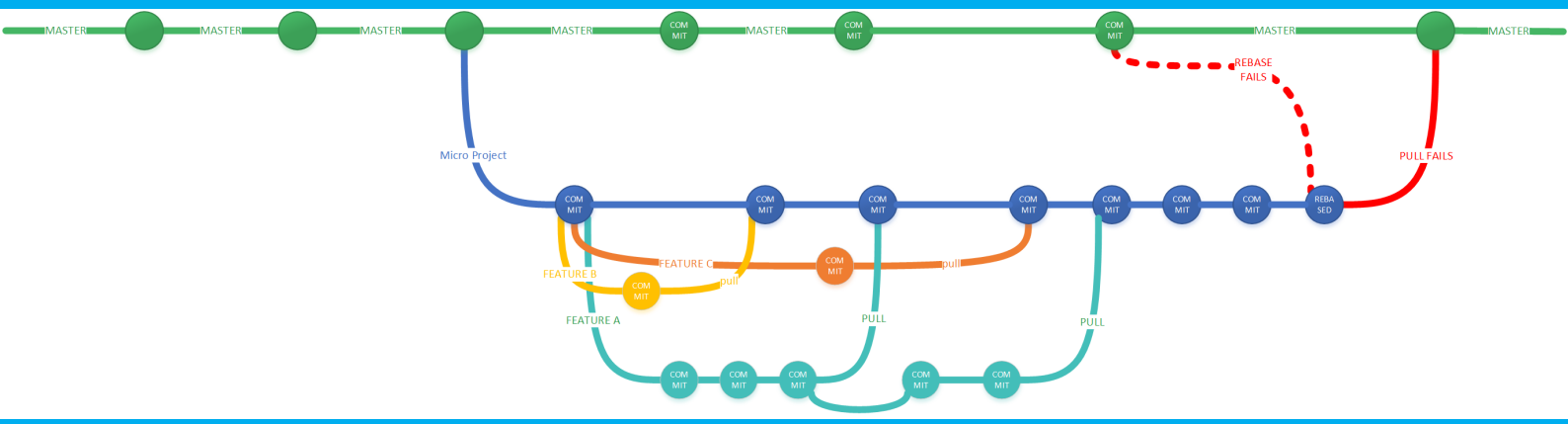
# Conflict Free R-Tree

## A CRDT-based index for P2P systems

E. M. Bongers




TU Delft

# Conflict Free R-Tree

## A CRDT-based index for P2P systems

by

E. M. Bongers

to obtain the degree of Master of Science at the Delft University of Technology, to be defended publicly on 2021-07-01Z10:00:00T


| | | |
|---|---|---|
| Student number: | 1188232 | |
| Project duration: | 2020-10 – 2021-08 | |
| Thesis committee: | (dr.) ir. M. de Vos, | TU Delft, advisor |
| | prof. dr. J. A. Pouwelse, | TU Delft, supervisor |
| | prof. dr. Nomen Nescio, | TU Delft |


An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

TU Delft

# Abstract

CRDTs are data structures that allow conflict free replication and modification. In theory, CRDTs seem like a natural fit for open P2P networks, however there are obstacles to overcome. Firstly, many proposed CRDTs are grow-only because (i) CRDTs may track deletions in permanent tombstone values or (ii) they may gather permanent information on every peer in the system. As such, CRDTs are not well adapted to open P2P networks. Many peers may come and go over time and persistent accurate information about all peers will grow too large eventually. Secondly, some types of CRDT (mainly op-based CmRDT) require causal message delivery which is hard in open P2P networks. Thirdly, CRDTs are typically built with the assumption that all peers need all data in a replica and thus all data is fully replicated, even though a client may only be interested in a small subset.

To address these issues a new state-based CvRDT is proposed: BloomCRDT, which is a variation on the OR-set that replaces its $R$ grow-only set with Bloom filters. It does not need knowledge of other peers or their state and avoids tombstones. This makes BloomCRDT well suited for use in open P2P networks. The grow-only aspect is vastly reduced compared to the standard OR-set. However, the BloomCRDT itself is indivisible and can grow to be too large if it stores many elements. This limit is not high enough to accommodate demanding applications. Scalability can be dramatically improved by combining multiple BloomCRDTs to form a Conflict Free R-Tree (CFRT). Each node of the R-Tree is represented by a BloomCRDT. Concurrent modifications are allowed and, due to the characteristics of the R-tree, this results in a consistent data structure with overlapping ranges. Since an R-Tree's efficiency is reduced when ranges overlap, a periodic optimization algorithm can be used to eliminate the overlapping ranges. A proof-of-concept implementation was made of BloomCRDT and CFRT using Python and PyIpv8. Experiments where performed using the Gumby experimental framework and the DAS5 cluster. The experiments show that BloomCRDT performs as designed: it keeps a smaller state than other solutions while being invariant to the number of identities that have interacted. The CFRT is shown to be able to balance key/value entries over multiple BloomCRDT instances using favorable messaging metrics. Moreover the CFRT, and by extension BloomCRDT, tolerate network faults and can work with up to 90% message loss.

# Preface

Around 2006 I worked on an application that used an object database. The database would replicate transactions to its peers as a means of communicating updates. I wondered if I could simply serialize and send the object state to the other peers, and then use a merge function that would handle conflicts. Like Git, which can already do a lot automatically, but without the human part. My intuition was that Git has very little notion of what exactly is going on in the various source files, but perhaps, by annotating more information it would be possible to fully automate object merging. This extra information could be: field X is an integer monotonic counter, or field Y is a reference to another object of type Z, etc. After coding up an example I realized I could come up with merging rules for certain simple things like counters and references, but not for more complicated data types like strings. But, why is that? What differentiates them? At the time I didn't pursue this any further since it was delaying the progress on the application, but unresolved questions like this tend to stick in my mind.

Fast-forward to the beginning of this thesis, and we investigated past issues that have sparked my curiosity. During the the next meeting with Martijn, he casually mentioned that my merging objects problem sounded like CRDTs. When subsequently reading about them, the veil was lifted, heavens parted and choirs sang. Well ok, perhaps not that dramatic, but it was a nice and sunny day. Suffice it to say that is how I found my thesis topic. From there on the process has been (mostly) rewarding. In no small part thanks to dr. ir. Martijn de Vos, who advised me and kept me on track, and to whom I can only express my deepest gratitude and thanks.

I cannot overstate my gratitude to my wife, Merel. For supporting me and her patience[1].

*E. M. Bongers*
*Schijndel, June 2021*

[1] And I promise Merel, to not pursue further academic advancement in the immediate future.

# Contents

# 1 Introduction

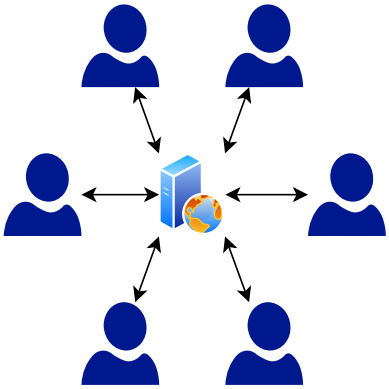
Big Tech (Google, Apple, Facebook, Amazon and Microsoft) offers easy to use services that attract many users. This in turn attracts developers to create products based on Big Tech's platforms. Users are attracted to Big Tech products and services because they offer a one-stop shop: a convenient, centralized and universally accessible means of working and collaboration. And so it is that Big Tech not only dictates the technological innovation but also controls the means of production, namely the data centers and the data stored there [47]. Some even go so far as to use this control to apply censorship [49] [32] [17].

The move towards Big Tech creates a problem because agency is stripped from users. The control a user can exert on digital artifacts (modify, copy, delete) is increasingly limited by Big Tech. By placing data in a data center, the user is also reliant on continued service to access to this data. It is precisely this shift, away from locally stored data towards centrally stored data, that drives Big Tech's growth. However the growth of Big Tech is becoming a problem for open source initiatives. Consider for example the LibreOffice [20] productivity software suite. It is as functionally complete as Microsoft Office, but it cannot compete with Google Docs on online collaboration. The disruption that Google Docs caused has even prompted Microsoft to transition Office to Office365. Most Open Source initiatives are in danger of missing the boat when it comes to online collaboration, since they can not hope to finance, deploy and maintain the necessary infrastructure.

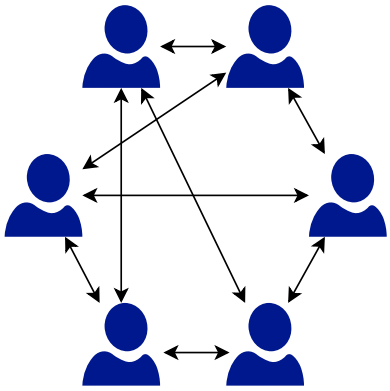
## 1.1. Local-First Software

Literature on Local-First Software [31] (LFS) signals the perils of Big Tech based software, and proposes ideals that should enable software to provide online collaboration without large scale supporting infrastructure. The main argument against the use of Big Tech online collaborative services is the question of data ownership. Not so much in the legal sense (that does not exist [19]) but rather in terms of the user's agency. When using online collaborative services the user typically only has a local cache copy at best, not an authorative or complete copy. This leads to a continued reliance on the online service and thus the continued existence of said service. If Big Tech decides to discontinue the service, the user's work stalls. LFS proposes to develop software that avoids the dangers of Big Tech by making each user's local resources authorative and complete. To this end LFS introduces the ideals for local-first software. Three of these in particular are of interest in relation to Big Tech:

1. **Responsiveness and Availability** By having a local authorative copy of data, applications do not have to wait for round trip calls to remote resources. Through the local authorative copy, LFS is free from Big Tech vendor lock-in which holds digital artifacts hostage through export controls.
2. **Seamless collaboration** If (concurrent) modifications cannot generate artifact versions or conflicts, then the collaboration is seamless since modifications can always be merged. This is an important consideration for user adoption. Big Tech services are not always free of this constraint: their platforms impose rules and workflows that must be followed.
3. **Optional network connectivity** Global network coverage is not a realistic assumption, as there will always be instances where machines are offline. By having a local copy and the guaran-



(a) A centralized network organisation example



(b) A decentralized network organisation example

Figure 1.1: Example network organisation for centralized systems and decentralized systems.

tee of error free merging, LFS works just as well offline as online, while maintaining seamless collaboration. This is a sharp contrast to Big Tech's online-only service.

A narrative that is not yet described in the context of Local-First Software is that, by embracing its ideals, open source can stand on equal footing with Big Tech. Not through bulk resources but through smart software. Applying the LFS ideals can move open source forwards to enable the online collaboration features and workflows that many users are becoming accustomed to. However applying ideals is easy in an ideal world, but in practice it requires non-trivial distributed systems. Proponents of the LFS ideals propose using Conflict-free replicated Data Types (CRDTs), a family of data structures suitable for use in decentralized systems. CRDTs allow users to work on a local replica of data while the CRDT merges this replica with work from others. Eventually all replicas reach a globally consistent state after all modifications stop. Chapter 2 introduces CRDTs and the relevant distributed systems background to understand CRDTs in the context of this work.

## 1.2. Central Components and Decentralized Systems

Some networked systems rely on centralized components (such as central servers) to perform some critical function. Figure 1.1 shows an example network organisation of centralized components (1.1a) versus decentralized components (Figure 1.1b). Often the central component is used to provide a controlled environment or to provide authority: a single state that the rest of the network accepts as absolute truth. It is relatively straightforward to design a system with a central authorative truth or process executing in a controlled and trusted environment. There is no conflict between network participants since they can rely on the centralized component to resolve these. Using central components in this way is a hallmark of Big Tech, which takes the role of central authority. Having centralized components can greatly simplify designs, but centralized components are also a weak point for a distributed system. An example would be HTTP web pages. Through links it is an interconnected distributed system and it uses web servers as a central component. However such servers are not able to perform their central function when they are overloaded, offline or disconnected. Thus the use of central components is often a design compromise. A good example of this is MMO gaming, a distributed system where the actual game simulation happens in a controlled environment on a central server cluster. Distributed gaming without a central component has yet to be perfected. So for some distributed system designs a central component might be the only practical choice.

Centralized components have three aspects that make them fundamentally undesirable in a distributed system. Firstly, from a technical point of view central components are a single point of failure, so the distributed system cannot function (fully) if a centralized component is unreachable or unresponsive. A second weakness is organizational: each instance of a central component is inevitably controlled by a single entity and, any entity is influencable by private and government action. Finally, central components also have a financial aspect: there is a real world cost associated with central components and the distributed system's user community has to provide for this in some way.

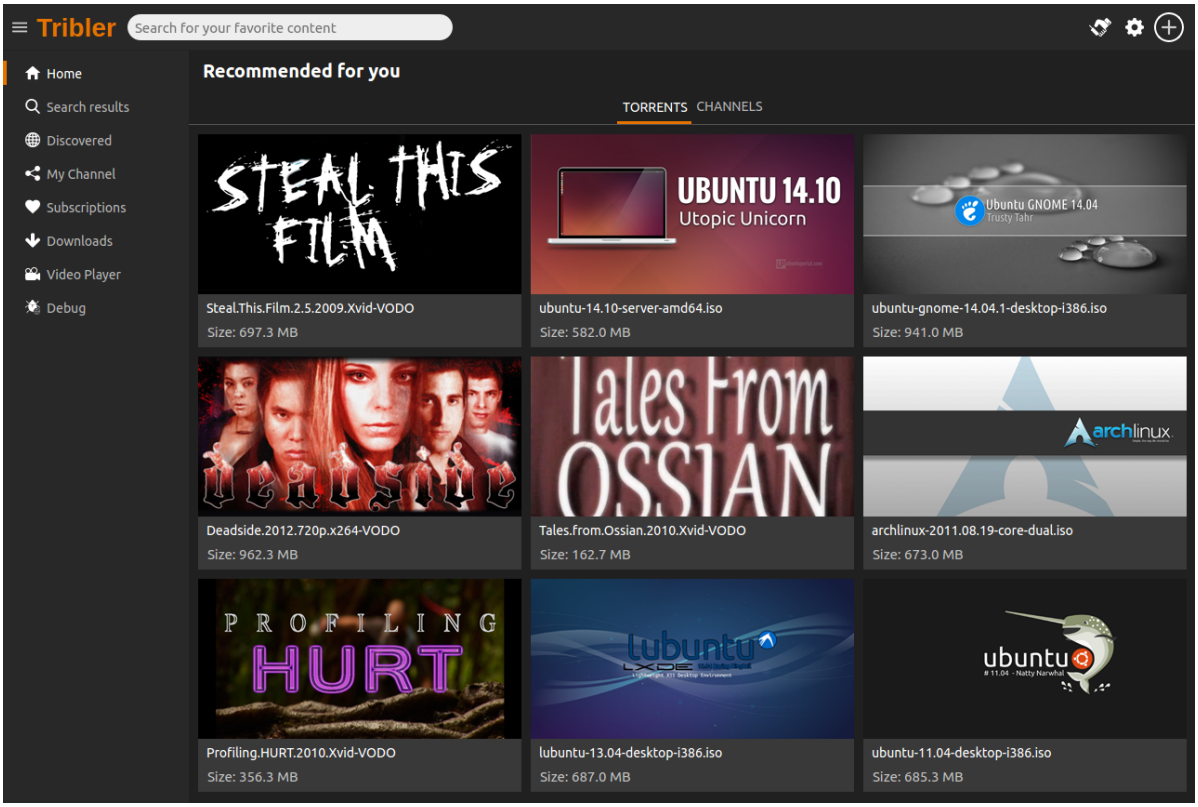



Figure 1.2: Tribler client showing channel content

Distributed systems that, by design, contain no central components are called decentralized systems. In decentralized systems there is no central component to wield authority and make decisions. Such systems must expect that conflicted states can arise, and have mechanisms to resolve these conflicts. Contrary to systems that contain a central component, decentralized systems are well aligned with the LFS ideals. In a decentralized system it is not possible to depend upon a central authority: every actor has to be its own authority. This leads to a very local view of the system truth, leaving it to the conflict resolution mechanism to resolve this with the rest of the network.

## 1.3. Tribler as LFS

Tribler [39] is an excellent example of a decentralized system that embraces the LFS ideals. Tribler is an open source BitTorrent client created by the Distributed Systems department of the TU Delft and has been downloaded by over 1.8 million users. For over a decade, Tribler has enabled researchers to investigate a variety of P2P topics including distributed content indexing and searching, video streaming and anonymous downloading. What makes Tribler unique as a research vehicle is the fact that it is used by thousands of real world end users. New features get put to the test in the real world, not just in academic benchmarks. Tribler already embraces the LFS ideals, most notably Tribler avoids reliance on centralized components and holds a completely local view on the system as a whole.

The *channels* feature of Tribler (shown in Figure 1.2) exemplifies how LFS can enable open source to compete with Big Tech. The channels feature allows users to create and publish content on their own channel, akin to YouTube. Tribler provides this functionality without global infrastructure. So Tribler brings Big Tech features, but without Big Tech. However Tribler's channels have a practical limit on the number of items they can contain. The current channels implementation cannot accommodate the scale needed by some of the larger channels. In addition to this, the channel contents is replicated between peers by gossiping a random sample from the subscribed channels. So obtaining a complete copy of, or searching in, a channel that a user is subscribed to, is not trivial. This is a situation where Tribler deviates from the LFS ideals since it does not provide an authorative local copy of the resource. The LFS go-to solution, a CRDT, is not trivial to apply due to the open nature of the Tribler P2P network.

This will be further discussed in Chapter 3.

## 1.4. Contributions

This work develops a CRDT primitive, BloomCRDT, with *set* semantics that has an affordable space complexity for workloads that include deletes, and that works in open P2P environments. BloomCRDT is by itself not a data structure capable of directly supporting large channels in Tribler, but it is a necessary first step. Composing multiple instances of BloomCRDT results in the Conflict Free R-Tree (CFRT), a key/value index structure with favorable scaling characteristics that works in open P2P environments, and is based on CRDTs. The design of both BloomCRDT and the CFRT is presented in Chapter 4 and an experimental evaluation of the designs is shown in Chapter 5. The experimental results validate the design and show that the CFRT robustly handles adverse network conditions and, as such is a candidate technology for integration into Tribler.

# 2

# Background & State-of-the-Art

A Conflict-free Replicated Data Type (CRDT) is an appealing primitive to use in designing scalable and decentralized applications. To this end, Section 2.1 introduces the distributed systems concepts necessary to understand CRDTs. Section 2.2 explains CRDTs, their formal definitions and some examples. Section 2.3 lists and discusses some basic and advanced implementations of CRDTs. Lastly, Section 2.4 discusses further research done on CRDTs and related research.

