Electronics and Computer Science

Faculty of Physical Sciences and Engineering

University Of Southampton

Author

Felix Whitefield

April 26, 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

[Due to be rewritten]

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`

Acknowledgements

I would like to express my sincere gratitude to the authors of the paper "A highly-available move operation for replicated trees"[11] and "A coordination-free, convergent, and safe replicated tree"[17] for their work. The algorithms which they have proposed has been an essential part of this report, and I am truly grateful for their efforts. This paper has attempted to implement their proposed algorithms in Go.

The source code for this report also makes use of the following open source packages:

Name	**URL**
uuid	`https://pkg.go.dev/github.com/google/uuid@v1.3.0`
protobuf	`https://pkg.go.dev/github.com/vmihailenco/msgpack@v4.0.4+incompatible`
msgpack	`https://pkg.go.dev/google.golang.org/protobuf@v1.30.0`

The protocol buffer compiler *protoc*, with plugin *protoc-gen-go* was used to generate the Go code for the protobuf messages. The generated code is within `*.pb.go` files, and was not written by Felix Whitefield.

Contents

Chapter 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. These systems use a tree-like structure to represent the directory hierarchy. Moreover, current systems which use a client-server architecture can feel 'slow' due to the latency added by having all updates go through the server.

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 [21]. This project aims to test the reliability (whether the system correctly resolves conflicts) and performance of the new tree CRDT algorithms. 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 visualise how the system would scale. The scope will be limited to implementing the CRDT algorithms and an accompanying, simple peer-to-peer network. Adding security or a more complex networking layer is out of the scope for this project. Also, more in-depth optimisation of the CRDTs is out of the scope of this project, however, the data structures used will attempt to be as efficient as possible.

Chapter 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; logical clocks; and CRDTs and their recent developments in highly available move operations.

2.1 Review of Existing Software

The two main network architectures of file synchronisation 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[1], OneDrive[2] and Dropbox[3]. On these services,

[1] `https://www.google.co.uk/intl/en-GB/drive/`

[2] `https://www.microsoft.com/en-us/microsoft-365/onedrive/online-cloud-storage`

[3] `https://www.dropbox.com/features/sync`

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.

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[4]. 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[5]. 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 [6].
- 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[7].
  - 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

[4] https://torrentfreak.com/utorrent-is-the-most-used-bittorrent-client-by-far-200405/
[5] https://www.resilio.com/
[6] https://help.resilio.com/hc/en-us/articles/205450655-Can-I-move-or-rename-a-syncing-folder-
[7] https://help.resilio.com/hc/en-us/articles/209606526-What-happens-when-file-is-renamed

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 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[8]. Even databases

[8] `https://cassandra.apache.org/doc/latest/cassandra/architecture/guarantees.html`

such as MongoDB[9] (which, by default, is strongly consistent) will become eventually consistent when reading from secondary members. Lower consistency levels do not 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. 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.

[9] `https://www.mongodb.com/docs/manual/core/read-isolation-consistency-recency/`

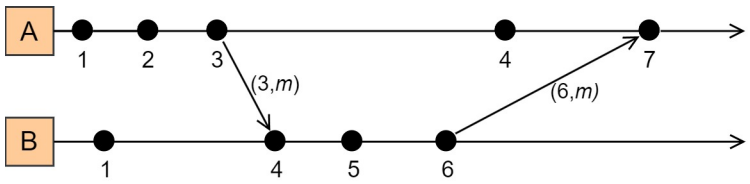


Figure 2.1: Example of lamport clocks. m represents a message.

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:

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

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

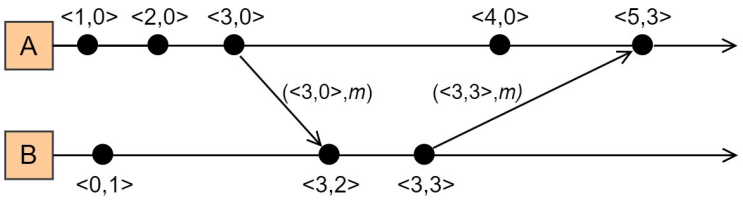


Figure 2.2: Example of vector clocks. The first and second values correspond to nodes A and B respectively. m represents a message.

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

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

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 (CRDTs) are types of data structures, such as sets, counters, and registers, that can be replicated across multiple

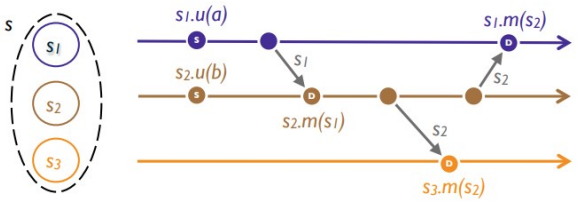


Figure 2.3: State-based CRDT replication. Reproduced from [21]

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. Although, certain CRDTs may rely on causal consistency to operate correclty. 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 without the system having to wait on replies from remote servers.

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]. As the state of a CvRDT grows in size, the entire state must be sent, which can result in high bandwidth usage. However, since CvRDTs send their full state, missed updates do not break the system because the next state transmission includes all previously missed

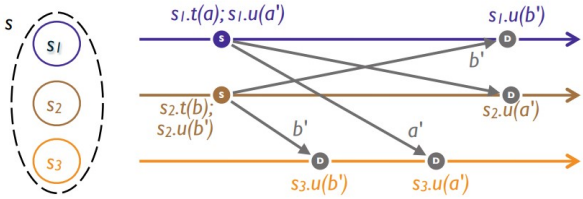


Figure 2.4: Operation-based CRDT replication. Reproduced from [21]

updates. As a result, there are fewer requirements on the replication layer. Also, CvRDTs can make use of gossip protocols (as shown in Figure 2.3) which helps reduce 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 operations to other replicas. Each update to the data 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 to the state[21]. The operations have to be commutative, but not idempotent. As CmRDTs only send operations, each replica must receive all operations otherwise the state will be incorrect. Therefore, there are more requirements on the replication layer. Figure 2.4 shows an example of this replication.

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. Additionally, the state of the object is composed of a partially ordered log of operations. It is possible to reduce the state size by implementing a method that strips causality information from operations once they become *"causally stable"*[4].

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], which can be complex. However, for the purpose of this paper, a rudimentary reliable causal broadcast algorithm will be implemented. 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* (or *atomic*) 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 highly available 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[10]
- *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[10]
  - 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

[10]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)

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. It must be noted that Najafzadeh et al. [18] has shown that any coordination-free solution will have anomalies, such as having to ignore certain operations. The following section will discuss these algorithms and how they work:

- **Kleppmann et al. [11]** proposes an algorithm based off previous work by Kleppmann, called OpSets [13]. The general approach is to have a total order for all operations, which can be achieved using lamport timestamps. Along with *undo* and *redo* operations. Using these, the algorithm will apply each operation in the correct order, undoing and redoing operations as necessary. A *log* is required to store all previous operations so that the algorithm can undo and redo operations. There is only one correct way for the operations to be applied and therefore one final state. An operation is ignored if it would cause an invalid tree state (a cycle), or if the node or new parent does not exist. It is possible to trim the log to reduce the amount of memory required using a causally stable threshold, however this will not be discussed here.
  - This algorithm has the overhead of having to undo and redo operations, however, it is easily extensible to add other conflicts (such as file name conflicts).
  - Operations can be applied immediately, as if they are dependent on any other operation, they will eventually be applied in the correct order.
  - This algorithm may produce unwanted behaviour, as it uses lamport timestamps to order operations. This means that if two operations are performed concurrently, the order they are applied is arbitrary (but deterministic).

