Electronics and Computer Science

Faculty of Physical Sciences and Engineering

University Of Southampton

Author
Felix Whitefield

April 3, 2023

A Study of Highly Available Move Operations in Tree CRDTs

Project Supervisor
Doctor Corina Cirstea

Second Examiner
Doctor Shoaib Ehsan

A project report submitted for the award of BSc Computer Science.

Abstract

This report is an attempt to use new developments in tree CRDTs with highly available move operations to create a file synchronisation system that will resolve all directory conflicts without human interaction; and will not exhibit 'buggy' behaviour such as duplicating files which some current systems exhibit. This report should provide research into the viability of these systems in a real-world scenario. So far, this paper has reviewed the existing literature and has analysed the concurrency issues facing current systems. As well as detailing two algorithms which have been proposed that state to have solutions to the problem of creating a highly available move operation.

The remaining work includes creating a more refined design of the system and then implementing the separate parts to create a cohesive application. Once implemented, the system will be tested for reliability and performance. This testing will inform the final evaluation of system, where it will be compared to existing solutions.

(rewrite this whole abstract once report is finished)

Statement of Originality

I have read and understood the ECS Academic Integrity[1] information and the University's Academic Integrity Guidance for Students[2].

I am aware that failure to act in accordance with the Regulations Governing Academic Integrity[3] may lead to the imposition of penalties which, for the most serious cases, may include termination of programme.

I consent to the University copying and distributing any or all of my work in any form and using third parties (who may be based outside the EU/EEA) to verify whether my work contains plagiarised material, and for quality assurance purposes.

- I have acknowledged all sources, and identified any content taken from elsewhere.
- I have not used any resources produced by anyone else.
- I did all the work myself, or with my allocated group, and have not helped anyone else.
- The material in the report is genuine, and I have included all my data/code/designs.
- I have not submitted any part of this work for another assessment.
- My work did not involve human participants, their cells or data, or animals

[1] http://ecs.gg/ai
[2] https://www.southampton.ac.uk/quality/assessment/academic_integrity.page
[3] http://www.calendar.soton.ac.uk/sectionIV/academic-integrity-regs.html

Contents

1 Introduction **5**
- 1.1 Problem . . . 5
- 1.2 Goal . . . 5
- 1.3 Scope . . . 6

2 Literature Review **7**
- 2.1 Review of Existing Software . . . 7
  - 2.1.1 Client-server Systems . . . 7
  - 2.1.2 Peer-to-peer Systems . . . 8
- 2.2 Distributed Systems . . . 8
  - 2.2.1 The CAP Theorem . . . 8
  - 2.2.2 The PACELC Theorem . . . 9
- 2.3 Consistency Models . . . 10
  - 2.3.1 Strong and Eventual Consistency . . . 10
  - 2.3.2 Strong Eventual Consistency . . . 10
- 2.4 Logical Clocks . . . 11
  - 2.4.1 Lamport Clocks . . . 11
  - 2.4.2 Vector Clocks . . . 12
- 2.5 Conflict-free Replicated Data Types . . . 13
  - 2.5.1 State-Based (CvRDTs) . . . 14
  - 2.5.2 Operation-Based (CmRDTs) . . . 14
  - 2.5.3 CRDTs and File Systems . . . 16
  - 2.5.4 Move Operations in Tree CRDTs . . . 16
  - 2.5.5 Algorithms With a Highly Available Move . . . 17

3 Design **20**
- 3.1 Requirements . . . 20
- 3.2 System Architecture . . . 21
- 3.3 System Design . . . 22
  - 3.3.1 Clocks and Timestamps . . . 22
  - 3.3.2 CRDT and Interface . . . 23
  - 3.3.3 Network . . . 24
- 3.4 Justification of Design . . . 25
- 3.5 Language and Tools . . . 26
  - 3.5.1 Go . . . 26

4 Project Management **26**
4.1 Account of Work to date 26
4.2 Plan of remaining work 26
4.3 Estimate of Support Required 27
4.4 Gantt Chart 27
4.4.1 Completed Work 27
4.4.2 Remaining Work 28
4.5 Risk Assessment 28

A Original Brief **33**
A.1 Title 33
A.2 Problem 33
A.3 Goal 33
A.4 Scope 34
A.5 Interim Abstract 34

1 Introduction

1.1 Problem

Distributed computing systems are becoming more popular for two main reasons, availability and scalability. Distributed storage systems that have replicas need a way to merge the replicas, however, conflicts can arise when merging concurrent operations. Current software such as Google Drive and Dropbox exhibit bugs in their concurrency control when the file system is concurrently updated on different computers. This can cause different issues, such as duplication, rollbacks and unintended actions. These issues are not helpful to a user and can hinder their productivity. Also, current systems which use a client-server architecture can feel 'slow' due to the latency added by communicating with a server. These systems use a tree-like structure to represent the directory hierarchy.

1.2 Goal

The goal of this project is to implement and test the viability of using new advancements in tree Conflict-Free Replicated Data Types (CRDTs). CRDTs are data types that can be concurrently updated on different nodes (without any coordination), will automatically resolve any differences within the data and are eventually consistent [21]. This project aims to test the reliability (whether the system correctly resolves conflicts) and performance of the implementation. The results of these tests can be used to inform future uses of these CRDTs and should demonstrate the trade-offs of using them. The system should be able to tolerate network failures and offline usage. The implementation will be peer-to-peer, meaning that each node will be equally privileged. Another focus of this project will be on testing the throughput of the new CRDTs as, while CRDTs have naturally low latency, their conflict resolution can cause lower throughput.

Tree structures are used in many scenarios, so the code for this project could be used where a tree structure with a highly available move operation is wanted.

1.3 Scope

The scope of this project will be limited to implementing CRDTs to resolve conflicts and testing their performance and reliability. The system should be tested with a varying number of replicas and conflicts to attempt to visualize how the system would scale. Optimisations will be implemented if time allows.

2 Literature Review

This research will focus on understanding current file synchronisation systems and their drawbacks; the trade-offs that come with distributed systems and their consistency models; and CRDTs and their recent developments in highly available move operations.

2.1 Review of Existing Software

The two main network architectures of file synchronization systems are client-server and peer-to-peer. Below I will discuss both, along with their respective drawbacks.

2.1.1 Client-server Systems

A vast majority of file synchronisation systems are integrated into file backup systems. These systems store the files in the cloud and all devices will be connected and synced with the cloud's version of the files. These services usually sell themselves as 'Cloud Storage', in that your files will be backed up on the cloud and be accessible from all your devices. A few of the popular services include Google Drive[4], OneDrive[5] and Dropbox[6]. On these services, the conflict resolution and merge of changes is all completed on a central server. This means that all updates to the file system have to go through a single server before being relayed to interested clients.

This has the drawback of increased latency as all requests have to go through and be processed by the central server. If many people use the service, then the strain on the server from file transfers would be high and could increase the latency further. These systems also rely on the user trusting the provider to keep their files safe.

As the web evolves there is more talk about Web3[8], which has foundations in decentralisation.

