COMP90020 Distributed Algorithms Report

Conflict-Free Replicated Data Types (CRDT): Implementation, Analysis and Applications

Mohammad Nafis Ul Islam
926190
mislam3@student.unimelb.edu.au

Pan Zhan
962088
pzz@student.unimelb.edu.au

Ziren Xiao
675485
zirenx@student.unimelb.edu.au

Sergey Kabanov
944862
skabanov@student.unimelb.edu.au

Abstract

Reaching eventual consistency is a very basic and necessary condition for distributed systems, involving shared mutable objects. The core aim to reach convergence for replicas of those objects, without explicit synchronization, is tough to achieve. Early methods and approaches for this, are complicated and prone to error. Here, we discuss about a data type, which follows some conditions upon its design, and guarantees eventual consistency at the data structure level. They are now known as Conflict-Free Replicated Data Type or CRDT. This survey natured report discusses CRDTs in a very general way, giving information about its different categories and implementation overview. It also provides some comparative analysis regarding these data types, along with information about its application domains.

Keywords— *CRDT, Strong eventual consistency, Distributed Systems, Convergence*

1 Introduction

Many of the distributed systems we use today, involve some common characteristics like accessing resources in offline mode, accessing them from remote locations and different devices, and most importantly, these resources are shared and can be modified by multiple entities. This is where the issue of consistency and also the need for replication arise. The general approach, that attains *Strong Consistency* is by maintaining a global total order [1]. But, such approach introduces some bottlenecks in scalability, which also reduces the overall performance of the system. Moreover, Strong consistency also needs to deal with the trade-off between availability and partition tolerance of CAP theorem [2].

There has been a different approach, known as *Eventual Consistency* [3], which works pretty well, giving improved performance and more availability in network environments, which are more tolerant to delays. Here, the updates, which take place in one of the replicas, are sent to other replicas asynchronously in later time. That is how the updates execute in all the replicas eventually, ensuring the eventual consistency. The updates, that take place concurrently, need to be resolved using some consensus algorithm [4], but still, the conflicting issues are hard to resolve.

This is where the *Conflict-Free Replicated Data Types (CRDT)* come, which is the key concept for the approach of *Strong Eventual Consistency (SEC)* [5]. To put that simply, SEC is more like eventual consistency, where the replicas do not need any consensus and at the same time, have the freedom from conflicts because of the CRDTs. CRDT replicas eventually converge to a correct state, common to all. Again, CRDT works in asynchronous fashion, so the executions of updates in any replica can take place immediately, which does not get affected by network delays, disruptions or faults. This ensures high availability, much improved scalability and better performance of the distributed systems, involving collaborative environments.

The aim of this report is to provide a general overview of CRDTs, which will include their different characteristics, general implementation mechanisms, comparison with relative approaches, application domains and future scope.

The rest of the report is organised as follows. In Section: 2, we will give some brief information of the related works, incorporating CRDTs and similar concepts. Next, in Section: 3, we explain some technical terminologies, which will be needed in the latter sections. Following that, in Section: 4, different types of CRDTs are explained, which are originated from different synchronization models. Next, in Section: 5, we give general implementation details of some popular CRDTs. In Section: 6, we try to give some comparative analysis for different synchronization models of CRDTs. We also compare CRDT with a similar approach, called Operational Transformation (OT) [6; 7]. Later in Section: 7, we talk about the application domains for different CRDTs. Finally in Section 8, we give some concluding remarks regarding the report and also discuss about the future scope of this data structure, in distributed systems.

2 Related Work

The usage of the term *Conflict-Free Replicated Data Type* or *CRDT* is not much old. But, the concept of it goes way back. Researchers have designed and implemented data types like this, without introducing the idea of CRDT explicitly.

Johnson et al. [8] invented a data type which is now known as LWW-Register, in the year of 1976. This data type combines a multiple number of registers and forms a bigger CRDT like data type. It acts more like a database of registers, where operations like create, update, delete follow the LWW mechanisms and ensures an arbitrary total order of executions, without consensus.

Wuu et al. [9], in the year of 1984, described the concept of two CRDT like data types, which was called Dictionary and Log. These concepts were more similar to today's Commutative Replicated Data Types or CmRDTs. It uses a replicated log of different executions, that operates as an epidemic broadcast medium.

The conflict-freedom, without the need of consensus is much used and needed in the field of collaborative applications. Before the CRDT concept, Operational Transformation (OT) studies by Ellis et al. [6; 7] were used for shared editing systems. Here, the design of the operations are not commutative, but the replica usually transforms it in accordance with the previous concurrent updates. But, instead of OT algorithms, embedding commutative design in data types is much simpler and effective [10]. This is where the concept of CRDT finally starts to come into light. Some early CRDTs for this purpose were WOOT, proposed by Oster et al. [11] and Logoot, proposed by Weiss et al. [12]. These two studies actually fuelled the invention of the concept of CRDTs. After that, most of the work related to CRDTs have been pretty recent. The design of Logoot by Weiss et al. was extended for a basic undo operation, based on PN-Counter [12].

Preguica et al. [13] described the design of a novel sequence CRDT: Treedoc, which is also used for cooperative text editing.

Another note-worthy application of CRDT was proposed by Martin et al. [14], that gives a generalized implementation of Logoot by maintaining XML.

A similar kind of data structure like CRDT, known as Replicated Abstract Data Type was developed by Roh et al. [15]. They provide a general version of LWW, involving partial ordered executions. It was called precedence transitivity, upon which they built many classes similar to LWW.

Since the explicit emergence of CRDTs is pretty recent, there have only been a handful of survey works on the field of CRDTs. Two very comprehensive studies of different types of CRDTs were published by Shapiro et al. [5; 10]. These papers provide the study of some formal sufficient conditions, to ensure eventual consistency by CRDTs. They formally introduce the mechanisms of object replication in asynchronous way, in both sate based and operation based environments. They also provide useful descriptions and properties of different types of CRDTs.

Another similar paper was published by Letia et al. [16], which addresses practical design issues in CRDT, like size of the identifier, reconstruction of the data type, indefinite extension, garbage collection etc. It mainly focuses on ordered-set CRDT.

Ahmed-Nacer et al. [17] published a report, describing the usage of CRDT, in managing a distributed hierarchical file system structure. Besides providing overview of CRDTs, it also provides precise solutions, to incorporate CRDTs in the file systems, following the set based approach.

A study regarding eventual consistency was published by Bailis et al. [18], which addresses CRDT as a data structure, which has provability of eventual consistency. Apart from talking about the limitations and extensions of eventual consistency, it acknowledges that, CRDTs are guaranteed to never violate the safety conditions of a distributed system and discusses about the mechanisms of how it can attain strong eventual consistency.

3 Terminology

Some technical terminologies need to be defined in the beginning, to avoid repeating explanations, and also for a clearer view of the concepts, to be explained in the latter sections. They are as follows.

- **Atom:** Atoms are considered to be immutable base data types. Two atoms are considered equal when their contents match exactly. Some example of atom types are: integer, string, tuple, set etc. [10]
- **Object:** Objects in this report, are considered as mutable, replicated types of data. An object contains a unique identifier; payload or content, which may include atoms or other objects; an initial state and an interface of operations. Two object are called replicas of each other when they have the same identifier, but have different process locations.