## 2.1. Distributed Systems and Consistency

In order to sufficiently comprehend the mechanisms and workings of CRDTs, some understanding of elementary Distributed Systems concepts is needed. Readers already versed in the core concepts of the CAP theorem and strong eventual consistency could skip ahead to 2.2.

### 2.1.1. The CAP theorem

Within distributed systems there is a well known theorem that bounds the capabilities of any distributed system: the CAP theorem [21]. The CAP theorem asserts that a distributed system cannot achieve *Consistency*, *Availability* and *Partition tolerance* at the same time. In this case Consistency should be taken to mean that the system should behave as if it where a single database [1]. Availability means that (user) operations don't block or wait until certain conditions are met, so the system should always be available for work. Partition tolerance means systems can recover from network interruptions. It is possible for a distributed system to achieve two of these conditions at any time, but not all three. An example of a system that gives up Consistency is the Domain Name System (DNS). DNS uses timers and caches when distributing data and thus achieves Availability and Partition tolerance. However it is possible that different results are returned to users depending on copies in intermediate caches. So the DNS system does not have strong consistency. Giving up Availability is exemplified by distributed locking, making some shared resources unavailable to prevent a lapse in Consistency. In addition, forfeiting Partition tolerance is a trade-off made by classical databases: they cannot recover from a loss of communication and, if networked, will typically be restricted to redundant locally networked clusters.

The author in [52] make some interesting observations about distributed systems that operate at internet scale. First off it is impossible to prevent partitions: links will need maintenance, smartphones will roam out of coverage, WiFi will be congested in urban areas, etc. So the system must absolutely be able to recover from partitions. Second, users nowadays have come to expect that systems are always Available, implying that Consistency should always be forfeited. While the argument for selecting Partition tolerance is certainly compelling, the argument for selecting either Consistency or Availability depends more on the application. Banks might well prefer Consistency over Availability in order to be safe from financial risks that could occur due to inconsistencies.

[1] An ACID [25] database that is.

### 2.1.2. Strong Eventual Consistency

The CAP Theorem deals with the concept of *strong consistency*, where all identical queries to the distributed system as a whole return the same result. This can be thought of as all nodes in the system having an equivalent state. However if, in the context of the CAP Theorem, strong consistency is forfeited it can be replaced with a lesser consistency. Usually this takes the form of *eventual consistency* [52]: a delay is accepted that allows the consistency to propagate through the distributed system. If all updates or modifications to the distributed system were stopped, then after some time every node of the distributed system should respond with the same result. The DNS system is a good example. After updates have stopped, and all caches expire, each identical request should result in the same records being returned. Note however that this is not because all nodes of the DNS system have reached an equivalent internal state. Unfortunately the definition of eventual consistency is somewhat imprecise and varies among authors. There are many more classes of consistency, interested readers are referred to [52].

A variation on eventual consistency is that of *strong eventual consistency* [45]. This definition starts with a weak form of eventual consistency where all nodes in a distributed system are informed about all updates, eventually. In addition to that the Strong aspect requires that all nodes apply all updates in such a way that the final internal state of the nodes is equivalent. Note that this is indeed stronger than the previous definition of eventual consistency since that placed no restrictions on the internal state of the nodes.

## 2.2. Conflict-free Replicated Data Types

In attempts to mitigate the results of selecting both Availability and Partition tolerance from the CAP theorem some solutions have been proposed. A recent paradigm is that of Conflict-free Replicated Data Type (CRDT) as described in [45]. The idea of a CRDT is to achieve strong eventual consistency of a replicated data structure by structuring data and updates in such a way that no conflicts can arise when combining different versions. This then allows automatic merging of different versions or updates of the data structure. In terms of the CAP theorem this provides Availability and Partition tolerance, but with a strong formal guarantee that strong consistency will be reached eventually.

Proponents of Local-First Software envision a central role for Conflict-free Replicated Data Types (CRDTs) in applications that strive for the Local-First ideals. This is because CRDTs naturally embody the ideals of Local-First Software. It allows users to directly collaborate, without a centralized infrastructure and where any conflict introduced by concurrent updates of users can always be resolved later. To aid adoption of the Local-First ideals several projects have developed CRDT implementations for browser environments [30] [23] [33].

The description of CRDT in [45] provides two formal models for reasoning about CRDTs: the state-based Convergent Replicated Data Type (CvRDT) and Op-based Commutative Replicated Data Type (CmRDT). The two models are equivalent in expressive power but CvRDTs are more convenient for mathematical reasoning and CmRDTs are easier to implement. The following subsections present the two CRDT types in more detail.

### 2.2.1. State-based CRDT

A Convergent Replicated Data Type (CvRDT), often called a State-based CRDT, exchanges the full CRDT state between replicas. The CvRDT model is based on a join-semilattice, in this case it means a partial ordering on the set of all states held by all replicas, and a function (known as 'join' or 'least upper bound') for pairs in the set. The join function produces a state that orders strictly greater than its two inputs and is commutative, idempotent and associative. This implies that as long as states can be ordered they can always be joined, and the result monotonically proceeds up the partial ordering chain. It must therefore converge to a maximal element in the partial ordering, one where all initial states have been joined, and because of the state equivalence relation of strong eventual consistency any maximal element will do. Although in practice the maximal element could well be singular, the greatest element of the partial ordering. The CvRDT definition of a CRDT permits testing of a data type to determine if it is a CvRDT, but it leaves a lot to the imagination when it comes to designing and implementing one.

To construct an implementation of a CvRDT, nodes send out a copy of their local replica state to other nodes. These other nodes then check the partial ordering and join states that contain "new" information. An example of a CvRDT is shown in Figure 2.1. This figure shows the operation of a

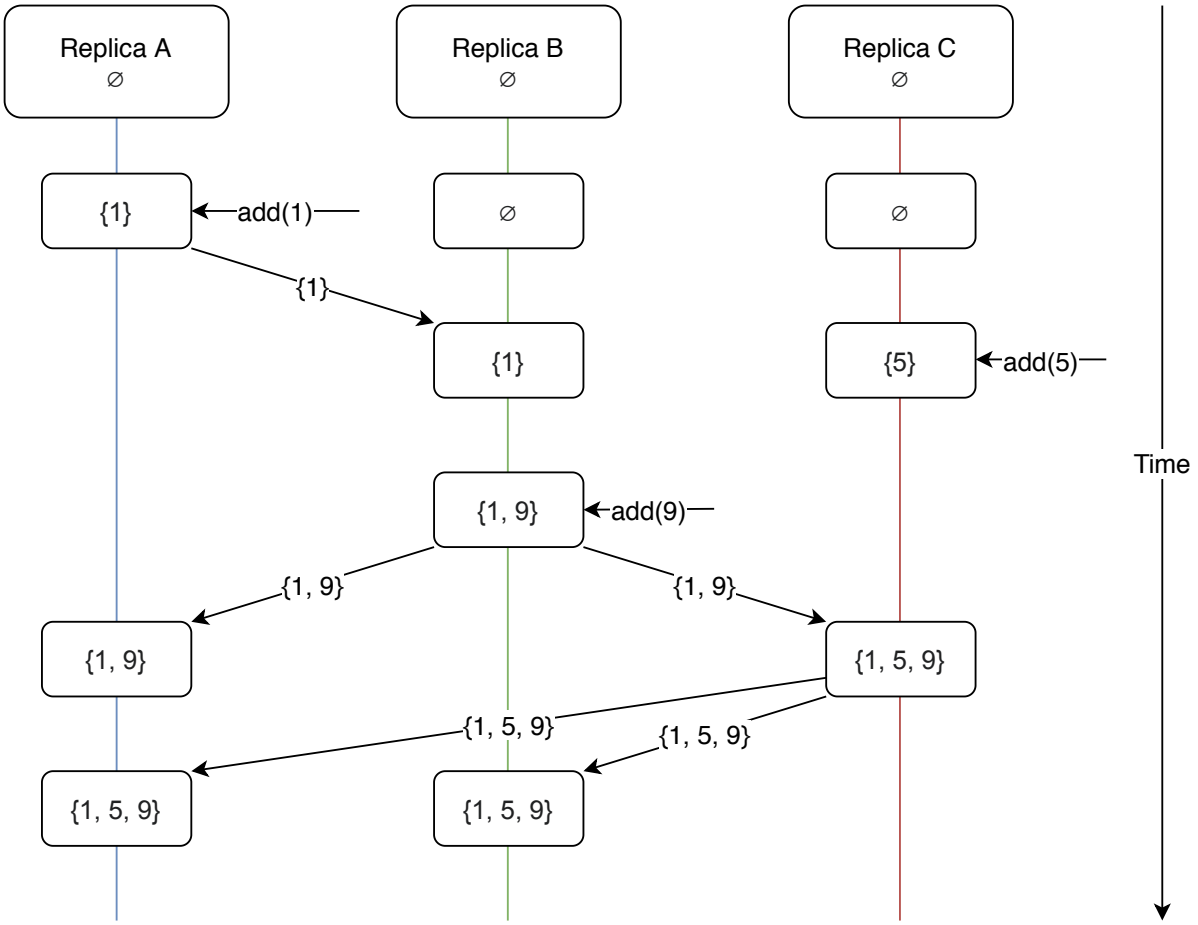



Figure 2.1: CvRDT operation. Replicas send full state messages to each other

CvRDT that contains a grow-only set. There are three replica nodes each with an initial empty state. Nodes then send their state to other nodes after an update, in this example nodes take the union of their local state and a received state. In this way all nodes have reached strong eventual consistency as defined by the CvRDT definition.

Note that ”taking an union” is a good example of why CvRDTs allow easy mathematical reasoning, but are not necessarily easy to implement. This requires comparing of two states to determine any differences and subsequently integrating those differences. Also, sending the entire state can be very inefficient: if the CvRDT state is large many bytes might be sent needlessly. These problems gave rise to a variation of the CvRDT, the $\delta$-CRDT [1], where only a difference of state is communicated between replicas. An example of a $\delta$-CRDT is TrustChain [38]. In this case a nodes state is the database of blocks that it knows about and new blocks are deltas on this state. Each node only needs to merge blocks it does not yet know about to advance its state towards a complete global state, so there is an ordering on states. Blocks can always be merged into the database independent of others, so the state of two nodes can always be merged. If all blocks where to be communicated to all other nodes, each node would end up with an equivalent (global) state.

### 2.2.2. Operation-based CRDT

The Operation-based style of CRDT known as a Commutative Replicated Data Type (CmRDT) is based around communicating the operations performed on the CRDT and having each replica apply these operations on their local state. The CmRDT model assumes a reliable causally ordered broadcast communication protocol and uses that to deliver a sequence of operations to all replicas. To each operation is bound a side-effect free precondition test that determines if an operation may be applied to the CmRDT’s state. If at a replica two operations are pending, i.e. their preconditions are satisfied,

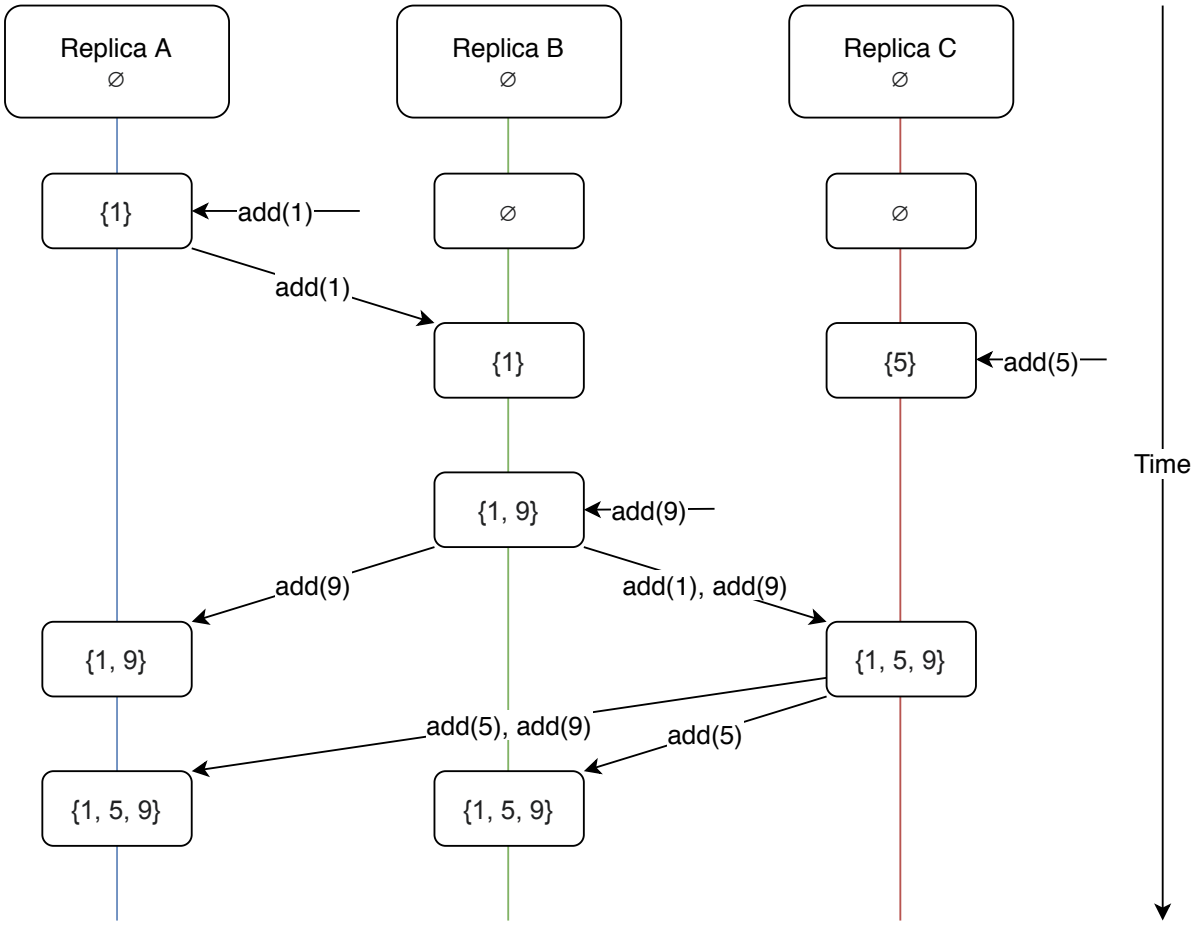



Figure 2.2: CmRDT operation. Replicas send update operations to each other

applying either operation may not invalidate the preconditions of the other. This allows operations to be applied in any order once their preconditions are met, or put another way: all concurrent operations must be commutative. This ensures that operations can always be applied eventually, and thus lead to an equivalent state at each replica. An example of a CmRDT is shown in Figure 2.2. This is similar to the CvRDT example but instead of sending a full state to other nodes, the operations are sent. Each node then applies the operations on its local state. This is also distinct from a $\delta$-CRDT which transports the change in state, that might not reflect the operation that was applied.

The CmRDT model is more suited to actual implementations since operations (e.g.: add, update, remove) on data occur naturally in programming and thus allow an easy transfer of theoretical concept to practical implementation. This is a sharp contrast with the CvRDT model where the whole state of a replica arrives at another node, which then has to be ”merged”.

## 2.3. Known CRDTs and implementation designs

CRDTs have attracted interest from both the academic community and from software projects. Described in Section 2.3.1 are some of the academic designs, and similarly Section 2.3.2 presents the work done on applying CRDTs to practical situations.

### 2.3.1. Basic CRDT Types

There are several known data structures that meet the definition of a CRDT: such as a vector clock, monotonic counters and add-only sets. Compositions of these basic data structures develops more advanced data structures. For example using two monotonic counters P and N it is possible to create a non-monotonic counter by computing P - N. But more complex compositions allow for sets, dictionaries

and directed graphs. Some of the basic CRDT data types are summarized in Table 2.1.

| Name | Description | Ref. |
|---|---|---|
| G-Counter | A monotonic increasing counter. | [46] |
| PN-Counter | Two G-Counters, one for positive, one for negative. The value of the PN counter is calculated as P - N. | [46] |
| G-Set | A grow-only set. | [46] |
| 2P-Set | Two phase set, akin to the PN-counter this set uses two G-Sets, one for items added and one for items removed (the tombstone set). The contents of the 2P-Set are the elements in the add set that are not also in the tombstone set | [46] |
| U-Set | A set that only adds and removes unique items. Combined with causally ordered messaging, this is enough to ensure no conflicts can arise. | [46] |
| LWW-element-Set | A set where the Last Write Wins in case of conflict. | [46] |
| PN-Set | A set where each element is paired with a PN-counter. Adding increments the counter, removing decreases the counter. An element is in the set if it associated counter value is greater than 0. | [46] |
| Observed-Remove Set | A set where elements are paired with a unique identifier. Before a remove can be issued the identifier needs to be observed, thus no concurrent add-remove conflicts can happen. | [46] |
| 2P2P-Graph | A graph made up of vertices in a 2P set, and edges in another 2P set. | [46] |
| Add-only Monotonic DAG | An add-only graph that uses a simple edge direction following rule such that a DAG is formed. | [46] |
| Add-Remove Partial Order | A DAG graph that uses a 2P-Set with tombstones for vertices and a G-Set for edges. This combination ensures a new edge can always be found between two vertices if an intermediate vertex is removed. | [46] |
| Replicated Growable Array (RGA) | List based on linked list. Allows updates to the elements in the list. Clocks that allow tombstone garbage collection. | [42] [46] |
| WOOT | List with unique identified elements. Tombstone set to filter out deletes. | [37] |
| Logoot | List with unique identified elements, insertion by generating identifier between two others. No tombstones but potentially unbounded identifier length. Claims that practical use sees no such unboundedness. | [53] |
| TreeDoc | List/document based on prefix Trie for element identifiers. Trie may become unbalanced and requires rebalancing, which uses 2-phase commit involving all replica's. Uses tombstones. | [40] |

Table 2.1: An overview of basic CRDT types

In Table 2.1 there are 4 common design elements that are used frequently when constructing CRDT data types. Firstly, there are monotonic operations, where the data only has one direction, such as the G-Counter, G-Set and monotonic DAG. Data only goes towards infinity and is never ”decreased”. This sidesteps combinations of concurrent add and remove operations that are not commutative. Secondly there is a frequent use of tombstones, special markers that indicate something used to be there but should be considered gone. This is frequently used in sets and ordered lists where concurrent updates could conflict. For example a `remove` and `addAfter` operation on the same element would conflict if the `remove` is processed first since the `addAfter` would then reference an invalid item. However

if the `remove` operation leaves a tombstone in the place of the element, the `addAfter` can still be processed. A third commonality is pairing list or set elements with unique or random identifiers. This can be an alternative to the use of tombstones in some cases. The principle is that removes must be causally ordered with respect to adds, since the paired identifier has to be observed first, before the remove of that particular element can be issued (see OR-set). Conversely a concurrent add of the same element produces two elements in the CRDT, each paired with an unique identifier. The fourth and last common design element is the use of clocks to mitigate the unbounded growth of tombstones in CRDTs. Often a vector clock is used since that allows a node to deduce if all other nodes have seen a particular tombstone and if so, discard such a tombstone from the CRDT.