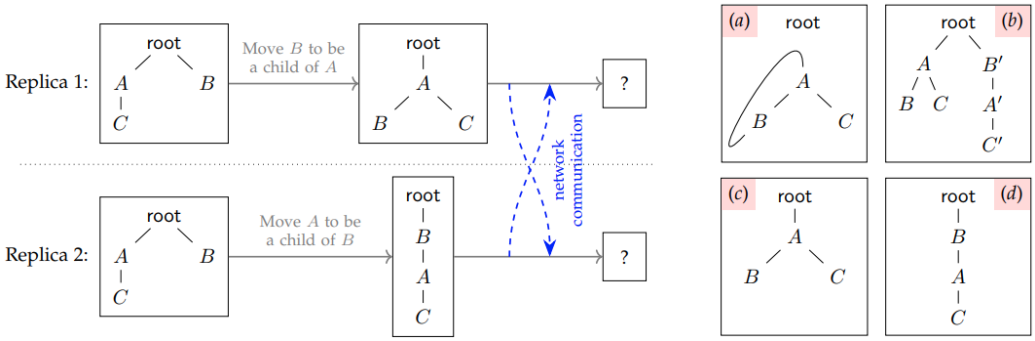


Figure 2.5: Concurrent move operations causing a cycle. Kleppmann's algorithm prefers the move from the replica with the lower ID and ignores the other operation (producing c in this case). Reproduced from [11]

- **Nair et al. [17]** proposes *Maram*, a "light-weight" tree CRDT algorithm. It only applies conflict resolution for concurrent move operations. The paper defines two types of move operations, *up-moves* and *down-moves*. An *up-move* is a move operation that moves a node towards the root, or to the same distance from the root. A *down-move* is a move operation that moves a node away from the root. Each move operation also has a priority. The idea is, that concurrent up-moves are safe (as they do not cause a cycle), and therefore do not need conflict resolution. Whereas, two concurrent down-moves, or an up-move and a down-move require conflict resolution if both operations move a node into the critical descendants (as shown in Figure 2.6) of the other move operation. In the case of a concurrent up-move and down-move, the up-move wins. In the case of concurrent down-moves, the move with the highest priority wins. The priority is a unique number that is assigned to each move operation.
  - This algorithm has the overhead of having to resolve conflicts for concurrent move operations, but not for any other operations.
  - It requires a priority to be assigned to each move operation, which may be difficult to implement in some systems.
  - It has more overhead for timestamps, as the timestamps must be able to detect concurrent operations.
  - It is harder to extend to other types of conflicts, such as file name

conflicts, as its conflict resolution policy is specific to move operations that cause cycles.

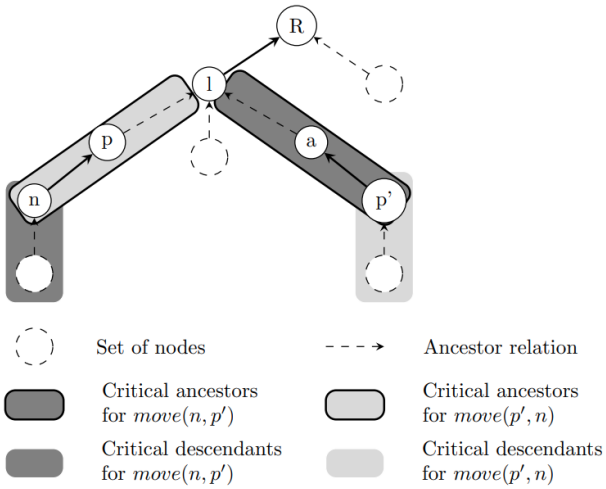


Figure 2.6: Critical ancestors and critical descendants for concurrent move operations, *move(node, newparent)*. *l* is the closest common ancestor. Reproduced from [17].

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

Table 2.1 shows a comparison between the two algorithms:

Table 2.1: 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*

[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.

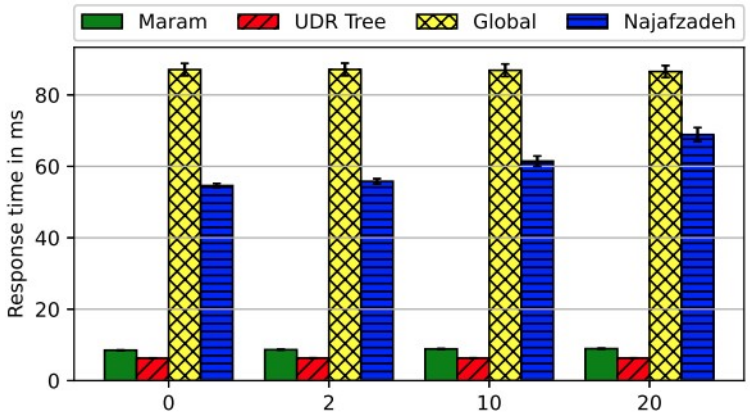


Figure 2.7: Response time for different conflict rates (0-20%). Reproduced from [17]

Figure 2.7 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 2.7 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.

Chapter 3

Planning

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, Maram[11] and Kleppmann's algorithm[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 a fixed number of 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 Language and Tools

3.3.1 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.

3.3.2 GitHub

The source code for this project is stored in a GitHub repository. This allows for the code to be easily shared and modified by other developers. It also allows for the code to be easily version controlled, and for the history of the project to be easily tracked. This report is also stored in the GitHub repository.

3.3.3 Diagrams

LucidChart[1] is an online tool which can be used to create different types of diagrams. This paper uses LucidChart to create UML class diagrams, which are used to visualise the different components of the system. Eraser[2] is another online tool which can be used to create diagrams from code. This paper uses Eraser to create sequence diagrams, which will are used to visualise the flow of messages within the system.

3.4 Testing

Go has a built-in testing framework, which is used to write unit tests. Unit tests can be used to simulate concurrent operations on the CRDTs, and therefore test the correctness of the CRDTs. Integration tests will be used to test the correctness of the system as a whole and ensure that the different layers work together correctly.

3.5 Code Structure

The code is written in accordance with the Go style guide[3]. This ensures that the code is easy to read and understand. Go itself is opinionated about

[1] `https://www.lucidchart.com/`
[2] `https://www.eraser.io`
[3] `https://golang.org/doc/effective_go`

how code should be structured, and comes with a built-in tool to automatically format code. This makes code written in Go consistent and easy to read.

The code will be split into different packages, with each package containing a single component of the system. The packages will be as follows:

- **TreeCRDT** This package will contain the implementations of the CRDTs. This package will be the main focus of this paper, and will contain the implementations of the two CRDTs described in Section 2.5.5. However, usually this package should not be used directly, as it does not provide an interface for application code to interact with the CRDTs. Instead, the **Interface** package should be used.
- **TreeInterface** This package will contain the interface for application code to interact with the CRDTs, as well as the connection between the CRDTs and the Network layer.
- **Connection** This package will contain a generic interface for the Interface layer to interact with the Network layer. It will contain the classes that will handle reliable communication between replicas.
- **Clocks** This package will contain the implementations of the clocks and timestamps.

Chapter 4

Design

4.1 Package Structure

The system will be split into different packages as described in Section 3.5. The `TreeCRDT` package is the CRDT layer, the `TreeInterface` is the Interface layer, and the `Connection` package is the Network layer. The dependencies between these packages are shown in Figure 4.1.

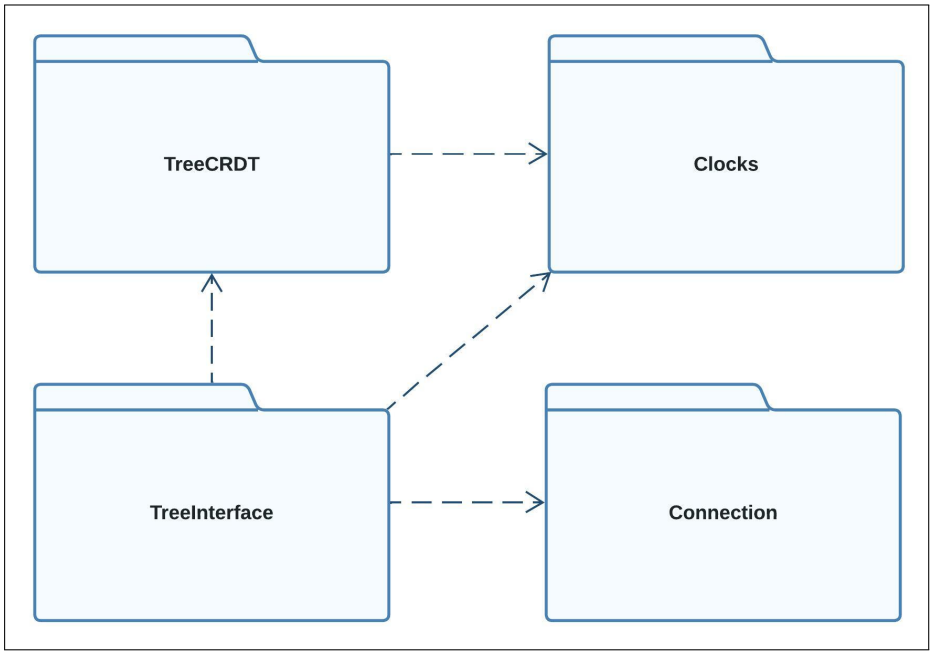


Figure 4.1: Package structure for the project, with dependencies shown.

