav19.pdf.

Bourne, Steve, and Bruce Lindsay. “A Conversation with Bruce Lindsay.” *ACM Queue* 2, no. 8 (December 6, 2008). https://queue.acm.org/detail.cfm?id=1036486.

Canini, Marco. “Causal Consistency CS240: Computer Systems and Concurrency Lecture 16,” n.d. https://web.kaust.edu.sa/Faculty/MarcoCanini/classes/CS240/F17/slides/L16-causal.pdf.

Demirbas, Murat. “Conflict-Free Replicated Data Types.” Blog. *Metadata* (blog), April 23, 2013. https://muratbuffalo.blogspot.com/2013/04/conflict-free-replicated-data-types.html.

Gilbert, Seth, and Nancy Lynch. “Brewer’s Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services.” Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139, 2002. https://users.ece.cmu.edu/~adrian/731-sp04/readings/GL-cap.pdf.

Google. “What’s Different about the New Google Docs: Conflict Resolution.” *Google Drive Blog* (blog), September 22, 2010. https://drive.googleblog.com/2010/09/whats-different-about-new-google-docs_22.html.

———. “What’s Different about the New Google Docs: Working Together, Even Apart.” *Google Drive Blog* (blog), September 21, 2010. https://drive.googleblog.com/2010/09/whats-different-about-new-google-docs_21.html.

Kleppmann, Martin, and Alastair Beresford. “A Conflict-Free Replicated JSON Datatype.” *Cornell University*, n.d. https://doi.org/arXiv:1608.03960.

Letia, Mihai, Nuno Pergucia, and Marc Shaprio. “CRDTs: Consistency without Concurrency Control.” *Cornell University*, n.d. https://doi.org/0907.0929.

Lindsay, Bruce, Patricia Selinger, Cesare Galtieri, James Gray, Raymond Lorie, Thomas Price, Franco Putzolu, Irving Traiger, and Bradford Wade. “Note on Distributed Databases.” IBM Research Laboratory San Jose, California 95193: IBM, July 14, 1979. https://domino.research.ibm.com/library/cyberdig.nsf/papers/A776EC17FC2FCE73852579F100578964/$File/RJ2571.pdf.

Liu, Gaoang, and Xiuying Liu. “The Complexity of Weak Consistency.” *Zenodo*, January 29, 2018. https://doi.org/10.5281/zenodo.1161960.

Savant, Nitin, Elise Olivares, and Sun-Li Beatteay. “Conclave - A Private and Secure Real-Time Collaborative Text Editor.” Conclave. Accessed May 12, 2020. https://conclave-team.github.io/conclave-site/.

Schwarz, Reinhard, and Friedemann Mattern. “Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail.” Germany: University of Kaiserslautern, University of Saarland, n.d. https://www.vs.inf.ethz.ch/publ/papers/holygrail.pdf.

Shapiro, Marc, Carlos Baquero, Nuno Preguica, and Marek Zawirski. “The Distributed Computing Column - Convergent and Commutative Replicated Data Types.” *Bulletin of the EATCS;*

European Association for Theoretical Computer Science, no. 104 (n.d.). file:///Users/aarondiamond-reivich/Downloads/120-477-1-PB.pdf.

Shapiro, Marc, Nuno Preguica, Carlos Baquero, and Marek Zawirski. "A Comprehensive Study of Convergent and Commutative Replicated Data Types." *Hal-Inria*, January 13, 2011. https://doi.org/inria-00555588.

———. "Conflict-Free Replicated Data Types." *Inria Reocquencourt* 25 (January 19, 2011). https://pages.lip6.fr/Marc.Shapiro/papers/RR-7687.pdf.

Sun, Chengzheng. *Issues and Experiences in Designing Real-Time Collaborative Editing Systems*. Lecture. YouTube, GoogleTechTalks, n.d. https://www.youtube.com/watch?v=84zqbXUQIHc.

Sypytkowski, Bartosz. "An Introduction to State-Based CRDTs." *Bartosz Sypytkowski - Software Dev Blog* (blog), December 18, 2017. https://bartoszsypytkowski.com/the-state-of-a-state-based-crdts/.

Tseitlin, Ariel. "The Antifragile Organization." *ACM Queue* 56, no. 8 (August 2013). https://doi.org/doi:10.1145/2492007.2492022.

"Usenet.Com," n.d. https://www.usenet.com/what-is-usenet/.

V, Saurabh. "Eventual vs Strong Consistency in Distributed Databases." Hackernoon, July 16, 2017. https://hackernoon.com/eventual-vs-strong-consistency-in-distributed-databases-282fdad37cf7.

Vogels, Werner. "Eventually Consistent." *ACM Queue* 6, no. 6 (December 4, 2008). https://queue.acm.org/detail.cfm?id=1466448.

Zagorskii, Anton. "Operational Transformations as an Algorithm for Automatic Conflict Resolution." *Coinmonks* (blog), n.d. https://medium.com/coinmonks/operational-transformations-as-an-algorithm-for-automatic-conflict-resolution-3bf8920ea447.