[4]https://www.google.co.uk/intl/en-GB/drive/
[5]https://www.microsoft.com/en-us/microsoft-365/onedrive/online-cloud-storage
[6]https://www.dropbox.com/features/sync

2.1.2 Peer-to-peer Systems

There are many peer-to-peer (p2p) file-sharing programs, with many popular ones being built using the BitTorrent protocol. The most popular BitTorrent clients are uTorrent and BitTorrent[7]. While the BitTorrent protocol is widely used, one of its main drawbacks is that it does not accommodate editing the shared files. A modified version of the BitTorrent protocol has been created to allow for the editing of files called Resilio[8]. This is an improvement on BitTorrent, however, it has some limitations and drawbacks, such as:

- Renaming a folder will not rename the folder on other devices [9].
- Renaming a file will cause other devices to believe it is deleted, and move the file into an archive folder. Then when the peer detects the renamed file, it will check the archive folder for a file with the same hash, and put it back with a new name[10].
  - If the file is renamed and then changed, the hashes will not be the same and therefore will require the transmission of the whole file contents.

One drawback of peer-to-peer file sharing systems is that for the sharing to occur, at least two nodes need to be online at the same time. In comparison to a client-server architecture, where the server is always available. Also, in practice, peer-to-peer networks require some form of centralisation for the initial connection, as you cannot multicast on the web.

2.2 Distributed Systems

2.2.1 The CAP Theorem

In any distributed system that has a persistent state, such as a file system, there is a trade-off between Consistency, Availability and Partition Tolerance. This is set out in the CAP Theorem, which was first introduced by Brewer in 2000[7]. In this talk, Brewer stated that a distributed system could

[7]https://torrentfreak.com/utorrent-is-the-most-used-bittorrent-client-by-far-200405/

[8]https://www.resilio.com/

[9]https://help.resilio.com/hc/en-us/articles/205450655-Can-I-move-or-rename-a-syncing-folder-

[10]https://help.resilio.com/hc/en-us/articles/209606526-What-happens-when-file-is-renamed

only select 2 of the 3 properties. A formal proof for the CAP theorem was provided by Gilbert and Lynch in 2002[9], they formally defined 3 properties as follows:

- **Consistency**: A total order on all operations must exist such that each operation looks as if it was completed at a single instance. An equivalence being requiring requests of a distributed shared memory act as if they were executing on a single node, responding to operations one at a time.
- **Availability**: Every request received by a non-failing node must result in a response. This means any algorithm used by the service must eventually terminate.
- **Partition Tolerance**: The network is allowed to lose arbitrarily many messages sent from one node to another. This can either be all the messages (full partition) or only a portion of the messages (temporary partition).

While this paper proved the CAP Theorem, further research into the area revealed that the initial CAP Theorem was too simplified. As, in reality, it is more of a trade-off between Availability and Consistency, instead of having to choose between the two. In a later paper by Brewer, "CAP Twelve Years Later: How the "Rules" Have Changed"[6], he explains how instead of having to choose 2 of 3 properties, different trade-offs can be made. He also explains how partitions are rare, and therefore usually Consistency or Availability would not need to be sacrificed. Only needing to make sacrifices when partitions are present. Some designers of distributed systems have misunderstood the CAP theorem and have built systems with unnecessary limitations at all times, whereas limitations are only needed in the event of failures [6].

Because the CAP Theorem is somewhat limited, a new theorem was created which takes into account new factors which better reflect the trade-offs placed on distributed systems.

2.2.2 The PACELC Theorem

The PACELC Theorem was proposed by Abadi [1], as an extension to the CAP theorem and to fill the areas in which the CAP theorem does not take

into account. This theorem, therefore, creates a more complete picture of the trade-offs within distributed systems.

PACELC incorporates the CAP Theorem, with the first part, 'PAC', having a similar meaning: when there is a partition (P), how does the system choose between availability (A) and consitency (C). The added part, 'ELC', means: else (E), when there are no partitions, the system has to choose between latency (L) and consistency (C) [1]. It also states that the 'ELC' part (trade-off between latency and consistency) only applies to systems that replicate data. This paper is proposing to create a PA/EL [1] system (highly available, with low latency).

2.3 Consistency Models

2.3.1 Strong and Eventual Consistency

Most distributed data stores today are either:

- **Strongly Consistent**: Any subsequent read after a write will return the most recent value.
- **Eventually Consistent**: The latest data will eventually become available, if no new updates are made.

Many databases are eventually consistent, such as Cassandra[11]. Even databases such as MongoDB[12] (which, by default, is strongly consistent) will become eventually consistent when reading from secondary members. Lower consistency levels don't require the coordiniation which comes with higher levels, and therefore can achieve higher performance because of this. As the need for low latency and higher throughput increases, eventual consistency becomes more desirable over strong consistency.

2.3.2 Strong Eventual Consistency

Strong Eventual Consistency (SEC) takes eventual consistency even further, by guaranteeing that two nodes which receive the same updates (regardless of order) will be in the same state. In eventually consistent systems, the nodes would have to communicate to resolve conflicts by consensus or roll-back.

[11] https://cassandra.apache.org/doc/latest/cassandra/architecture/guarantees.html
[12] https://www.mongodb.com/docs/manual/core/read-isolation-consistency-recency/

Whereas in strong eventually consistent systems the conflicts are resolved in a deterministic manner on each node (without the need for communication).

SEC was proposed in 2011[21] to describe Conflict-free Replicated Data Types (CRDTs). It was defined as:

- **Strong Eventual Consistency** Eventually consistent, as well as conforming to: any two replicas that have received the same updates will have the same state.

2.4 Logical Clocks

Compared to physical clocks, which track time, logical clocks are used to track events. They were designed to capture the happened-before relation between events, which in turn captures causal relationships between them. The happened-before relation is denoted by "$\rightarrow$", and was defined by Lamport in 1978[14] as:

- "If a and b are events in the same process, and a comes before b, then $a \rightarrow b$"

This happened-before relation can be used to track the causal dependencies, as if $a \rightarrow b$, then b is *causally dependent* on a. Which means that b may have been influenced by a.

2.4.1 Lamport Clocks

Lamport clocks were created by Lamport in 1978[14]. They work by having each node maintain their own counter, which is incremented on each local event. This counter value is attached to the event, and is the lamport timestamp for that event. When nodes send messages to each other, they send their current counter value along with it. Then, the receiving node updates their counter to the received lamport timestamp if it is greater than their current counter value. It is important to note that sending and receiving messages both count as events.

Lamport clocks give us the following guarantee, where $L(a)$ is the lamport timestamp of event a:

- $a \rightarrow b \implies L(a) < L(b)$

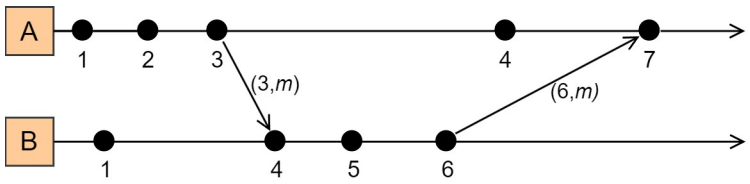


Figure 1: Example of lamport clocks. m represents a message.

