NTNU

Norwegian University of Science and Technology

Department of Computer Science

TDT4900 - Computer Science, Master's Thesis

A CRDT-based file synchronization system

Erik Liu

Supervisor: Svein Erik Bratsberg

June 14, 2021

Abstract

This paper designs and implements a Conflict-free Replicated Data Type file synchronization system with a custom made CRDT used as an index to track file updates. The system designed to be modular such that each module can be swapped to accommodate different needs. Conflict-free Replicated Data Types has the property of strong eventual consistency which when implemented correctly ensures that any concurrent operations across an arbitrary network are able to converge to be same state, regardless of the operation order. This is used to implement system to work in a peer-to-peer network. The paper carefully breaks down the different aspects of a file synchronization system, implements a proof-of-concept guided by the designed architecture, and verifies the functional requirements by a series of tests.

Contents

1 Introduction

1.1 Purpose

Popular cloud storage systems like Google Drive, One Drive, and Dropbox utilize a centralized server architecture to provide the necessary functionalities of file sharing and collaboration. When it comes to everyday users, the thought of "where" the user's files are stored rarely crosses their mind. Only the fact that the files are easily accessed matters. In these aforementioned systems, the files are primary stored on a server, possibly far away from the users. Consequently, the files need to relay through the server, adding unnecessary transmission latency, and forcing any conflicts from concurrent user operation to be resolved at the server. This results in the server being a potential bottleneck, and a single point of failure.

This paper will take a look at how Conflict-free Replicated Data Types work and how they simplify the conflict resolving process in a P2P network. CRDTs are an integral part of resolving file update conflicts, such as concurrent file modification, that can occur in a file synchronization system. The goal of this paper is to both design and implement a Conflict-free Replicated Data Type index that can be used to keep track of files and folder, and then, integrate it into a peer-to-peer file synchronization system. Additional care will be given in the designing phase to ensure a modular system. The storage scheme can be changed from storing files on the local file system to storing files in a database. The communication protocol should be swappable, such that the system can easily upgrade to a better communication protocol. This paper aims to develop a minimal viable product of the file synchronization system, and the correctness of the system, i.e. the different ways to fix conflicts, will be verified by tests.

A peer-to-peer file-sharing system can be easily used to distribute of illegal content, this paper is not a encouragement of this kind of behavior.

1.2 Research Goals

G1: Design a CRDT-index to keep track of files.

G2: Make a proof-of-concept CRDT-based file synchronization system.

G3: Analyze the set of resolve policies for each conflict.

1.3 Scope

Due to the time of effort for one person to design, develop, and test the new system. Some limitations need to be set to avoid spending unnecessary time in non-core features. This involves limiting the communication protocol to only http with no added encryption schemes.

1.4 Report structure

This report is the continuation of the previous report "Designing a CRDT-based file-syncing system" from "TDT4501 - Specialization Project" Fall 2020. Thus the relevant background theory and previous design aspects from the previous report will be reiterated and reinforced in this report. The core of the report is split into 7 chapters.

- **Chapter 1** introduces the general problem of the most popular file-sharing/syncing services, provides a potential solution, and presents the goals which the paper aims for.
- **Chapter 2** presents the current state of file-sharing/syncing services, the common network architectures in such services, and all the relevant background theory and concepts which will form the basis of the new system's architecture.
- **Chapter 3** describes the overall architecture and the individual components of the system.
- **Chapter 4** takes a deeper look into the implementation details of the system prototype.
- **Chapter 5** presents the testing environment which shall verify the correctness of the system, and discusses the trade-offs between the system prototype and existing file-sharing/syncing services.
- **Chapter 6** summarizes the paper, and discusses the goals of the paper.
- **Chapter 7** provides a roadmap for the future development of the system.

2 Background

This chapter is split into three parts. The first introduces the concept of file sharing and file synchronization and the common network architecture used. The second part presents the theories which will forge a new file synchronization system. The last part takes a look at, and discusses the related within the field.

2.1 File sharing

As the amount of digital data keep increasing, the need to easily access data becomes a major part of our life. There are many services that accommodate this need with some popular choices being: BitTorrent[1], Dropbox[2], and Google Drive[3]. The later two being cloud-based services. Cloud-based file-sharing systems use a centralized server to process and store the data. For most users, this does not pose a problem as they do not care where their data is stored. However, a file-sharing system which prioritized the local storage on each user's daily devices(phones, laptops, and external storage systems), could better utilize the empty storage space these devices. In addition, leaving the responsibility for keeping the data safe to the users themselves, instead of trusting the businesses could be valuable. In cloud-based, when a user shares a file to another user, the file has to relay through the data-center, which may result in additional travel time before reaching the user.

