**An Outdated Philosophy of Distributed Systems**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.” [[1]](#footnote-1) 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.”[[2]](#footnote-2) 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 , then if each piece of data exists in only one location, each piece of data is accessible with probability If, on the other hand, each piece of data is replicated times, then each piece of data is available with probability . If we assume that and 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 (think AWS) that handle trillions of transactions, improbable events like server downtime become almost guaranteed.[[3]](#footnote-3) As a result, systems are designed with replication of datastores to guarantee consistent availability.[[4]](#footnote-4) This solution helps systems maintain their availability in exchange for creating complexity in making sure that data is consistent across the replicas.[[5]](#footnote-5)

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. Of course, it is not possible for an update to automatically update every replica without communication between replicas. However, there is a consistency model that mimics this desired behavior, the unanimous agreement update strategy.[[6]](#footnote-6) This strategy dictates that unless every replica accepts the update, the update is rejected. Thus, with a replica availability probability of and replicas, each update is only accepted with probability Again, if and , each update is only accepted about 81% of the time.[[7]](#footnote-7) 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.[[8]](#footnote-8)

There are many other data consistency strategies that offer a higher probability that write requests will succeed….

The consistency models discussed above and written about by IBM in 1979, attempt to achieve distribution transparency – the idea that, to the user, the distributed system appears like it is one singular system instead of a network of databases working together.[[9]](#footnote-9) They took the philosophy that it was better to fail transactions than break the façade of distribution transparency.[[10]](#footnote-10)

**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.”[[11]](#footnote-11) 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.[[12]](#footnote-12) Availability says that every request received by a non-failing node must eventually terminate with some response.[[13]](#footnote-13) A partition is the a division of the nodes in a network such that there are no successful communications between nodes in different partitions. Thus, partition-tolerance states that consistency and availability still occur even if the network is partitioned. [[14]](#footnote-14)

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

First, in a distributed system it is impossible for a read/write data object to have availability and consistency in an environment in which messages may be lost. The basis of the proof follows: assume a network contains two nodes . Create a partition of the network such that and can no longer communicate with each other. Let function write data to . Later, let read from . The value returned from and will be the same. Thus, this system is not consistent. [[15]](#footnote-15)

Second, in a distributed system it is impossible for a read/write data object to be available in all executions and consentient in all executions in which no data is lost. Let us again discuss the high-level ideas of the 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 in all executions. However, our first proof showed that a network cannot guarantee availability and atomic consistency in all fair executions. Thus, the network is unable to guarantee availability in all fair executions and atomic consistency in only fair executions with no message loss. [[16]](#footnote-16)

**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. [[17]](#footnote-17)

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 [[18]](#footnote-18) – , the idea of systems being unavailable became increasingly less tolerable. [[19]](#footnote-19) MAYBE A GRAPH OF USE OF INTERNET AND SIZE OF DISTRIBUTED SYSTEMS Thus, the industry’s mindset began to shift from one prioritizing consistency to one prioritizing availability.

The CAP Theorem helped researchers understand the tradeoffs that could be made to maintain availability. It proved that only two of the following three properties could be achieved in a distributed system: data consistency, system availability, and tolerance to network partitions. [[20]](#footnote-20)[[21]](#footnote-21)

Importantly, in large-scale distributed systems, network partitions are inevitable [[22]](#footnote-22) and therefore, it is impossible for large-scale distributed systems to maintain data consistency and system availability according to the CAP Theorem. [[23]](#footnote-23)[[24]](#footnote-24) If we want to prioritize availability, then it must be at the expense of consistency.

**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. [[25]](#footnote-25)

Let’s informally define weak data consistency as the following: “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.” [[26]](#footnote-26) Those conditions are determined by the specific implementation of weak consistency. [[27]](#footnote-27)

One such form of weak consistency is eventual consistency. Formally, strong consistency is defined by Shapiro, et al. in the following way:

Eventual Consistency is combination of three properties.

Property 1 – Eventual Delivery: An update that reaches one replica will eventually reach all replicas.

Property 2 – Convergence: Replicas that have received the same updates will eventually have the same state.

Property 3 – Termination: All transactions terminate. [[28]](#footnote-28)

Plainly, the combination of these three properties guarantees that if no new updates are made to a data object, then eventually all accesses to that data object will return its most recently updated value. [[29]](#footnote-29)

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 syncing my paper to Dropbox. With this hardware, I can back up my work in a few ways.