This is useful for detecting potential causal dependencies. However, by using lamport clocks, concurrent events may appear to be causally dependent. One use for lamport clocks is to create a total order ($\prec$) which captures the causal dependencies. This can be achieved by assigning each node a unique ID, and using that to break ties when $L(a) = L(b)$. When an event happens on a node, the node ID is then also stored in the lamport timestamp. It is worth noting that this total order is arbitrary, but deterministic. This was defined by Lamport[14], and can be denoted as:

$$a \prec b \iff L(a) < L(b) \vee (L(a) = L(b) \wedge ID(a) < ID(b))$$

2.4.2 Vector Clocks

Vector clocks are another logical clock that provide more gurantees than lamport clocks. Each node maintains a vector timestamp, which is a vector of counters. Each counter corresponds to a node in the system. When an event occurs on a node, the node increments its own counter in the vector and sends the updated vector to the other nodes. When a node receives a timestamp, it updates its own vector timestamp to be the maximum of the two timestamps. [16].

Vector timestamps can be ordered by the following rules, where $V(a)$ is the timestamp of event a and $V(a)_i$ is the value of the counter for node i:

$$V(a) \leq V(b) \iff \forall_i [V(a)_i \leq V(b)_i]$$

$$V(a) = V(b) \iff \forall_i [V(a)_i = V(b)_i]$$

Lamport clocks can only indicate *potential* causal dependencies. Vector timestamps are larger than Lamport timestamps, but they precisely indicate

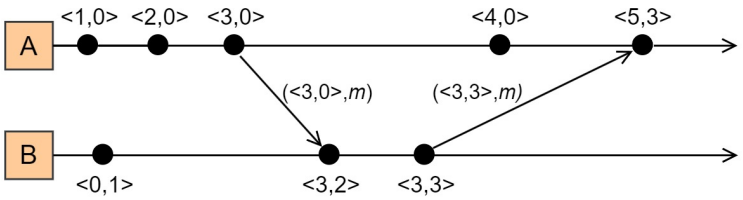


Figure 2: Example of vector clocks. The first and second values correspond to nodes A and B respectively. m represents a message.

the happened-before relation between events in a system. This is denoted by the following (|| denotes concurrent):

- $a \rightarrow b \iff V(a) \leq V(b) \wedge V(a) \neq V(b)$
- $a = b \iff V(a) = V(b)$
- $a \,||\, b \iff V(a) \not\leq V(b) \wedge V(b) \not\leq V(a)$

This can be used to create a partial order, as concurrent events are neither causally related nor ordered with respect to each other.

2.5 Conflict-free Replicated Data Types

Conflict-free Replicated Data Types are types of data structures, such as sets, counters, and registers, that can be replicated across multiple nodes in a distributed system. They are unique in that the data type itself is responsible for resolving conflicts, rather than the application.

CRDTs were created to allow for a better eventually consistent model, that can increase availability and performance while removing the need for conflict arbitration. One main factor that allows CRDTs to guarantee SEC is that the data types are commutative[21], meaning that the updates can be applied in any order. They also do not require a main or primary elected server, as each node can resolve the conflicts by themselves. Therefore, CRDTs are applicable in a peer-to-peer environment.

One useful application for CRDTs is their ability for offline use [12]. If an application is built using CRDTs, it can feel more responsive as updates to the state can be made locally first without the system having to wait on replies from remove servers.

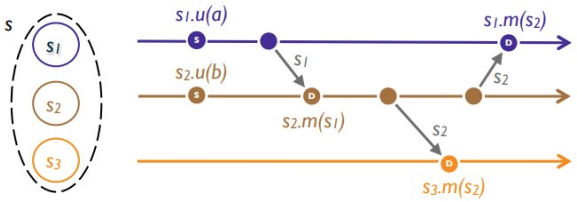


Figure 3: State-based CRDT replication. Reproduced from [21]

There are two main types of CRDTs, state-based and operation-based. Both have different ways of sending updates to other nodes, and different ways of resolving conflicts.

2.5.1 State-Based (CvRDTs)

State-based CRDTs, or otherwise called Convergent Replicated Data Types (CvRDTs), send their full state to other replicas where they are merged by a function to form a new state. The merge function must be associative, commutative and idempotent [21]. CvRDTs can use lots of bandwidth because as the state grows in size, they will still have to send the whole state. However, because they send their full state, if the transmission of a state is 'missed' it does not break the system as the next state transmission will contain all previous 'missed' updates to the state. Also, CvRDTs can make use of gossip protocols (as shown in Figure 3) which helps reduces network usage.

Delta State CRDTs are a form of state-based CRDTs that only send the changed, or the delta, part of the state. This allows for the messages sent between replicas to be smaller and work across unreliable networks [2]. They, therefore, have the best of both operation and state-based CRDTs.

2.5.2 Operation-Based (CmRDTs)

Operation-based CRDTs, or otherwise called Commutative Replicated Data Types (CmRDTs), only send the update operation to other replicas. Each operation or update is a pair consisting of a *prepare* method and an *effect* method. The *prepare* method is executed at the local replica and produces a message representing the operation, and the *effect* method applies this operation at all replicas[21]. The operations also have to be commutative, but

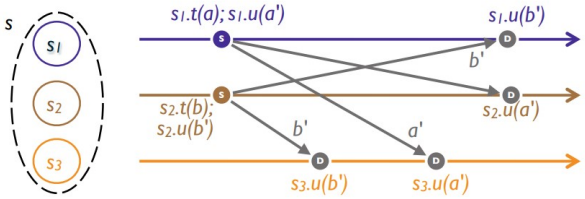


Figure 4: Operation-based CRDT replication. Reproduced from [21]

not idempotent. Because of the lack of idempotence, there are more requirements on transmission between replicas as they have to ensure each operation is sent to each replica only once (This can be seen in Figure 4).

Pure operation-based CRDTs were proposed to create better operation-based CRDTs[4]. These impose restrictions on the *prepare* method, requiring that it does not inspect the current state and can only return the operation. They also require the state of the object to be comprised of a partially ordered log of operations. As well as defining an extension which strips causality information from operations once they are *"causally stable"*[4], alllowing for the state to be smaller.

It is important to note that operation-based CRDTs are more applicable to *static* distributed systems, where the number of replicas is fixed. Papers about operation-based CRDTs usually assume a fixed number of replicas, such as [3]. There are ways to make operation-based CRDTs work well in *dynamic* systems, such as using dotted version vectors [20]. For the remainder of this paper, the terms 'static' and 'dynamic' systems will be used as defined above. The focus of this paper will be on static systems, as dynamic systems are beyond its scope.

It is important to note that operation-based CRDTs usually require reliable causal broadcast [4]. These algorithms can be complex, but they are not the focus of this paper. Therefore, this paper will implement a simple reliable causal broadcast algorithm. If the CRDTs discussed in this paper are to be used in a real-world application, a more complex reliable causal broadcast algorithm should be used [5].