P2P-based file-sharing systems are not something new or foreign. Many are currently in use. A popular P2P file-sharing system is BitTorrent. However, BitTorrent is mostly used with static files. Once a file is distributed, it is assumed that it will never be updated. While there exists a workaround to support file updates, it involves the user, or an external mechanism redistributing(removing and re-adding) the file. Which This extension has been used by file synchronizing services like Resilio[4], a proprietary system. However, working around the BitTorrent protocol might involve unnecessary overhead, and this report will try to design a similar system in functionality, but with P2P file synchronization, i.e. the ability to update files, as the foundation.

One particular advantage cloud-based file synchronizing services have over P2P-based file synchronizing services is the ability to collaborate on files, i.e. multiple users can edit the same document at the same time. Take Google Docs as an example or any of the other Google documents. By relaying all update information through a centralized server, the server knows everything can apply all updates without the worry that the files get desynchronized.

[1]https://www.bittorrent.com
[2]https://www.dropbox.com/
[3]https://www.google.com/intl/en_in/drive/
[4]https://www.resilio.com/

From hereon, the usage of "file-sharing" represents both the act of replicating the file across multiple nodes and the act of synchronizing any arbitrary changes to the said files in a deterministic and consistent manner. File collaboration will be referred to as content-based synchronization. Additionally, the usage of "files" will represent both "files" and "folders", and "file update" consist of the actions:

- Create new file
- Modify existing file
- Delete existing file
- Rename existing file
- Move existing file

2.1.1 Use-cases

To get a better grasp on what a file-sharing system should do, several use-cases of such a system are presented below. These will also be used to evaluate the system in the later chapters.

- **Team collaboration**: A group of users that know all the other users, or at least can trust that the other users will not engage in malicious behavior to danger the integrity of the shared resources. Examples of this kind of groups could be: students collaborating on a common task and require some form of shared workspace. Another characteristic of such use-cases is that everyone have access to the full set of file manipulations, i.e. create, change, delete, rename, and move files.
- **Content distribution/sharing**: A group of users where each is either a provider or consumer. The providers are the ones that have the permission to add new files and to remove or modify existing files in the network. The consumers can only access the files, but can not change these in any way. This use-case represents individual users who want to share some content, and larger organizations which distribute a large amount of content.

2.2 Network architectures

As mentioned, a file-syncing system could be cloud-based or P2P. Take mainstream file-syncing services like Google Drive or One Drive as examples. Both utilize the centralized server architecture, which puts a huge amount of responsibility on the server. On the other end, the peer-to-peer(P2P) architecture is designed to handle an arbitrary network topology.

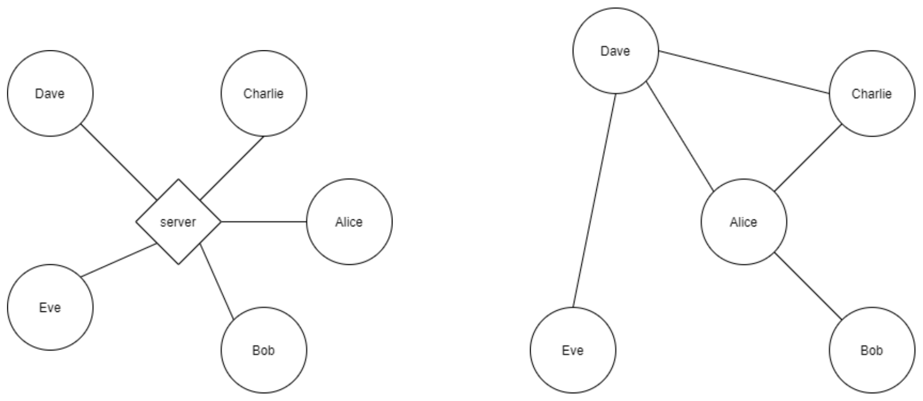


Figure 1: An centralized server architecture vs an ideal p2p network.

2.2.1 Centralized

The particular network architecture used by most popular file-syncing systems, or most application in general, is cloud-based. This involves a centralized server architecture, with two distinct types of nodes, the servers, which are known to all nodes and consist of powerful hardware, and the clients, which consist of arbitrary hardware. The types of jobs the servers and the client are responsible for are usually split in such that the servers, with its powerful hardware can perform heavy jobs relatively fast and within an expected amount of time. The clients on the other hand, mostly performs easy jobs since putting a strong hardware requirement on the client side is usually too high of an entrance-barrier to the application. In P2P networks, the term ”peer” are used to represent a node, and both will be used interchangeably.