Option 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.

Option 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 what I 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. [[30]](#footnote-30)

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.

**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.[[31]](#footnote-31) This is arbitration process and the sometimes rolling back of updates is a waste of resources which we would like to avoid. [[32]](#footnote-32)

Enter strong eventual consistency. Strong eventual consistency is a specification of eventual consistency. Recall the formal definition of eventual consistency. Strong eventual consistency is eventually consistent with the additional specification of strong convergence, which says that replicas that have received the same updates also have the same state. [[33]](#footnote-33) Therefore, instead of replicas which have seen the same updates being consistency *eventually,* they are now consistent *immediately.*

Let’s remember that eventual consistency guarantees that if no new updates are made to a data object, then *eventually* all accesses to that data object will return its most recently updated value, giving no specification for how long until the data replicas’ state converges.[[34]](#footnote-34)

**Achieving Strong Eventual Consistency Through Conflict-free Replicated Data Types**

A conflict-free replicated data type (CRDT) is a data structure, able to be replicated across multiple nodes in a network such that transactions can be processed independently by nodes and shared across the network such that regardless of the order in which each node receives each transaction, each node will result in the same final state. [[35]](#footnote-35)[[36]](#footnote-36) CRDT’s are distributed datatypes that allow replicas of the CRDT instance to diverge in their state and guarantees all replicas will eventually converge to the same final state.[[37]](#footnote-37) The “conflict-free” nomenclature is a nod to strong eventual consistency. CRDT’s “don’t require exclusive write access and are able to detect concurrent updates and perform deterministic, automatic conflict resolution.”[[38]](#footnote-38) It’s not that conflicts never occur, its that the replica can deterministically resolve the conflict without external information and every replica will resolve the conflict in the same way. [[39]](#footnote-39)[[40]](#footnote-40) Deterministic conflict resolution is possible due to metadata stored in the structure of the datatype. The two categories of CRDT’s, state-based (convergent) data types and operation-based (commutative) data types, differ in how they store this extra metadata. As you might have guessed, state-based data types encapsulate this metadata as part of the data structure itself whereas operation-based data types rely on more heavily on the replication protocol. [[41]](#footnote-41)

As all good things do, this ability comes with a tradeoff – CRDT’s can only service simple, locally verifiable invariants. [[42]](#footnote-42) “CRDTs come in two flavors: state-based, where a state is changed locally and shipped and merged into other replicas; operation-based, where operations are issued locally and … broadcast to all other replicas.” [[43]](#footnote-43) The fundamental difference is how an update to one replica is shared with the 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. [[44]](#footnote-44)

**State-based 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.”[[45]](#footnote-45)[[46]](#footnote-46)[[47]](#footnote-47)[[48]](#footnote-48)

With this basic understanding of state-based conflict free replicated data types, let’s start to formalize.

**Definition 2.1** **– Causal History:** For any replica of some State-based conflict-free replicated data type distributed system, the causal history, follows: [[49]](#footnote-49)

* Initially
* After executing an update, u,
* After executing a merge between replicas and , written, ,

A causal history is a set of events with a causal ordering. For some event, *e,* the causal history, , contains all of the events which causally preceded (read: may have effected) *e*.

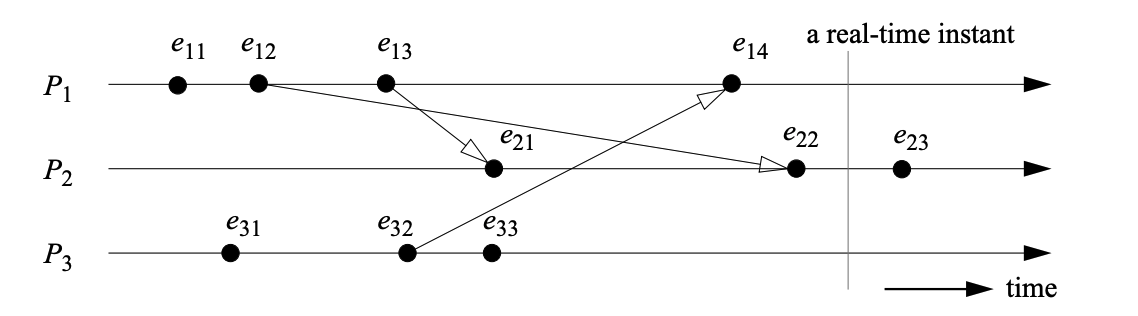
****

Figure . A time diagram of events across three replicas in a distributed system. source: Detecting Causal Relationships in Distibuted 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 if there is a directed path from e to . For example, event may effect local events , and remote events , , and . However, has no effect on or .[[50]](#footnote-50)