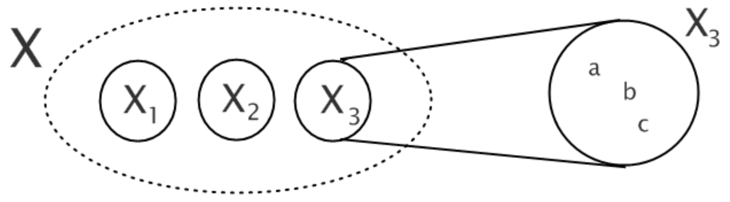


Figure 1: Object

Fig. 1 shows the structure of an object X. It has 3 replicas at 3 different processes and the current payload of replica 3 has the values: 'a', 'b' and 'c'.

- **Partial Order:** A binary relation has a partial order if it has the following properties:
  - Reflexivity, meaning an item can only be comparable to itself
  - Antisymmetry, meaning two different items cannot precede each other for every pair
  - Transitivity, meaning the beginning of a sequence of precedence relation, must precede the end of the sequence.
- **Least Upper Bound (LUB):** According to the definition by Shapiro et al. [10], c = a $\cup_v$ b is an LUB of {a,b} under partial order $\leq_v$, if and only if, $x \leq_v c$ and $y \leq_v c$ and there is no $c' \leq_v c$, such that $x \leq_v c'$ and $y \leq_v c'$. For this, the merge operation $\cup_v$ needs to have the following properties.
  - Idempotence, meaning it can be applied multiple times, which does not change the outcome.
  - Commutativity, meaning the change of order of the operands, does not change the result.
  - Associativity, meaning the order of the operations in the same sequence, does not change the outcome.
- **Semilattice:** A semilattice is a partially ordered set, which has a least upper bound.

4 Types of CRDTs

Based on the synchronization mechanism, CRDTs are mainly divided into two categories. They are as follows:

- State-based CRDT or Convergent Replicated Data Type (CvRDT)
- Operation-based CRDT or Commutative Replicated Data Type (CmRDT)

4.1 State-based CRDT

State based CRDTs follow a passive synchronization model. Here, a client of the system can initiate any update or change to one of the replicas of any object. This replica is considered as the source. This update or change first takes place entirely at the source. Then, the whole updated state of the source is sent to some other replica. When the next replica receives this state, it merges it with its own local state and eventually sends the final state to some other replica. In this way, the update gradually propagates to all other replicas of the system [19]. Fig. 2 shows this mechanism clearly.

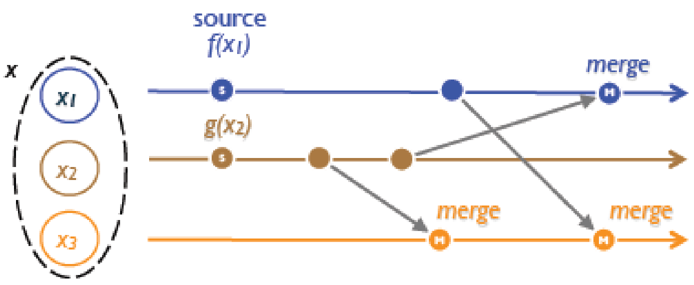


Figure 2: State Based Synchronization. Source: [10]

According to Shapiro et al. [10], all the replicas, following this synchronization model, are bound to converge to the same state, if the following combination of conditions holds.

- The values of the object's payload, form a semilattice.
- Updates need to be increasing; meaning the payload value should be equal or greater than its previous value after the update.
- The merge function has to produce a least upper bound (LUB), and for that, it needs to be idempotent, commutative and associative.
- The replicas need to have eventual connectivity in the network

The above combination of properties is called *Monotonic Semilattice* in the paper of Shapiro et al. [10]. The data type with this property, is named as *Convergent Replicated Data Type (CvRDT)*, which also serves as another name for state-based CRDTs.

4.2 Operation-based CRDT

Operation-based CRDTs follow an active synchronization mechanism. Unlike state-based approach, it does not send the whole state of the source replica to others, when an update operation takes place at itself. Instead, only the update operation is broadcast to all other replicas from the source. Each recipient replica is then expected to replay that update operation.

During broadcast, the operation order might get changed for different replicas. Shapiro et al. [10] shows the following 3 types of scenarios, that can occur here and gives the conditions on how the replicas would still converge to the same state in each of these cases.

1. If the operations are not related by causal order, then they are considered concurrent, and all concurrent events need to be commutative.

2. If operations have a causal order, but they are not delivered in the same order, then they have to commute too. For example, if two operation O_1 and O_2 take place at the source in this order and then broadcast to two other replicas, then one replica might have O_2 being executed before O_1 [19]. In this case, these operations must have commutativity.

3. If the operations have causal order, and they are also delivered in that same order, then it will be just applied in the same order at every replica.

So, we can see that, the key condition for all the replicas to converge to the same state, is that the operations need to have the commutative property in case of disorder. That is why, another name of operation-based CRDTs is *Commutative Replicated Data Type (CmRDT)*. Fig. 3 shows the operation-based synchronization mechanism.

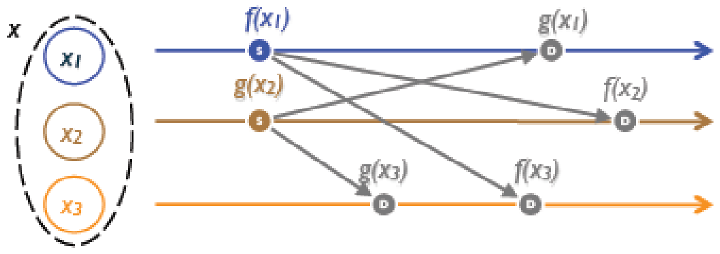


Figure 3: Operation Based Synchronization. Source: [10]

The operations in this synchronization model are not idempotent. This where another condition for these data types to reach convergence arrive, which is a reliable broadcast channel. It is needed to ensure no duplication, while sending the operations to other replicas.

5 Implementation Overview of Some Popular CRDTs

Now, we shall give some brief overviews of some popular CRDTs, to explain more about their designs and concepts. To keep it simple, we shall not go into much specific details, but discuss about them in a more general way. Some of the popular containers for implementing CRDTs are as follows:

- Counter
- Register
- Set
- Graph
- Sequence

We shall discuss about each of them in the following subsections.

5.1 Counter

Counter is an integer value, that is replicated across the system. It supports *increment* and *decrement* operations for updating and a *value* operation, which queries the state of it. The convergence value of it, is the difference between the global number of increments and decrements [10]. These counter based CRDTs can be used in many situations like counting the number of likes in a social media, or counting the current number of logged in users in a system. Among others, two popular counter based CRDTs are *G-Counter* and *PN-Counter*

5.1.1 G-Counter

G-Counter is also knows as Grow-Only Counter, because this counter only increments its value. Fig. 4 shows the sketch of the specification of a State-based G-Counter.

```
payload integer[$n$] $P$                                  ▷ One entry per replica
    initial $[0, 0, \ldots, 0]$
update *increment* ()
    let $g = myID()$                                      ▷ $g$: source replica
    $P[g] := P[g] + 1$
query *value* () : integer $v$
    let $v = \sum_i P[i]$
compare (X, Y) : boolean $b$
    let $b = (\forall i \in [0, n-1] : X.P[i] \leq Y.P[i])$
merge $(X, Y)$ : payload $Z$
    let $\forall i \in [0, n-1] : Z.P[i] = \max(X.P[i], Y.P[i])$
```

Figure 4: State-based G-Counter Specification. Source: [10]

From Fig. 4, we can see that, each replica in a cluster of *n*, has its own local value, which will be eventually incremented. Obviously, they will be assigned unique identifiers, which is first retrieved value in the update operation. Since we are not considering any decrement here for simplicity, the *value* operation is just the sum of all increments for all the entries. The *compare* function ensures the partial order among the states. And finally, the merge operation is taking the maximum value from all the entries, which is idempotent. For example, if two replicas have initially 0 count, and an *increment* is triggered in both replicas, then it will still converge to a single value 1, because maximum function is idempotent and produces an LUB. In another case, if it was an addition, then it would not have been an idempotent merge function, and would not result in convergence. So, this implementation clearly follows the conditions of CRDT and more specifically, the properties of state-based CRDT. Here, a monotonic semilattice is formed with the states and merge function gives an LUB. As a result, it forms a CvRDT.