The packages will be developed in a way which implements the main focus of this paper first, the CRDTs. Therefore, they will be developed in the following order: `Clocks`, `TreeCRDT`, `Connection` and finally `TreeInterface`. This will allow for the main focus of this paper to be implemented first, and then the rest of the system can be built around it.

4.2 System Structure

4.2.1 Clocks

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 4.2, 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 4.2. The

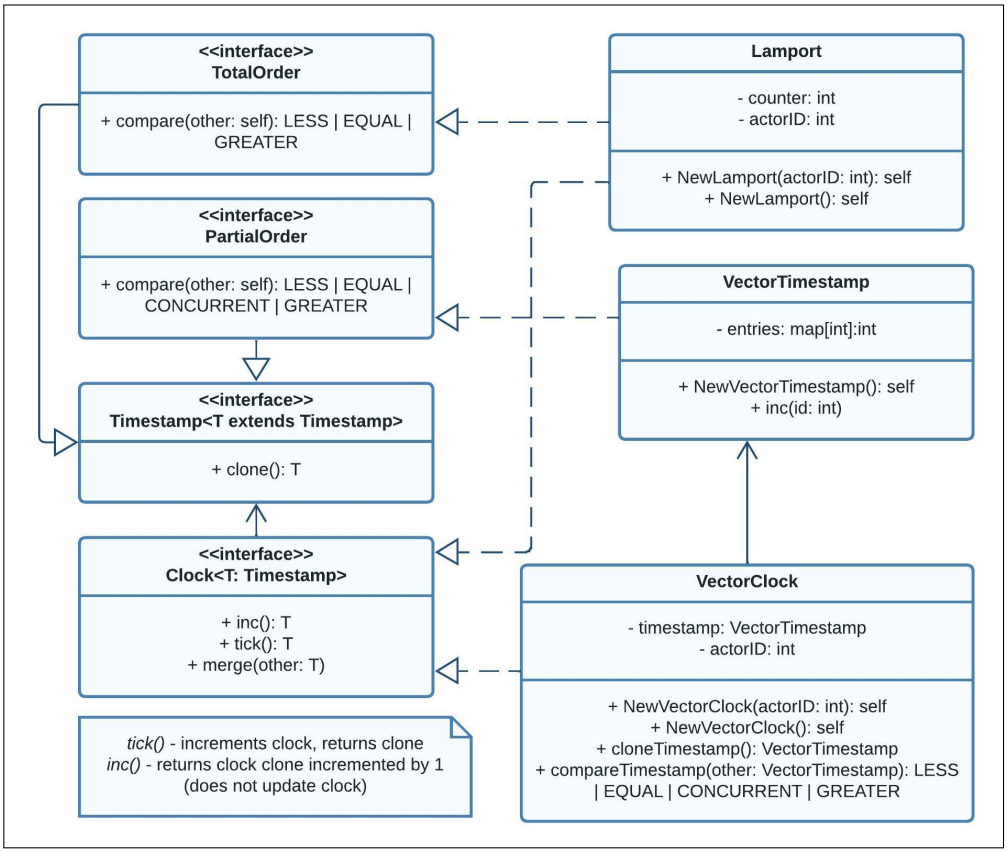


Figure 4.2: Class diagram for the clocks and timestamps. Including Lamport and Vector clocks.

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.

4.2.2 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 implementation is not known at this point, and will be discussed in Section 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 4.3 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 them to the local CRDT or buffering them, depending on the consistency model. They will also be responsible for providing a generic interface for application code to interact with the CRDT, and translating the application code into operations for the specific CRDT.

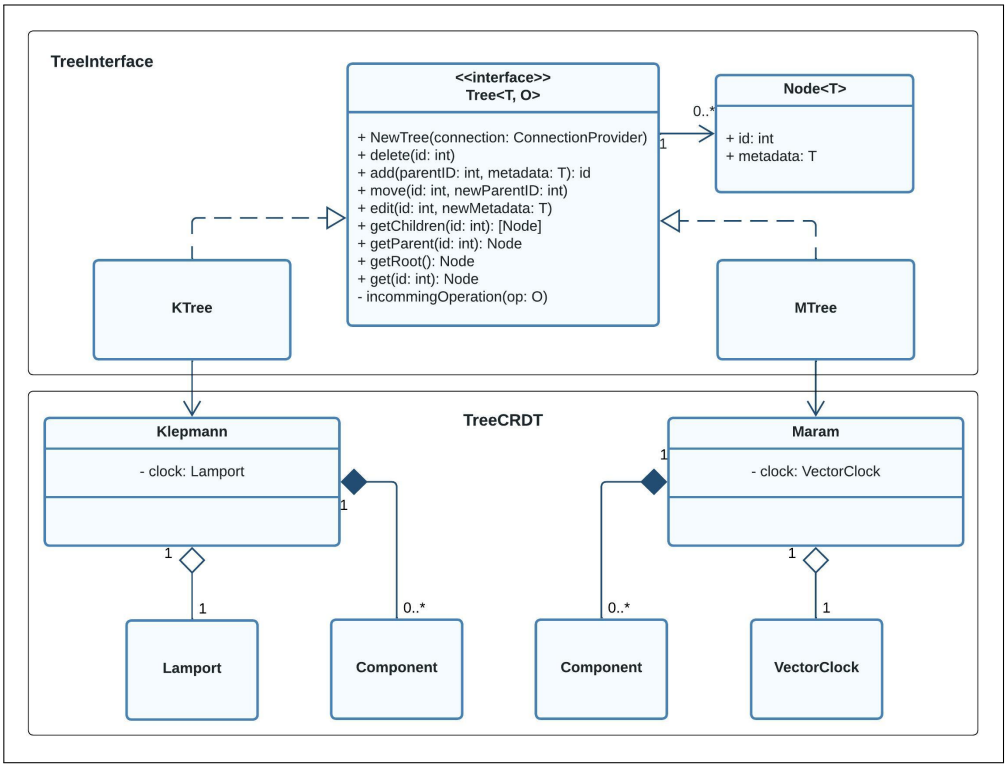


Figure 4.3: Class diagram for the CRDT and Interface layer. The `Tree` interface provides an interface for application code. Classes `MTree` and `KTree` provide implementations for the specific CRDTs.

4.2.3 Network

The network layer takes inspiration from Yjs[1], 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 nodes, as they would require access to the current state of the CRDT.

[1] `https://docs.yjs.dev/`

The network layer assumes that each node will be started with the same number of peers, and the total set of peers will not change. This is a reasonable assumption for a static system, however it is not suitable for a dynamic system.

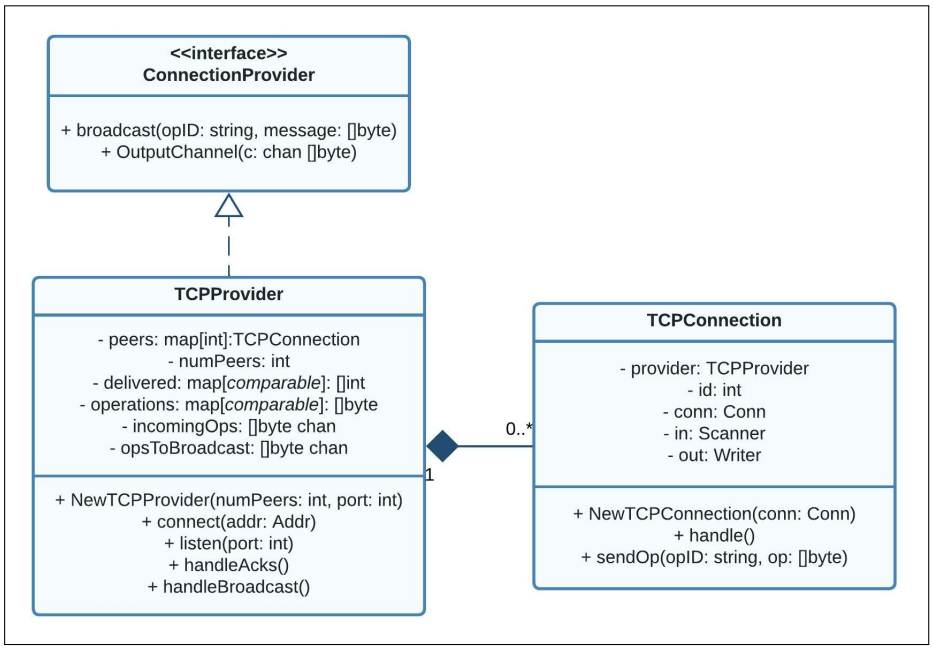


Figure 4.4: Class diagram for the Network layer. With the interface `ConnectionProvider` and implementing class `TCPProvider`.