### 2.3.2. Applied CRDTs

In addition to research on basic CRDT structures, some software projects have also applied CRDTs to real world situations. Table 2.2 lists some of them. Two types of application are typical among the applied uses of CRDTs. One is no-SQL or key-value store databases, and the second is collaborative editing.

| Name | Description | Ref. |
|---|---|---|
| Redis | Redis is an in-memory key-value store that can use CRDTs to implement multi master replication. | [41] [54] |
| Riak | Eventual consistent key-value store based on CRDTs. | [5] [13] |
| Roshi | SoundCloud uses Roshi, a CRDT that uses LWW-set in combination with garbage collection. Based in part on Redis. | [48] |
| Akka | Actor based programming language which uses CRDTs for its replicated data types. | [26] |
| Scalable XML Collaborative Editing with Undo | Applies CRDTs to XML document editing in a collaborative setting. Garbage collection using vector clocks. | [33] |
| Conflict-Free Replicated Relations for Multi-Synchronous Database Management at Edge | Applies CRDT to traditional RDBMs'. | [55] |
| Designing a Planetary-Scale IMAP Service with Conflict-free Replicated Data Types | Replicated maildirs over IMAP using CRDT to sync geo replicas. Uses a CRDT map/dictionary internally. | [27] |
| A Study of CRDTs that do Computations | CRDTs that result in a computation being performed in the CRDT state. Comparable to the join step in parallel processing. | [36] |

Table 2.2: An overview of CRDT applied uses

## 2.4. Other work related to CRDTs

There are many interesting works that further explore CRDTs, their limits and ways in which they could be improved. Such as [7] which aims to "Constraining the Eventual in Eventual Consistency". In order to achieve this leases are added to CRDTs such that operations can timeout and be canceled. This provides the same consistency but with a bound on the "eventual" part of the consistency, at the cost that some operations might eventually produce an error. There are also dead ends in CRDT research as explained in [29]. This work examines a common problem in collaborative editors, moving a range of characters. This operation turns out to be particularly difficult to capture in CRDTs since some combinations of concurrent operations are non commutative. In [6] the authors examine the problem of ever growing metadata in CRDTs, particularly unbounded growth of tombstones and operation histories.

The field of CRDTs is closely related to the much older field of Operational Transformation (OT). In OT as proposed by [18], concurrent operations on a replica are serialized to a predictable order,

and then applied to the state sequentially. After an operation is executed, all pending operations are adjusted to ensure they reflect an operation against the current state. In other words, the pending operations are transformed to work against an updated version of the state. The difficult part in the OT scheme is deciding the order of operations. Popular products that use OT such as Google Docs use a centralized component, a server in this case, to decide the order of operations. This ensures that all clients reach the same state. When the internal state is designed properly it is even possible to create a hybrid OT/CRDT system [34], thus allowing choice about the mechanism to use.

Also closely related to CRDTs are Cloud Types [14]. These are based on a similar idea of always allowing divergent versions of a value to be be recombined without much conflict. In the case a conflict does arise Cloud Types defer to a centralized replica (in the cloud somewhere) to handle arbitration. In contrast to CRDTs the use of a centralized component allows Cloud Types to use non-commutative concurrent operations. On the other hand this means that the replicas held outside of this master copy are not authorative. Cloud Types also use a variation of a vector clock to ensure values cannot be recombined to older versions of themselves. In short, Cloud Types are aimed at enterprise scenarios with centralized components, which is exactly what Local First Software tries to avoid.

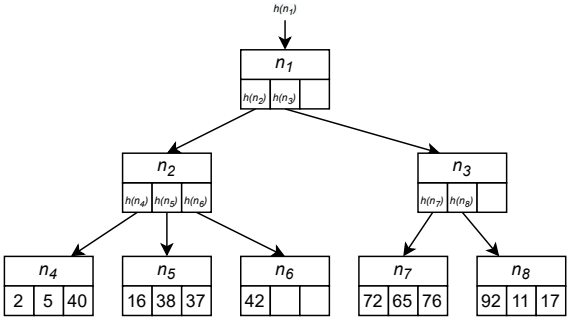
Another slightly more generic branch of research that is closely related is research on anti-entropy algorithms. The main idea here is to use an algorithm to synchronize peers, and thus reduce entropy in the distributed system. Conflicts are resolved using Last-Writer-Wins rules or are left unresolved as multiple versions of a value. The DottedDB [22] is a distributed key-value database that uses a double clock mechanism to garbage collect causality information that is no longer useful. To avoid unbounded grow DottedDB introduces a watermark set, a method for detecting what other peers know and discarding information is accepted by a quorum. This scheme is tolerant of peer churn, but use of a quorum indicates the peers are a limited set that are well connected.

Lastly there are CRDT designs Vegvisir [28] and Merkle-CRDT [43] which apply to low power IoT devices and IPFS respectively. Both build on the idea of using a Directed Acyclic Graph (DAG). Each operation on a CRDT is represented by a node in the DAG, and each such node has an edge directed at one or more previous operation-nodes. The edges of the DAG thus encode the causal relation of the updates and are sufficient to allow CRDTs to work on this.

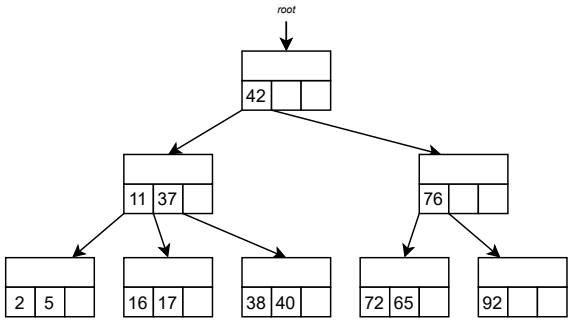
## 2.5. CRDTs embedded in Merkle Trees

Merkle Trees are data structures that can be used to efficiently find differences in large sets of elements. The idea begins with hashing each element, then a certain number of these hashes are concatenated and hashed again forming the leaves of a tree structure where each level towards the root repeats the concatenation and hashing (as depicted in Figure 2.3a). The root of a Merkle Tree is a single hash that identifies this particular tree. With this scheme any change in an element produces a different hash for all its parents, up to and including the root. This way a hash can compare subtrees or indeed a whole tree with one message exchange. Many practical applications use Merkle Trees including ZFS, bitcoin, git and Riak to name a few. While a Merkle Trees can detect detect differences in large element sets, it does not provide direction over a key space, a Merkle Tree is not an indexing structure.

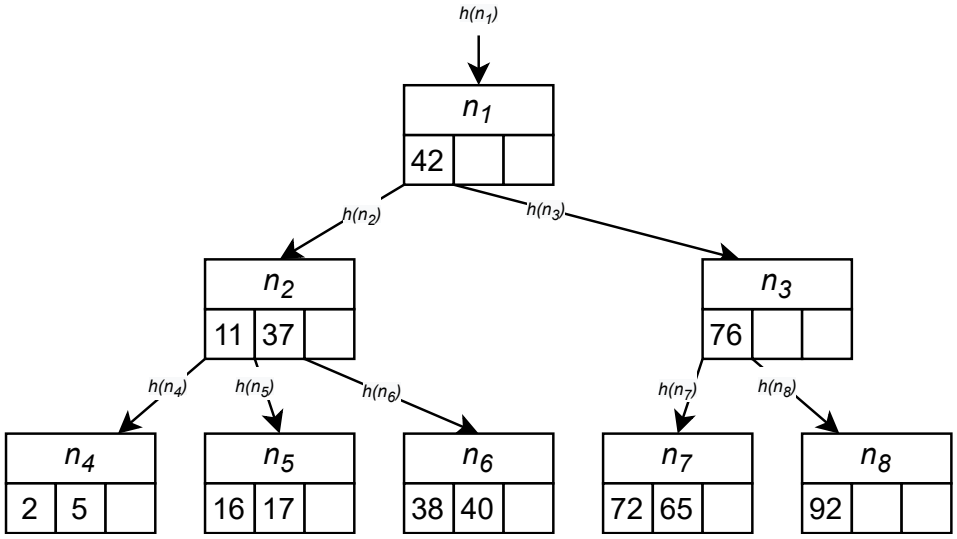
The Merkle Search Tree (MST, [4]) aims to unify Merkle Trees and indexing structures. Since a Merkle Tree is indifferent to exactly how the children of each node are selected, the MST can exploit this freedom and proposes to compose elements into a tree structure similar to a B-Tree [9] (see Figure 2.3b). It starts by ordering all elements and hashing them. Each element is then assigned a level (height or distance from leaf layer) based on the number of leading zeros in its hash. This height combined with the ordering results in a tree structure, to which the Merkle Tree concept can be applied (see Figure 2.3c). Replicas of the MST can be compared using the techniques developed for Merkle Trees, in this case gossiping of the current root hash and subsequent rounds of obtaining missing nodes/hashes. Of special note is that the MST is formed deterministically from the set of elements. Given the same set of elements, each replica will form an identical tree. This ensures there is no infinite variation on hash due to different causal paths, leading to a converging state and an eventually consistent MST. The MST proposes using key/value pairs as entries and using CRDTs in the leaf values to resolve concurrent updates, much like Riak and Redis. The paper presenting the MST hints at, but is not clear on, the mechanism of removing elements from an MST. After inquiry with the first author [3] it is clear that the MST supports deletes in a limited capacity. Any local delete would be considered a missing value in any subsequent gossiping round and be restored. Thus the best the MST can offer is a special




(a) Example of a Merkle Tree. Concatenating node contents and hashing repeatedly to create a tree. Terminating in a single root hash. Note that the leaf nodes do not need any specific ordering to construct an Merkle Tree. In practice however it is usually favorable to have an ordering since the Merkle Tree expresses a permutation of elements and ordering reduces that to a combination of elements.




(b) A B-Tree [9] is a generalization of a binary search tree that allows more than one value in each node. Entries are added to leaf nodes that split when they become full. Because splits propagate up from the leaf layer and creates only siblings, the tree is kept balanced. Adding a new layer at the root when needed.




(c) The Merkle Search Tree combines the Merkle Tree with the B-Tree and allows searching, but also efficient comparison of (sub)trees.

Figure 2.3: The Merkle Search Tree combines the concepts of a Merkle Tree with those of a regular indexing structure. $n_x$ indicates a tree node and $h(n_x)$ indicates the hash of a node's contents and pointers.

tombstone value in the key/value pair, making it a grow-only structure.

# 3

# Problem Description

The previous chapter describes the model of a CRDT, and its properties. The sections on applications show there is no significant use of CRDTs in open P2P systems. Most applications are collaborative editors for users and key/value data stores. These are controlled situations with a limited number of replicas where new replicas need permission to join and all replicas are informed about each other. This in contrast to open P2P systems and hinders the ideals of Local First software. This chapter describes the problems that arise when applying CRDTs to open P2P systems. Section 3.1 describes these problems in three subsections. Next, Section 3.2 describes how these problems impact a real world application.

## 3.1. Limits and Problems of current CRDTs

Explained in the next sections three inherently limiting issues are explained that, at minimum, hinder a wider adoption of CRDTs in open P2P systems. This section uses Git [15] as an example since this software is very familiar to most readers and displays many characteristics of CRDTs[1].

### 3.1.1. Data structure Scalability

In theory, the basic definition of a CRDT does not place any hard limits on state size. However, when applying theory to practice there will always be real world considerations. Some authors have found that the size of the CRDT's state was a topic for future research [6] [30], as it is a fundamental limitation on CRDT applicability and adoption. In Git, an example would be many or large binaries being added which leads to a large working copy. In many CRDTs a similar issue occurs with grow-only structures, and more specifically tombstones inflating the CRDTs state. In Git it suffices to delete the binaries to reduce the size of the working copy. However, the grow-only nature of many CRDTs means it is not possible to simply remove tombstones without introducing conflicts. So CRDTs that rely on tombstones require other solutions to keep their state manageable. One solution is to ignore the problem if the CRDT is of an ephemeral nature. Since most existing applications of CRDTs are in collaborative software, it is reasonable to assume that the collaboration will cease eventually. If the CRDT's state has not grown so much that it becomes impractical then there is no problem to solve in practice.

If the CRDT is not ephemeral, but of a more persistent nature, ignoring a grow-only state or the state size in general is a strategy that will fail eventually. One solution that has been suggested is a distributed garbage collection scheme to remove tombstones that no longer serve a purpose. As briefly discussed in Section 2.3.1, the go-to solution is to use a vector clock and attach a timestamp to each tombstone. This allows each node to reason about the causality at other nodes. Once it can be inferred that all replicas have seen the tombstone, it can be removed safely. The problem with a vector clock is that it:

[1] In fact Git could be considered a CRDT itself. It has states that can be merged towards eventual consistency. However in such a view of Git the join function is a human that resolves conflicts. Git itself is not aimed at being conflict free, just being able to resolve the conflicts that happen. Furthermore, the use of a human to do something that pure mathematics can not is where the uncertain status of Git as a CRDT originates. In essence the join function is an oracle machine; useful for mathematical reasoning but not very practical.

1. **Requires knowledge about the existence of other replicas.** When replicas are not know to each other, they are not included in the vector clocks. This prevents replicas from correct causal reasoning about operations.

2. **Limits the number of replicas in practice.** If many replicas join, then a vector clock becomes unwieldy and timestamps grow in size. A simple calculation shows that a standard Ethernet frame would overflow with just 94 replicas, assuming 64-bit counters and identities.

3. **Grows as replicas come and go.** Since it is impossible to tell what replicas are partitioned from the network and what replicas have left the network, the only safe option is to assume a partition has happened and thus save all required metadata (see also Section 3.1.2).

To demonstrate how infeasible the standard vector clock is in an open P2P setting consider trying to impose a vector clock on the Linux kernel git repository. The first step would be to discover, world wide, all replicas of the Linux kernel repository. If there are many, then the clock state will become large and require special consideration in software. If all replicas have been found, the clock must keep track of the clock states of all replicas, even of replicas that have been removed. Since removal of replicas is indistinguishable from a network partitioning and there is no limit on the duration of the network partitioning, their clock states should be held indefinitely. As such a vector clock on the Linux kernel Git repository would become a grow-only data structure. Recurs to Section 3.1.1 to read about the problems of grow-only data structures.

The problems the vector clock encounters in the above scenario stem from the openness of Git: anyone can come along and decide to create a replica without being forced to register this replica. Previous works on CRDTs have either been theoretical, or have applied CRDTs to controlled settings where the participants in the distributed system are known, reasonably limited in number and no churn occurs. This means that CRDTs have not found their way into P2P distributed systems, even though CRDTs would seem to be a natural fit for P2P networks at first glance (some notable exceptions being Vegvisir [28] and Merkle-CRDT [43]).

### 3.1.2. Metadata Scalability

CRDTs consist of two types of data: application state is used by the application built on top of the CRDT, and metadata is needed to make the CRDT function but has no direct value to the using application. Git also clearly shows this distinction: the working copy can be considered an application state while the commit history is metadata kept by Git to make the users workflow possible. Just as unbounded growth of the application state is problematic, so is unbounded growth of a CRDT's metadata. Op-based CRDT implementations are a prime example of this unbounded metadata growth problem because they require that all operation messages contain causality information. This is not a problem by itself but usually this causality information is only informative in the context of the complete history of operations on the op-based CRDT. In other words, operations refer to the operations that came before them. Also, keeping the full history of a CRDT ensures that a replica recovering from a network partition can be supplied with the operations it still has to perform.

At first glance state-based CRDTs should not have the issue of grow-only metadata, since state-based CRDTs do not require a causal ordering. However, implementations may require this anyway in order to determine the join of two states. The state-based CRDT needs to determine what state is newer, or has components that have not yet been observed. In terms of Git an example would be a merge (or rebase), that requires a common ancestor state to determine how two states have diverged. Changes since such a common ancestor state can then be compared and a joined state emerges. The definition of the state-based CRDT is much looser than this Git example would suggest. Nonetheless, it highlights that even state-based CRDTs can be reliant on persisting metadata of previous states.

### 3.1.3. Forced Convergence Assumption

A fundamental assumption of many CRDT implementations is that they aim for an active full state synchronization among peers. For the small and closed communities that most CRDTs target this works well. However, there is no requirement in the definition of CRDTs that replicas are *forced* to converge. Put another way, the delay after which eventual consistency is reached could be at infinity. CRDTs where initially designed for applications with the assumption that it is desirable to actively converge all replicas to a globally equivalent state. This is what many projects have implemented and indeed this

has so far proven a good fit for collaborative editor applications and key-value databases. However, this *forced convergence assumption* has some disadvantages. Most importantly, a replica might not need the complete data structure in order to function. This is a key design insight that allows for example ConTrib [16] [2] to function efficiently. ConTrib expressly avoids globally distributing block chains. Secondly, CRDT applications that aim for an actively converged state assume that all replicas know about each other. This allows CRDT implementations to push new states or operations to other replicas, forcing them to converge towards a globally equivalent state. However, as with vector clocks in Section 3.1.1, keeping track of all replicas is not trivial. Lastly, the forced convergence assumption does not consider replicas on heterogeneous hardware. If a receiving replica is low-powered, then new updates that are pushed towards this replica force it to use a lot of power to keep up. That is, if the low-powered device is capable of keeping up at all. Especially in the case where only a small fraction of the whole data structure is needed, the burden on this replica might be disproportional compared to the benefit for the user.

## 3.2. Applying CRDTs to open P2P systems

As was hinted at in Section 1.3, CRDTs do not translate easily to open P2P networks. The previous section makes clear why this is the case. To get an idea of the infeasibility of using existing CRDT implementations for Tribler consider the following. Each torrent consists of: a name, a magnet link, tracker info, thumbnail(s) and other metadata. On average this is amounts to about 2KB per torrent. Since the release of Tribler's channels feature some channels have grown beyond a million torrents. An estimate for the application state size of a CRDT supporting such a channel would be 2GB at minimum. On top of that, any remove operation would leave a tombstone and since the channels are more permanent than ephemeral, the tombstones would eventually dominate the CRDT. In the case of a single pure state-based CRDT, this would send the full state to other peers to communicate a single change. However the alternatives, either an op-based or a $\delta$-CRDT, also have problems. They require causal message delivery or intricate vector clocks. Both of which are easily susceptible to infinite growth in a P2P system with dynamic identities over time. Lastly, it is hard to imagine that any user would have a need for every single torrent in such a channel. Therefore, its reasonable to assume each user is only interested in a small subset of the full channel. Thus if such a large channel would be implemented as a CRDT, the forced convergence assumption would force users to acquire the full application state and metadata of the CRDT and to keep up with any changes. This is a heavy burden for a user that is only interested in a small subset of the full state. Since all of the problems described in Section 3.1 apply, it is clear that applying a CRDT to the Tribler open P2P network is not trivial and warrants further research.