5.1.2 PN-Counter

One shortcoming of G-Counter is that it does not support decrement. If it did, it would not comply with the monotonicity property, that is needed to form the semilattice, according to the conditions of CvRDT [10]. A solution to this is using a combination of two G-Counters, which then forms a PN-Counter. The 'P' signifies increments, and the 'N' signifies decrements.

```
payload integer[$n$] $P$, integer[$n$] $N$                ▷ One entry per replica
    initial $[0, 0, \ldots, 0]$, $[0, 0, \ldots, 0]$
update *increment* ()
    let $g = myID()$                                      ▷ $g$: source replica
    $P[g] := P[g] + 1$
update *decrement* ()
    let $g = myID()$
    $N[g] := N[g] + 1$
query *value* () : integer $v$
    let $v = \sum_i P[i] - \sum_i N[i]$
compare (X, Y) : boolean $b$
    let $b = (\forall i \in [0, n-1] : X.P[i] \leq Y.P[i] \land \forall i \in [0, n-1] : X.N[i] \leq Y.N[i])$
merge $(X, Y)$ : payload $Z$
    let $\forall i \in [0, n-1] : Z.P[i] = \max(X.P[i], Y.P[i])$
    let $\forall i \in [0, n-1] : Z.N[i] = \max(X.N[i], Y.N[i])$
```

Figure 5: State-based PN-Counter Specification. Source: [10]

Fig. 5 shows the specification sketch of PN-Counter. It is very similar to G-Counters. We now use another array to store the decrements. The *value* operation now returns the difference between increments and decrements. *Compare* operation ensures partial order in both cases by conjunction property. Merge function merges two vectors, instead of one, but uses the same idempotent maximum operation. So, the resultant data type is still a CvRDT like before.

5.2 Register

Registers are memory cells that are used to store atoms or objects in the context of CRDT. Two major register oriented CRDTs are: *LWW-Register* and *MV-Register*.

5.2.1 Last-Writer-Wins Register (LWW-Register)

LWW-Register does exactly the same kind of thing as its name suggests (Last writer wins). Here, a total order among the updates are maintained using timestamps. Fig. 6 shows the structure of State-based LWW-Registers. The objects are like tuples here, containing a value and a timestamp. The *compare* operation takes care of the ordering, and the merge operation always takes the maximum timestamp value, which makes it a CvRDT.

```
payload $X$ $x$, timestamp $t$                                   ▷ $X$: some type
    initial $\bot, 0$
update assign $(X\ w)$
    $x, t := w, now()$                                          ▷ Timestamp, consistent with causality
query $value$ () : $X\ w$
    let $w = x$
compare $(R,\ R')$ : boolean $b$
    let $b = (R.t \leq R'.t)$
merge $(R,\ R')$ : payload $R''$
    if $R.t \leq R'.t$ then $R''.x, R''.t = R'.x, R'.t$
    else $R''.x, R''.t = R.x, R.t$
```

Figure 6: State-based LWW-Register Specification. Source: [10]

Fig. 7 shows an example of LWW-Register where, the second element of the tuple signifies timestamp and we can see that, all the replicas agree on the latest timestamp value.

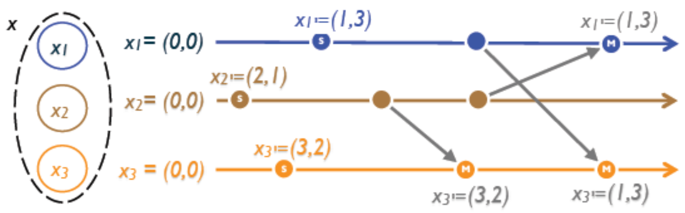


Figure 7: State-based LWW-Register Example. Source: [10]

5.2.2 Multi-Value Register (MV-Register)

LWW-Register takes a precedence over timestamp to reach convergence to a single correct value. Multi-Value Register is another approach, which retains the concurrent updates my taking union of their sets. Later, it could be reduced to a single value [10]. MV-Register uses version vectors for this implementation. The sketch of state-based specifications, used for MV-Register is showed on Fig. 8.

A version vector has an entry for each replica in the cluster, which gets incremented for the events. It has a similar mechanism like G-Counter. This version vector is also sent within the payload during synchronization, which is then used to determine which version vector dominates the other ones. This is used to decide on the new or obsolete values, as the *merge* function takes the union of the elements in a set, which are not dominated by any other element [10]. Fig. 9 illustrates how MV-Registers work.

```
payload set $S$                                              ▷ set of $(x, V)$ pairs; $x \in X$; $V$ its version vector
    initial $\{(\perp, [0, \ldots, 0])\}$
query $incVV$ () : integer$[n]$ $V'$
    let $g = myID()$
    let $\mathcal{V} = \{V | \exists x : (x, V) \in S\}$
    let $V' = [\ \max_{V \in \mathcal{V}}(V[j])\ ]_{j \neq g}$
    let $V'[g] = \max_{V \in \mathcal{V}}(V[g]) + 1$
update assign (set $R$)                                      ▷ set of elements of type $X$
    let $V = incVV()$
    $S := R \times \{V\}$
query $value$ () : set $S'$
    let $S' = S$
compare $(A, B)$ : boolean $b$
    let $b = (\forall (x, V) \in A, (x', V') \in B : V \leq V')$
merge $(A, B)$ : payload $C$
    let $A' = \{(x, V) \in A | \forall (y, W) \in B : V \parallel W \vee V \geq W\}$
    let $B' = \{(y, W) \in B | \forall (x, V) \in A : W \parallel V \vee W \geq V\}$
    let $C = A' \cup B'$
```

Figure 8: State-based MV-Register Specification. Source: [10]

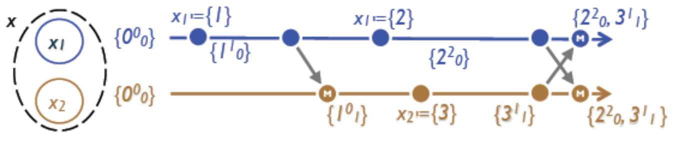


Figure 9: State-based MV-Register Example. Source: [10]

5.3 Set

Sets are basic data types, that serve as the starting point of building many complex data structures. But, a set cannot act as a CRDT because of its usual *add* or *remove* operations, as they do not commute [10]. An example is shown on Fig. 10, where these usual operations do not converge to same value for replica 1 and replica 3, even after following causal order.