2.5.3 CRDTs and File Systems

A file system can be represented as a tree structure, where the files and directories are the nodes and the directory hierarchy is represented by the branches between nodes. Therefore, a tree CRDT can be used to create a distributed/replicated file system that will have high availability and low latency. Tree-structured CRDTs have been created before [22], [10]. However, designing a tree CRDT with a *highly available* move operation poses significant challenges due to the need to preserve the strict tree invariant. *Highly available* meaning: does not require locking, consensus and does not create duplicates.

This can be shown by ElmerFS (A file system created using CRDTs), where concurrent moves can create a cycle [23]. Najafadeh et al. [18] proposes a similar system which instead locks on move operations to ensure cycles are not created. These papers show how hard creating a highly available move operation is, however recent work has shown that it is possible [11], [17], [13].

2.5.4 Move Operations in Tree CRDTs

Tree CRDTs themselves are not difficult to implement, as shown by the paper "Abstract unordered and ordered trees CRDT" [15] from 2012. Tree CRDTs have also been using in real-time collaborative editing applications [19]. However, these implementations only allow for adding and removing nodes, not moving them. This is because a move operation is difficult to implement, as concurrent moves can break the tree invariant. Therefore, this section will exclusively consider conflicts that arise from move operations, as these represent new advancements in tree CRDTs.

There are three main operations that will need to be considered when implementing conflict resolution. These being: *add(p,n)*, *remove(n)* and *move(p,n)*. Where p is the parent node, and n is the globally unique id of the node to be added, removed or moved. *Add* will add node n under parent node p, *remove* will remove node n and *move* will move node n to parent node p. Considering these operations, the following describes the conflicts that can arise from concurrent operations involving move operations:

- *Move-Add* conflict - This conflict has one case:

  - The move operation moves node n_1 with name m is under node p, while the add operation adds node n_2 with name m under node p[13]

- *Move-Remove* conflict - This conflict has two cases:
  - The move operation moves node n under node p, while the remove operation removes node p
  - The move operation moves node n under node p, while the remove operation removes node n
- *Move-Move* conflict - This conflict has three cases:
  - One move operation moves node n_1 with name m under node p, while the other move operation moves node n_2 with name m under node p[13]
  - Two move operations move the same node n to different parents p_1 and p_2
  - One move operation moves node n_1 under node n_2, while the other move operation moves node n_2 under node n_1, creating a cycle

The conflict resolution will not be discussed here, as it will be a part of the algorithms discussed in the next section. Each algorithm will have its own conflict resolution strategy. It is important to note that the algorithms, as they are, do not uniquely identify nodes by their name and parent. And therefore, do not resolve those conflicts. These algorithms may be adapted to resolve these conflicts, and if time permits, this paper will explore such modifications.

2.5.5 Algorithms With a Highly Available Move

Currently, there two main proposed algorithms that allow a highly available move operation:

- **Kleppmann et al.** [11] proposes an algorithm that represents the tree as a set of parent-child relationships. A move operation is performed by removing the node from wherever it is in the tree and moving it to its

[13]This conflict will only arise if nodes are uniquely identified by their name and parent, as is typical in file systems. (However, for other systems this may not be the case)

new location. This operation can also be used for adding and removing nodes, the former by creating a new node under the specified parent and the latter by moving the node under a "trash" node. It also stores a *log* of all previous operations, which allows the algorithm to implement a move operation. The algorithm can use the *log* to ensure that all operations are applied in the correct order by undoing and reapplying operations (as proposed in [13]). It deals with cycles by ensuring that if any operations were to create a cycle, it would be ignored however they are still added to the *log*. In general, the algorithm will prefer the latest operation that does not cause a conflict. The only conflict which the algorithm does not resolve is child nodes with the same parent having the same name.

- **Nair et al. [17]** proposes *Maram*, a "light-weight" tree CRDT with a highly available move operation that represents the tree a set of nodes and child-to-parent relations. Instead of storing an in-order log of operations, this algorithm relies on a causal delivery layer which guarantees a happened-before relation. Maram splits move operations into *down-moves* (the node is moved away from the root) and *up-moves* (the node is moved towards to root or to the same distance). The conflict resolution is as follows: in concurrent up-moves, they are safe (so no conflict resolution required); in a concurrent up-move and down-move, the up-move wins; in concurrent down-moves, the move with the highest priority wins (the priority is deterministic, and specific to each application).

These will be referred to as *Kleppmann's algorithm* and *Maram* respectively, and will be the main focus of this paper.

Table 1 shows a comparison between the two algorithms:

[i] *Causal Consistency* gurantees that if two operations are causally related, then the second operation will be applied after the first operation on all replicas. For example, if a replica receives operation A then creates operation B, then operation A should be applied before operation B on all other replicas.

Table 1: Comparison of Algorithms

	Kleppmann's [11]	**Nair's (*Maram*) [17]**
Delivery Layer	Eventual Consistency *Low Cost*	Causal Consistency[i] *Higher Cost*
Operation Order	Total Order *High Cost*	Partial Order *Low Cost*

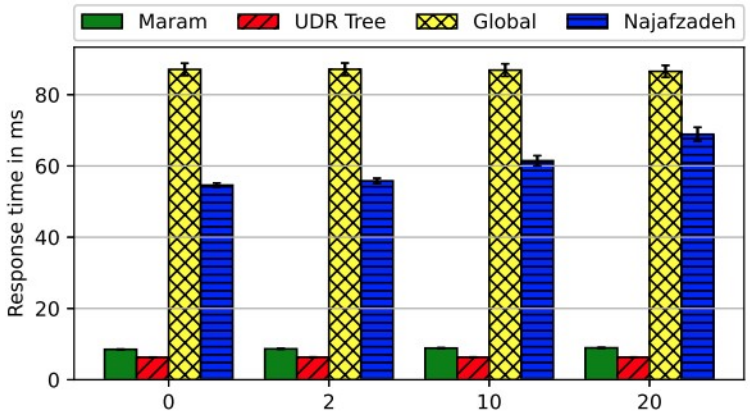


Figure 5: Response time for different conflict rates (0-20%). Reproduced from [17]

Figure 5 shows differences in response times between different algorithms. Kleppmann's algorithm [11] can be regarded as a UDR Tree, and therefore we can see that it has a slightly lower response time than Maram [17]. Figure 5 also shows how a lower response time can be achieved by using non-locking algorithms.

This paper will use both of the algorithms stated above, Kleppmann's [11] and Nair's (*Maram*) [17]. The implementation of these algorithms, and subsequent testing, will be the main part of this project. The testing will inform the evaluation, which will compare the two algorithms and determine their viability. Notably, these algorithms require a separate delivery layer to communicate between replicas. This layer will be implemented separately from the CRDTs.