Even though current CRDT implementations are problematic in open P2P systems, the idea remains that a CRDT would be perfectly suited to service Tribler's channels. In essence, Tribler's channels are collaborative adds and removes on a set. So the natural question would be: does a CRDT exist that could support Tribler's channels? Or more generally, a CRDT that can work in an open P2P system. If such a CRDT does exist then it should meet these criteria:

1. **State-based CRDT** Since gathering any causality information is going to be be difficult in an open P2P setting, causal message delivery and vector clocks are not possible. This directly rules out op-based CRDTs and $\delta$-CRDTs. Causality information will have to be encoded into the data structure, which is a natural fit for state-based CRDTs.

2. **No unbounded growth** In both application state and metadata there should not be unbounded growth since this is not compatible with the persistent nature of Tribler's channels.

3. **Not push based** A peer should be in charge itself about what information to fetch and at what time. This allows for targeted retrieval for those peers that need just a subset of all information, and prevents low-powered nodes from being overwhelmed by other peers.

[2] The ConTrib structure can be interpreted as a $\delta$-CRDT, on the condition that all nodes behave correctly. The state of a replica in this case is the local database of blocks that a replica holds. There is only one update function: `addBlock`. Which translates to a $\delta$-state message containing the new block. This message is distributed to other replicas and merged into the state (persisted in the database). The merge operation always produces a consistent new state that is further along the path to global convergence. Thus it has all the attributes of a state-based CRDT and is an example of a CRDT that does not force convergence of replicas.

4. **No persistent peer knowledge** Any solution that permanently stores (meta)data about other peers will show unbounded growth since P2P networks are subject to a constant influx of new peer identities. Criterium 2 prohibits unbounded growth, thus it is immediately obvious that any solution must not keep persistent peer knowledge.

# 4

# The Conflict Free R-Tree

Two concepts are presented that overcome the problems described in the previous chapter. First, BloomCRDT an innovation on the standard OR-set that vastly reduces the unbounded growth and operates in an open P2P environment. Second, the Conflict Free R-Tree is a composition of BloomCRDTs into an index data structure that scales far beyond what a single BloomCRDT could practically contain.

## 4.1. System model and assumptions

The previous chapter describes the problems faced by CRDTs in open P2P systems. These problems are modeled by the following assumptions:

- Message delivery is imperfect, messages may arrive out of order, duplicated or not at all. This is because real world IP networking is not perfect and these events happen, so any design should account for this.
- Peers and network links may fail, even leading to network partitions. This is because hardware will fail in real world scenarios, again systems should be designed to account for this. Intentional failues or manipulation of the network are out-of-scope for this work.
- Over time, large numbers of peers may join and leave the network, and interact with a BloomCRDT or CFRT instance. This is a natural feature of open P2P systems that has design implications in the case of CRDTs. As such it is important to explicitly acknowledge this assumption.

Moreover, there are several basic communication features on which BloomCRDT and CFRT are based. It is assumed that there exists a communication library such that:

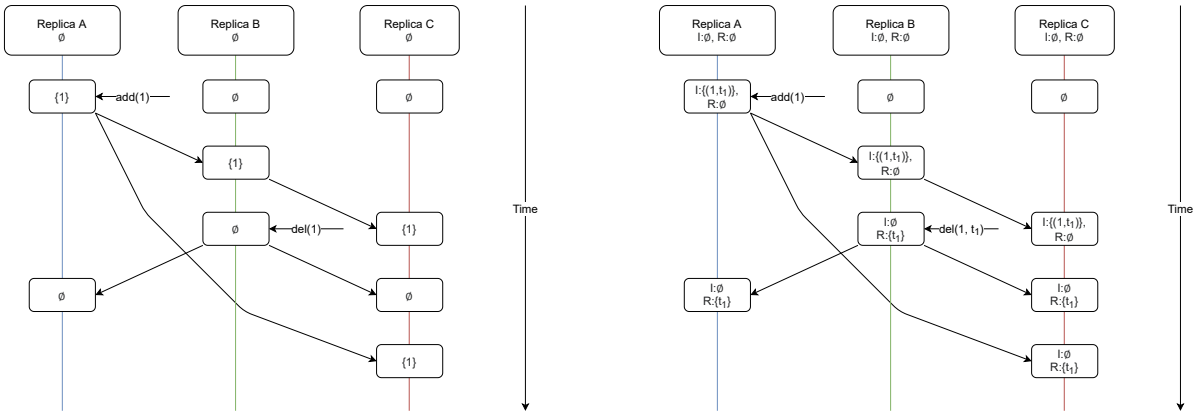
- Messages are checked for tampering using cryptographic signatures. For example a system where a peer's identity is a public key which is used uses to verify messages.
- Groups can be resolved from global identifiers. For CFRT it is necessary to form identifiable groups of peers that exchange messages. Knowing the ID of a group should allow a new peer to join that group.
- Each peer can message a (small) subset of other peers in joined groups.

## 4.2. BloomCRDT

Presented here is BloomCRDT, a novel CRDT that provides set semantics in an open P2P environment. BloomCRDT is a state-based CRDT that behaves like a set and is based on the traditional OR-set [46]. BloomCRDT does not rely on causal message delivery nor does it need knowledge of other peers or their state, and finally does not show unbounded growth over time in practice. There appears to be no CRDT described in related work that matches these characteristics.

### 4.2.1. The Observed-Remove Set

BloomCRDT is a modification of the Observed-Remove set (OR-set), which should be explained before presenting the modified version. An OR-set tags each added element with a random number or *tag*, this effectively makes each element unique and unpredictable. Internally, the OR-set consists of a grow-only set of inserted elements (denoted $I$) and a grow-only set of removed elements ($R$). The contents of the OR-set from the user's perspective is the relative complement $I \setminus R$. New elements ($e$) are paired with a tag ($t$) and added to $I$. To remove an element, the $(e,t)$ pair is added to $R$. Since $t$ is unpredictable a replica must first *observe* a $(e,t)$ pair in $I$ before the pair can be added to $R$ to remove it from the OR-set, hence the name Observed-Remove set. The required observation step ensures any remove must causally follow the add.




(a) Traditional set demonstrating the overtaking delete problem.

(b) OR-set solution to the overtaking delete problem

Figure 4.1: The overtaking delete problem

To emphasize the problem that the OR-set solves, consider the $add()$ and $delete()$ operations on a traditional set. In the case of state-based CRDTs that have non-causal message delivery, messages can get reordered as demonstrated in Figure 4.1. So the message that signals the deleted state might arrive before the message that carries the added state. This is known as the overtaking delete problem [35]. In this case Replica C observes the delete of an item before it observes the add. As shown in sub Figure 4.1a a traditional set is not guaranteed to be eventually consistent. Sub Figure 4.1b shows how the OR-set solves the problem of an overtaking delete. In the last step at Replica C, the $R$ set reveals that the element was already deleted.

Some research has been done to improve upon the OR-set, mostly to address its grow-only nature. The Optimized Conflict-free Replicated Set [10] proposes the OptORSet, a modified OR-set that includes information about the state of other peers. This allows reasoning about what state has been propagated globally and thus what state can be discarded. To do this the OptORSet uses causal message delivery and keeps a per peer state. These concepts are not suited to an open P2P environment. The Optimized OR-set Without Ordering Constraints [35] improves on this because it does not require causal message ordering and makes a compelling argument for its interval version vectors. However, this solution also uses per peer state. In the open P2P environment this will show unbounded growth over time.

### 4.2.2. Structure of the BloomCRDT

In an OR-set, $I$ does not actually need to be a grow-only set. Once a $(e,t)$ pair is in $R$ it can be removed from $I$. The relative complement $I \setminus R$ will still compute the OR-set contents. However there is no obvious way to remove the grow-only aspect of $R$. In a more theoretical sense, $R$ provides information for the join function such that the result state is ordered greater on the join-semilattice than either input state. Or viewed another way, $R$ encodes the history of an OR-set such that a join function can move forward and will not regress. In $R$ it is especially the unique tags $t$ that are of interest, since those are what make the elements unique and force the observed relationship. If only there where a space efficient method to encode set membership of many elements without having to persist the members.

Luckily Bloom filters [11] can encode set membership without having to persist the members and

are a suitable replacement of the $R$ set in an OR-set. Bloom filters start as a list of bits, each set to 0. When an element is added to the Bloom filter a number of hashes is computed on the added element, and each hash function produces an index in the Bloom filter's bits. At the calculated indices the bit is set to 1. To test if an element is in the Bloom filter a queried element is hashed with the hash functions and if the bit at any of the computed indices is zero, then the queried element is not in the Bloom filter. Else it most likely in the Bloom filter, however it could also be a false positive if other elements flipped the relevant bits. The idea is to choose the number of bits and number of hash functions in such a way that the probability of a false-positive is small enough for the application.

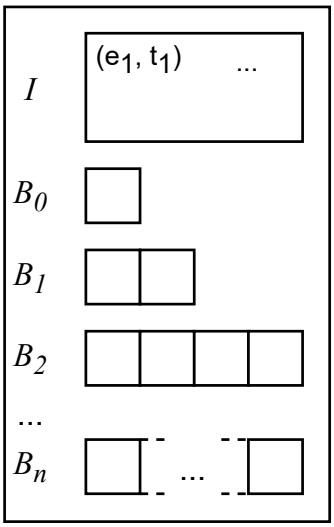



Figure 4.2: Structure of a BloomCRDT. Each entry is tagged with a unique tag. One or more Bloom filters of increasing size are added as needed.

Using a Bloom filter, it is possible to change the OR-set into a BloomCRDT. To do this $R$ is replaced with a Bloom filter, and coupled with the previous observation that $I$ need not be grow-only, the BloomCRDT is constructed (see Figure 4.2). While this is the basic premise of the BloomCRDT, there are some details that require further explanation.

- **Bloom Filter alternatives** Instead of a Bloom filter there are more recent alternatives to consider such as Cuckoo filters and XOR filters. However there are specific conditions that the $R$ replacement must handle. For example, Cuckoo filters require storing fingerprints of elements in the set which would not amount to a net savings since the $t$ tags are similar to the required fingerprints. The $R$ replacement should result use less space compared to the size of $R$ as a grow-only set. The more recent XOR filters are built from all elements prior to filtering and are not updatable. But $R$ is not something that can be computed a priori, it updates over time. This covers the two most frequent reasons other techniques cannot be used. In addition to this, the $R$ replacement must be able to join efficiently as will be explained in 4.2.3. In the end, Bloom filters are very simple and can show a net space saving over $R$ after adding just a few entries.

- **False Positives** A major problem for Bloom filters is their probabilistic nature: there can be no false negatives but there is a chance of a false positive. When applied as a replacement of $R$, this means there is a chance that an element is falsely considered as removed. This is only a problem in exceptional circumstances[1]. Moreover Bloom filters allow the implementer to select the probability of false positives which should be customized to fit the risk of the application. The specific conditions required for a false positive to adversely effect the BloomCRDT coupled with the low probability of actually creating a false positive in the first place, means this event can reasonably be excluded during further discussion.

[1] Elements are only checked against the Removed Bloom filter during a join, and only if they are only in one of the two input states. Thus it requires a removal in one replica that creates an update in the Bloom filter that collides with a concurrent addition that is still propagating among replicas.

- **Parameter selection** The only further consideration for Bloom filters is that they require an a priori estimation of the number of elements that will be added in order to guarantee the probabilistic bounds. This presents a problem for long lived BloomCRDTs, the number of elements added to the Bloom filter cannot be fixed in advance. To address this, BloomCRDT actually uses a list of Bloom filters ($B_i$ where $i$ is the index in the list of Bloom filters). When it is estimated that a Bloom filter is nearing saturation, a larger Bloom filter is added to the list of Bloom filters. This unfortunately reintroduces an unbounded growth in the theoretical sense, albeit with a much reduced magnitude. Section 4.3.3 discusses the composition of BloomCRDTs into a CFRT and includes a mitigation strategy for this potentially unbounded growth.

### 4.2.3. Joining two BloomCRDTs

The traditional OR-set has a simple algorithm for the CRDT join function, but for BloomCRDT this is slightly more complicated. Assume that a join function has two inputs named $l$ and $r$, and an output state named $j$. The traditional OR-set calculates its new states as $I_j = I_l \cup I_r$ and $R_j = R_l \cup R_r$. The BloomCRDT join is slightly more complex and must consider joining the Bloom filters and a correct handling of the difference in $I_l$ and $I_r$. The Bloom filters of $j$ can be computed as $B_{n,j} = B_{n,l} \mid B_{n,r}$ where $B_{n,.}$ is considered as all zeros if the index is undefined. Since Bloom filters only change their bits from 0 to 1, a bit wise-or suffices to combine them. Initially $I_j = I_l \cap I_r$, and each element in $I_l \vartriangle I_r$ that has a tag that is not in $B_j$ is also added to $I_j$. The idea is that if a $(e, t)$ pair is in one of $I_l$ or $I_r$, then it is either a new element or it was removed. Since removed elements should be present in $B_j$, they are omitted from $I_j$. The bit wise-or of two Bloom filters implies the condition that both Bloom filters are based on identical parameters. So when growing the list of Bloom filters, there should be a deterministic process to set the parameters of the next Bloom filter. This ensures that when two or more replicas concurrently add a new Bloom filter, the new Bloom filters are compatible for merging. A simple strategy could be to double the capacity compared to the last Bloom Filter.

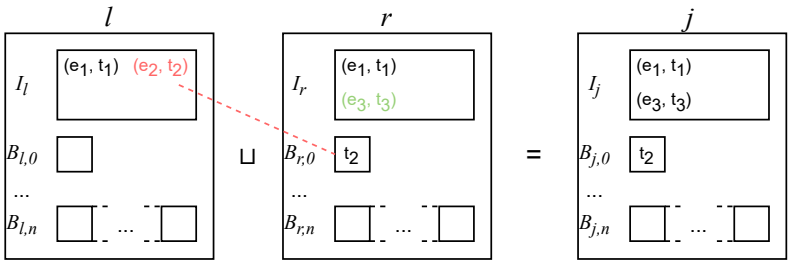



Figure 4.3: Example of joining two BloomCRDT states.

An example of a BloomCRDT join is shown in Figure 4.3. Here BloomCRDTs $l$ and $r$ are joined and result in BloomCRDT $j$. The symmetric difference $I_l \vartriangle I_r$ reveals that $(e_2, t_2)$ and $(e_3, t_3)$ are not in both states and should thus be checked against the Bloom filter. In this case $(e_3, t_3)$ is a new element since its tag is not in the Bloom filters and its pair is thus added to $I_j$. However $t_2$ is in the Bloom filter, so its pair is omitted from $I_j$. The resulting state $j$ captures all information in both input states and is thus a join in the CRDT semi-lattice sense. The state $j$ orders greater on the semi-lattice than both $l$ and $r$.

With the join algorithm as described, there is a complicating factor that need to be considered when using BloomCRDT. If many deletes are being processed concurrently it is possible that a subsequent join pushes a Bloom filter over its designated capacity. Thus each BloomCRDT should have a soft limit on the number of elements in each of the Bloom filters. The soft limit should be the floor of designed capacity of the Bloom filter minus the global rate of distinct elements removed multiplied by the time to global convergence. For example, if a Bloom filter is designed to hold 20 elements, the global removal rate is 0.2Hz and the time to global convergence is 17 seconds, then the soft limit would be $\lfloor 20 - 0.2 * 17 \rfloor = 16$. When reaching this soft limit a BloomCRDT should stop adding to the Bloom filter and add a new one, with the idea that the remaining space in the Bloom filter could be filled up by deletes that are still propagating. Estimations of the variables are obviously implementation dependent, and could be set at design time or set at run-time dynamically. Moreover exceeding the designed capacity of a Bloom filter is not automatically a problem it only slightly increases the chance of a false positive.

## 4.3. The Conflict Free R-Tree

The Conflict Free R-Tree (or CFRT) is a novel data structure that is composed of many BloomCRDTs, allowing it to scale far beyond what a single BloomCRDT could practically contain. The BloomCRDT behaves much like a typical set with linear computational complexity on insert, lookup and remove operations. To support larger datasets a lower complexity is required, which implies ordered elements. There are several possible directions that could work. For example BloomCRDTs could be composed to form a DHT, or a skip list. However, when composing BloomCRDTs the most difficult part is correctly maintaining the links between the BloomCRDTs. A classic index tree structure provides ordering on the contained elements and a low number of links compared to other solutions.

### 4.3.1. The R-Tree

The R-Tree [24] is similar to the well known B-Tree, as used in Merkle Search Tree described in Section 2.5, but has different characteristics. The R-Tree is an index on a key space $\mathbb{K}$ such that it can efficiently $add(k)$, $lookup(k)$ and $remove(k)$ where $k \in \mathbb{K}$. The central idea is to build a tree that labels its edges with a range on $\mathbb{K}$ that indicates the range of keys that can be found in the sub tree that the edge points to. Walking the tree from root towards leaf nodes should result in an ever narrower range. The R-Tree nodes typically contain many edges towards children, but the fan out factor is implementation dependent. Figure 4.4 shows an example of a R-Tree.

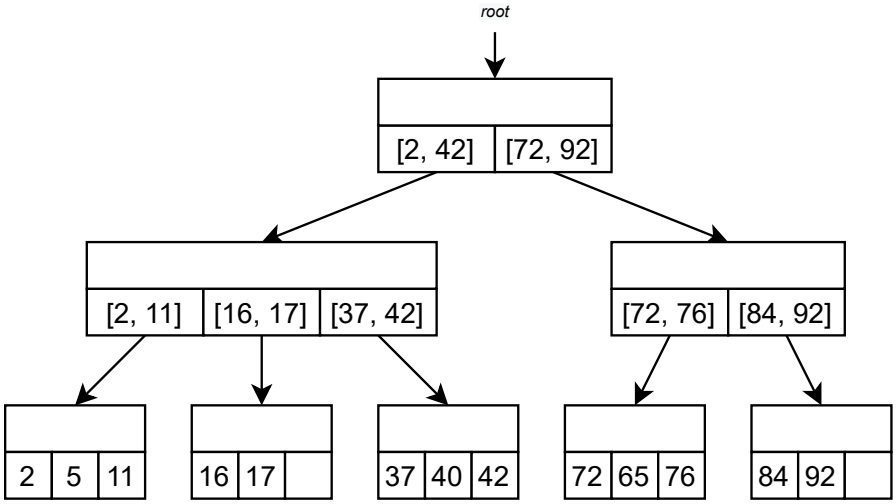



Figure 4.4: The diagram shows an example R-Tree where each edge is labeled with a range to indicate the range of keys that are contained in the child .