This architecture is easy to work with, as the centralized server is a common node for processing, where all data have to pass through at some point, i.e. there are no direct communication between the clients. The server has no need to fetch additional data from other clients to perform its jobs. If there are multiple servers, then there will be some coordination between the servers. In a centralized server architecture, all the servers, by design, know each other, such that the coordination can be handled in an efficient manner, i.e. there is no need to speculate that there are unknown servers with critical data that are needed for other servers to perform correctly.

Key notes of centralized server architecture:

- Two types of nodes, the servers and the clients.
- The server is a node where all necessary data are present, or can be easily acquired, to perform any jobs.
- Lax hardware requirements for the client.

- The application owner or maintainer needs to ensure the capacity of the server at any time.
- Sending data from client to client is usually indirect, as it has to pass through the server.
- The jobs for servers and clients are straightforward in the sense that the is no need to compensate for unknown nodes.
- The uptime of the system is directly correlated to the number of servers and their capacity, the amount of request a server can handle.

2.2.2 P2P

The indirectness of the data transfer, adds unnecessary latency when sending data from client to client. This is from the fact that data are stored primary on the centralized server, not on the client's local device. A P2P architecture is a network architecture where each individual node keeps track of a subset of the network. No node knows the entire network, but only which nodes itself is connected to. Figure 2 shows a node A not knowing node D, but both are indirectly connected. Each node are both the server and the client. All the nodes are then identical and have the same responsibility.

But such network architecture poses a challenge when a node is performing an important job and requires additional data from other unknown nodes. The node do not have the complete data which is stored across the network. An example could be each node holds a counter and one node got the job of computing the sum of all the counters in the network. In this case, the node has no guarantee that the result is the correct result it was looking for.

A system built on a P2P network is able to be scaled indefinitely, since each node essentially serves as a server. With no necessary need for the owner of maintainer to be involved. But that is at the cost of each node requiring to perform extra work to ensure that the network is not partitioned. Figure 3 shows an example where node B leaves. To avoid the network to be partitioned, B tells A about C, and tells C about A, such that they can set up a connection before B leaves.

Key notes of P2P architecture:

- One type of node, everyone has the same responsibility.
- There is a need for additional coordination between the nodes when performing jobs.
- The network scales automatically with the number of nodes.
- The uptime scales with the number of active nodes, nodes that are currently connected to the network.

Benefits for a P2P architecture over a centralized server architecture:

- **Availability:** A centralized server architecture need to keep the central server online and therefore gives a single point of failure. A p2p architecture can statistically be always online, provided a sufficient user base.
- **Latency:** All data need to relay through the central server, possibly making the data take a longer path than necessary. While a p2p architecture does not guarantee a lower transmission/propagation latency than central server architecture, it does guarantee a lower *average* transmission latency.
- **Scalability:** The processing capacity of the central server needs to always accommodate a growing user base. In P2P, each node is functionally a server and therefore automatically distributes the processing load.

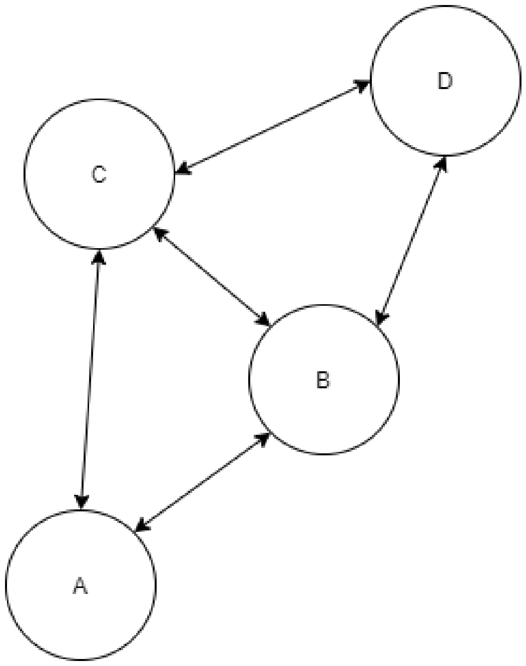


Figure 2: A can not see D.