3 Design

The original aim of this paper was to create a file synchronisation system using conflict-free replicated data types (CRDTs). However, due to the complexity and challenges of implementing such a system, this paper has narrowed its focus to explore the implementation and evaluation of the CRDTs. By narrowing down the scope of this research, this paper aims to provide a more in-depth analysis of the advantages and limitations of the CRDTs. This will allow for a more thorough evaluation of the CRDTs, and will allow for a more detailed comparison between the different CRDTs.

3.1 Requirements

The main focus of this paper is to implement the two CRDT algorithms, [11] and [17], and to test them individually as well as to compare them. While the evaluation of these algorithms will be the main focus, this paper will also attempt to create implementations that can be used by other developers. As such, the requirements will be as follows:

- The system shall be comprised of multiple layers that can be modified independently.
- The system should have a simple and easy to use API.
- The implementations should follow the literature as closely as possible, to ensure they are logically correct.
- The system shall support immediate local execution of operations on the CRDTs.
- The replication between replicas shall be handled by a separate layer than that which handles the CRDTs. This will allow for the replication layer to be modified independently of the CRDT layer, and vice versa.
- The replication layer should handle the communication between replicas asynchronously.
- Each layer should have an interface that allows for other implementations to be created and swapped in. This will allow for the system to be easily modified and extended.

3.2 System Architecture

The design will be split into different layers that will each be independent and can be modified separately. The layers are as follows:

- **CRDT** This layer will be in charge of conflict resolution and is the main focus of this paper. The CRDTs will be implemented using the algorithms described in Section 2.5.5. This layer is responsible for the following:
  - Applying operations (which will be *local* or *remote*) on the CRDT
  - Resolving conflicts that arise between concurrent operations
  - Ensuring that all replicas eventually converge to the same state, if they have all received the same operations
  - Keeping track of the state of the CRDT (the state of the tree)
  - Providing an interface for application code to interact with the CRDT
- **Interface** This layer will provide a uniform interface to the user, as well as provide the connection between the CRDT and Network layer. It will need to be able to:
  - Ensure that operations are applied to the local CRDT only when they are ready to be applied (This will depend on the chosen consistency model)
  - Buffer operations that are not ready to be applied on the local CRDT
  - Provide an interface for application code to interact with the CRDT
  - Ensure that all operations are sent to the network layer for replication
- **Network** This layer will be in charge of network communications. It will need to be able to:
  - Connect to other peers, and share peers with other peers
  - Send and Receive operations to and from other peers

- Ensure that operations are sent to all peers, even if some peers are offline
- Buffer operations to be sent to peers that are offline

Certain CRDTs require different consistency models, therefore the interface layer will ensure that incoming operations are applied correctly. It is important that each CRDT is correctly paired with the correct consistency model, as otherwise the CRDT may not work correctly.

3.3 System Design

The system will be designed using the requirements stated above. The design will attempt to use interfaces where possible, to allow for the system to be easily modified and extended.

3.3.1 Clocks and Timestamps

The system has two interfaces for tracking the order of events, clocks and timestamps. Moving forward, *clocks* will refer to a timestamp for a specific actor, and *timestamps* will refer to a timestamp for a specific event. As shown in Figure 6, the system will have two implementations of clocks, Lamport and Vector clocks. These will be used within their respective CRDTs, as described in Section 2.5.5.

The two interfaces, `Clock` and `Timestamp`, are shown in Figure 6. The class `Lamport` implements both, as it is both a timestamp and a clock. The Vector clock is split into two classes, `VectorClock` and `VectorTimestamp`. The `VectorClock` implements the `Clock` interface, and the `VectorTimestamp` implements the `Timestamp` interface.

The `Lamport` and `VectorTimestamp` classes implement `TotalOrder` and `PartialOrder` respectively. They denote the differences in the gurantees that the two types of clocks provide. The `Lamport` class implements `TotalOrder` as it provides an arbitrary, but deterministic total order of events. The `VectorTimestamp` class implements `PartialOrder` as it provides a partial order of events.

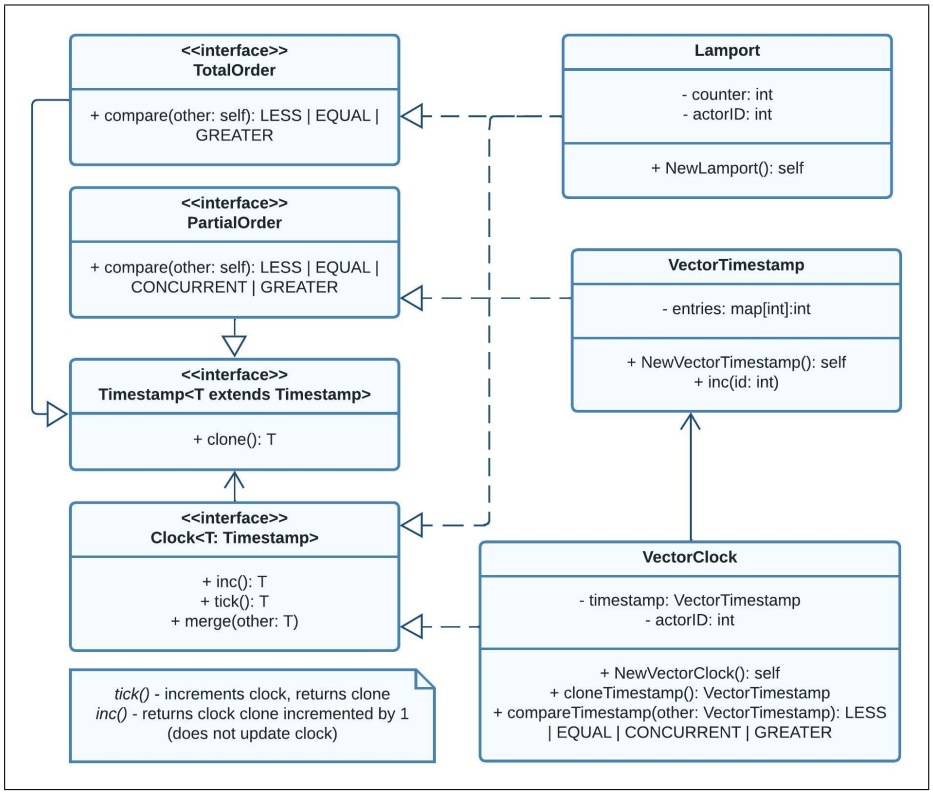


Figure 6: Class diagram for the clocks and timestamps. Including Lamport and Vector clocks.

3.3.2 CRDT and Interface

Since the CRDT layer is the primary focus of this paper, it will comprise multiple components, making it the largest layer. The components will be specific to each CRDT, and will be implemented using the algorithms described in Section 2.5.5.

The exact `Components` of each CRDT will be known once the algorithms have been implemented. It may also be possible to provide a `CmRDT` interface for the CRDTs, with *prepare* and *effect* methods. However, this will not be explored in this paper.