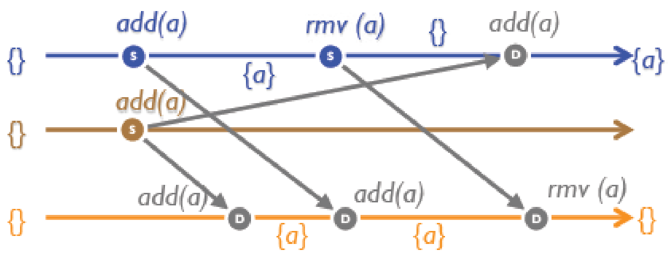


Figure 10: Diverging Example of OP-based Set with Concurrent Add and Remove. Source: [10]

That is why, some approximations are needed with set operations to convert them into CRDTs. We will be discussing about two such approximations, which are: *G-Set* and *2P-Set*.

5.3.1 Grow-Only Set (G-Set)

A Grow-Only Set only allows *add* operation and *lookup*. The specification of a state-based G-Set is given in Fig. 11. Since adding is just taking union of a set with an element, it always commutes. The partial order is maintained by *compare*, which is actually equivalent to a sub-set operation. The *merge* function is also just a union which produces an LUB. So, these states construct the required monotonic semilattice for a CvRDT.

```
payload set $A$
    initial $\varnothing$
update $add$ (element $e$)
    $A := A \cup \{e\}$
query $lookup$ (element $e$) : boolean $b$
    let $b = (e \in A)$
compare $(S, T)$ : boolean $b$
    let $b = (S.A \subseteq T.A)$
merge $(S, T)$ : payload $U$
    let $U.A = S.A \cup T.A$
```

Figure 11: State-based G-Set Specification. Source: [10]

5.3.2 Two-Phase Set (2P-Set)

Two-Phase Set is another variant where *remove* is allowed, but for that, *add* is a little mutated. Here, a previously removed element cannot be added again. Fig. 12 shows this mechanism.

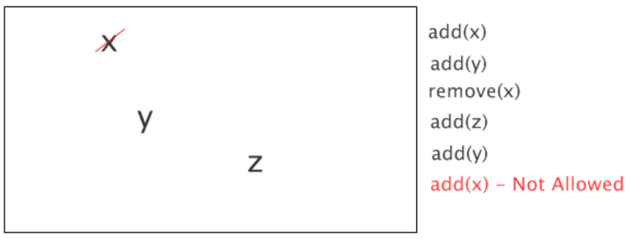


Figure 12: 2P-Set Example

The specification of a state-based 2P-Set (Fig. 13) is pretty similar to the G-Set. Here, actually two G-Sets are used, one for adding items and the other for removing them. The removing set is also called *Tombstone Set* [10]. Once an item gets into it, it is not allowed to be added again.

```
payload set $A$, set $R$                          ▷ $A$: added; $R$: removed
    initial $\varnothing, \varnothing$
query $lookup$ (element $e$) : boolean $b$
    let $b = (e \in A \wedge e \notin R)$
update $add$ (element $e$)
    $A := A \cup \{e\}$
update $remove$ (element $e$)
    pre $lookup(e)$
    $R := R \cup \{e\}$
compare $(S, T)$ : boolean $b$
    let $b = (S.A \subseteq T.A \vee S.R \subseteq T.R)$
merge $(S, T)$ : payload $U$
    let $U.A = S.A \cup T.A$
    let $U.R = S.R \cup T.R$
```

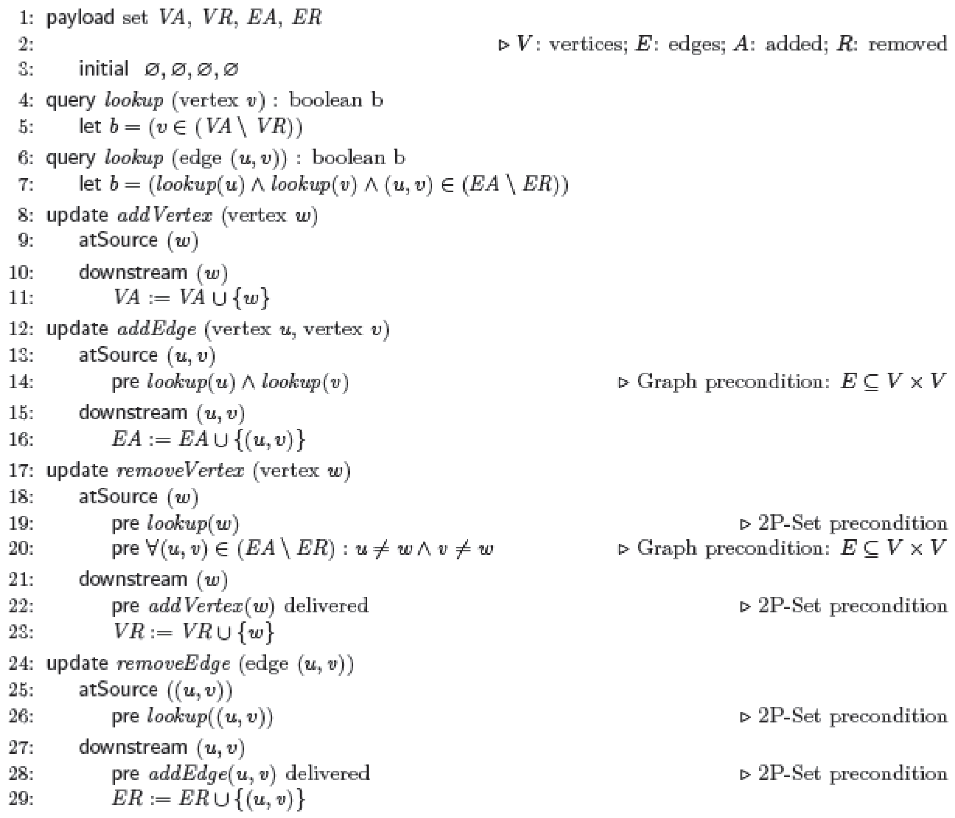
Figure 13: State-based 2P-Set Specification. Source: [10]

Now, the multiple execution of *add* or *remove* operations does not affect convergence. The *merge* also gives an LUB with the individual union of added and removed sets. So, it becomes a CvRDT.

5.4 Graph

Graph based CRDTs are really complex structures. For the sake of generality, we will not get into too much technical details here. We shall just discuss about the common aspects on how CRDTs can be constructed through graphs, by overcoming the issues with graph operations.

Graph operations, needless to say, involve vertices and edges, and because of the relation between the edges and vertices, these operations cannot be considered independently. For example, we cannot add an edge between two vertices if one of them is non-existent. Again, we cannot remove a vertex if it is part of an edge. In case of collaborative distributed systems, this issue arises when there are concurrent *addEdge(u,v)* and *removeVertex(u)* operations.



```
payload set $VA$, $VR$, $EA$, $ER$
                                              ▷ $V$: vertices; $E$: edges; $A$: added; $R$: removed
    initial $\varnothing, \varnothing, \varnothing, \varnothing$
query lookup (vertex $v$) : boolean b
    let $b = (v \in (VA \setminus VR))$
query lookup (edge $(u, v)$) : boolean b
    let $b = (lookup(u) \wedge lookup(v) \wedge (u, v) \in (EA \setminus ER))$
update addVertex (vertex $w$)
    atSource ($w$)
    downstream ($w$)
        $VA := VA \cup \{w\}$
update addEdge (vertex $u$, vertex $v$)
    atSource ($u, v$)
        pre $lookup(u) \wedge lookup(v)$      ▷ Graph precondition: $E \subseteq V \times V$
    downstream ($u, v$)
        $EA := EA \cup \{(u, v)\}$
update removeVertex (vertex $w$)
    atSource ($w$)
        pre $lookup(w)$                        ▷ 2P-Set precondition
        pre $\forall (u, v) \in (EA \setminus ER) : u \neq w \wedge v \neq w$      ▷ Graph precondition: $E \subseteq V \times V$
    downstream ($w$)
        pre $addVertex(w)$ delivered           ▷ 2P-Set precondition
        $VR := VR \cup \{w\}$
update removeEdge (edge $(u, v)$)
    atSource ($(u, v)$)
        pre $lookup((u, v))$                   ▷ 2P-Set precondition
    downstream ($u, v$)
        pre $addEdge(u, v)$ delivered          ▷ 2P-Set precondition
        $ER := ER \cup \{(u, v)\}$
```