The interface `ConnectionProvider` is shown in Figure 4.4. 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 providing the reliable broadcast functionality, as well as listening for incoming connections.

The `TCPConnection` class is responsible for maintaining a connection with a single node, and receiving messages from that node. When a node joins after being offline, `TCPProvider` is responsible for sending them all the messages which they missed. The `TCPConnection` will not apply incoming operations, instead they will be passed to the TreeInterface for processing. This is useful as it reduces the need for locks, which would be required if the `TCPConnection` was responsible for applying operations.

4.3 Communication Sequences

The CRDT layer and Clocks in the application should not be affected by concurrency because they are designed to be used in a single thread. Therefore, it is the responsibility of the developer to ensure that both are used exclusively within a single thread. The Interface and Network layer however will run in multiple threads, and therefore concurrency and message passing must be considered.

4.3.1 Network Layer

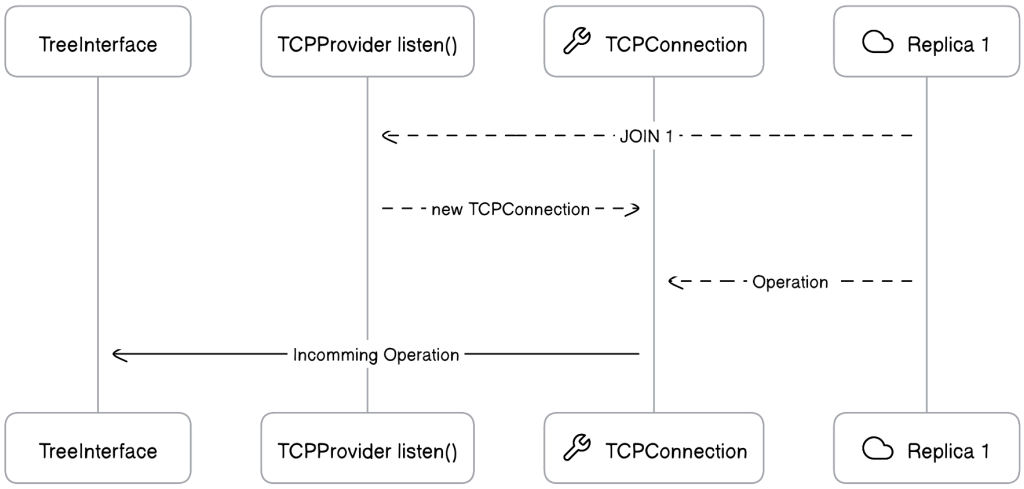


Figure 4.5: Sequence diagram for the network layer, showing a replica connecting and sending an operation.

The network layer will comprise multiple concurrent processes. One process will be responsible for listening for incoming connections, and creating a new `TCPConnection` for each connection. The `TCPConnection` will only be responsible for receiving messages from the peer but not sending them. If the message is an operation, it is sent to the TreeInterface buffer to be applied to the Tree. Otherwise, if it is an acknowledgement, it is sent to `TCPProvider handleAcks()` for processing. As shown by Figure 4.5 and 4.6.

The `TCPProvider handleBroadcast()` will be responsible for sending messages to other peers, which will happen in a separate process as shown by Figure 4.6. The process will receive operations to send from the Interface

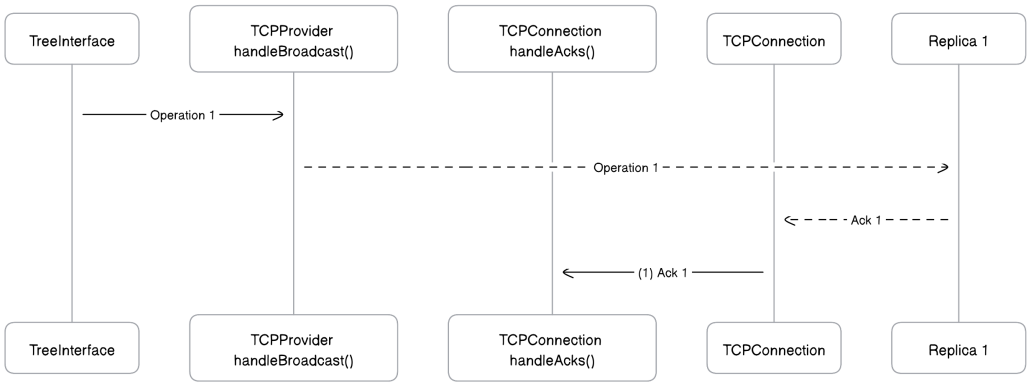


Figure 4.6: Sequence diagram for the network layer, showing an operation being sent and an acknowledgement being received.

layer. Whenever an operation is broadcasted, it is added to a map that associates the operation's ID with a list of node IDs. When a node sends an acknowledgement for an operation, that node ID is added to the list of node IDs for that operation. When the list of node IDs is the same size as the number of nodes, the operation is removed from the map. This is how the system ensures each node receives each operation.

4.3.2 Interface Layer

As shown by Figure 4.7, the interface layer will buffer operations from the network layer, and only apply them when they are ready. This is important for the causal consistency model, as it ensures that operations are applied in the correct order. For local operations, they are translated into operations for the specific CRDT, and then applied to the CRDT. It is likely that the CRDT will be protected by a lock, as it is not thread safe.

Broadcasting operations is not shown in Figure 4.7, but it would happen after the operation is applied to the CRDT. A message would be sent to the network layer, which would then broadcast the operation to all other nodes.

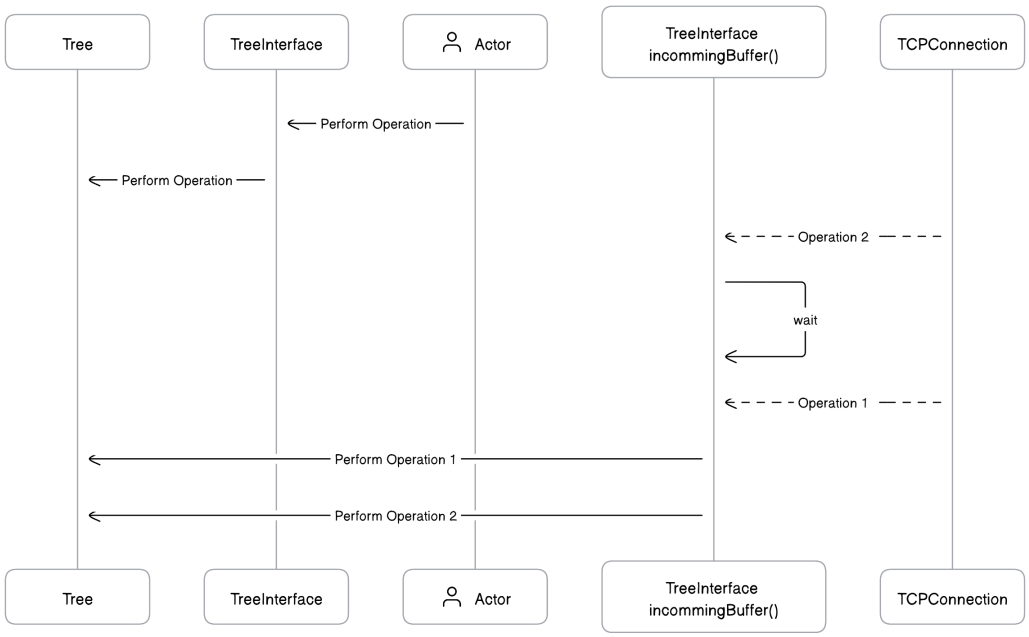


Figure 4.7: Sequence diagram for the interface layer, showing operations being applied to the Tree. Broadcasting is not shown.

4.4 Justification of Design

The design is split into four packages, which helps to keep the code modular and easy to understand. Each package has a single responsibility, which helps to keep the code easy to maintain. This design approach of having a single responsibility for each package and having interdependencies between packages is a common design pattern in software engineering.