Appendix

CLIENT.PY

```
from utils import get_node_with_ID, get_document
From uID_tree_node import uID_tree_node
# Tree representation of each document
class client():
  def __init__(self, site_id):
    self.rootNode = uID_tree_node(None, "0")
    self.site_id = site_id
    self.element_mapping = []
    self.unshared_operations = []

  # inserts the element directly after the previous_id on the client
  def local_insert(self, element, previous_id):
    #get the previous node
    previous_node = get_node_with_ID(self.rootNode, previous_id)
    #create new element
    safe_element = element + ":" + self.site_id
    #insert the new element as the right child
    inserted_id = previous_node.setRightChild(safe_element)
    self.element_mapping.append((safe_element, inserted_id))
    self.unshared_operations.append(("insert", safe_element, inserted_id))

  # deletes the element from the node with id on the client
  def local_delete(self, element, id):
    self.delete(element, id)
    self.unshared_operations.append(("delete", element, id))

  # deletes the element in the given node
  def delete(self, element, id):
    node = get_node_with_ID(self.rootNode, id)
    if element in node.elements:
      node.elements.remove(element)
      self.element_mapping.remove((element, id))
```

```
# execute received remote operations
  def remote_operations(self, remote_operations):
    for op in remote_operations:
      (operation, element, id) = op
      if operation == "insert":
        node = get_node_with_ID(self.rootNode, id)
        if node.elements == [None]:
          node.elements = [element]
        else:
          node.elements.append(element)
        self.element_mapping.append((element, id))
      if operation == "delete":
        #print(get_document(self.rootNode))
        self.delete(element, id)

  # returns the elements of a given node
  def lookup(self, id):
    node = get_node_with_ID(self.rootNode, id)
    return node.elements

  # determines if two nodes are equivalent (same characters in same location)
  def compare(self, x):
    doc1_elements = self.element_mapping
    doc2_elements = x.element_mapping
    for element in doc1_elements:
      if element not in doc2_elements:
        print("not found in doc 2:")
        print(element)
        return False
    for element in doc2_elements:
      if element not in doc1_elements:
        print("not found in doc 1: ")
        print(element)
        return False
    return True
```

uID_tree_node.PY

```
class uID_tree_node():
    def __init__(self, element, rootID):
        self.rootID = rootID
        self.elements = [element]
        self.site_counter = 0
        self.leftChild = None
        self.rightChild = None

    # add the left_child_element to the left child of the node
    def setLeftChild(self, left_child_element):
        if self.leftChild == None:
            newID = self.rootID + "0"
            self.leftChild = uID_tree_node(left_child_element, newID)
            return newID
        else:
            if self.leftChild.elements == [None]:
                self.leftChild.elements = [left_child_element]
            else:
                self.leftChild.elements.append(left_child_element)
            return self.leftChild.rootID

    # add the right_child_element to the right child of the node
    def setRightChild(self, right_child_element):
        if self.rightChild == None:
            newID = self.rootID + "1"
            self.rightChild = uID_tree_node(right_child_element, newID)
            return newID
        else:
            if self.rightChild.elements == [None]:
                self.rightChild.elements = [right_child_element]
            else:
                self.rightChild.elements.append(right_child_element)
            return self.rightChild.rootID
```

main.PY

```
import random
import string
from client import client

def main():

   # replicas r0, r1 which receive local state manipulations
   r0 = client("site 0")
   r1 = client("site 1")

   # Distributed system R comprised of r0, r1
   R = [r0, r1]

   #Start each client on the same document
   r0.local_insert("A", "0")
   r1.local_insert("A", "0")

   actions = []

   for i in range(100):
      # choose an actor
      actor_id = random.choice([0, 1])
      actor = R[actor_id]
      # choose an action, element and location
      action = random.uniform(0, 1)
      if action < .8:
         # insert action
         element = random.choice(string.ascii_letters)
         # choose the id of the node to insert after
         if len(actor.element_mapping) == 0:
            id = "0"
         else:
            id = random.choice(actor.element_mapping)[1]
         # insert locally
         actor.local_insert(element, id)
      else:
         # delete action
         if not len(actor.element_mapping) == 0:
            (element, id) = random.choice(actor.element_mapping)
```

```
                actor.local_delete(element, id)

            actions.append((actor.site_id, action, element, id)) # mark that we are adding this



            # 1/20th of the time we send local ushared actions to the other actor
            # and check that actions are included correctly
            if random.choice(list(range(20))) == 0:
                other_actor = R[1 - actor_id]
                local_unshared_actions = actor.unshared_operations
                other_actor.remote_operations(local_unshared_actions)
                actor.unshared_operations = []
                print("merged the actions of: " + actor.site_id + " into: " + other_actor.site_id)



        # share unshared actions one last time at the end
        # and check that the states are equal!
        r0_unshared_actions = r0.unshared_operations
        r1_unshared_actions = r1.unshared_operations
        r0.remote_operations(r1_unshared_actions)
        r1.remote_operations(r0_unshared_actions)

        assert r0.compare(r1)
        print("simulation complete and replica states are equivalent")

    main()
```

utils.PY

```
def get_node_with_ID(rootNode, id):
    if float(rootNode.rootID) == float(id) and len(rootNode.rootID) == len(id):
        return rootNode
    else:
        #compare the next path item in the string
        root_id_len = len(rootNode.rootID)
        next_path_director = id[root_id_len]
        if next_path_director == "1":
            if rootNode.rightChild == None:
                rootNode.setRightChild(None)
            return get_node_with_ID(rootNode.rightChild, id)
        else:
            if rootNode.leftChild == None:
                rootNode.setLeftChild(None)
            return get_node_with_ID(rootNode.leftChild, id)

# returns the document in list format
def get_document(rootNode):
    doc = get_document_helper(rootNode, [])
    doc = [i for i in doc if i != []]
    temp_cleaned_doc = []
    for e in doc:
        temp_cleaned_doc.extend(e)
    cleaned_doc = []
    for elements in temp_cleaned_doc:
        cleaned_doc.append(elements.split(":site"))
    return cleaned_doc

# In order traverse the tree and return the document
def get_document_helper(rootNode, document):
    if rootNode == None:
        return None

    get_document_helper(rootNode.leftChild, document)
    #Remove None values from list
    elements = [i for i in rootNode.elements if i]
    document.append(elements)
    #document.append((rootNode.elements, rootNode.rootID))
    get_document_helper(rootNode.rightChild, document)
    return document
```