The R-Tree starts as a single node. Entries get added and at some point the node decides it is too full and needs to split. So there is a limit on growth after which a node will be considered full. When a node detects that it is full it splits into two nodes and informs the parent of this change. The parent then adds an entry to register the new child. This in turn can cause the parent to become too full and split. This splitting can proceed up the tree all the way to the root node. If the root node splits, it creates a new node that becomes the new root of the tree and the parent of the old root. The inverse of splitting can also happen: if a node contains only a few entries, it can be merged with a sibling. A sibling is selected and the entries are transferred to this sibling. The node is removed and the parent is updated to remove the corresponding child entry. If the root node has only 1 child, that child becomes the new tree root and the old root is removed. If enough entries are removed, the tree can collapse all the way down to a single node.

This scheme is very robust. It does not matter *how* a node is split, its entries could be randomly distributed over the old and the new node and the R-Tree will still be consistent. Similarly the sibling selected for merging does not matter, any sibling is valid and results in a consistent tree. The cost of this imprecise work is efficiency. The random split and merge strategies result in children overlapping the range of the parent almost fully. This requires a lookup to visit most of the nodes in the R-Tree, effectively a complicated scheme for sequential scan. The average fraction of overlap between children

is a well known (and studied) performance characteristic of R-Trees. Any overlap that can be avoided will improve the efficiency of a lookup.

The mechanic of splitting and merging is all very similar to a regular B-Tree so what justifies the storage cost of ranges on $\mathbb{K}$ as opposed to just elements from $\mathbb{K}$? The R-Tree can, in contrast to the B-Tree, contain children whose ranges overlap without affecting the consistency of the tree as a whole. In a B-Tree any key in an interior node serves to direct algorithms that traverse the tree to one of the children at either side of the key. This implicitly bounds the key space range of children, but cannot express children with overlapping ranges. The R-tree explicitly tracks the range of children and is thus able to express overlapping children, at a cost in storage since each child requires two bounds from $\mathbb{K}$.

### 4.3.2. Structure of CFRT

After adding a few thousand elements the BloomCRDT becomes very inefficient. By the nature of a state-based CRDT, the entire state needs to be transferred to communicate even the smallest change. So a natural idea is to split the elements over multiple BloomCRDTs and link them together. Since the R-Tree is tolerant of overlapping children, this provides a suitable data structure to organize the linking of BloomCRDTs. Each node of the Conflict Free R-Tree is backed by a BloomCRDT. It is reminiscent of the MST, but the MST is not based on a CRDT for the actual tree nodes and is grow-only. Moreover the MST is modeled after a B-Tree but that is an unsuitable choice in the context of concurrency. If replicas independently decide to split a tree node then without coordination there is no consensus on how the elements are divided. Since building consensus is contrary to the idea of CRDTs, the only other solution is to deal with the conflicting splits that will happen. The B-Tree is not able to express such conflicting splits, but the R-Tree is uniquely suited to this situation.

In order to use a BloomCRDT for each tree node, there must be a mapping of R-Tree node *entries* to BloomCRDT set *elements*. Entries are be formed as triples of the form $e = (k_{min}, k_{max}, v)$. The $k_{min}$ and $k_{max}$ indicate a range in $\mathbb{K}$ covered by the entry. When $k_{min} = k_{max}$ this indicates a leaf value, and when $k_{min} \neq k_{max}$ this indicated a child reference. In the case of a leaf value, $v$ is the value associated with the key $k_{min}$. In the case of a child reference, $v$ is a global identifier of another BloomCRDT that can be resolved to produce a local replica. Furthermore, a special entry is added to each BloomCRDT of the form $e = (\text{"}parent\text{"}, \text{"}parent\text{"}, p)$ where $p$ is the global identifier of the node's parent. Figure 4.5 shows an example of R-Tree entries mapped to BloomCRDT set elements for each node.

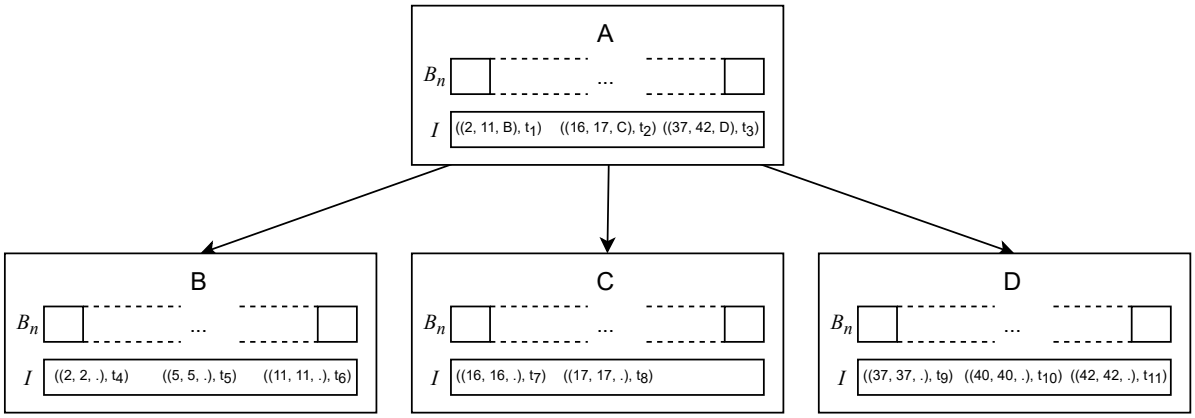



Figure 4.5: Example structure of a CFRT, showing the mapping onto BloomCRDTs

In addition to mapping the R-Tree node entries onto BloomCRDT set elements, the CFRT also needs a specification for its three basic operations: $add(k, v)$, $lookup(k)$ and $remove(k)$, and for its internal operations $split()$ and $merge()$

- $lookup(k)$ The lookup function is straightforward. Given a tree node, for each entry $(k_{min}, k_{max}, v)$ that satisfies $k_{min} \leq k \leq k_{max}$, return $v$ if it is a leaf value and otherwise recurs to the BloomCRDT identified by $v$.

- $add(k, v)$ The add function descends the CFRT identical to the lookup function, but it can run into the situation that zero or multiple entries satisfy $k_{min} \leq k \leq k_{max}$. If zero entries satisfy

the condition the add function *could* select any child to descend, but to reduce the potential for overlap and the resulting reduced performance, the child that *should* be selected has the smallest increase in range to accommodate $k$. If multiple entries satisfy the condition, choose one. When at a leaf node, add entry $(k, k, v)$ to the node. While unwinding the recursion stack, each node should update its entry in its parent to reflect any changes to the range of the child.

- $remove(k)$ The remove function descends the CFRT identical to the lookup function. If in any node no entry satisfies $k_{min} \leq k \leq k_{max}$, then $k$ is not in the CFRT. When at a leaf node, remove any entry that satisfies the condition. While unwinding the recursion stack, each node should update its entry in its parent to reflect any changes to the range of the child.

- $split()$ To split a node ($n$), select the median ($m$) of all $k_{min}$ and $k_{max}$ values of all entries in $n$. For each entry in $n$, decide if it orders less or greater than $m$. For entries that overlap $m$, so $k_{min} \leq m \leq k_{max}$, if $m$ is closer to $k_{min}$ than to $k_{max}$ the entry orders less, otherwise it orders greater. If the distances are equal, choose less or greater at random. Create a new CFRT node $n'$ and add to it all entries that ordered greater than $m$. Update the parent of $n$ with the range and id of $n'$. Remove from $n$ all entries that ordered greater than $m$. Update the parent of $n$ with the new range of $n$. If the transferred entries where not leaf values, update the parent references of the nodes identified by those entries from $n$ to $n'$.

- $merge()$ To merge two sibling nodes ($n$ and $n'$), add all entries from $n'$ to $n$. Update the parent of $n$ with the new range of $n$. From the parent of $n'$ remove entries where $v$ identifies $n'$.

There is however one caveat with the system outlined above. Assume a node splits (or merges) and moves a portion of its children to a new node, then the children will need to be informed of their new parent. However there is no way to atomically update all the replicas of each child node, since the parent reference is stored in a BloomCRDT entry. In fact, in the case that the parent is split by multiple nodes concurrently the child's parent pointer is updated concurrently and points to multiple new parents. There are two solutions to manage the parent/child relationship: one is to work without parent pointers at all, and the second is to simply allow multiple parents for each node. When working without a parent pointer, the parent node would need to observe the child for modifications and update its entries accordingly. However after a node splits, the parent has no way to discover the newly created siblings and would thus require sibling pointers on each node to aid in discovery. These sibling pointers have problems similar to the parent pointer. Also without parent pointers the immediate propagation of information towards the root is interrupted. Suppose a split would propagate all the way up to the root, then at each layer it has to pass the propagation has to wait for the parent to observe the split. The second way to manage the parent/child relationship is to simply allow the child to point to multiple parents. Both mechanisms result in the child being referred to by multiple parents. This type of R-Tree is called a R+-tree [44], and even though this forms a Directed Acyclic Graph this does not violate any R-Tree constraints. This does obviously introduce overlap, since all parents must include the range of the child.

### 4.3.3. Checking for optimizations

So far the definition of the CFRT had been straightforward, but the observant reader might have already wondered, what if two replicas concurrently $split()$ a node? The parent will contain three or more entries that represent overlapping child ranges. This is where the R-tree has a clear advantage over the B-tree, since it can express overlapping ranges of children while remaining consistent. The flow of events and the resulting state is exemplified in Figure 4.6. A concurrent $add()$ leads to a concurrent $split()$. The adds have introduced entries at different positions in the key space, so the split will not use the same median. Since the CFRT is composed of BloomCRDTs at each tree node, the BloomCRDTs will join their states and eventually settle. After the tree nodes have joined their states the CFRT contains overlapping child ranges in the root node, but the data structure is still consistent.

The CFRT could be left in this state with the hope that adds/removes will, over time, naturally reduce the overlap. However reduced overlap can also be actively pursued by means of a periodic $check()$ function. This active restructuring of a R-Tree is the principle behind the R*-tree [8] and can result in improved efficiency. Active restructuring also means that each BloomCRDT is likely to be removed at some point, thus preventing the infinite growth of its Bloom filters. Furthermore, the $check()$ function can test for some other structural optimizations that can be applied to the CFRT.

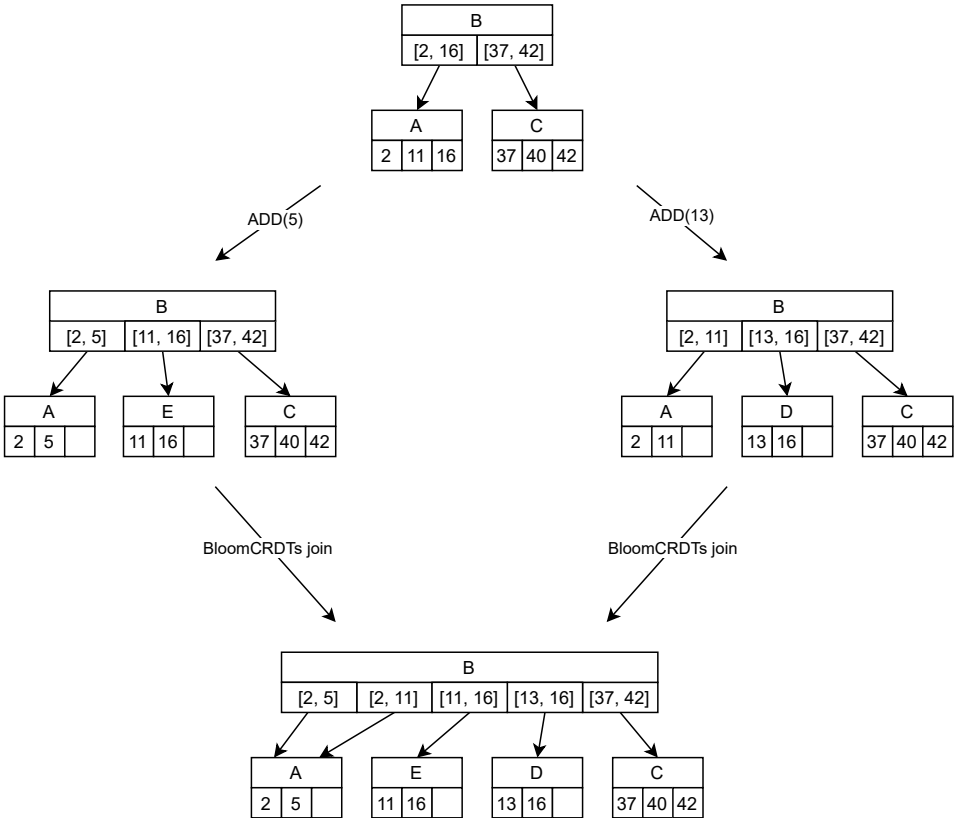



Figure 4.6: Example of CFRT concurrent $split()$.

- **Check parent-child links** As explained in Section 4.3.2, child nodes can have multiple parents. After the joining process of BloomCRDTs the parent/child links can become outdated. Removing superfluous parent and child links helps to reduce the overlap between nodes by shrinking ranges.
- **Check merge/split threshold** R-Tree nodes in traditional databases have a hard limit on their size, often aligned to a memory or disk page size. Thus an $add()$ that overflows a node immediately triggers a split. For the CFRT there is no such limit, allowing a decoupling of the decision to split/merge from the add/remove logic. The $check()$ function is an excellent place to do this since its periodic execution allows batching additions and removes.
- **Check entry in parent** A concurrent split can result in multiple entries in a single parent pointing to the same child node. During the periodic check, a child node should check for and correct this condition.
- **Check (large) overlapping children** If there are entries in a node that have an overlapping range then these could be merged. This is a natural result of a concurrent split of a child node, that will likely produce multiple siblings that overlap a lot due to duplicate contents. Merging the siblings that overlap the most removes duplicated entries. There is however also the situation that the overlap in range is large, but the number of duplicate items is low. In such cases, the merge can result in a node that is over the split threshold and will split again, with a more favorable split.

As an example of how the $check()$ function fixes inconsistencies consider Figure 4.7. It starts with the last state from Figure 4.6. This state shows a double reference for node A and shows that ranges $[11, 16]$ and $[13, 16]$ overlap. The only reason that the splits are not identical is that the two concurrent adds affected the selection of the median. In practical implementations a node contains more than three entries, where as in this example entries 2 and 16 represent larger sequences of entries. So in practice, the overlap resulting from concurrent updates would be far more dramatic. The check function fixes the double pointer and merges node D into node E. The result is an optimized CFRT where overlap is avoided.

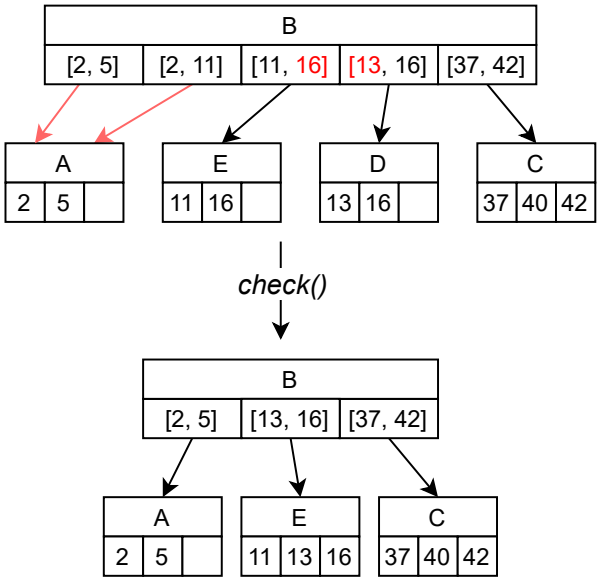



Figure 4.7: Example of CFRT $check()$ function correcting overlap in the ranges spanned by its entries.

### 4.3.4. CFRT compared to DHT

CFRT performs roughly the same as a DHT (also a distributed index). So what justifies the CFRT compared to it?

- It does not require obfuscating the key space with a hash. This is because it is a balanced tree, it doesn't need to spread keys uniformly.
- Load is distributed because the (path to) popular keys has a correspondingly higher number of replicas that share the load.
- It natively supports multidimensional keys, the DHT would need to do a linearization step, with assorted problems.

# 5 Evaluation

This chapter provides implementation details on the BloomCRDT and CFRT data structures, along with an experimental evaluation of these solutions.

## 5.1. Implementation

The system model (see Section 4.1) requires the use of a communication middle-ware library that provides the required features. In this case PyIpv8 [51] was selected because it provides the required features, is very flexible and works well with the Gumby [50] experimental framework. Gumby allows developers to quickly setup and run experiments in distributed environments. Since these components need to be extended, this means the experiments are programmed in the Python programming language. To shorten development time, a search was performed for Python compatible CRDT libraries and off-the-shelf components. Below is a list of existing solutions and their technical characteristics.

- https://github.com/ericmoritz/crdt, up to 2013. Based on Python 2, whereas PyIpv8 is based on Python 3. The CRDT base class only offers two convenience methods / prototypes for obtaining a serialized state.
- https://github.com/kishore-narendran/crdt-py, up to 2016. Uses CRDTs in Redis as multi-valued registers. This is not the intended application domain of BloomCRDT and CFRT.
- https://github.com/anshulahuja98/Python3-crdt, current. Active. No base classes to inherit for BloomCRDT and CFRT.
- https://github.com/merchise/xotl.crdt, up to 2020 (Dec). Offers a base class that has 3 extra lines to (un)pickle CRDTs.

All modules that provide Python CRDT implementations are simple implementations of the CRDT primitives from the original CRDT tech report: the G-Set, Counter, 2P-Set, 2P2P-graph, etc. These primitives are not directly useful for BloomCRDT nor for CFRT, any further usefulness is a few lines of code to (un)pickle CRDT state. The evaluated libraries do not provide the needed functionality, and extending them would be sub-optimal time management. Therefore, BloomCRDT and the CFRT are implemented from scratch in the Python programming language.

The Python code written for this work consists of several Python modules. Figure 5.1 shows the architecture and relations between these modules.