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

**Definition 2.2 – Eventual Convergence:** For any two replicas and of a distributed system R, and eventually converge if the below conditions are satisfied: [[51]](#footnote-51)

* Safety: implies that the abstract states of and are equivalent.
* Liveness: implies that eventually

In practice, we can think of eventual convergence as query convergence. That for any queries, q,

This pairwise definition of eventual convergence implies that any subset of replicas in R converge. [[52]](#footnote-52)

**Definition 2.3 – Least Upper Bound:** is a Least Upper Bound of under the partial order if and only if and and there is no such that if and . [[53]](#footnote-53)

**Definition 2.4 – Join Semilattice:** An ordered set (S, ) is a Join Semilattice if and only if exists. [[54]](#footnote-54)

With these definitions in hand, let’s formalize state-based conflict free replicated data types.

**Definition 2.5 – State-Based Conflict Free Convergent Replicated Data Types (CvRDT):** A CvRDT is a distributed data structure composed of 1) local state and algorithms 2) an anti-entropy protocol.[[55]](#footnote-55)[[56]](#footnote-56)

The local state and algorithms are: [[57]](#footnote-57)

* *S,* a join semi-lattice
* *M,* a set of mutators that takes a state and returns an updated state where is an inflator such that
* *Q,* a set of query functions which return data without modifying the state.

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

* Sends the state of to other replicas
* Receives the state of other replicas and performs a merge operation to merge the received state into its own state. The merge operation is commutative, associative, and idempotent. [[58]](#footnote-58)

Because query and mutator operations are performed on the local state of the replica and are executed without communication between replicas, concurrent mutations causing replicas to diverge. [[59]](#footnote-59) Convergence is eventually achieved through the anti-entropy algorithm, which allows all replicas must receive the results of all mutator operations. WHY IS THIS TRUE? WHAT IF THERE IS ONE ADD X AND ONE REMOVE X TRANSACTION? LOOK AT THE DEF OF EVENTUAL CONVERGENCE. WE REQUIRE THE CAUSAL HISTORIES ARE THE SAME. MUST THEY BOTH BE PART OF THE CAUSAL HISTORY?

Before we prove that CvRDT’s converge, let us walk through a State based CvRDT grow-only set with the following specification: [[60]](#footnote-60)

1. Class GrowOnlySetReplica:
3. /// The Set of values stored by this replica
4. Set{} V;
6. /// An Add element mutator which adds the element e to V
7. Mutators:
8. Add(element e): V <- V U {e}
10. /// A Lookup query which returns true if e is in V
11. Query:
12. Lookup(element e): returns e V
13. /// The Anti-Entropy Algorithms
14. Anti-Entropy:
15. /// Merges the state of a different replica into V
16. Merge(ReplicaState V’): V <- V U V’
17. /// Sends V to another replica for merging
18. SendState(Replica r): r.Merge(this.V)

Checkout page 22 a comprehsnevie study. They use compare and merges on S at T. Check that out and make sure the above specification is correct.

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 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 call the Merge function on a different replica.

Although this is a very trivial example, CRDT’s get much more complicated. But before looking at those more complicated structures, lets formally prove that CvRDT’s do indeed converge. As Shapiro et al. do, we will prove the following claim:

**Proposition 1:** Any two replicas of a CvRDT eventually converge assuming that each replica eventually receives all updates.[[61]](#footnote-61)

Consider any two replicas, and . 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 merges the state of and merges the state of , and will have the same causal history. Therefore, by the commutativity of the least upper bound, both and will have the same abstract 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. [[62]](#footnote-62)

**Operation-based or Commutative Replicated Data Type (CmRDT)**

Operation-based replicated data types are another structure 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 he 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.” [[63]](#footnote-63)

A screenshot of a cell phone

Description automatically generated

Figure : 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.[[64]](#footnote-64) ADD MORE DETAIL ABOUT THE PREPARE AND EFFECT OPERATIONS. The evaluation operation takes as arguments a specific query and state and return the result of running the query on the given the state. [[65]](#footnote-65)

Let us again formalize.