Figure 14: Operation-based 2P2P-Graph Specification. Source: [10]

According to Shapiro et al. [10], there could be three types of strategies to encounter this issue. They are as follows:

- *removeVertex(u)* is given priority over all. As a consequence, any adjacent edge to it gets removed too. This is a comparatively easy approach, by utilizing the *Tombstone Set* concept from Section: 5.3.2.
- *addEdge(u,v)* is given priority over all. In case either *u* or *v* or both are missing or removed earlier, they get re-added. This mechanism is very complicated
- *removeVertex(u)* operation is made to wait until all other concurrent *addEdge* operations have finished executing. To enforce this delay between operations, synchronization is needed.

The simple approach, which is the first strategy stated above, uses two 2P-Sets to utilize the concept of *Tombstone Set*. Its specification is pretty similar to 2P-Sets, which is given on

Fig. 14. One 2P-Set is used for vertices and the other is used for edges. For using two 2P-Sets, this is called 2P2P-Graph. All the conditions between the vertices and edges are resolved just like in the 2P-Sets. So, in the end it constructs a valid CRDT [10].

5.5 Sequence

Sequence CRDT or Ordered-Set CRDT is basically used for collaborative text editing systems. There are a handful of sequence CRDTs like Woot [11], Logoot [12], TreeDoc [13; 16], RGA [15] etc. As we have implemented TreeDoc (Binary Tree) for our demo collaborative editing software, we shall keep our discussion to only TreeDoc for sequence CRDTs. We shall exclude the specific implementation details of our code, but discuss it in a more general way.

5.5.1 TreeDoc

A TreeDoc is an operation-based CRDT or CmRDT. It is basically a binary tree which can be extended for the purpose of multi-user collaborative editing. The elements of the TreeDoc are called *atoms* which are in most cases, characters or could be some other immutable elements too. Each atom is associated with a unique position identifier *PosID*. To achieve the properties of a CRDT, these *PosIDs* need to have the following properties [13] .

- Same atoms have same identifiers (in different replicas).
- Different atoms must have different identifiers.
- The identifiers are immutable.
- A total order is maintained among the identifiers.
- For two identifiers 'A' and 'C', we should be able to find another identifier 'B' in between, meaning the identifier space is dense.

The two fundamental operations for TreeDoc are insert($PosID_n$, newAtom) and *delete*($PosID_n$). delete($PosID_n$) is pretty straight forward, which just removes the atom with the PosID. For *insert*($PosID_n$, *newAtom*) operation, the invoking site generates a new unique $PosID_n$ for it, which of course is relative to other atom positions and puts it in the desired location. This is possible because of the last property of PosIDs, mentioned above. Generally, to insert before an atom, a '0' is concatenated with the parent and to insert after an atom, a '1' is concatenated to the parent ID. This is the general case for no child. If we need to insert an atom 'Y' to the right of an atom 'X' and, 'X' has a right child 'Z', then 'Y' is inserted at the left most position of the sub-tree, rooted by 'Z' [16].

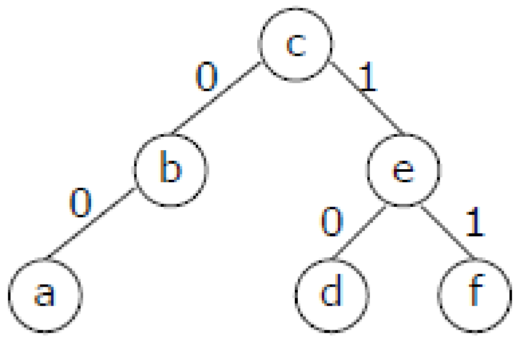


Figure 15: Single User TreeDoc Example. Source: [13]

A simple structure showing the string "abcdef" in a document for a single user, is shown of Fig. 15 for a clearer understanding. The IDs for 'c' is empty; 'b' is '0', 'd' is '10' and so on.

In collaborative distributed environment, other sites may try to allocate conflicting identifiers. To overcome this issue, instead of a simple binary tree, TreeDoc uses an extended structure, where a node can contain other internal nodes called *mini-nodes*. The nodes containing *mini-nodes* are called *major-nodes*. The mini-nodes are distinguished by a *disambiguator*, that keeps track of the site which is responsible for that *mini-node*. Fig. 16 shows the extended structure with *major-nodes* and *mini-nodes*.

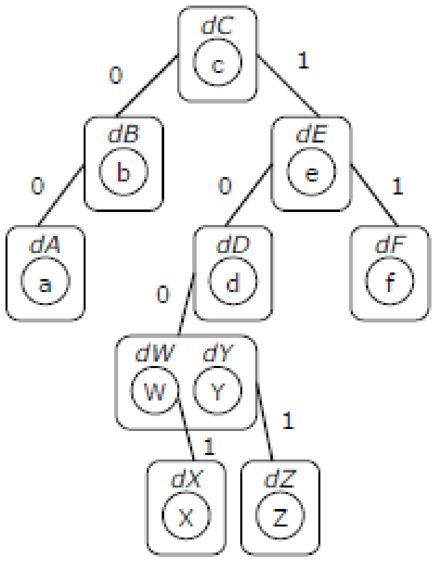
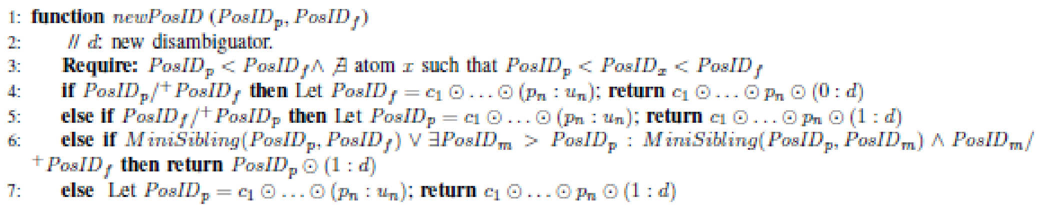


Figure 16: Multi-User TreeDoc Example With Major Node, Mini Node and Disambiguator. Source: [13]