- **CRDT set primitives** This part only implements the algorithm for in-process use.
  - **BloomCRDT** Including an implementation of the classic Bloom filter. The BloomCRDT algorithm is fairly simple and the expressiveness of Python means BloomCRDT uses just 127 Lines-of-Code (LoC), slightly under half of which is the Bloom filter implementation. The initial implementation was straightforward but inefficient in computing hashes for the Bloom filter. This became a problem during the experimental phase, since BloomCRDT would

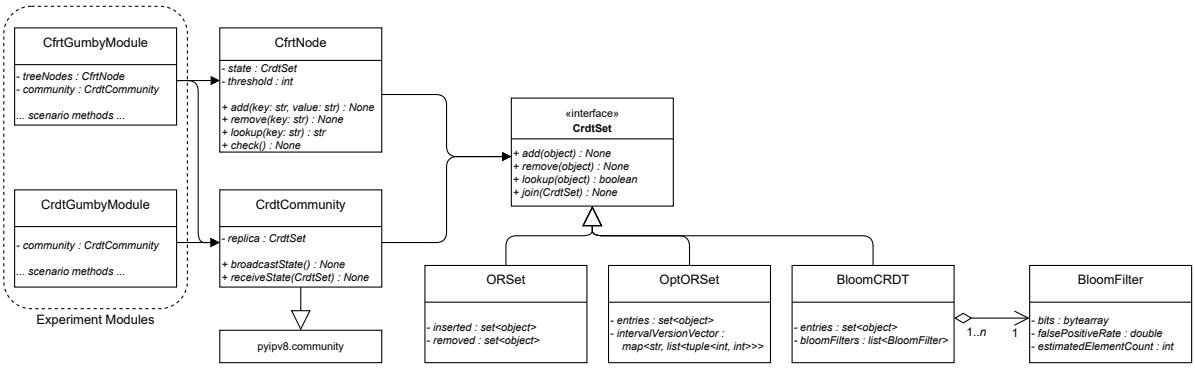



Figure 5.1: Diagram of implementation and experimentation architecture.

become CPU-bound much sooner than other algorithms. After profiling, the problem was identified and a more efficient method of computing hashes was implemented. Instead of calculating multiple full hashes, a single hash state is incrementally updated and at each step a hash is produced. Apart from this, no optimization was done. In specific usage scenarios there is room for further optimization. For example when a BloomCRDT contains only key/value pairs, the reliance on Python's $set()$ primitive to store them is sub-optimal.

  - **Classic OR-Set** Implemented as described in [46] is very simple at just 43 LoC, most of which is Python boilerplate and convenience methods.
  - **OptOR-Set** Or the Optimized OR-Set Without Ordering Constraints as described in [35]. It uses 88 LoC, most of which go towards logic for collapsing version ranges.

- **CFRT Node** Uses one instance of BloomCRDT for each R-Tree node. For 328 LoC the implementation is significantly more complex than BloomCRDT and the other CRDT set primitives. Of particular note are the separation of $check()$ and the randomized split/merge thresholds. The $check()$ method is only directly executed when absolutely necessary. In addition to not being cheap to execute, it offers the chance to coalesce multiple $split()$ and $merge()$ operations. Each instance of a CFRT node is provided with a randomized threshold for merging and splitting. The rationale is that this reduces the probability that peers concurrently $split()$ or $merge()$ a node. While CFRT is tolerant of such an event, it is more efficient to reduce this occurrence. The ranges for the random thresholds are set such that a split produces two nodes that cannot cross the merge threshold of another peer and vice versa.

- **CRDT Pyipv8 Community** This module is responsible for sending a CRDT's state to other peers. All CRDT set implementations (BloomCRDT, OR-Set and OptOR-Set) are state-based CRDTs that expose a common set of methods. This allows the CRDT community to be agnostic as to the actual type of CRDT being used. The CRDT community thus holds a reference to an instance of any of the CRDT set implementations. When requested it will serialize (`pickle`) the local CRDT instance and send it to up to 10 peers. When a CRDT state message arrives from another peer, the CRDT community deserializes the CRDT set instance contained in the message and provides it as an argument to the $join()$ function of the local replica. During experimentation it was quickly obvious that the 64KByte limit on UDP packets used by Pyipv8 could be exceeded. To overcome this limit a very rudimentary fragmentation scheme was added to the CRDT community. This raises the message size limit enough to allow the experiments to run. This brings the total LoC for the CRDT community to 110.

- **CRDT Gumby Module** manages the CRDT Pyipv8 Community based on Gumby scenario directives. It exposes several methods to control concurrent tasks in a Gumby scenario, $add()$, $remove()$ and $join()$, with configurable parameters to simulate different workloads. Note that it is the $join()$ task that ultimately initiates the CRDT Community to broadcast its state. An alternative would be to have the CRDT Community as an observer of its CRDT set, but this precludes the option of batching several updates in a single state broadcast. A final task $stats()$ outputs statistics gathered by the CRDT community that are later plotted in graphs. The CRDT Gumby module also allows switching the type of CRDT set used by the CRDT Community.

- **CFRT Gumby module** leverages the CRDT PyIpv8 Community to synchronize the BloomCRDT instances contained in CFRT Nodes. It is very similar to the CRDT Gumby module however: it tracks different statistics, manages the reference to the CFRT root node and modifies the CFRT with key/value pairs as opposed to random elements like the CRDT Gumby module uses.

The code as used in the experiments is public and can be found at [12]. Unless otherwise noted it is assumed that BloomCRDT initialized the Bloom filter with $p(falsepositive) = 1^{-8}$ and an expectation of 500 elements. The CRDT Community aims to maintain 10 connections to other peers for broadcasting replica states. These values are suitable to demonstrate the effectiveness of BloomCRDT and CFRT in an experimental setting. Unless otherwise noted all experiments are run on the DAS5 [2] TU Delft cluster.

## 5.2. BloomCRDT Experiments

Section 4.2 predicts two advantages of the BloomCRDT that should be confirmed by experiments. Specifically, improved space complexity when removing from the set and usability in a P2P setting.

1. Does BloomCRDT provide a reasonable space complexity when the workload includes $delete()$? The space complexity is very important since, as with all state-based CRDTs, the whole state has to be communicated to other peers in order for changes to propagate. Thus a lower space complexity requires less bandwidth and can reduce the need for message fragementation.

2. Does BloomCRDT provide a reasonable space complexity when the network has peer identity churn? If BloomCRDT is affected by peer identity churn then it could grow to infeasible sizes in open P2P environments.

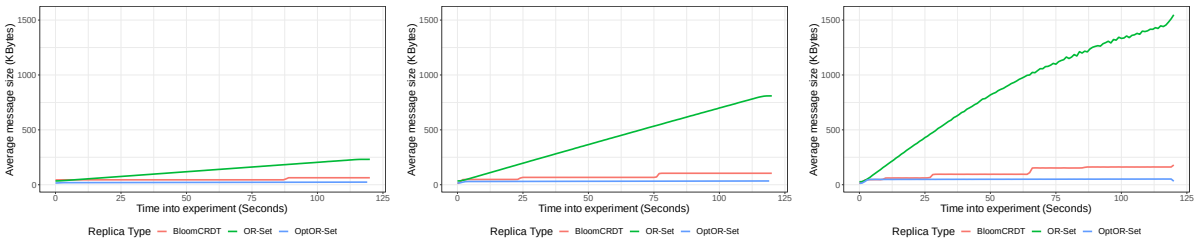
To answer these questions, BloomCRDT is compared to the OR-Set and the OptOR-Set under various conditions. The OR-Set was chosen as a baseline since it is the design that BloomCRDT is derived from. The OptOR-Set was chosen because: it has set semantics as opposed to the list semantics of many other CRDTs; it has a state-based CRDT mode of operation; it is an advanced design; it is also a derivative of the OR-Set; and it shows some algorithmic similarity to BloomCRDT in the $join()$ function.

### 5.2.1. BloomCRDT $delete()$ workload storage cost

The hypothesis is that BloomCRDT has an acceptable space complexity for a workload that includes $delete()$. Since BloomCRDT does not keep the full elements (nor tags) of deleted elements it should have a lower space complexity compared to the OR-Set. In fact for the given false positive probability, a Bloom filter should use ≈38.34 bits per element. An OR-Set would need a pointer to each deleted element which uses 64-bits on current hardware. This is almost double the bits used by a BloomFilter, and does not include the actual bytes that comprise the element. So the BloomCRDT should be superior, but by how much? In contrast, the OptOR-Set actively works to reduce the size of its state and as such its space complexity should be constant with respect to the number of $delete()$ operations performed.

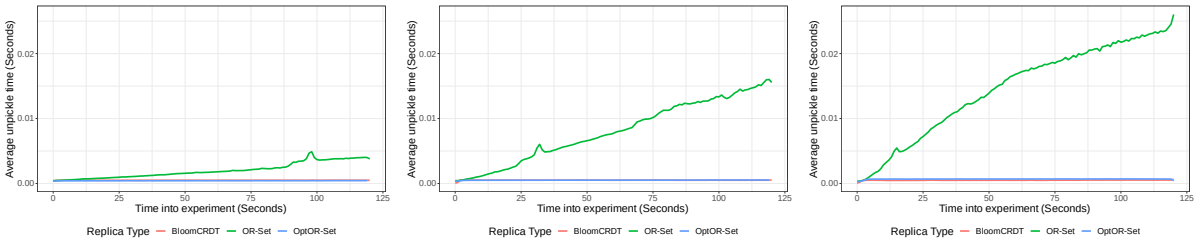
**Experiment Setup** To test the space complexity of BloomCRDT, the standard OR-Set and the OptOR-Set with respect to the number of $delete()$ operations, an experiment is setup as follows. A variable number ($n$) of peers is started. Each peer holds one replica of a BloomCRDT, OR-Set or OptOR-Set. All sets are preloaded with 512 randomly generated elements, in this case 128-bit integers. During the whole experiment peers send their state to up to 10 other peers and $join()$ this state every two seconds. At the start, the sets are allotted 10 seconds to reach eventual consistency. After that, for 110 seconds, each replica starts two recurring tasks to concurrently add and remove a random element every second. Subsequently the peers are allotted 10 seconds to join their states and reach eventual consistency again. During the experiment a record is kept of: the number of bytes used to store a replica, the time uses to $join()$ exchanged states and the time used for deserialization of the replicas.

The recurring add and remove tasks aim to keep the number of elements in the set around 512. This is to ensure that the size measurement only measures the effect of element churn in each CRDT set type and not increased or decreased element count. The number of bytes used is measured with Python's `pickle` module. This is because it is difficult to have Python produce an accurate count of memory bytes used for a given object graph. The `pickle` methods are not a perfectly accurate measure of the memory bytes used for an object graph. `Pickle` will dereference pointers and insert back references




(a) $n = 16$ (b) $n = 64$ (c) $n = 256$

Figure 5.2: Space complexity of BloomCRDT, OR-Set and OptOR-Set




(a) $n = 16$ (b) $n = 64$ (c) $n = 256$

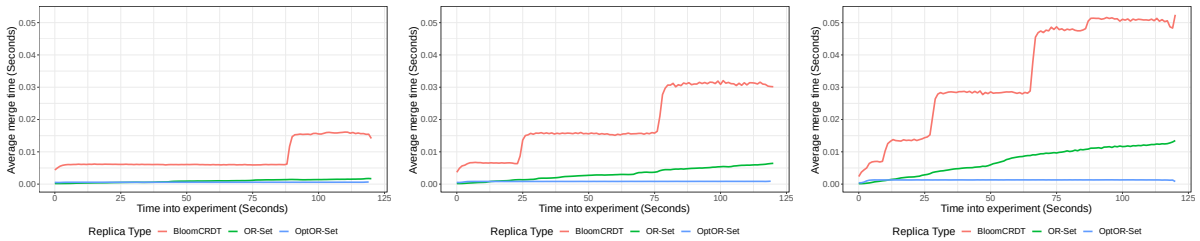
Figure 5.3: Time complexity (Deserialization) of BloomCRDT, OR-Set and OptOR-Set

in the byte stream if it encounters a pointer to a previously serialized object. Also `pickle` does not include padding that a Python interpreter would use to align object members in memory. This means the number of bytes produced by `pickle` will likely be less than actual memory bytes used. However the `pickle` methods are also used to serialize replica state when communicating with other peers, so using the `pickle` method gives an accurate measure of message space complexity in regards to the most limiting factor: message size.

**Results** Figure 5.2 shows the space complexity results of the experiment for various values of $n$. Each graph shows the time into the experiment in seconds progressing on the horizontal axis and the average measured replica size in KBytes on the vertical axis. Within each graph a distinction is made between BloomCRDT, the OR-Set and the OptOR-Set. The results clearly show the expected benefit of BloomCRDT over the OR-Set. In this case the difference in message complexity is around one order of magnitude. However a practical use of CRDTs might very well use more than 16 bytes of information per element as was used in this experiment. So this result is indicative of the lower bound on space complexity improvement of BloomCRDT over the OR-Set. The OptOR-Set shows the expected behavior and is able to collapse metadata and maintain a constant space complexity with respect to the number of elements deleted.

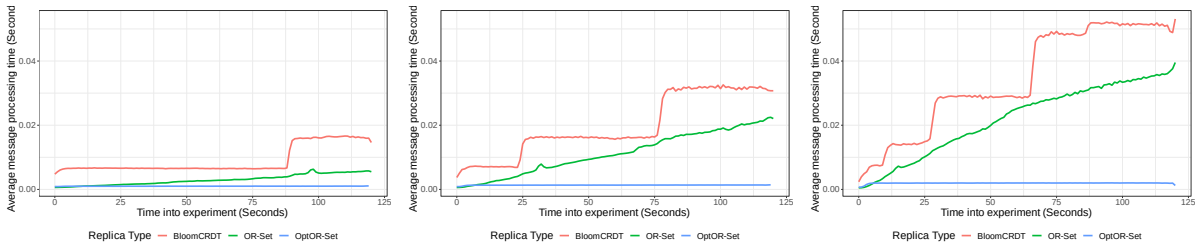
In the interval between broadcasting replica states, the CRDT Community can expect to receive one replica state from each peer it is connected to. As previously described this number is set to 10 in these experiments. The process of receiving and joining a CRDT state should be fast enough to keep up with the flow of messages. Figure 5.3 shows the deserialization time complexity results of the experiment for various values of $n$. Each graph shows the time into the experiment in seconds progressing on the horizontal axis and the average time in seconds taken for replica deserialization on the vertical axis. Within each graph a distinction is made between BloomCRDT, the OR-Set and the OptOR-Set. The effect of a reduced space complexity of BloomCRDT and the OptOR-Set compared to the OR-Set is also apparent in the time used to deserialize replica states. This is in part due to to simply more bytes to process for the OR-Set instances, but also the structure of the OR-Set. It consists almost entirely of small entities like tuples and (random) integers. Each such entity will add extra overhead to the deserialization time. In contrast, the BloomCRDT set has a structure where most bytes are in large arrays that can be efficiently processed. Even though the state is bigger, it still deserializes as fast as the OptOR-Set. Again, the OptOR-Set is able to collapse its metadata state and therefore is able to maintain a very small state. Even though its state is also composed of many small entities, it is still very efficient when deserializing.

Figure 5.4 shows the time complexity results of the actual merge algorithm for various values of




(a) $n = 16$ (b) $n = 64$ (c) $n = 256$

Figure 5.4: Time complexity (Join) of BloomCRDT, OR-Set and OptOR-Set




(a) $n = 16$ (b) $n = 64$ (c) $n = 256$

Figure 5.5: Time complexity total for each message in BloomCRDT, OR-Set and OptOR-Set

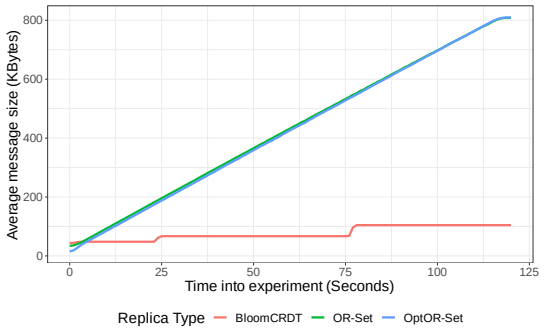
$n$. Each graph shows the time into the experiment in seconds progressing on the horizontal axis and the average time in seconds taken for the $join()$ algorithm on the vertical axis. Within each graph a distinction is made between BloomCRDT, the OR-Set and the OptOR-Set. Clearly the join of two replica states is more complex to compute for BloomCRDT than for OR-Set and OptOR-Set. This is because the BloomCRDT has to compute hashes for new elements. The OR-Set also shows some increase in the time taken to join two states. This is due to the nature of the Python $set()$ primitive which, for each added element, has to check if the element is already a member of the set.

Figure 5.5 shows the sum of figures 5.4 and 5.3 for various values of $n$. Both the deserialization and join have to be performed once for each message received. After examining the steps separately this graph shows the combined result. Depending on the situation, BloomCRDT and OR-Set have a comparable time complexity, with BloomCRDT being more favorable as more deletes happen over time.

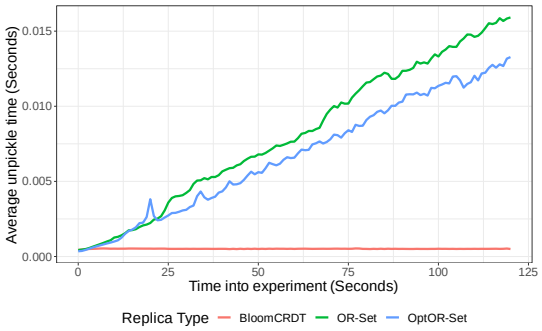
### 5.2.2. BloomCRDT P2P tolerance

Given the previous experiment, one could conclude that the OptOR-set is a superior solution that performs better than BloomCRDT and the OR-Set. Other solutions, like Redis [41], Riak [5] and DottedDB [22] should show results similar to the OptOR-Set since they all work to collapse state size through garbage collection. All these solutions require either a limited number of peers, or keep persistent information about other peers in the network. How does BloomCRDT compare to such systems when there is peer churn in a P2P network? Peer identities come and go over time, and keeping track of them all could inflate a replica's state size. Using the OptOR-Set as an example of systems that keep per peer state, an experiment was performed to investigate the effect of peer identities over time on the replica state size. The experimental setup is identical to Section 5.2.1 with the only change being an additional recurring task that changes the identity of each replica every second to simulate peer churn. In this experiment $n$ is not varied and fixed to $n = 64$.