**Definition 3.1 – happened-before:** The happened-before relationship, denoted 🡪 orders two transactions, and , 🡪 if and only if for all replicas . [[66]](#footnote-66)

**Definition 3.2 – Concurrent Operation:** Operations, and , are concurrent if they are not ordered by the happened-before relation. Symbolically, not 🡪 and not 🡪 *f.* [[67]](#footnote-67)

**Definition 3.3 – Commutative Data Types:** A concurrent data type is commutative if and only if the following properties are satisfied:

* For any operations and ,
* All concurrent invocations are equivalent to some linear application of the operations [[68]](#footnote-68)

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. [[69]](#footnote-69)

**Definition 2.5 – Operation-Based Conflict Free Convergent Replicated Data Types (CmRDT):** CmRDT is a distributed data structure composed of 1) local state and algorithms 2) an anti-entropy protocol. [[70]](#footnote-70)

The local state and algorithms are:

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

**A screenshot of a cell phone

Description automatically generated**

Figure : 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. [[71]](#footnote-71) The instantiation above causes all replicas to start 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.[[72]](#footnote-72) The prepare function creates a message, in this case just the increment command and sends that message to other replicas. [[73]](#footnote-73) When the message is received, the evaluate function increments the state of the replica. [[74]](#footnote-74) And finally, the evaluate operation only takes in a single query, value, which returns the value of the specified state. [[75]](#footnote-75)

**Proposition 2:** Any two replicas of a CmRDT eventually converge assuming that they are delivered in the delivery order, , and that each replica eventually receives each message. [[76]](#footnote-76)

Consider any two replicas, and . With our liveness assumption, eventually both replicas will have applied all operations to its own state. Therefore, the casual history of and are equivalent. Therefore, for any two operations and in the causal history of ), they fall into one of three cases: (1) if they are not causally related than they by definition are concurrent under and must commute. (2) if they are causally related in the order 🡪 but they are not ordered in the delivery order under , then they must commute. 3) if they are causally related in the order 🡪 and they are also delivered in the order then they are applied in that order at every replica. All three cases lead to every replica having an equivalent abstract state. [[77]](#footnote-77)

**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. [[78]](#footnote-78) 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. [[79]](#footnote-79)

Operation-based CRDTs, on the other hand, often has a smaller per transaction payload because it does not require sending the entire state each time. [[80]](#footnote-80) 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. [[81]](#footnote-81) See the below schema for an op-based Add-Wins Set as an example of the potential complexity of operation-based CRDTs. [[82]](#footnote-82) 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 concurrent add and remove operations are called with the same argument, *x*, for example, the Add-Wins set keeps the element *x* in the set so that subsequent calls to contains(x) will evaluate to true. [[83]](#footnote-83)

A screenshot of a cell phone

Description automatically generated

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

We are now going to shift gears to discuss collaborative editing before discussing how CRDT’s can be used to implement collaborative editing software.

**Collaborative Editing**

As the name implies, collaborative editing is a system that allows multiple individuals to edit a document at the same time from their own computer. [[84]](#footnote-84)[[85]](#footnote-85) 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. [[86]](#footnote-86) 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. [[87]](#footnote-87) Therefore, the biggest issue that collaborative editing software faces is maintaining consistency between each of the clients. [[88]](#footnote-88)[[89]](#footnote-89) Each client, or replica, to borrow stay consistent with our previous language, must converge on the same, “correct” version of the document. [[90]](#footnote-90) Sound familiar?

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

**Google Docs Implements Collaborative Editing using Operational Transforms**

Let’s motivate our discussion with the following example. [[91]](#footnote-91) 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. [[92]](#footnote-92)The software needs more information to correctly incorporate these incompatible concurrent changes.

A close up of a map

Description automatically generated

Figure : Display of the collaborative editting 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. [[93]](#footnote-93) Editing the document is not editing the actual character of the Google Doc, but instead, it is appending an operation onto the end of the revision log. [[94]](#footnote-94) In order to display a document, a client must apply all of the revisions in the revision log in chronological order. [[95]](#footnote-95)

However, there is one more twist. Consider the following example presented by Google. [[96]](#footnote-96) 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.

Operational transforms help clients preserve the intent of each edit by applying updates in accordance with the happened-before relation and transforming operations accordingly. [[97]](#footnote-97) Transformation functions take two forms.