From Fig. 16, we can see that, Site *A* inserts atom 'a' with disambiguator *dA*. Again, for concurrent insertion of 'W' and 'Y', they get inside the same major-node as mini-nodes. Mini-node traversal takes place following the disambiguator order [16]. The mechanism for creating unique IDs during inserts is shown on Fig. 17. This criteria is same as above mentioned single-user scenario. Just the difference is, for disambiguator, it is concatenated along with its disambiguator name, with a ':', preceding '0' or '1' according to its desired position.



```
function $newPosID$ ($PosID_p, PosID_f$)
    // $d$: new disambiguator.
    Require: $PosID_p < PosID_f \wedge \nexists$ atom $x$ such that $PosID_p < PosID_x < PosID_f$
    if $PosID_p /^+ PosID_f$ then Let $PosID_f = c_1 \odot \ldots \odot (p_n : u_n)$; return $c_1 \odot \ldots \odot p_n \odot (0 : d)$
    else if $PosID_f /^+ PosID_p$ then Let $PosID_p = c_1 \odot \ldots \odot (p_n : u_n)$; return $c_1 \odot \ldots \odot p_n \odot (1 : d)$
    else if $MiniSibling(PosID_p, PosID_f) \vee \exists PosID_m > PosID_p : MiniSibling(PosID_p, PosID_m) \wedge PosID_m /$
  $^+ PosID_f$ then return $PosID_p \odot (1 : d)$
    else Let $PosID_p = c_1 \odot \ldots \odot (p_n : u_n)$; return $c_1 \odot \ldots \odot p_n \odot (1 : d)$
```

Figure 17: New ID Generation for Insert. Source: [13]

Different patterns of insert and delete can create unbalanced trees. To solve this issue, a *flatten* operation is used to turn the tree into a flat array. The IDs get modified to resolve any type of ambiguity, while reconstructing the tree. To reach commutativity while updating, an update-wins approach is followed, where in case of concurrent issue between flatten and update, update is given priority, and flatten gets aborted with no effect on the structure. It is based on a voting mechanism. If any site has concurrent update, it votes "no"; otherwise votes "yes" to agree with flattening. The coordinator in the voting mechanism aborts the flattening process, in case of any "no" vote from any site [16].

6 Comparative Analysis

Here, we give two different comparison approaches. As mentioned earlier in Section: 2, before the concept of CRDT, another mechanism, called Operational Transformation (OT) was used, mainly for collaborative systems. Here, in our first comparison approach, we give some basic and general comparisons between CRDT and OT Concepts, since these two are very often talked about in pairs, because of their similarities.

In our second approach, we compare the major two types of CRDTs, which are the state-based ones and the operation-based ones. Besides giving the key differences between them, we also discuss about their similarities and how one can be emulated from the other.

6.1 CRDT vs Operational Transformation (OT)

We have already talked about the basic structures and mechanisms of CRDTs. To have a clearer view of the comparison, we shall first briefly go through the basic idea of operational transformation without going into much detail.

The very basic idea of operational transformation is to change the operation parameters and mechanisms at one replica, based on the execution of the same concurrent operation in some other replica, to achieve the same consistent state in both of them. For example, we think about two operations O_1 and O_2 taking place at two different sites of a collaborative document editing system, which currently has the string "abc" at both sites. Let us suppose O_1 wants to insert the character ’x’ at position ’0’ (the first position) of the string *(insert(position: ’0’,element: ’x’))* and O_2 wants to delete the character at position ’2’ of the current string *(delete(position: ’2’))* which is supposed to be ’c’. The correct state of the string on both sites are expected to be "xab" after these operations are are done. Now, if O_1 occurs first and then O_2 in site 1, then after executing O_1, the string becomes "xabc". Now, if the original O_2 takes place, it would wrongly delete the character ’b’ at position ’2’ instead of ’c’, which was the original intention. For this to converge to the same consistent state, the operation O_2 at site 1 needs to be transformed, so that it would change its deletion position parameter to ’3’ like this O_2': *delete(position: ’3’)*. Only then it would produce the correct intended state. The reverse order does not affect the outcome, even without transformation, in this case; but might be different based on different operations. Fig. 18 illustrates this example.

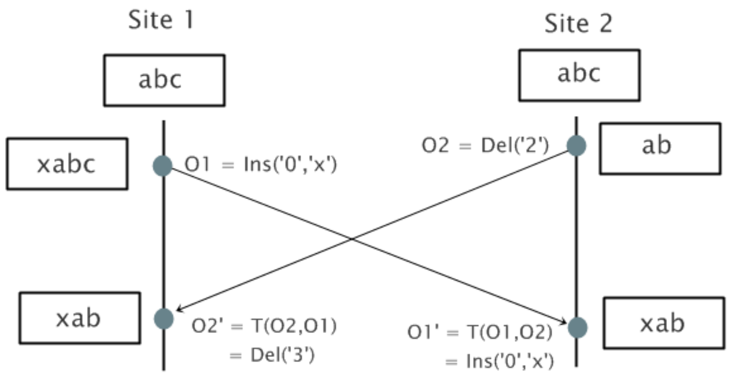


Figure 18: Operational Transformation Example

So, from here, we can see that the core difference between OT and CRDT is that, OT transforms the whole operation to reach consistency, but CRDT either sends the whole updated state, or the update operation as it is, but still reaches consistency with the specific properties (Section: 4) of it.

Being a relatively newer concept, we do not find much usage of CRDTs in industry level collaborative editing systems. On the other hand, many such softwares (like Google Doc) are using OT.

Even though there is industry level usage of OT, there are some consistency issues with this approach, which were proven by Oster et al. [20] and also Imine et al. [21]. We will not get into much technical details regarding how they were proven wrong, as the scope of this report is mainly focused on CRDTs. To put it in a more general way, these papers have stated and proved scenarios, where the OT operations might not converge to a single correct state and the fundamental conditions [21] for convergence are violated.

Moreover, designing OT algorithms are pretty complex, difficult and error-prone compared to CRDTs. But, in case of time and space complexity, OT has more advantage. It is because, OT does not have any cost for buffered operations and transformation time, in case of non-concurrent events. But, CRDT always has to record the object sequence in case of both sequential and concurrent events. So, it costs more time and space when there is no concurrent event. Again, OT saves time and space in case of local operations, but CRDT has to bear the operational costs, even in case of local operations [22].

Tab. 1 summarizes the comparisons between CRDT and OT.

Property	**CRDT**	**OT**
Basic mechanism	Sends state or actual operation	Transforms operation
Convergence Issue	No convergence issue	Sometimes convergence property is violated
Design	Simple	Complex
Efficiency	Relatively lower efficiency as time-space complexity is higher	Higher efficiency as time-space complexity is lower
Industry Use	Not vastly used, because of being a new concept	Used in many industry level systems

Table 1: Basic Comparison Between CRDT and OT

6.2 Comparison Between State-based and Operation-based CRDT

The key difference between state-based (CvRDT) and operation-based (CmRDT) CRDTs is that, the CvRDT propagates the whole state to all the replicas, whereas, the CmRDT only broadcasts the changing operation to the replicas, to gain strong eventual consistency. For that reason, the bandwidth consumption by CvRDT is greater than CmRDT, as the size of operational transactions are less than the whole state of a replica. Again, the CmRDT does not have idempotent operations, so a reliable broadcast channel is needed to ensure no duplication of the sent operations, whereas, the merge operation in CvRDT have commutativity, associativity and idempotence too, so it will converge to the same outcome as all other replicas, even in case of multiple occurrences of the merge. Tab. 2 summarizes these basic differences.