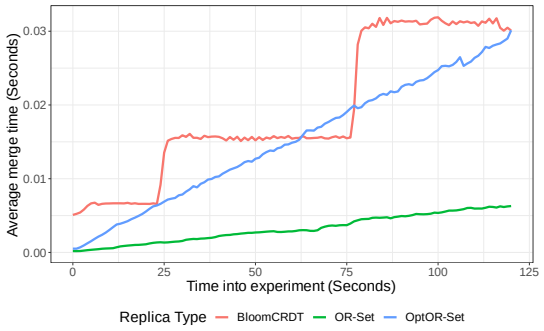
Figure 5.6 shows the results of the experiment. As can be seen in Figure 5.6a, the OptOR-Set trades the regular OR-Set's linear space complexity with respect to the number of deletes for a linear space complexity with respect to the number of peer identities encountered. In this experiment the OptOR-Set encounters 7040 identities (1 identity per peer per second $*$ 64 peers $*$ 110 seconds) and shows a space complexity for this similar to the regular OR-Set. The time complexity of the OptOR-Set also shows the same trends as the OR-Set. P2P networks can have millions of identities over the lifetime of the network. So clearly the OptOR-Set is not well suited to a P2P environment. On the other



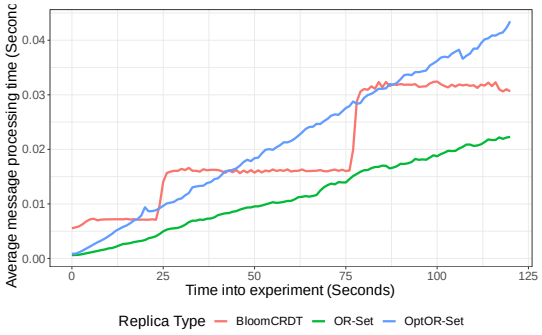

(a) Storage/Message Cost of CRDTs




(b) Deserialization time of CRDTs




(c) $join()$ time of CRDTs




(d) Total time for each CRDT message

Figure 5.6: Experiment comparing BloomCRDT, OR-Set and OptOR-Set with network churn

hand, Figure 5.6 shows that BloomCRDT is unaffected by the number of encountered peer identities. So BloomCRDT is suitable for P2P networks with peer churn that require a CRDT with set-semantics that also supports deletes.

## 5.3. CFRT Experiments

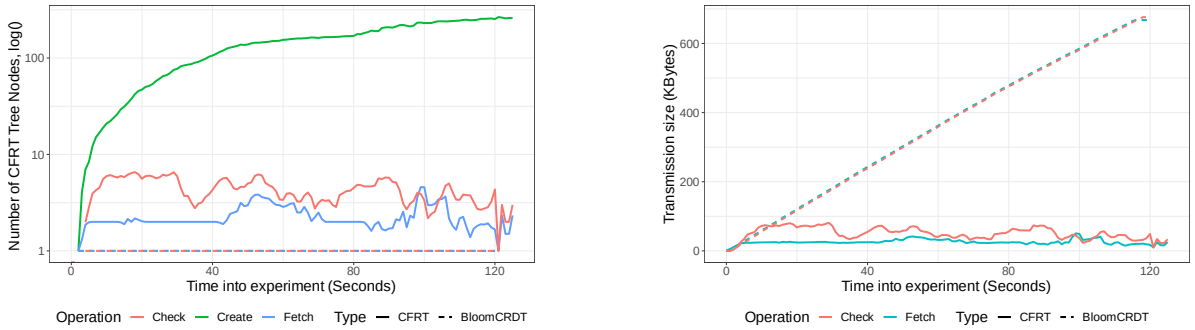
The motivation for the design of CFRT is that BloomCRDT can only scale linearly with respect to the number of elements it contains. BloomCRDT is atomic and it can only be distributed as an indivisible state. So if, for example, a use-case arises where an application wants to store a million torrents and magnet links in a BloomCRDT, then this would work in theory. In practice users will probably not want to wait while a gigabyte sized BloomCRDT state is downloaded.

### 5.3.1. Practical limits of BloomCRDT and CFRT

This experiment explores the practical limits of BloomCRDT and CFRT. Especially how they function from the perspective of a new peer joining the network with the aim of adding a specific key/value pair. To complete the $add()$ the new peer will have to fetch a BloomCRDT or several CFRT nodes. The number of fetched bytes directly translates to the time it takes to execute the operation. As already found in experiment 5.2.1 a singleton BloomCRDT shows linear growth in state size on both add and remove, and thus in bytes that new peers will have to exchange for an update. The CFRT, by nature of most index trees, should show $log_k$ behavior for fetched bytes where $k$ is the fan out factor of the tree nodes.

The setup of this experiment is as follows. A number ($n = 16$) of peers is started. Each peer starts out with a replica of an empty BloomCRDT and an empty root CFRT node. To be able to test BloomCRDT in an experiment with CFRT, the BloomCRDT replica is represented by a single CFRT node with a split threshold at infinity. Since a single CFRT node contains a single BloomCRDT, setting the infinite bound will keep the CFRT node from splitting. The CFRT code provides the necessary logic to allow key/value pair manipulation in a BloomCRDT. For 110 seconds, each replica runs a recurring task to add random key/value pairs. Every two seconds replicas perform a $check()$ on CFRT nodes and subsequently exchange and $join()$ all BloomCRDTs. During the experiment a record is kept of:

the number of CFRT nodes created, the number and size of CFRT nodes visited during each $add()$, and the number of CFRT nodes that required a $check()$.




(a) CFRT nodes created, fetched and checked

(b) Size of CFRT nodes fetched and checked

Figure 5.7: Scaling comparison of BloomCRDT to CFRT

Figure 5.7 shows the results of the experiment. Both figures have the time into experiment in seconds on the horizontal axis. The first figure, 5.7a shows the counts of CFRT nodes. Note that the vertical axis of this figure is $log_{10}()$. The graph shows how many CFRT tree nodes are created during the experiment. The graph also shows the number of nodes that each $add()$ operation had to fetch and the number of nodes that required a $check()$. The BloomCRDT is obviously limited to 1 node, no splits or merges happen. The CFRT shows a linear growth in the total number of nodes created, as is to be expected when adding at a constant rate and a limited number of entries per node. The number of nodes fetched is the number of nodes that a new peer with no prior information would have to retrieve to add a new value to the data structure. In the case of CFRT, it shows a $log_k$ relation to the number of nodes in the CFRT tree and by extension, to the number of entries in the data structure. The number of nodes that are checked also seems to show a $log$ relationship to the number of nodes in the CFRT tree. The second Figure 5.7b shows the size in bytes of the CFRT nodes fetched and checked. Again they reflect the $log$ nature of the CFRT, and the linear nature of the BloomCRDT. It is clear that just after the first split is made, the CFRT is more efficient with bandwidth usage. In the long run it should be expected that CFRT bandwidth usage scales much better than BloomCRDT with respect to the number of entries.

### 5.3.2. Fault tolerance of CFRT

If the CFRT is to be a practical choice, it must show robustness and continue to function in spite of faults. In distributed systems these faults can take many forms, but a common is dropped messages due to network congestion or link failure. The next experiment aims to show that CFRT is indeed robust against such failures. The experiment starts a number of peers ($n = 10$), initiates a CFRT root node and allows time all peers to reach consistency. Then in 10 batches 200 key/values entries are randomized and added to the CFRT by a single peer. After this, the 200 entries are removed from the CFRT in 10 batches by the same peer. During the process of adding and removing CFRT entries, all messages have a uniform probability ($p$) of being dropped. For each peer a plot is made of the number of entries reachable in its CFRT replica. The experiment is repeated for different values of $p$. The idea is to test the impulse response of the system. Given a unit of change does the system reach consistency and if so, how long does it take to reach consistency? This experiment is small and was therefor not executed on the DAS5.

Figure 5.8 shows the results of the experiment. The left column shows graphs for different values of $p$ that count, for each peer, the number of CFRT key/value entries reachable from the root. The red line, peer 1, is the peer that adds and removes CFRT entries. When all lines coincide with peer 1, then the CFRT has distributed all information perfectly and all peers have reached consistency. The difference of any peer and peer 1 is a measure of inconsistency. The right side column shows the number of messages dropped and passed by the experimental framework. All figures have the time into the experiment in seconds on the horizontal axis. Graphs 5.8a and 5.8b show the ideal baseline situation when $p = 0$. There are some low dips; this is due to a tree structure change happening concurrently with counting the number of reachable entries. Graphs 5.8c and 5.8d show the situation

for $p = 0.25$. In this case CFRT manages to almost keep ideal performance, although the time to convergence is slightly longer in some cases. At $p = 0.5$ (graphs 5.8e and 5.8f) some peers are noticeably slower to converge, especially when a split happens that is missed. In these cases the number of entries in the tree appears to reduce temporarily. For practical scenarios the CFRT would still be usable. Even at $p = 0.75$ (graphs 5.8g and 5.8h) the peers roughly converge and would be able to respond correctly to the vast majority of all user requests. Noteworthy is the number of messages sent. For $p = 0$ through $p = 0.75$ the experiment sends around a thousand messages total regardless of the percentage dropped. There is no increased number of messages to compensate for the dropped messages. This is because if a message is lost between peers, the message at the next interval will include all information that the dropped message attempted to send. So the amount of information that is communicated per message increases as more messages are dropped. This comes at the cost of time to convergence, as can be seen in the left hand column. It takes slightly longer for peers to converge their state as $p$ becomes larger. At $p = 0.9$ (graphs 5.8i and 5.8j) the CFRT breaks down and no longer shows any practical level of convergence.

The breakdown at $p = 0.9$ is mostly due to all new information from peer 1 being blocked. Like a min-flow max-cut situation, the rest of the peers cannot know more than what peer 1 is able to get through, and even then dissemination though the network is erratic. Another reason for the breakdown is how the messaging is setup in the experiments. Only CFRT nodes that changed since, or as a result of, the last $join()$ interval are broadcast to other peers. Thus there is little redundancy in the sending side to overcome faults. Even though networks with 90 percent message drop are not realistic, CFRT can be made to work with them. If the messaging of peer 1 is altered to be more redundant such that the changed state is ignored and the full state sent each $join()$ interval, then the result is noticeably different. Graph 5.9 shows that even with $p = 0.9$ the CFRT can be made to work when given this slightly altered messaging policy. The cost is more messages being sent, but that is difficult to avoid given the value of $p$.

### 5.3.3. A practical application of CFRT

The previous sections have show experimental results that confirm the theoretical benefits of the CFRT. These experiments have been synthetic, each exploring how BloomCRDT and the CFRT perform in light of the theoretical problems identified in Section 3.1. However Section 3.2 shows how a practical application such as Tribler's channels might experience the theoretical problems. To investigate if the CFRT retains it's benefits when applied in a more realistic setting a further experiment is needed.

A dataset of torrents was obtained by starting a clean instance of Tribler and ordering the channels descending by number of torrents contained. Next the client was subscribed to subsequent channels until the total number of torrents exceeded 1 million. Tribler was left to synchronize the subscribed channels and was shutdown afterwards. The result of this protocol is a sqlite database file `metadata.db` that is filled with torrents. In this case the client was subscribed to the four biggest channels and after two hours of synchronizing this resulted in a dataset containing 1,198,638 torrents. The database file size is 1,191,514,112 bytes which is about half of the 2GB suggested in Section 3.2. However the selected Tribler channels only include a thumbnail image for the channel, not for every torrent. The collected database contains a timestamp of when each torrent was added locally and thus when each torrent became visible to the user.

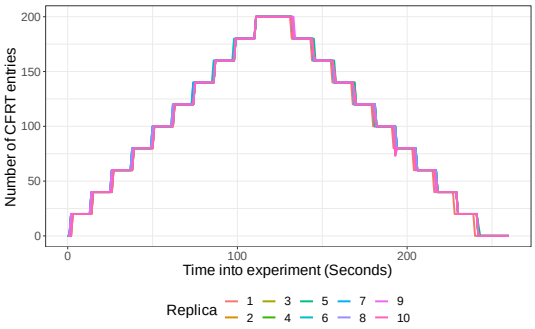
*This rant could go to section 3.2 somewhere? I feel it is important to explain this, even though it could be explained in less detail for the purpose of understanding the experiment* In Tribler each channel is modeled as a tree with nodes, akin to a file system. To distribute a channel the entire node graph is serialized and packaged as a torrent that can be downloaded by other users. This monolithic torrent is redistributed for each small changes, and a user cannot obtain a subset of the channel. Especially this last problem is incongruent with Triblers responsive and big tech inspired user experience. Users browsing a newly subscribed channel are presented with a blank screen until the channel torrent has downloaded its first few pieces and a part of a directory structure shows up. Not until much later do torrents start showing up. Tribler is all Big-Tech flash, but no Big-Tech bang.

The experiment determines, for each torrent in the dataset, how long it takes a new peer with no prior knowledge to look it up. In the case of Tribler torrents become available as the download of each channel's torrent progresses. The collected dataset already records these timestamps. To compare this with the CFRT the following experiment was performed. A peer starts a new CFRT and loads it, in a random order, with all the torrents from the dataset. Using infohash as key and (title, tracker info)

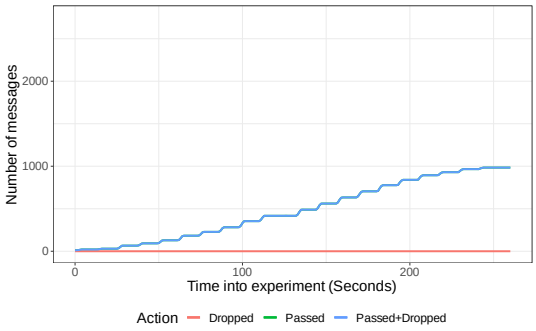
as the value. Then a second peer opens the dataset and, again in a random order, queries the CFRT of the first peer for each infohash. To perform such a query, the second peer first obtains a replica of the tree root, decends to a child and obtains a replica of this child, etc. A measurement is made of the time it takes to query each infohash. Since both peers rondomize the order of inserts and queries, the reader should be confident that no infohash is favoured.

(alternate key strings, such as path in tribler channel, are possible. Infohash is just very convenient. )

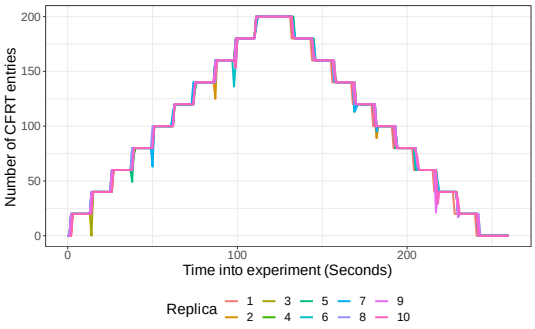
The result of this experiment is shown in Figure 5.10. The horizontal axis represents the elements of the dataset, for each element is plotted on the vertical axis the time, in seconds, that it takes to become available through Tribler and through the CFRT. Notice that this graph has a log distribution on the vertical axis. Tribler starts at around one second for the first element and then the last element becomes available just under two hours later. The CFRT implementation has the fastest element in around one millisecond and the slowest element around 79 seconds. However for the vast majority of elements the CFRT implementation is five orders of magnitude faster. Using CFRTs in Tribler to maintain channels should provide a far more responsive user experiance.



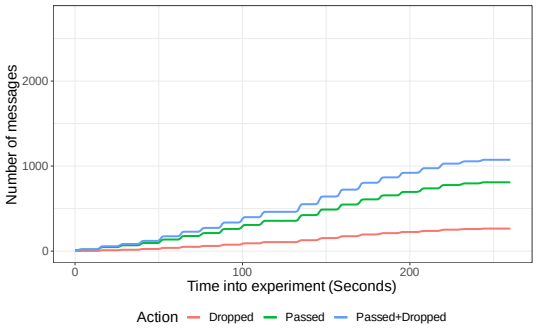

(a) Total number of key/value entries in CFRT ($p = 0$)



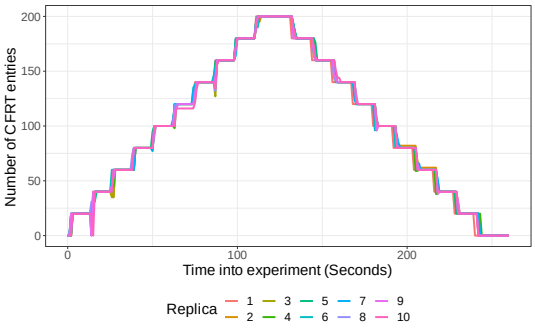

(b) Average number of messages CFRT ($p = 0$)



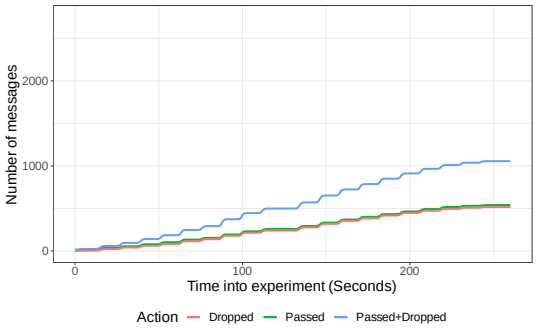

(c) Total number of key/value entries in CFRT ($p = 0.25$)



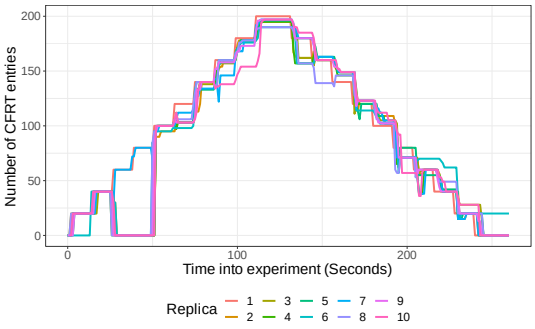

(d) Average number of messages CFRT ($p = 0.25$)



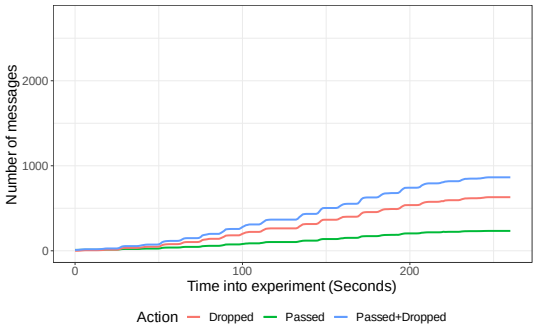

(e) Total number of key/value entries in CFRT ($p = 0.5$)



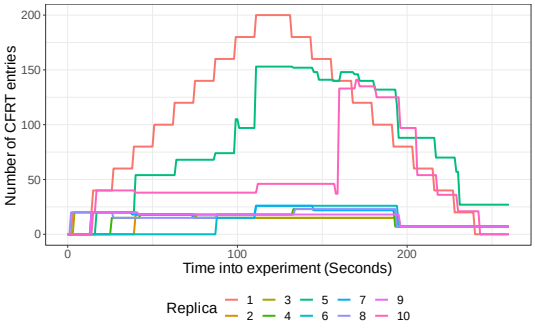

(f) Average number of messages CFRT ($p = 0.5$)