The system takes advantage of concurrency to allow for increased performance and scalability. Message passing will be used to reduce the need for locks, which can otherwise cause performance issues. The use of multiple threads also allows for the system to be more responsive, as it can handle multiple requests at once. Each thread will be responsible for a single task, which will help to keep the code easy to understand.

Chapter 5

Implementation

This section will explain the difficulties in implementing the system, and how they were overcome. As well as the differences between the design and the implementation. The diagrams in this chapter were created using *goplantuml*, and some manual editing.

The main struggle in the implementation was using Go to implement the system. Go does not have explicit implementations of interfaces, and uses composition instead of inheritance. There are also no classes in Go, instead there are types and structs which can have functions. This caused some initial difficulties in the implementation.

The implementation uses the following open source packages:

- **uuid** - Used to generate unique IDs for nodes, operations and peers.
- **protobuf** - Used to serialise messages between replicas.
- **msgpack** - Used to serialise operations. *protobuf* cannot be used as it requires the types to be defined in a `.proto` file. This is not possible for metadata field in the tree nodes as they can be *any* type.

5.1 Clocks

The clocks package was the first to be implemented. Initially, it was a struggle to understand how to use Go's generics and to understand Go's implicit in-

terface implementation, however this was overcome. It is also idiomatic in Go for a function's return type to be simple and generic, with the documentation explaining the return value and how to use it. Therefore, the implementation contains comments explaining the return values of functions. As shown by Figure 5.1, there are a few main differences between the design:

- `uuid.UUID`s are used in place of `int` for the node ID. This allows for better unique IDs.
- The compare methods return `int` instead of the enums defined in the design. This is more idiomatic in Go, as Go does not have enums.
- The `Clock` interface has two extra methods, `ActorID` and `Timestamp`. It was realised that these methods were needed for the CRDTs, and so they were added to the interface. The `Timestamp` method returns a clone of the timestamp.

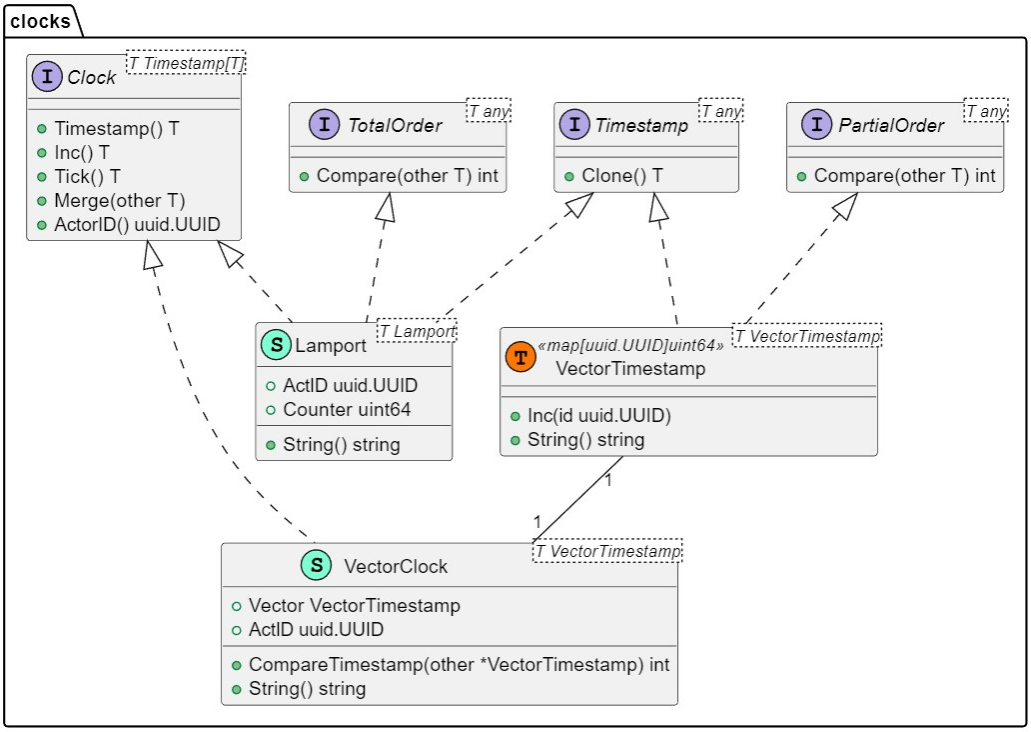


Figure 5.1: Class diagram for the clocks package.

5.2 Tree CRDTs

The CRDT algorithms were the largest part of the implementation as they contain the most parts. They were constructed by following the algorithms within the respective papers. The CRDT implementations are not thread-safe.

The `Tree` and `TreeNode` structs are general structs and can be reused for both algorithms. They are in the `treecrdt` package, and the different algorithms are in subpackages of the `treecrdt` package. The two structs do the following:

- `TreeNode`: This contains the node's parent id and metadata.
- `Tree`: This contains the representation of the tree. It has a map of node ids to `TreeNode`s. As well as a map of node ids to their children, for easy access to the children of a node.

The `Tree` struct uses maps for nodes and children to enable fast retrieval and editing. The children were originally stored in a slice, but this was changed to a set to allow O(1) for querying and editing a particular child.

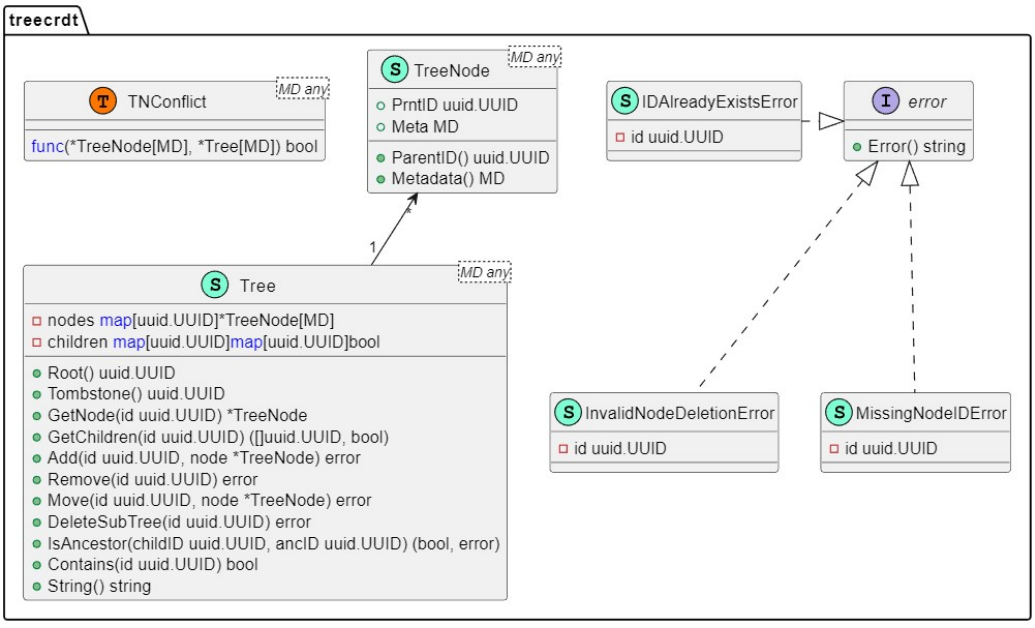


Figure 5.2: Class diagram for the *treecrdt* package.

5.2.1 Kleppmann's Algorithm

The first algorithm implemented was Kleppmann's algorithm, primarily due to its simplicity and ease of implementation. It is based on OpSets, that were introduced in 2018[13], which that contributes to its simplicity. More information on Kleppmann's algorithm can be found in Section 2.5.5. Overall, the algorithm was relatively simple to implement and was composed of the following parts:

- `OpMove`: This contains a move operation, which is the child id, parent id, metadata and a timestamp.
- `LogOpMove`: This contains a move operation, and its previous parent id and metadata. This is used for the log.
- `State`: This contains the state of the CRDT. It contains a `Tree` and a log of operations. The state contains the main algorithm for applying operations.
- `TreeReplica`: This represents the state of the CRDT for a single replica. It contains a `State` and a `Clock`. This struct implements the *prepare* and *effect* functions from the literature on CmRDTs.

The log of operations being implemented as a linked list. This was chosen over an array or slice as it is faster for altering the middle of the list and frequent appending. A normal slice would have to copy all the elements to a new array when it is full.