1. Inclusion/forward transformation: The goal of inclusion transformations is for two operations, 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. [[98]](#footnote-98)
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. [[99]](#footnote-99)

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 given using the information he has about the local operations that he has already applied to his client. 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.” [[100]](#footnote-100)

The trouble with operational transforms is that they are actually quite difficult to implement because it requires so much indexes transformations. [[101]](#footnote-101) 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.” [[102]](#footnote-102)

**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. [[103]](#footnote-103) Instead, of computing these complicated transformations, we can implement a collaborative text editor using CRDTs. [[104]](#footnote-104)

CRDT based collaborative text editors actually exist! We will explore the implementation of the Conclave. [[105]](#footnote-105)

In order to implement a collaborative text editor using CRDT’s we need to keep track of some extra metadata. [[106]](#footnote-106) The metadata stored must make each character globally unique in the document and it must make each character globally ordered. [[107]](#footnote-107)

**Maintaining globally unique characters**

Conclave ensures that characters are globally unique by assigning a Site ID to each character in the document. [[108]](#footnote-108) Consider the following example provided by Conclave which highlights the importance of globally unique characters. [[109]](#footnote-109)

A picture containing clock

Description automatically generated

Figure : 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). [[110]](#footnote-110) If instead, received operations used index references instead of globally unique characters, both Alice and Bob would have deleted “A”, shifted the index of “T” and then deleted “T” as well.

Although not included in the Conclave case study, let’s understand what is going on underneath the hood. DISCUSS WHAT TYPE OF CRDT THIS IS: OP-BASED. HOW ARE THE OPERATIONS ALTERED FOR SHARING… THE CLIENT RECEIVES DELETE AT AN INDEX, FINDS THE GLOBAL UNIQUE CHARACTER, AND SHARE DELETE THAT CHARACTER.

**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. [[111]](#footnote-111) Consider the following word with its 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. [[112]](#footnote-112) 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]. [[113]](#footnote-113) 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. [[114]](#footnote-114) Now consider updating the word “CHAT” to “CHEAT”. We need to give the character “E” a global position identifier. To find one, we can consider traversing the following tree. [[115]](#footnote-115)

**A close up of a logo

Description automatically generated**

Figure : 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]. [[116]](#footnote-116)

A map with text

Description automatically generated

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

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 “CCT.” [[117]](#footnote-117)

Because we use fractionally indexed positions, deleting a character does not effect inserting another character – the insert and delete operations commute.[[118]](#footnote-118)