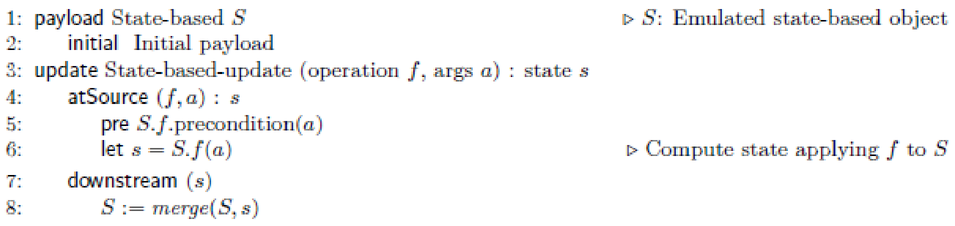
Property	**State-based CRDT (CvRDT)**	**Operation-based CRDT (CmRDT)**
Basic mechanism	The whole state is sent to the replicas	Only the updating operations are sent to the replicas
Sending mechanism	Gradual propagation, from one replica to the next and so on.	Broadcast
Network bandwidth	High bandwidth consumption	Lower bandwidth consumption
Communication reliability	No need for reliable communication	Needs reliable communication channel, to avoid duplicate sends and receives
Operation property	Merge operation is commutative, associative and idempotent	Operations are commutative, associative but not idempotent

Table 2: Basic Differences between CvRDT and CmRDT

Despite these differences, these two types of CRDTs have much similarities, and more interestingly, one of them can be emulated from the other [5; 10]. Avoiding much technical details, we shall briefly discuss about these emulation concepts, just to give a sketch of their interchangeability.

6.2.1 Emulation of a State-based Object with Operation-based Approach

Fig. 19 shows the basic steps of emulating a state-based object from operation-based perspectives, which was proposed by Shapiro et al. [10].



```
payload State-based $S$                                      ▷ $S$: Emulated state-based object
    initial Initial payload
update State-based-update (operation $f$, args $a$) : state $s$
    atSource $(f, a)$ : $s$
        pre $S.f.\text{precondition}(a)$
        let $s = S.f(a)$                                     ▷ Compute state applying $f$ to $S$
    downstream $(s)$
        $S := merge(S, s)$
```

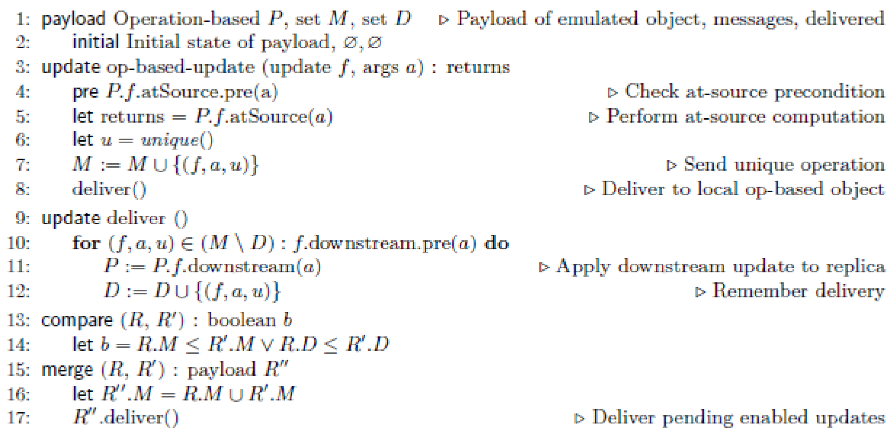
Figure 19: Emulation of a State-based Object from Operation-based. Source: [10]

From Fig. 19, we can see that, the mechanism is pretty simple, where the operation-based update is performed on the state of the source after verifying the precondition. Then, it is propagated in the downstream phase, to all other replicas and there is no need for any precondition here, as the merge has to take place in all the states, that are reachable from the source [10]. Because of being a operation-based emulation, we also do not need the compare operation here.

6.2.2 Emulation of an Operation-based Object with State-based Approach

Fig. 20 shows the basic steps of emulating an operation-based object from state-based perspectives, which was given by Shapiro et al. [10].

From Fig. 20, we can see that operation-based object update is merged withing the message *M*, following the state-based approach. Moreover, unique operation is also maintained for the sake of the final operation-based object emulation. To avoid duplication, the messages are maintained in a sorted order in the set *D*. The delivery operation takes the state to a partial order, while the merge guarantees an LUB operation.



```
payload Operation-based $P$, set $M$, set $D$          ▷ Payload of emulated object, messages, delivered
    initial Initial state of payload, $\varnothing, \varnothing$
update op-based-update (update $f$, args $a$) : returns
    pre $P.f$.atSource.pre($a$)                        ▷ Check at-source precondition
    let returns = $P.f$.atSource($a$)                  ▷ Perform at-source computation
    let $u = unique()$
    $M := M \cup \{(f, a, u)\}$                        ▷ Send unique operation
    deliver()                                          ▷ Deliver to local op-based object
update deliver ()
    for $(f, a, u) \in (M \setminus D) : f$.downstream.pre($a$) do
        $P := P.f$.downstream($a$)                     ▷ Apply downstream update to replica
        $D := D \cup \{(f, a, u)\}$                    ▷ Remember delivery
compare ($R$, $R'$) : boolean $b$
    let $b = R.M \leq R'.M \vee R.D \leq R'.D$
merge ($R$, $R'$) : payload $R''$
    let $R''.M = R.M \cup R'.M$
    $R''$.deliver()                                    ▷ Deliver pending enabled updates
```

Figure 20: Emulation of an Operation-based Object from State-based. Source: [10]

Because of the compare mechanism of state-based approach, and maintaining a sorted set *D*, the updates, contained in the set *M*, maintains a casual order, which guarantees the convergence to same state for non-concurrent events. For concurrent events, the commutative property of the emulated operation-based object: CmRDT, guarantees to converge to same state [10].

7 Applications

Most of the works related to CRDTs are still under the research based implementation platform, as the concept of it is still pretty new in terms of day to day application usage. Many ongoing projects involving CRDTs can be found on the internet [1]. At the same time, there have been quite a few industry level deployments using these data types as well, and its application domain is increasing gradually. Some of the current industry usage, involving CRDTs are as follows:

1. **Redis:** This distributed, in memory database uses CRDT under the hood, which uses bidirectional replication across geographically distributed data centers. It ensures local latencies for read and write operations by utilizing the consensus free protocols of CRDTs, to maintain strong eventual consistency. Moreover, the built-in conflict resolution schemes for these data types, makes application development easier and simpler [2].

2. **Soundcloud:** The popular audio media streaming platform, Soundcloud has developed a distributed storage system called *Roshi*, which implements a set type CRDT, quite similar to LWW-element-set. It used Redis for database, and the CRDT implementation provides self-repairing read-write operations in a stateless layer [3].

3. **Riak:** This distributed key-value type NoSQL database is based on CRDTs. Basically Riak 2.0 have integrated them to create its own *Riak Data Types*, which has counters, maps, registers, flags and sets. These data types make the development process quite easy, where the developer does not need to think much about the complex vector clocks and other conflict resolution mechanisms [4].

[1]https://libraries.io/search?keywords=crdt

[2]https://redislabs.com/docs/under-the-hood/

[3]https://developers.soundcloud.com/blog/roshi-a-crdt-system-for-timestamped-events

[4]https://riak.com/introducing-riak-2-0/

4. **League of Legends:** This famous multiplayer online battle arena game uses the CRDT implementation by Riak, for its chatting system, to achieve their huge linear horizontal scalability. This online system needs to handle 7.5 million concurrent players; 27 million daily players; 11 thousand messages per second, and 1 billion events per server, per day [5]. These numbers, in a nut-shell, can sum up the power of CRDTs pretty well.

5. **Bet365:** Bet365 is the biggest online betting company in Europe which has 2.5 million simultaneous users involvement. This system uses the OR-Set Implementation of Riak to manage the user data.