The classes that implement `Tree` from Figure 7 will also be responsible for sending operations to the network layer to be broadcast to other peers. As well as receiving operations from the network layer, and either applying

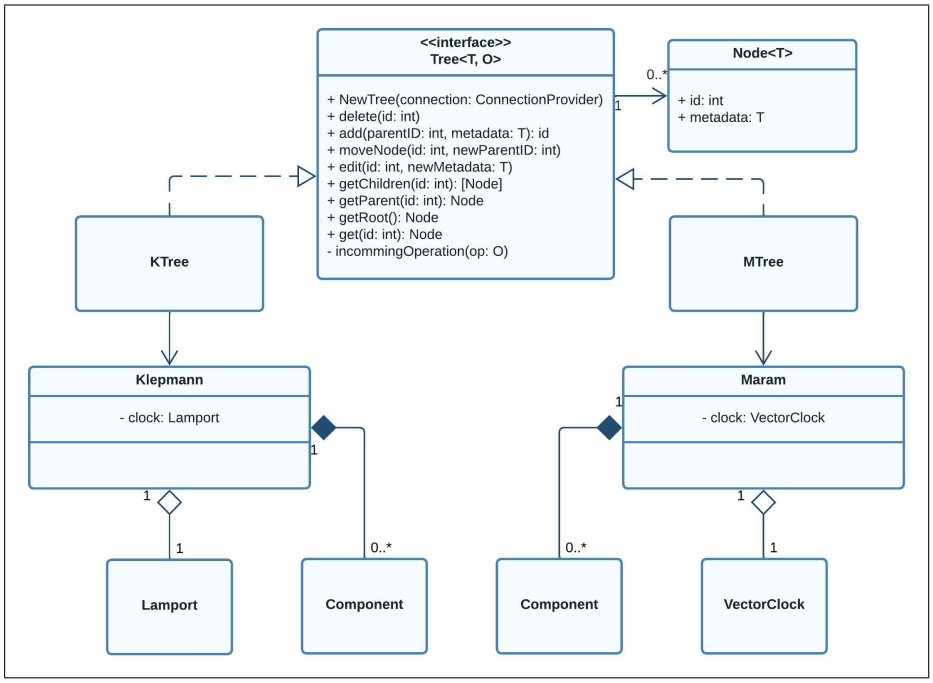


Figure 7: Class diagram for the CRDT and Interface layer. The `Tree` interface provides an interface for application code. Classes `MTree` and `KTree` provide implementations for the specific CRDTs.

them to the local CRDT or buffering them, depending on the consistency model.

3.3.3 Network

The network layer takes inspiration from Yjs[14], however the implementation is somewhat reversed. In Yjs, the network layer binds to the CRDT, in this paper the Interface layer binds to the network layer. The main reason for this is that this paper is designing a static system, with a fixed number of nodes and therefore this design is more suitable.

If the system was to be designed to be dynamic, then the CRDT layer would have more requirements. The CRDT would either need to store all local operations indefinitely or provide a means of transferring its state to new

[14]https://docs.yjs.dev/

nodes, as they would require access to the current state of the CRDT. This helps to simplify the design, as the network layer does not need to know about the CRDTs, and the CRDTs do not need to know about the network layer.

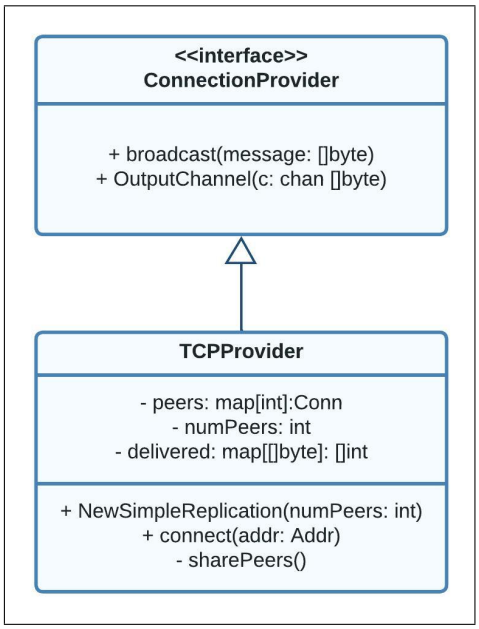


Figure 8: Class diagram for the Network layer. With the interface `ConnectionProvider` and implementing class `TCPProvider`.

The interface `ConnectionProvider` is shown in Figure 8. It describes what the network layer is required to do, broadcast messages and receive them. This interface is implemented by the class `TCPProvider`. The `TCPProvider` class is responsible for creating and managing connections to other peers. It will also be responsible for sending and receiving messages to and from other peers.

3.4 Justification of Design

The system is decoupled into three distinct layers, which will improve the maintainability, readability and re-usability of the code. It will also help during development, as each layer will only need to focus on its own requirements. As well as allowing for each layer to be tested individually.

3.5 Language and Tools

3.5.1 Go

Go is a language created by Google, and is designed to be used in distributed systems with the help of its built-in, lightweight concurrency features. As this paper aims to create a distributed system, then Go is a good choice to help with this.

The CRDTs which will be implemented by this paper are particularly applicable to distributed file systems. Recent distributed file systems have been written using Go, such as JuiceFS or Kertish-DFS. Therefore, by using Go this paper aims to create useful implementations for these systems, or to provide a base implementation which can then be improved upon.

4 Project Management

4.1 Account of Work to date

The current work has been around reviewing relevant literature and creating an initial design from this research. The research has been into current systems for file synchronisation, and their drawbacks. As well as into distributed systems and consistency models. Most importantly, detailed research has been completed into CRDTs and ones applicable to this paper's proposed system. From this research, an initial design has been created outlining the main aspects of the proposed system.

4.2 Plan of remaining work

The design will be refined and more detailed. From this design, the CRDT layer will be implemented by first selecting an algorithm and then coding it. Then, the Network layer, where the peer-to-peer networking will be coded first, followed by operation transmission and finally file data transmission. Then, the File System layer, where the watcher (which watches the file system for changes) will be coded and then reading and writing file content. Development of these layers may overlap, but it will follow the general structure listed.

Once all components have been coded, the system will have its reliability and

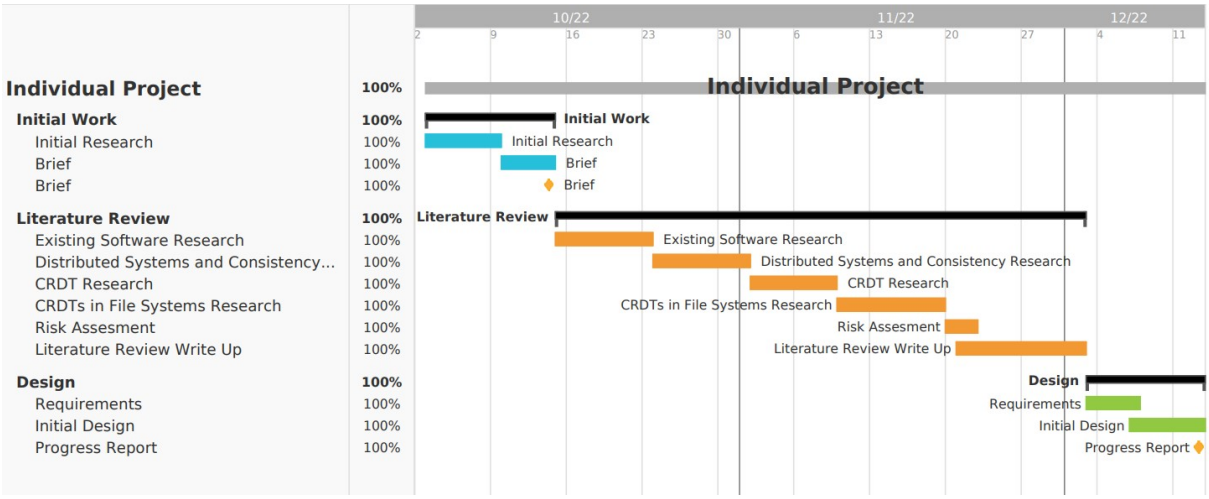
performance tested, which will inform the evaluation. The testing will seek to measure the throughput and response time of the algorithm(s), and how adding more replicas affects this. The evaluation then needs to be written and will contain the outcome of the project.