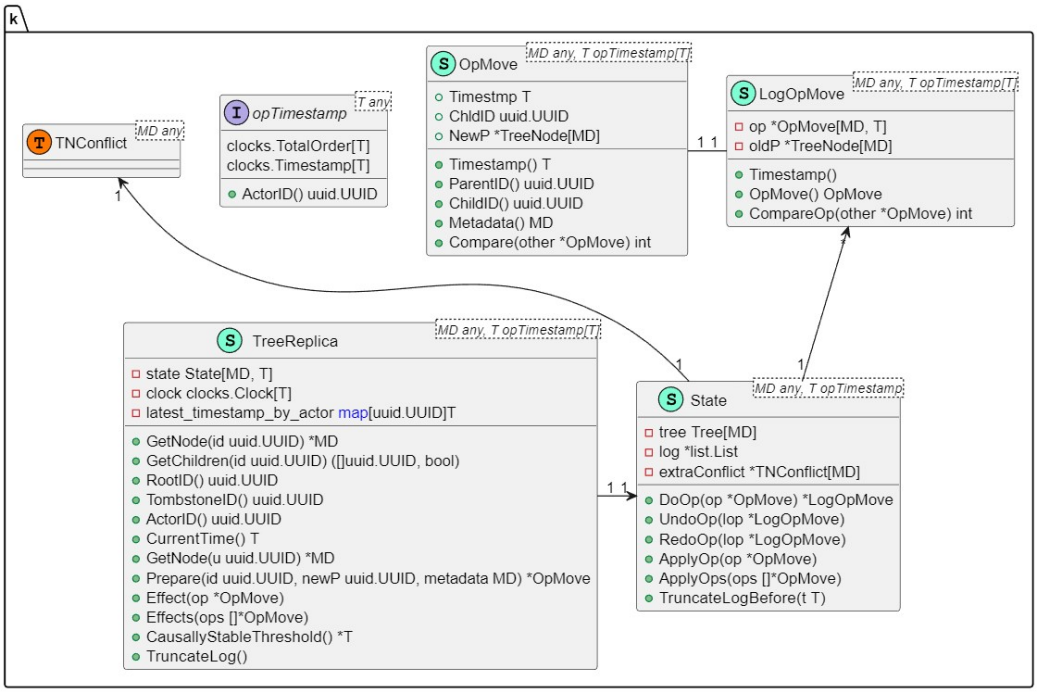


Figure 5.3: Class diagram for the Kleppmann's algorithm.

5.2.2 Maram

The Maram algorithm was more difficult to implement, and the final implementation slightly alters from the literature. The paper for the algorithm [17] was not as clear as the paper for Kleppmann's algorithm [11], and the algorithm was more complex. The paper defines how to resolve conflicts between two operations, but does not define how to resolve conflicts between multiple operations. Also, later concurrent move operations may affect the conflict resolution of earlier concurrent move operations. Therefore, the algorithm was implemented with the following change:

- Concurrent move operations implement the algorithm from Kleppmann's algorithm. They have a total order, and are undone and re-applied in the that order. This ensures that concurrent move operations have a deterministic outcome.

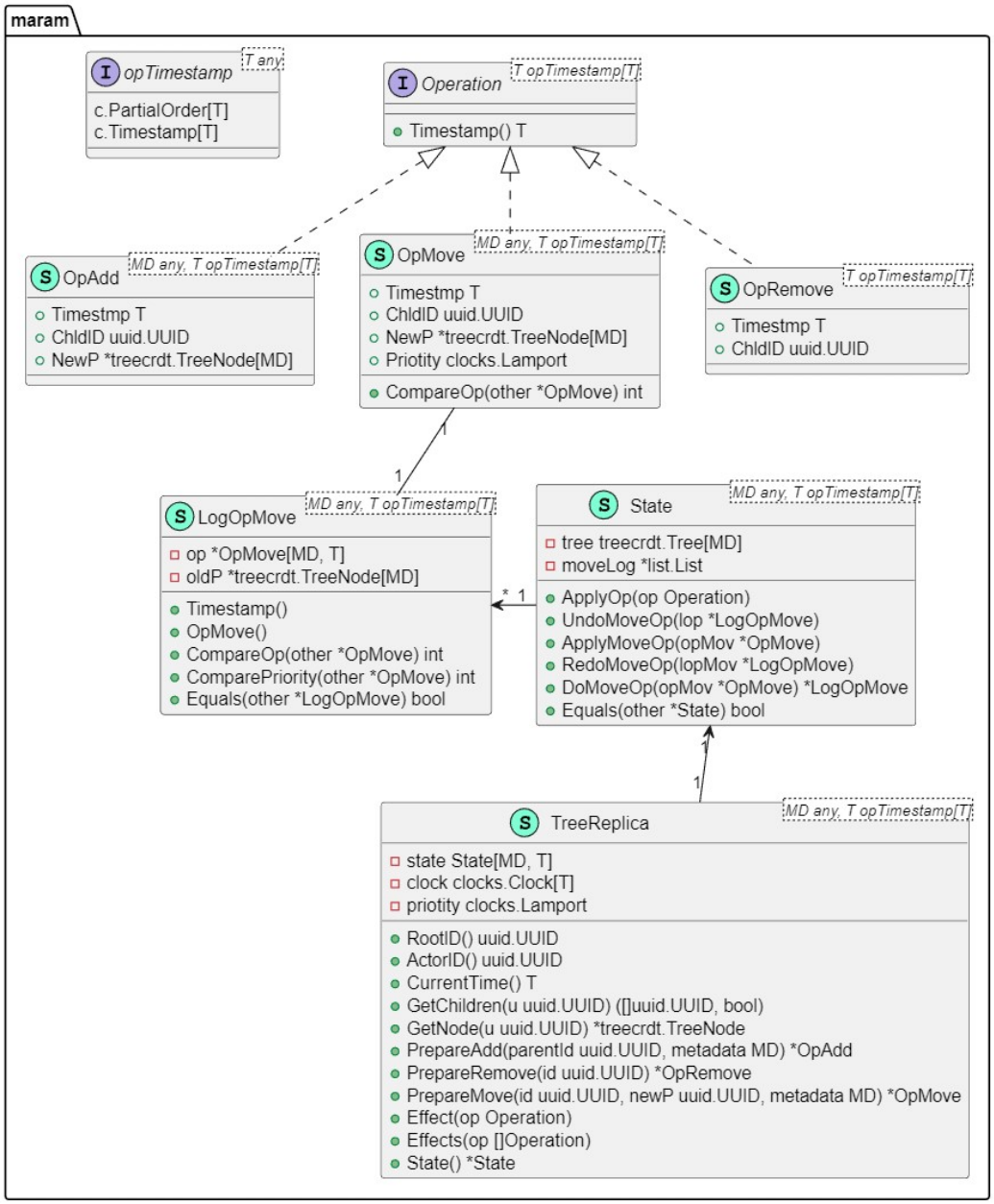


Figure 5.4: Class diagram for Maram.

In comparison to Kleppmann's algorithm, Maram only logs move operations. This is because of the following reasons:

- *Add* operations are not logged as they have a precondition that requires the parent node to be in the tree. Together with the causal delivery layer, this implies that the parent node will be added first, followed by the addition of its child node. It also relies on nodes being added having unique ids, which is possible with UUIDs.
- *Remove* operations are also not logged as they have a precondition that the node being removed is in the tree. In the presence of the causal delivery layer, this indicates that the removal of a node will occur after its addition to the tree. This also relies on each node having a unique id. This has the implication that for concurrent remove and move operations on the same node, the rmeove will always win. However, the children of a removed node can still be concurrently moved back into the tree.

Move operations will always need to be logged, as they can break the tree invariant. Therefore, it is necessary to log them so that concurrent move operations can be applied in a deterministic order and not break the tree invariant.

5.3 Network

The network package was the most difficult to implement, as it is a peer-to-peer network with a reliable broadcast protocol. It has to deal with concurrency, and keep track of which nodes have received which operation. This requires that the nodes exchange their ids and share their peers.