1. Gaoang Liu and Xiuying Liu, “The Complexity of Weak Consitency,” *Zenodo*, January 29, 2018, https://doi.org/10.5281/zenodo.1161960 page 3. [↑](#footnote-ref-1)
2. 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 [↑](#footnote-ref-2)
3. Werner Vogels, “Eventually Consistent,” *ACM Queue* 6, no. 6 (December 4, 2008), https://queue.acm.org/detail.cfm?id=1466448. [↑](#footnote-ref-3)
4. Vogels. [↑](#footnote-ref-4)
5. Vogels. [↑](#footnote-ref-5)
6. Lindsay et al., “Note on Distributed Databases.” page 2 [↑](#footnote-ref-6)
7. Lindsay et al. page 2 [↑](#footnote-ref-7)
8. Lindsay et al. page 2 [↑](#footnote-ref-8)
9. Vogels, “Eventually Consistent.” [↑](#footnote-ref-9)
10. Vogels. [↑](#footnote-ref-10)
11. 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 1. [↑](#footnote-ref-11)
12. Gilbert and Lynch page 3. [↑](#footnote-ref-12)
13. Gilbert and Lynch page 3. [↑](#footnote-ref-13)
14. Gilbert and Lynch pages 3-4. [↑](#footnote-ref-14)
15. Gilbert and Lynch page 5. [↑](#footnote-ref-15)
16. Gilbert and Lynch page 5. [↑](#footnote-ref-16)
17. Lindsay et al., “Note on Distributed Databases." page 2 [↑](#footnote-ref-17)
18. “Usenet.Com,” n.d., https://www.usenet.com/what-is-usenet/. [↑](#footnote-ref-18)
19. Vogels, “Eventually Consistent.” [↑](#footnote-ref-19)
20. Vogels. [↑](#footnote-ref-20)
21. Gilbert and Lynch, “Brewer’s Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services." page 3 [↑](#footnote-ref-21)
22. Ariel Tseitlin, “The Antifragile Organization,” *ACM Queue* 56, no. 8 (August 2013), https://doi.org/doi:10.1145/2492007.2492022 page 1. [↑](#footnote-ref-22)
23. Vogels, “Eventually Consistent.” [↑](#footnote-ref-23)
24. Gilbert and Lynch, “Brewer’s Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services." page 3 [↑](#footnote-ref-24)
25. Gilbert and Lynch page 3. [↑](#footnote-ref-25)
26. Vogels, “Eventually Consistent.” [↑](#footnote-ref-26)
27. Liu and Liu, “The Complexity of Weak Consitency. page 3” [↑](#footnote-ref-27)
28. Marc Shapiro et al., “Conflict-Free Replicated Data Types,” *Inria Reocquencourt* 25 (January 19, 2011) page 5. [↑](#footnote-ref-28)
29. Vogels, “Eventually Consistent.” [↑](#footnote-ref-29)
30. Saurabh V, “Eventual vs Strong Consitency in Distributed Databases,” Hackernoon, July 16, 2017, https://hackernoon.com/eventual-vs-strong-consistency-in-distributed-databases-282fdad37cf7. [↑](#footnote-ref-30)
31. Shapiro et al., “Conflict-Free Replicated Data Types." page 5 [↑](#footnote-ref-31)
32. Shapiro et al. page 5 [↑](#footnote-ref-32)
33. Shapiro et al., “Conflict-Free Replicated Data Types." page 5 [↑](#footnote-ref-33)
34. Vogels, “Eventually Consistent.” [↑](#footnote-ref-34)
35. Marc Shapiro et al., “A Comprehensive Study of Convergent and Commutative Replicated Data Types,” *Hal-Inria*, January 13, 2011, https://doi.org/inria-00555588. [↑](#footnote-ref-35)
36. 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/. [↑](#footnote-ref-36)
37. 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. [↑](#footnote-ref-37)
38. 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/. [↑](#footnote-ref-38)
39. Sypytkowski. [↑](#footnote-ref-39)
40. 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. [↑](#footnote-ref-40)
41. Sypytkowski, “An Introduction to State-Based CRDTs.” [↑](#footnote-ref-41)
42. Demirbas, “Conflict-Free Replicated Data Types.” [↑](#footnote-ref-42)
43. 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. [↑](#footnote-ref-43)
44. Baquero, Paulo, and Shoker. page 1. [↑](#footnote-ref-44)
45. Baquero, Paulo, and Shoker. page 1. [↑](#footnote-ref-45)
46. Sypytkowski, “An Introduction to State-Based CRDTs.” [↑](#footnote-ref-46)
47. Almeida, Shoker, and Baquero, “Delta State Replicated Data Types." page 4 [↑](#footnote-ref-47)
48. Shapiro et al., “A Comprehensive Study of Convergent and Commutative Replicated Data Types." page 7” [↑](#footnote-ref-48)
49. Shapiro et al. page 7 [↑](#footnote-ref-49)
50. 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 [↑](#footnote-ref-50)
51. Shapiro et al. page 9. [↑](#footnote-ref-51)
52. Shapiro et al. page 9 [↑](#footnote-ref-52)
53. Shapiro et al. page 10 [↑](#footnote-ref-53)
54. Shapiro et al. page 11 [↑](#footnote-ref-54)
55. Sypytkowski, “An Introduction to State-Based CRDTs.” [↑](#footnote-ref-55)