4.3 Estimate of Support Required

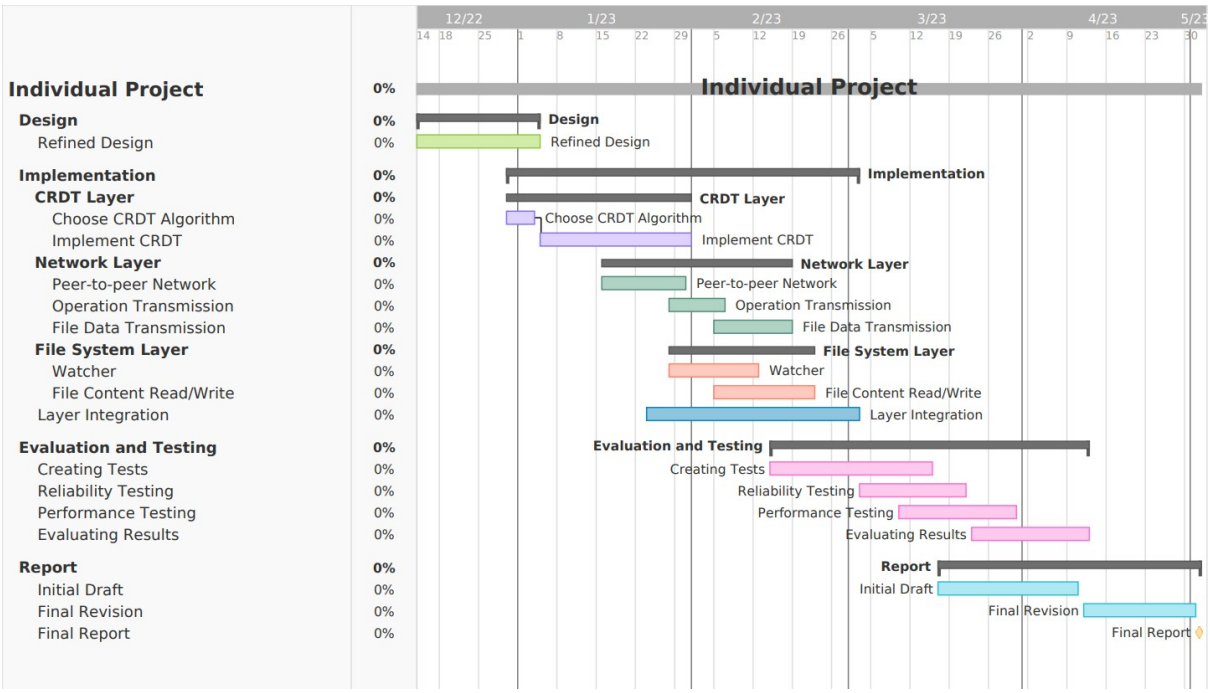
The performance testing may require the use of rented servers, to test latency.

4.4 Gantt Chart

4.4.1 Completed Work



4.4.2 Remaining Work



4.5 Risk Assessment

- **(P) Probability** 1 - low, 5 - high
- **(S) Severity** 1 - low, 5 - high
- **(RE) Risk Exposure** Probability * Severity

Table 2: Risk Assessment

Risk	**P**	**S**	**RE**	**Mitigation**
Loss of Report	2	5	10	Back-up on computer, laptop and OneDrive
Loss of Code	1	5	5	Back-up on computer, laptop, OneDrive and GitHub
Underestimating Tasks	3	3	9	Plan for extra time in case tasks require it, as well as making informed estimates for time needed for each task
Health Issues	2	4	8	Allow for spare time, and possible removal of parts of project. Attempt to not become sick.
Implementation Difficulties	3	4	12	Create a clear plan of work to do, and take time to understand the design. Can seek help if needed.
Change in Scope	2	4	8	Complete implementation early to allow for changes to the scope.

References

[1] Daniel Abadi. "Consistency Tradeoffs in Modern Distributed Database System Design: CAP is Only Part of the Story". In: *Computer* 45.2 (2012), pp. 37–42. DOI: 10.1109/MC.2012.33.

[2] Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. "Delta state replicated data types". In: *Journal of Parallel and Distributed Computing* 111 (Jan. 2018), pp. 162–173. DOI: 10.1016/j.jpdc.2017.08.003. URL: https://doi.org/10.1016%5C%2Fj.jpdc.2017.08.003.

[3] Carlos Baquero, Paulo Sergio Almeida, and Ali Shoker. *Pure Operation-Based Replicated Data Types*. 2017. arXiv: 1710.04469 [cs.DC].

[4] Carlos Baquero, Paulo Sérgio Almeida, and Ali Shoker. "Making Operation-Based CRDTs Operation-Based". In: *Distributed Applications and Interoperable Systems*. Ed. by Kostas Magoutis and Peter Pietzuch. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 126–140. ISBN: 978-3-662-43352-2.

[5] Kenneth Birman, André Schiper, and Pat Stephenson. "Lightweight Causal and Atomic Group Multicast". In: *ACM Trans. Comput. Syst.* 9.3 (Aug. 1991), pp. 272–314. ISSN: 0734-2071. DOI: 10.1145/128738.128742. URL: https://doi.org/10.1145/128738.128742.

[6] Eric Brewer. "CAP twelve years later: How the "rules" have changed". In: *Computer* 45.2 (2012), pp. 23–29. DOI: 10.1109/MC.2012.37.

[7] Eric Brewer. "Towards robust distributed systems". In: Jan. 2000, p. 7. DOI: 10.1145/343477.343502.

[8] Ethereum. *Introduction to Web3*. URL: https://ethereum.org/en/web3/.

[9] Seth Gilbert and Nancy Lynch. "Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services". In: *SIGACT News* 33.2 (June 2002), pp. 51–59. ISSN: 0163-5700. DOI: 10.1145/564585.564601. URL: https://doi.org/10.1145/564585.564601.