As shown by Figure 5.5, the network package only consists of two structs. It works as follows:

- The `TCPProvider` runs `Listen()` in a seperate goroutine to listen for incoming connections. When a new peer connects a `TCPConnection` is spawned in a new goroutine.
- The `TCPConnection` sends the other peer the current node's ID and a list of peers. Once an ID is received from the peer, it is added to the *peers* map in `TCPProvider`. (Before then the ID is unknown, and so the peer cannot be added to the map)
- The `TCPConnection` will also send all the operations which the newly

connected peer has not seen to it.

- The `TCPConnection` then listens for incoming messages. When a message is received, its type is determined and the appropriate code is run. There are 4 message types:
  - The first is the *node_id* message, which is sent at the start of a connection.
  - The second is the *peer_list* message, which is sent after the node id message.
  - The third is the *operation* message, which sends the operation to the `incommingOps` channel for processing, and an ack is sent back to the peer.
  - The fourth is the *op_ack* message, which adds the node to the list of nodes that have received the operation.
- The `TCPProvider` also runs the `handleBroadcast()` function in a goroutine. This listens on the `opsToBroadcast` channel for operations to broadcast. When an operation is received, it is sent to all peers.

Protocol Buffers[1] are used to serialise the messages. They were chosen as they are smaller, faster and more efficient than other methods. Also, using their `oneof` feature, the message type can be easily determined. More information on the protocol buffers used can be found in Appendix C.1.

[1] `https://protobuf.dev/`

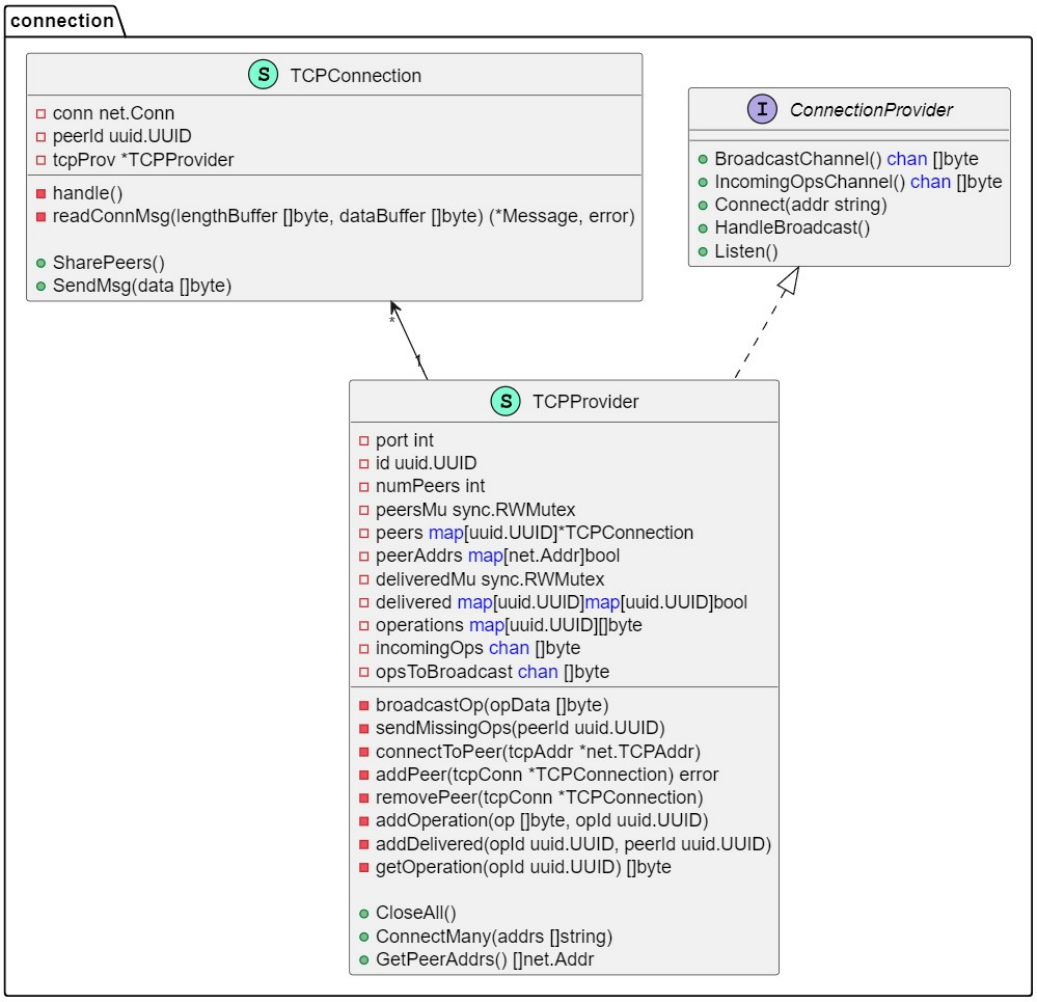


Figure 5.5: Class diagram for the connection package.

5.4 Tree Interface

The tree interface provides a simple interface for the user to interact with the tree with. It is a wrapper around the CRDT, and translates generic tree operations into CRDT specific operations. It also provides functions to query the tree. The interface also applies operations from the network to the CRDT. Because the CRDT is not thread-safe, the interface ensures that only one operation is applied at a time.

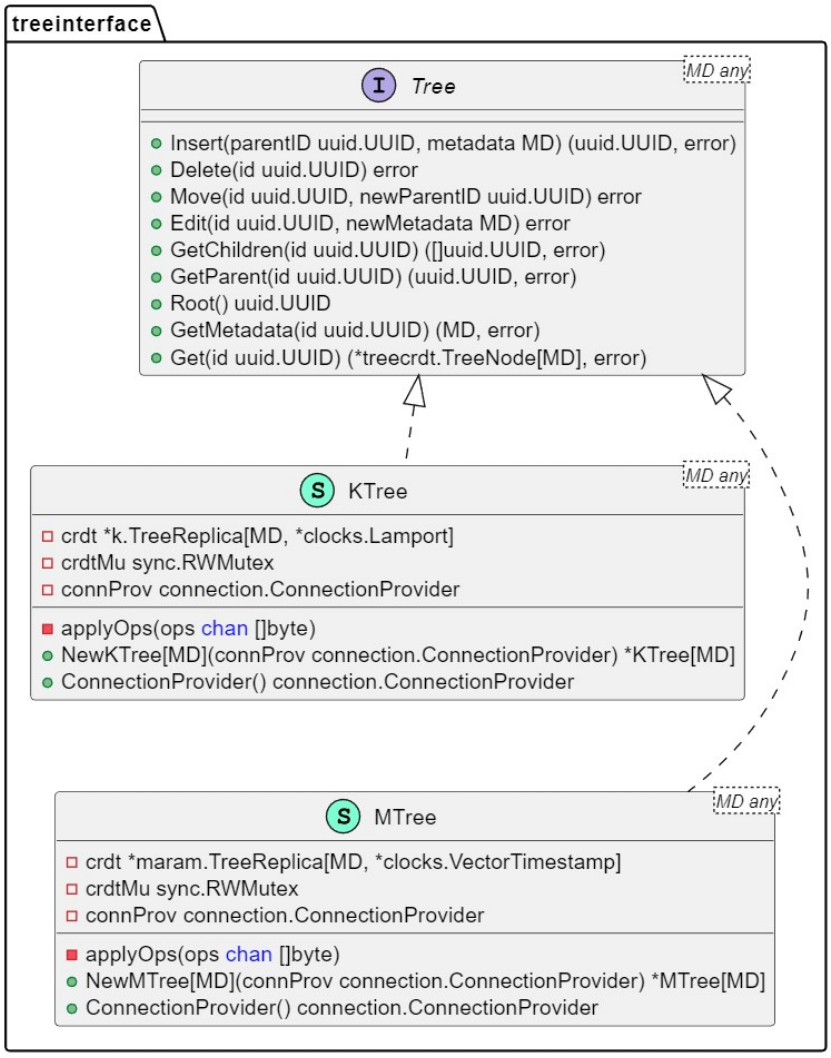


Figure 5.6: Class diagram for the tree interface.

The implementation of the interface was relatively simple, as it is a wrapper around the CRDT. As shown by Figure 5.6, a RWMutex is used to ensure that only one operation is applied to the CRDT at a time. This Mutex is locked when applying operations and when querying the tree.

Chapter 6

Testing

Chapter 7

Evaluation

Bibliography

[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.

Appendix A

Interim Project Management

A.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.