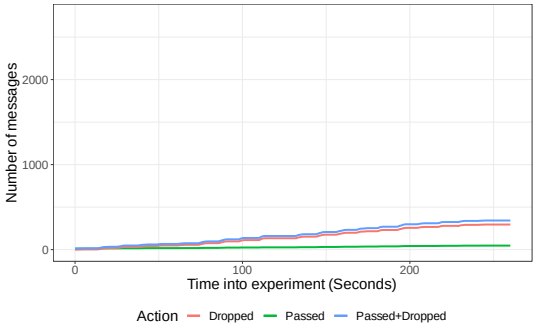

(g) Total number of key/value entries in CFRT ($p = 0.75$)




(h) Average number of messages CFRT ($p = 0.75$)

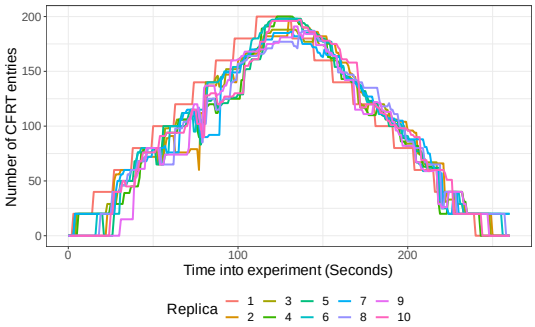



(i) Total number of key/value entries in CFRT ($p = 0.9$)

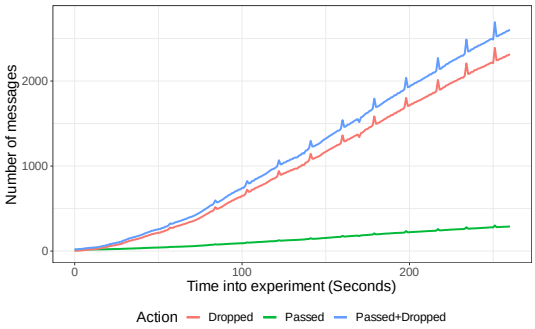



(j) Average number of messages CFRT ($p = 0.9$)

Figure 5.8: Fault tolerance experiment of CFRT




(a) Total number of key/value entries in CFRT ($p = 0.9$)




(b) Average number of messages CFRT ($p = 0.9$)

Figure 5.9: Fault tolerance experiment of CFRT with improved messaging policy

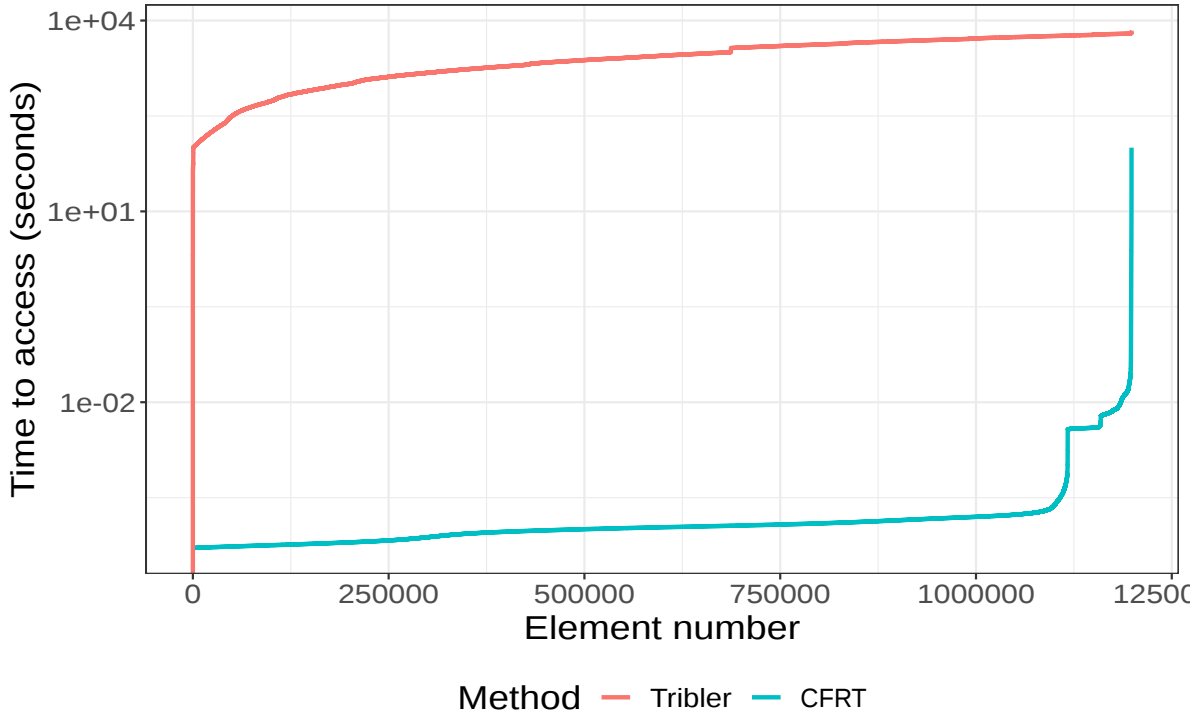



Figure 5.10: Element access time in Tribler Channels compared to CFRT

# 6

# Conclusion and further work

The CFRT and BloomCRDT have been designed and empirically investigated. Enough evidence has been gathered to support the first conclusions, especially with respect to the stated problem and solution criteria of Section 3.2.

## 6.1. Conclusions

Existing CRDTs have shortcomings that prevent their application in open P2P networks. Existing solutions are grow-only in (meta)data or they require knowledge about the state of other peers. Furthermore, they push the full state to all peers, to the detriment of peers that are only interested in a particular subset of data or peers that are low-powered. To address this, Section 3.2 introduced the criteria that should be observed by any CRDT that aims to work in P2P networks. Based on this, the CFRT was designed that should, together with BloomCRDT, meet the criteria.

The novel CRDT primitive with set semantics, BloomCRDT, has been designed as a state-based CRDT. Also, it was shown to be invariant to the number of peer identities that have interacted with it (see Section 5.2.2). BloomCRDT cannot be said to be free from unbounded growth: each deleted item takes up a fixed number of bits. Fortunately this number of bits is affordable, and set can be filled and emptied several times before the deleted items have a big impact on the total state size of BloomCRDT. However, BloomCRDT doesn't scale to many entries very well. Since it is indivisible, a large number of items also means a large state size and thus big network messages.

The CFRT was described as the composition of many BloomCRDT instances. It is structured as a R-Tree and forms a key/value store. Users only need to fetch on the order of $log(k)$ bytes to add, lookup or modify an entry, where $k$ is the number of entries in the whole CFRT. Since peers using the CFRT decide in which direction to traverse the tree, the CFRT as a whole is not push based. Furthermore, the periodic check function of the CFRT continually splits and merges nodes, implying that the BloomCRDT underlying each node only sees a limited number of deletes before it is not needed anymore. This means that any BloomCRDT used in CFRT is very unlikely to materialize its potential unbounded growth in practice.

When used together BloomCRDT and CFRT meet all the criteria of a CRDT that could work in a P2P environment. There is potential of applying BloomCRDT and CFRT in a wide variety of applications. Specifically, Tribler's channels feature could be based on the CFRT, thereby enabling support for channels with millions of torrents.

## 6.2. Further Research

Due to scope and time constraints not every aspect of BloomCRDT and the CFRT have been investigated. There is much more to investigate about the CFRT. Such as what is the lifetime of a CFRT node/BloomCRDT set, and what is the usage of such a set? If there are outliers, BloomCRDTs that exist for very long and see many deletes, then it is possible that the growth of BloomCRDT is more than desirable. This is also very much depends on the type of workload. So another avenue for research is to investigate how BloomCRDT and the CFRT behave when confronted with a real world dataset

or workload. As seen in Section 5.3.2, during fault conditions the messaging algorithm has a huge impact on the results. The decision of when to send the state to peers must strike a balance between redundancy and efficiency.

On a more theoretical level, research can explore the keys used in the CFRT. R-Trees can contain multi-dimensional keys, but does that translate to the CFRT without affecting validity? Since the performance of a regular R-Tree relies heavily on avoiding overlapping child regions, what will the performance of the CFRT be in such a scenario? Furthermore, what practical uses can multi-dimensional keys in the CFRT enable?

The BloomCRDT also has some room for further engineering. By keeping a simple list of Bloom Filters, each check has to pass all bloom filters. If the Bloom Filters could be arranged in a tree structure, then perhaps this cost can be of a log order instead of a linear.

The last avenue of research that should be mentioned is an investigation into how the CFRT could be integrated into Tribler to provide the backing storage for the channels feature. This is likely to bring new insights into real world operation of the CFRT. Also, since BloomCRDT is a CRDT, it is very capable of supporting new Tribler features like a collaborative metadata editor.

# Bibliography


[1] Paulo Sergio Almeida, Ali Shoker, and Carlos Baquero. Efficient state-based crdts by delta-mutation. In *International Conference on Networked Systems*, pages 62–76. Springer, 2015.

[2] Vrije Universiteit Amsterdam. The distributed asci supercomputer 5 (das5) project website. URL `https://www.cs.vu.nl/das5/home.shtml`.

[3] Alex Auvolat. private communication, 2021.

[4] Alex Auvolat and François Taïani. Merkle search trees: Efficient state-based crdts in open networks. In *2019 38th Symposium on Reliable Distributed Systems (SRDS)*, pages 221–22109. IEEE, 2019.

[5] Basho. Riak. URL `http://basho.com/products/riak-kv/`.

[6] Jim Bauwens and Elisa Gonzalez Boix. From causality to stability: Understanding and reducing meta-data in crdts.

[7] Jim Bauwens, Florian Myter, and Elisa Gonzalez Boix. Constraining the eventual in eventual consistency. In *Proceedings of the 5th Workshop on the Principles and Practice of Consistency for Distributed Data*, pages 1–3, 2018.

[8] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The r*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of the 1990 ACM SIGMOD international conference on Management of data*, pages 322–331, 1990.

[9] R Beyer and EM McCreight. Organization and maintenance of large ordered indices. *Acta Informatica*, 1(3):173–189, 1972.

[10] Annette Bieniusa, Marek Zawirski, Nuno Preguiça, Marc Shapiro, Carlos Baquero, Valter Balegas, and Sérgio Duarte. An optimized conflict-free replicated set. *arXiv preprint arXiv:1210.3368*, 2012.

[11] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[12] Ewout Bongers. Bloomcrdt and cfrt code. URL `http://git.captaincoder.nl/thesis-code`.

[13] Russell Brown, Zeeshan Lakhani, and Paul Place. Big (ger) sets: decomposed delta crdt sets in riak. In *Proceedings of the 2nd Workshop on the Principles and Practice of Consistency for Distributed Data*, pages 1–5, 2016.

[14] Sebastian Burckhardt, Manuel Fähndrich, Daan Leijen, and Benjamin P Wood. Cloud types for eventual consistency. In *European Conference on Object-Oriented Programming*, pages 283–307. Springer, 2012.

[15] Software Freedom Conservancy. Git. URL `https://git-scm.com/`.

[16] M. de Vos and J. Pouwelse. Contrib: Universal and decentralized accounting in shared-resource systems. In *Proceedings of DICG'20: 1st International Workshop on Distributed Infrastructure for Common Good*, New York, NY, USA, 2020. Association for Computing Machinery. doi: 10.1145/1122445.1122456. URL `https://doi.org/10.1145/1122445.1122456`.

[17] Stuart Dredge. Apple bans satirical iphone game phone story from its app store. URL `https://www.theguardian.com/technology/appsblog/2011/sep/14/apple-phone-story-rejection`.

[18] C. A. Ellis and S. J. Gibbs. Concurrency control in groupware systems. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, SIGMOD '89, page 399–407, New York, NY, USA, 1989. Association for Computing Machinery. ISBN 0897913175. doi: 10.1145/67544.66963. URL https://doi.org/10.1145/67544.66963.

[19] Arnoud Engelfriet. Waarom ik steeds zeg dat data niets is, juridisch gezien, 2019. URL https://blog.iusmentis.com/2019/07/31/waarom-ik-steeds-zeg-dat-data-niets-is-juridisch-gezien/. Accessed 2020-10-30.

[20] The Document Foundation. Libreoffice. URL https://www.libreoffice.org/.

[21] A. Fox and E. A. Brewer. Harvest, yield, and scalable tolerant systems. In *Proceedings of the Seventh Workshop on Hot Topics in Operating Systems*, pages 174–178, 1999. doi: 10.1109/HOTOS.1999.798396.

[22] Ricardo Jorge Tomé Gonçalves, Paulo Sérgio Almeida, Carlos Baquero, and Vitor Fonte. Dotteddb: Anti-entropy without merkle trees, deletes without tombstones. In *2017 IEEE 36th Symposium on Reliable Distributed Systems (SRDS)*, pages 194–203. IEEE, 2017.

[23] Pascal Grosch, Roman Krafft, Marcel Wölki, and Annette Bieniusa. Autocouch: a json crdt framework. In *Proceedings of the 7th Workshop on Principles and Practice of Consistency for Distributed Data*, pages 1–7, 2020.

[24] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, pages 47–57, 1984.

[25] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM computing surveys (CSUR)*, 15(4):287–317, 1983.

[26] Akka IO. Distributed datatypes. URL https://doc.akka.io/docs/akka/current/typed/distributed-data.html.

[27] Tim Jungnickel, Lennart Oldenburg, and Matthias Loibl. Designing a planetary-scale imap service with conflict-free replicated data types. In *21st International Conference on Principles of Distributed Systems (OPODIS 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[28] Kolbeinn Karlsson, Weitao Jiang, Stephen Wicker, Danny Adams, Edwin Ma, Robbert van Renesse, and Hakim Weatherspoon. Vegvisir: A partition-tolerant blockchain for the internet-of-things. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pages 1150–1158. IEEE, 2018.

[29] Martin Kleppmann. Moving elements in list crdts. In *Proceedings of the 7th Workshop on Principles and Practice of Consistency for Distributed Data*, pages 1–6, 2020.

[30] Martin Kleppmann and Alastair R Beresford. A conflict-free replicated json datatype. *IEEE Transactions on Parallel and Distributed Systems*, 28(10):2733–2746, 2017.

[31] Martin Kleppmann, Adam Wiggins, Peter van Hardenberg, and Mark McGranaghan. Local-first software: you own your data, in spite of the cloud. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pages 154–178, 2019.

[32] Jim Lynch. Tim cook takes apple down the dark road of censorship. URL https://www.cio.com/article/2940563/apple-removes-apps-with-the-confederate-flag.html.

[33] Stéphane Martin, Pascal Urso, and Stéphane Weiss. Scalable xml collaborative editing with undo. In *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*, pages 507–514. Springer, 2010.

[34] Ahmed-Nacer Mehdi, Pascal Urso, Valter Balegas, and Nuno Perguiça. Merging ot and crdt algorithms. In *Proceedings of the First Workshop on Principles and Practice of Eventual Consistency*, pages 1–4, 2014.

[35] Madhavan Mukund, Gautham Shenoy, and SP Suresh. Optimized or-sets without ordering constraints. In *International Conference on Distributed Computing and Networking*, pages 227–241. Springer, 2014.

[36] David Navalho, Sérgio Duarte, and Nuno Preguiça. A study of crdts that do computations. In *Proceedings of the First Workshop on Principles and Practice of Consistency for Distributed Data*, pages 1–4, 2015.

[37] Gérald Oster, Pascal Urso, Pascal Molli, and Abdessamad Imine. Real time group editors without operational transformation. 2005.

[38] Pim Otte, Martijn de Vos, and Johan Pouwelse. Trustchain: A sybil-resistant scalable blockchain. *Future Generation Computer Systems*, 107:770–780, 2020.

[39] Johan A Pouwelse, Pawel Garbacki, Jun Wang, Arno Bakker, Jie Yang, Alexandru Iosup, Dick HJ Epema, Marcel Reinders, Maarten R Van Steen, and Henk J Sips. Tribler: a social-based peer-to-peer system. *Concurrency and computation: Practice and experience*, 20(2):127–138, 2008.

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

[41] Redis. Redis documentation. URL `http://redis.io/documentation`.

[42] Hyun-Gul Roh, Myeongjae Jeon, Jin-Soo Kim, and Joonwon Lee. Replicated abstract data types: Building blocks for collaborative applications. *Journal of Parallel and Distributed Computing*, 71 (3):354–368, 2011.

[43] Hector Sanjuan, Samuli Poyhtari, Pedro Teixeira, and Ioannis Psaras. Merkle-crdts: Merkle-dags meet crdts. *arXiv preprint arXiv:2004.00107*, 2020.

[44] Timos Sellis, Nick Roussopoulos, and Christos Faloutsos. The r+-tree: A dynamic index for multi-dimensional objects. pages 507–518, 1987.

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

[46] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. A comprehensive study of convergent and commutative replicated data types. 2011.

[47] Nikos Smyrnaios. L'effet gafam: stratégies et logiques de l'oligopole de l'internet. *Communication langages*, (2):61–83, 2016.

[48] SoundCloud. Roshi: a crdt system for timestamped events. URL `https://developers.soundcloud.com/blog/roshi-a-crdt-system-for-timestamped-events`.

[49] Brian Stelter. A pulitzer winner gets apple's reconsideration. URL `https://www.nytimes.com/2010/04/17/books/17cartoonist.html`.

[50] Tribler Team. Gumby, . URL `https://github.com/Tribler/gumby`.

[51] Tribler Team. Pyipv8, . URL `https://github.com/Tribler/py-ipv8`.

[52] Werner Vogels. Eventually consistent. *Communications of the ACM*, 52(1):40–44, 2009.

[53] Stéphane Weiss, Pascal Urso, and Pascal Molli. Logoot: A scalable optimistic replication algorithm for collaborative editing on p2p networks. In *2009 29th IEEE International Conference on Distributed Computing Systems*, pages 404–412. IEEE, 2009.

[54] Georges Younes, Ali Shoker, Paulo Sérgio Almeida, and Carlos Baquero. Integration challenges of pure operation-based crdts in redis. In *First Workshop on Programming Models and Languages for Distributed Computing*, pages 1–4, 2016.

[55] Weihai Yu and Claudia-Lavinia Ignat. Conflict-free replicated relations for multi-synchronous database management at edge. In *IEEE International Conference on Smart Data Services, 2020 IEEE World Congress on Services*, 2020.