6. **Tom Tom:** This is a dutch company that is mainly involved in producing traffic, navigation and mapping products. They use CRDTs as robust data structures, for the synchronization of navigation data among user devices, specially for unreliable network conditions. Use of CRDTs enables the resolution of conflicts to be done at data type level, ensuring the eventual consistency among the replicas [6].

7. **Phoenix:** This web development platform, written in Elixir, uses CRDT to have data replication in a conflict-free fashion. It ensures high availability and performance, as there is no need to pass data through any central server. It also gives the property of automatic failure recovery for the system [7].

8. **Facebook:** One of the top tech companies of present time, Facebook, has implemented CRDTs in their *Apollo* database. It provides features like low latency and consistency with scalability [8].

9. **Teletype for Atom:** Although, CRDTs are great for collaborative text editing systems, there haven't been much industry level production using them, in this area. But Teletype for Atom, has developed a standalone library called *teletype-crdt*, which uses CRDT for collaborative editing. It enables developers in a team, to share and collaborate on their codes in real time [9].

10. **CosmosDB:** This NoSQL, multi-model database by Microsoft; uses CRDTs (LWW) for its multi-master write functionality, alongside other conflict resolution models.

8 Conclusion and Future Scope

The literal concept of Conflict-Free Replicated Data Types is relatively new. But, its simplistic approach towards convergence in a distributed system, is giving it much popularity over other similar approaches. We have tried to give a very general overview of CRDTs here. The discussion about the background researches, related to this kind of concepts has shown how the actual idea of CRDTs have emerged and advanced over time.

To keep the report structure like a survey kind of format, we have avoided much technical details or mathematical proofs. Instead, we have discussed about the technical overviews and the core characteristics of CRDTs in a more general way. Both the synchronization models of CRDTs reach a converging, correct state by maintaining the consistency mechanisms, within

[5]http://highscalability.com/blog/2014/10/13/how-league-of-legends-scaled-chat-to-70-million-players-it-t.html

[6]https://speakerdeck.com/ajantis/practical-demystification-of-crdts

[7]https://dockyard.com/blog/2016/03/25/what-makes-phoenix-presence-special-sneak-peek

[8]https://dzone.com/articles/facebook-announces-apollo-qcon

[9]https://blog.atom.io/2017/11/15/code-together-in-real-time-with-teletype-for-atom.html

the data structure level. We have also discussed about some common CRDTs based on counter, set, register, graph etc. and their implementation mechanisms in a general way.

For the comparisons, we have primarily chosen operational-transformation to compare with CRDT, because they have very similar nature. Even though OT was proposed much earlier, we have shown the issues with it, for which CRDT is now being considered in many cases alongside OT. Furthermore, there has also been discussion about the similarities and differences between the state-based and operation-based synchronization models of CRDT. Despite being a new concept, CRDTs have their handful of applications in different domains too, which were discussed briefly.

The future work related to CRDT will probably cover both theoretical and also practical fields. The simple design format and serverless communication among peers, in collaborative distributed systems, has many benefits as well as many challenges, that need to be overcome. Because of the conditions and properties of CRDTs, to achieve convergence without consensus, it would not be suitable for all types of problems and applications. More work on this could be done on research platform, to enhance the capabilities of CRDTs. From the perspective of implementation within application, more well defined CRDT libraries could help developers to work with more ease. As for this report, it could be elaborated even more, giving much technical details regarding CRDT and could be transformed into a more comprehensive survey, including more recent researches and studies, in this field.

References

[1] L. Lamport, “Time, clocks, and the ordering of events in a distributed system,” *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[2] S. Gilbert and N. Lynch, “Brewer’s conjecture and the feasibility of consistent, available, partition-tolerant web services,” *Acm Sigact News*, vol. 33, no. 2, pp. 51–59, 2002.

[3] W. Vogels, “Eventually consistent,” *Queue*, vol. 6, no. 6, pp. 14–19, 2008.

[4] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser, “Managing update conflicts in bayou, a weakly connected replicated storage system,” in *SOSP*, vol. 95, 1995, pp. 172–182.

[5] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, “Conflict-free replicated data types,” in *Symposium on Self-Stabilizing Systems*. Springer, 2011, pp. 386–400.

[6] C. A. Ellis and C. Sun, “Operational transformation in real-time group editors: issues, algorithms, and achievements,” in *Proceedings of the 1998 ACM conference on Computer supported cooperative work*. Citeseer, 1998, pp. 59–68.

[7] C. A. Ellis and S. J. Gibbs, “Concurrency control in groupware systems,” in *Acm Sigmod Record*, vol. 18, no. 2. ACM, 1989, pp. 399–407.

[8] P. R. Johnson and R. H. Thomas, “The maintenance of duplicate databases. internet request for comments rfc 677,” *Information Sciences Institute*, 1976.

[9] G. T. Wuu and A. J. Bernstein, “Efficient solutions to the replicated log and dictionary problems,” in *Proceedings of the third annual ACM symposium on Principles of distributed computing*. ACM, 1984, pp. 233–242.

[10] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, “A comprehensive study of convergent and commutative replicated data types,” Ph.D. dissertation, Inria–Centre Paris-Rocquencourt; INRIA, 2011.

[11] G. Oster, P. Urso, P. Molli, and A. Imine, “Data consistency for p2p collaborative editing,” in *Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work*. ACM, 2006, pp. 259–268.

[12] S. Weiss, P. Urso, and P. Molli, “Logoot-undo: Distributed collaborative editing system

on p2p networks," *IEEE transactions on parallel and distributed systems*, vol. 21, no. 8, pp. 1162–1174, 2010.

[13] N. Preguica, J. M. Marques, M. Shapiro, and M. Letia, "A commutative replicated data type for cooperative editing," in *2009 29th IEEE International Conference on Distributed Computing Systems*. IEEE, 2009, pp. 395–403.

[14] S. Martin, P. Urso, and S. Weiss, "Scalable xml collaborative editing with undo," in *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*. Springer, 2010, pp. 507–514.

[15] H.-G. Roh, M. Jeon, J.-S. Kim, and J. Lee, "Replicated abstract data types: Building blocks for collaborative applications," *Journal of Parallel and Distributed Computing*, vol. 71, no. 3, pp. 354–368, 2011.

[16] M. Letia, N. Preguiça, and M. Shapiro, "Crdts: Consistency without concurrency control," *arXiv preprint arXiv:0907.0929*, 2009.

[17] M. Ahmed-Nacer, S. Martin, and P. Urso, "File system on crdt," *arXiv preprint arXiv:1207.5990*, 2012.

[18] P. Bailis and A. Ghodsi, "Eventual consistency today: Limitations, extensions, and beyond," *Queue*, vol. 11, no. 3, p. 20, 2013.

[19] N. Yigitbasi, "A look at conflict-free replicated data types (crdt)," https://medium.com/@istanbul_techie/a-look-at-conflict-free-replicated-data-types-crdt-221a5f629e7e, 2015.

[20] G. Oster, P. Urso, P. Molli, and A. Imine, "Proving correctness of transformation functions in collaborative editing systems," Ph.D. dissertation, INRIA, 2005.

[21] A. Imine, M. Rusinowitch, G. Oster, and P. Molli, "Formal design and verification of operational transformation algorithms for copies convergence," *Theoretical Computer Science*, vol. 351, no. 2, pp. 167–183, 2006.

[22] C. Sun, D. Sun, W. Cai *et al.*, "Real differences between ot and crdt for co-editors," *arXiv preprint arXiv:1810.02137*, 2018.