56. Almeida, Shoker, and Baquero, “Delta State Replicated Data Types." page 4 [↑](#footnote-ref-56)
57. Almeida, Shoker, and Baquero. page 4 [↑](#footnote-ref-57)
58. Sypytkowski, “An Introduction to State-Based CRDTs.” [↑](#footnote-ref-58)
59. Almeida, Shoker, and Baquero. page 4 [↑](#footnote-ref-59)
60. Shapiro et al., “A Comprehensive Study of Convergent and Commutative Replicated Data Types." page 22 [↑](#footnote-ref-60)
61. Shapiro et al. page 10. [↑](#footnote-ref-61)
62. Shapiro et al. page 10. [↑](#footnote-ref-62)
63. Baquero, Paulo, and Shoker, “Pure Operation-Based Replicated Data Types." page 1 [↑](#footnote-ref-63)
64. Baquero, Paulo, and Shoker. page 4. [↑](#footnote-ref-64)
65. Baquero, Paulo, and Shoker. page 4. [↑](#footnote-ref-65)
66. 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. [↑](#footnote-ref-66)
67. Shapiro et al page 72. [↑](#footnote-ref-67)
68. Baquero, Paulo, and Shoker. page 6. [↑](#footnote-ref-68)
69. Savant, Olivares, and Beatteay, “Conclave - A Private and Secure Real-Time Collaborative Text Editor.” [↑](#footnote-ref-69)
70. Baquero, Paulo, and Shoker. page 2. [↑](#footnote-ref-70)
71. Baquero, Paulo, and Shoker. page 6. [↑](#footnote-ref-71)
72. Baquero, Paulo, and Shoker. page 4. [↑](#footnote-ref-72)
73. Baquero, Paulo, and Shoker. page 5. [↑](#footnote-ref-73)
74. Baquero, Paulo, and Shoker. page 5. [↑](#footnote-ref-74)
75. Baquero, Paulo, and Shoker. page 5. [↑](#footnote-ref-75)
76. Shapiro et al., “A Comprehensive Study of Convergent and Commutative Replicated Data Types." page 11 [↑](#footnote-ref-76)
77. Shapiro et al. page 11 [↑](#footnote-ref-77)
78. Shapiro et al. page 11. [↑](#footnote-ref-78)
79. Shapiro et al. page 11. [↑](#footnote-ref-79)
80. Shapiro et al. page 12. [↑](#footnote-ref-80)
81. Shapiro et al. page 12. [↑](#footnote-ref-81)
82. Baquero, Paulo, and Shoker, “Pure Operation-Based Replicated Data Types." page 6” [↑](#footnote-ref-82)
83. 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. [↑](#footnote-ref-83)
84. Chengzheng Sun, *Issues and Experiences in Designing Real-Time Collaborative Editing Systems*, Lecture (YouTube, GoogleTechTalks, n.d.), https://www.youtube.com/watch?v=84zqbXUQIHc. [↑](#footnote-ref-84)
85. 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. [↑](#footnote-ref-85)
86. Zagorskii. [↑](#footnote-ref-86)
87. Zagorskii. [↑](#footnote-ref-87)
88. 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. [↑](#footnote-ref-88)
89. Zagorskii, “Operational Transformations as an Algorithm for Automatic Conflict Resolution.” [↑](#footnote-ref-89)
90. Zagorskii. [↑](#footnote-ref-90)
91. Zagorskii. [↑](#footnote-ref-91)
92. Google, “What’s Different about the New Google Docs: Working Together, Even Apart.” [↑](#footnote-ref-92)
93. 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. [↑](#footnote-ref-93)
94. Google. [↑](#footnote-ref-94)
95. Google. [↑](#footnote-ref-95)
96. Google. [↑](#footnote-ref-96)
97. Zagorskii, “Operational Transformations as an Algorithm for Automatic Conflict Resolution.” [↑](#footnote-ref-97)
98. Zagorskii. [↑](#footnote-ref-98)
99. Zagorskii. [↑](#footnote-ref-99)
100. Zagorskii. [↑](#footnote-ref-100)
101. Zagorskii. [↑](#footnote-ref-101)
102. Zagorskii. [↑](#footnote-ref-102)
103. Savant, Olivares, and Beatteay, “Conclave - A Private and Secure Real-Time Collaborative Text Editor.” [↑](#footnote-ref-103)
104. Savant, Olivares, and Beatteay. [↑](#footnote-ref-104)
105. Savant, Olivares, and Beatteay. [↑](#footnote-ref-105)
106. Savant, Olivares, and Beatteay. [↑](#footnote-ref-106)
107. Savant, Olivares, and Beatteay. [↑](#footnote-ref-107)
108. Savant, Olivares, and Beatteay. [↑](#footnote-ref-108)
109. Savant, Olivares, and Beatteay. [↑](#footnote-ref-109)
110. Savant, Olivares, and Beatteay. [↑](#footnote-ref-110)
111. Savant, Olivares, and Beatteay. [↑](#footnote-ref-111)
112. Savant, Olivares, and Beatteay. [↑](#footnote-ref-112)
113. Savant, Olivares, and Beatteay. [↑](#footnote-ref-113)
114. Savant, Olivares, and Beatteay. [↑](#footnote-ref-114)
115. Savant, Olivares, and Beatteay. [↑](#footnote-ref-115)
116. Savant, Olivares, and Beatteay. [↑](#footnote-ref-116)
117. Savant, Olivares, and Beatteay. [↑](#footnote-ref-117)
118. Savant, Olivares, and Beatteay. [↑](#footnote-ref-118)