A.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

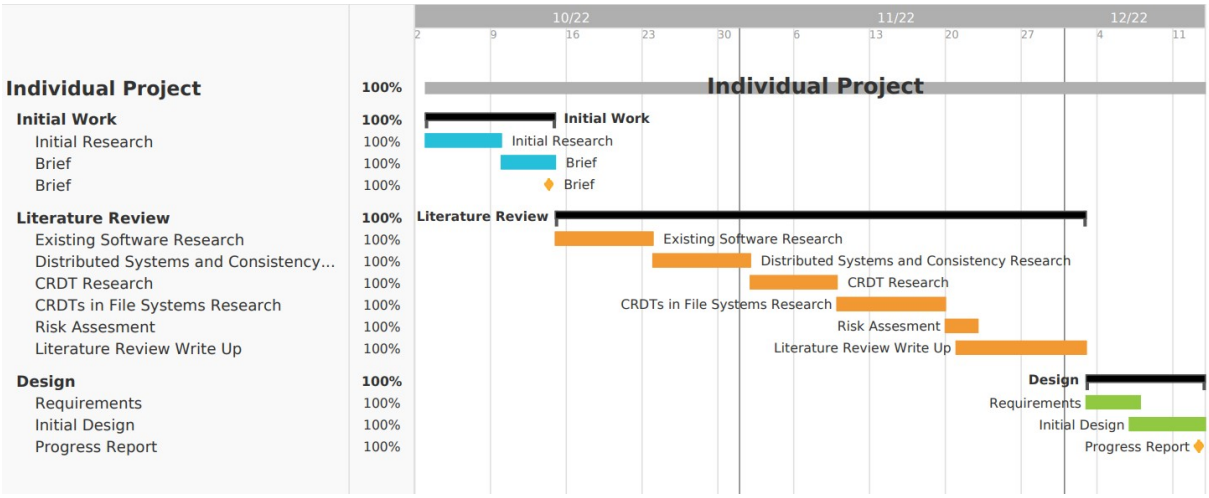
to measure the throughput and response time of the algorithm(s), and how adding more replicas affects this. The evaluation then needs to be written and will contain the outcome of the project.

A.3 Estimate of Support Required

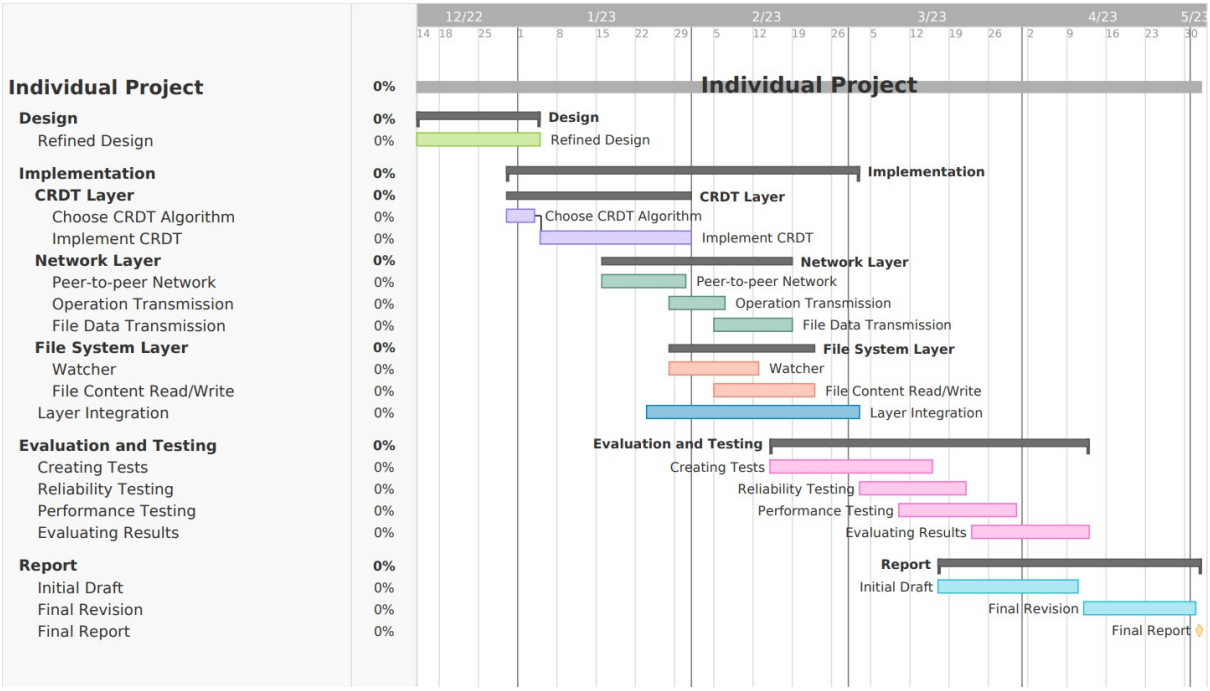
The performance testing may require the use of rented servers, to test latency.

A.4 Gantt Chart

A.4.1 Completed Work



A.4.2 Remaining Work



A.5 Risk Assessment

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

Table A.1: 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.

Appendix B

Original Brief

B.1 Original Title

Using CRDTs to Create a File Synchronization System

B.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).

B.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.

B.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.

B.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.

Appendix C

Implementation

This chapter details the interesting parts of the implementation. It contains some code snippets, however for a more detailed look at the code, the source code and associated comments should be consulted. The source code can be found at `github.com/FelixWhitefield/Tree-CRDTs-With-Move`. The repository will become public after the project has been marked.

C.1 Protocol Buffers

The 'message.proto' file is shown below. The message `Message` is the only message which is sent between peers. The `oneof` is used to allow for different types of messages to be sent. The *repeated* keyword in the `PeerAddresses` message is used to define a list of strings. The messages are relatively simple, and using protocol buffers the messages can be understood by any language which has protocol buffer support

```
syntax = "proto3";

package connection;
option go_package="github.com/FelixWhitefield/Tree-CRDTs-With-Move/connection";

message PeerAddresses {
    repeated string peerAddrs = 1;
}

message PeerID {
    bytes id = 1;
}

message OperationMsg {
    bytes id = 1;
    bytes op = 2; // Will be a msgpack encoded operation
}

message OperationAck {
    bytes id = 1;
    bool ack = 2;
}

message Message {
    oneof message {
        PeerID peerID = 1;
        PeerAddresses peerAddresses = 2;
        OperationMsg operation = 3;
        OperationAck operationAck = 4;
    }
}
```

Appendix D

Archive Contents

The archive has the following contents and directory structure:

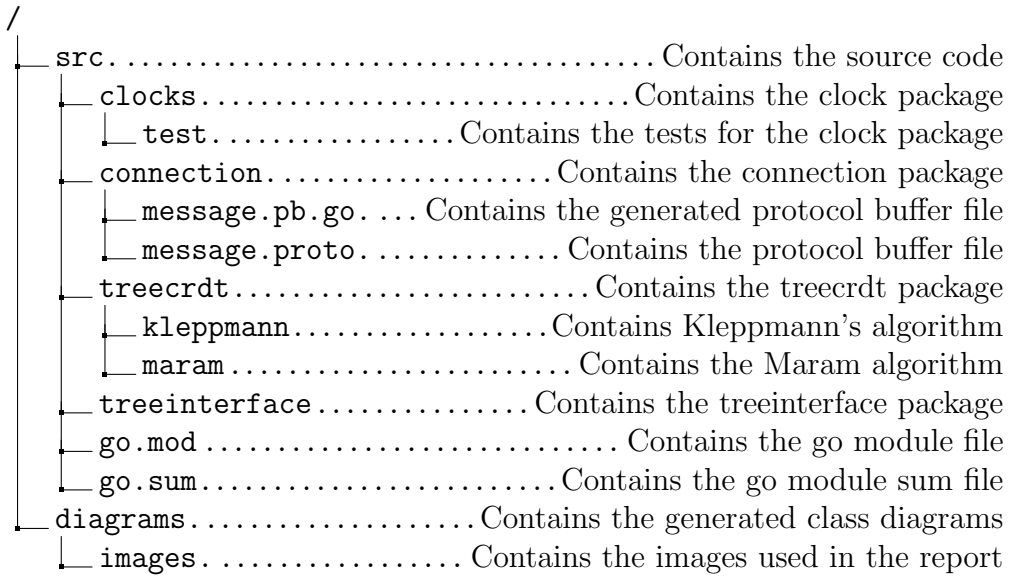


Figure D.1: Directory structure of the archive