[10] Martin Kleppmann and Alastair R Beresford. "Automerge: Real-time data sync between edge devices". In: *1st UK Mobile, Wearable and Ubiquitous Systems Research Symposium (MobiUK 2018). https://mobiuk.org/abstract/S4-P5-Kleppmann-Automerge. pdf*. 2018, pp. 101–105.

[11] Martin Kleppmann et al. "A Highly-Available Move Operation for Replicated Trees". In: *IEEE Transactions on Parallel and Distributed Systems* 33.7 (2022), pp. 1711–1724. DOI: 10.1109/TPDS.2021.3118603.

[12] Martin Kleppmann et al. "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*. Onward! 2019. Athens, Greece: Association for Computing Machinery, 2019, pp. 154–178. ISBN: 9781450369954. DOI: 10.1145/3359591.3359737. URL: https://doi.org/10.1145/3359591.3359737.

[13] Martin Kleppmann et al. *OpSets: Sequential Specifications for Replicated Datatypes (Extended Version)*. 2018. DOI: 10.48550/ARXIV.1805.04263. URL: https://arxiv.org/abs/1805.04263.

[14] Leslie Lamport. "Time, Clocks and the Ordering of Events in a Distributed System". In: *Communications of the ACM 21, 7 (July 1978), 558-565. Reprinted in several collections, including Distributed Computing: Concepts and Implementations, McEntire et al., ed. IEEE Press, 1984.* (July 1978). 2000 PODC Influential Paper Award (later renamed the Edsger W. Dijkstra Prize in Distributed Computing). Also awarded an ACM SIGOPS Hall of Fame Award in 2007., pp. 558–565. URL: https://www.microsoft.com/en-us/research/publication/time-clocks-ordering-events-distributed-system/.

[15] Stéphane Martin, Mehdi Ahmed-Nacer, and Pascal Urso. *Abstract unordered and ordered trees CRDT*. 2012. arXiv: 1201.1784 [cs.DS].

[16] Friedemann Mattern. "Virtual Time and Global States of Distributed Systems". In: *Parallel and Distributed Algorithms*. North-Holland, 1989, pp. 215–226.

[17] Sreeja Nair et al. *A coordination-free, convergent, and safe replicated tree*. 2021. DOI: 10.48550/ARXIV.2103.04828. URL: https://arxiv.org/abs/2103.04828.

[18] Mahsa Najafzadeh, Marc Shapiro 0001, and Patrick Eugster. "Co-Design and Verification of an Available File System". In: *Verification, Model Checking, and Abstract Interpretation - 19th International Conference, VMCAI 2018, Los Angeles, CA, USA, January 7-9, 2018,*

Proceedings. Ed. by Isil Dillig and Jens Palsberg. Vol. 10747. Lecture Notes in Computer Science. Springer, 2018, pp. 358–381. ISBN: 978-3-319-73721-8. DOI: 10.1007/978-3-319-73721-8_17. URL: https://doi.org/10.1007/978-3-319-73721-8_17.

[19] Nuno Preguica et al. "A Commutative Replicated Data Type for Cooperative Editing". In: *2009 29th IEEE International Conference on Distributed Computing Systems*. 2009, pp. 395–403. DOI: 10.1109/ICDCS.2009.20.

[20] Nuno Preguiça et al. *Dotted Version Vectors: Logical Clocks for Optimistic Replication*. 2010. arXiv: 1011.5808 [cs.DC].

[21] Marc Shapiro et al. "Conflict-Free Replicated Data Types". In: *Stabilization, Safety, and Security of Distributed Systems*. Ed. by Xavier Défago, Franck Petit, and Vincent Villain. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 386–400. ISBN: 978-3-642-24550-3.

[22] Vinh Tao, Marc Shapiro, and Vianney Rancurel. "Merging Semantics for Conflict Updates in Geo-Distributed File Systems". In: *Proceedings of the 8th ACM International Systems and Storage Conference*. SYSTOR '15. Haifa, Israel: Association for Computing Machinery, 2015. ISBN: 9781450336079. DOI: 10.1145/2757667.2757683. URL: https://doi.org/10.1145/2757667.2757683.

[23] Romain Vaillant et al. "CRDTs for Truly Concurrent File Systems". In: *Proceedings of the 13th ACM Workshop on Hot Topics in Storage and File Systems*. HotStorage '21. Virtual, USA: Association for Computing Machinery, 2021, pp. 35–41. ISBN: 9781450385503. DOI: 10.1145/3465332.3470872. URL: https://doi.org/10.1145/3465332.3470872.

A Original Brief

A.1 Title

Using CRDTs to Create a File Synchronization System

A.2 Problem

Distributed computing systems are becoming more popular for two main reasons, availability and scalability. Distributed storage systems that have replicas need a way to merge the replicas, however conflicts can arise when merging two different replicas. Software such as Google Drive and Dropbox exhibit bugs in their concurrency control when the filesystem is concurrently updated on different computers. CRDTs (Conflict-free Replicated Data Types) can be implemented to ensure strong eventual consistency (SEC) as well as remove the need for a centralized server or leader. SEC ensures that two nodes/peers will converge to the same state even if the order in which they receive a set of updates is different. CRDTs can have the benefit of a low-latency because updates can be made locally without having to contact another node, however their throughput may be lower than other solutions, such as state machines, due to them having to resolve conflicts. Due to the high guarantees of CRDTs, implementations for complex data types are still being theorized and tested in practice (such as tree CRDTs).

A.3 Goal

The goal of this project is to implement and test the performance and viability of using new advancements in tree CRDTs in a file synchronization system to solve directory conflicts. This project aims to be able to demonstrate whether using CRDTs in this application is viable, mainly focusing on the throughput and reliability of the system. The system should be able to tolerate network failures and offline usage. The implementation will be peer-to-peer, meaning that each node will be equally privileged. Another focus of this project will be on optimisation and so different methods will be documented to show how this was achieved, with a focus on increasing throughput (as CRDTs have naturally low latency). Tree structures are used in many scenarios so the code for this project could be used in other scenarios when Strong Eventual Consistency is wanted.

A.4 Scope

The scope of this project will be limited to implementing CRDTs to resolve directory conflicts and testing their performance and reliability. The system should be tested with a varying number of replicas to attempt to visualize how the system would scale. Optimisations will only be implemented after the system works as intended. File conflicts will not be a focus of this project, however they could be brought into the scope if time allows.

A.5 Interim Abstract

This report is an attempt to use new developments in tree CRDTs with highly available move operations to create a file synchronisation system that will resolve all directory conflicts without human interaction; and will not exhibit 'buggy' behaviour such as duplicating files which some current systems exhibit. This report should provide research into the viability of these systems in a real-world scenario. So far, this paper has reviewed the existing literature and has analysed the concurrency issues facing current systems. As well as detailing two algorithms which have been proposed that state to have solutions to the problem of creating a highly available move operation. The remaining work includes creating a more refined design of the system and then implementing the separate parts to create a cohesive application. Once implemented, the system will be tested for reliability and performance. This testing will inform the final evaluation of system, where it will be compared to existing solutions.