2.3 Consistency

The term "consistency" was briefly mentioned in the previous section. Consistency in this context means that the data, e.g. a file, is identical across the whole the network. The same file on all the nodes are called replicas, and the process in which the replicas become identical is called convergence. In a file-syncing system, when a file gets updated, the replicas on the other nodes are

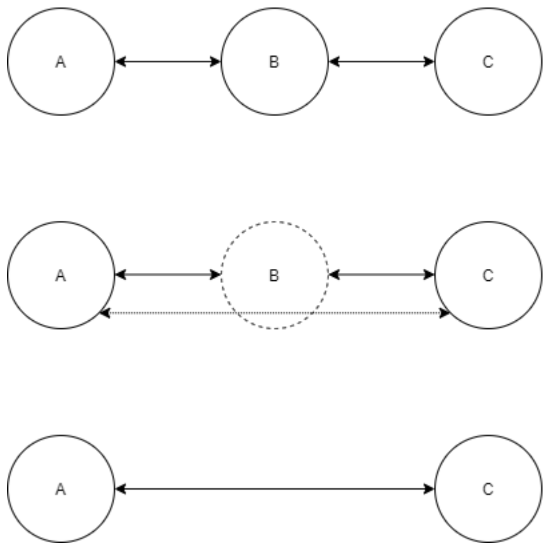


Figure 3: Node B leaving a P2P network

updated as soon as possible. This is the preferred property in a file-syncing application, but this property would not be without challenges. Specifically, the CAP-theorem[1].

2.3.1 CAP

File-syncing applications as described in section 2.1 are a distributed system, where data are spread across on different nodes in the network. According to the CAP-theorem, a distributed system have to choose two of the three properties: *Consistency*, *Availability*, and *Partition tolerance*. All three properties are central in a distributed system, and choosing which one to be excluded is dependent on the application itself.

A simplified description of the three properties in the context of file-syncing.

- Consistency: The same files across the network are identical.
- Availability: It is always possible to access the files in the network.
- Partition tolerance: The system must be able to tolerate arbitrary network partitions.

Often in distributed systems, the property of *Partition tolerance* is non-negotiable. The system has to tolerate any kind of network partition. Then, the choice fall between *Availability* or *Consistency*. Excluding the first means that, in the case of a network partition where a node is unable to communicate with the rest of the network, the node has to abstain itself from performing any changes to the local data. Such that the local data do not diverge, and can be easily updated when the network partition dissolves. On the other hand, in the same network partition, but this time the *Consistency* property is excluded, the node can perform any changes to the local data. Here a file conflict will occurs. The same file in is now different in both the network partitions, and they can not be synchronized in a trivial manner. While files can diverge in the first example, the files may only be "behind" the main replica, and the same sequence of updates can be applied to the diverged files to become the same as the main. Figure 4 shows a visual representation of the file when *Consistency* is excluded.

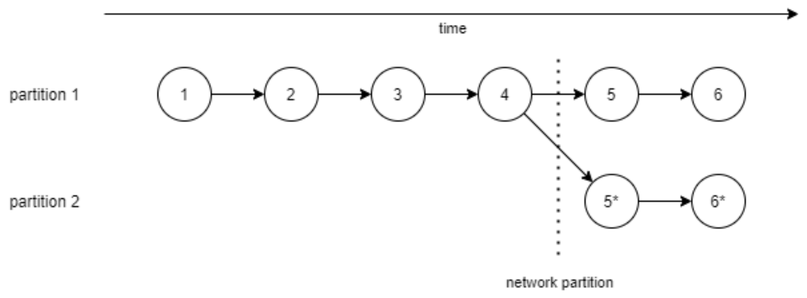


Figure 4: File timeline with network partition (No Consistency).

Since the P2P file-syncing system need both *Partition tolerance* and *Availability* to enable file updates when the node is offline/disconnected from the network, the *Consistency* property needs to the compromised. One common solution is to adopt the weaker property known as *Eventual Consistency*.

2.3.2 Eventual Consistency

Instead of system-wide consistency, this property is slackened into *Eventual Consistency*. While immediate file synchronization is desirable, in situations where a network partitions, files are allowed to diverge, but the system requires a mechanism for synchronizing the files when the network partition dissolves. This is usually be done in such a way that after file updates have been retrieved, the node will hold on to the update until the node is idle, i.e. all updates have propagated through the network. Only after that, the system will try to apply the updates in the correct order. Replicas in different network partitions are no longer required to be identical, but the problem is the assumption that the system will have an idle period. Which in some scenarios might not happen. For example, in systems where a file is constantly being written to, the system will never have a idle period, thus the stored updates will never be applied. In other words, this property gives to time guarantee that the convergence will be achieved. While this property could be applicable in file-syncing as one could reasonable argue that file updates do not happen frequently enough, having to wait a relative long time for the updates to be applied can diminish the user experience, and a stronger consistency model is needed.

2.3.3 Strong Eventual Consistency

A stronger version of *Eventual Consistency* which can capture "when" convergence has been achieved is known as *Strong Eventual Consistency*[2]. Which can be easily defined as "Replicas have converged if they have seen and applied the same set of operations". This is split into two part. The first, all the updates will eventually be transmitted to every node. The second, the retrieved update can be applied immediately on the local data. Together, the replicas will converge only when the same set of updates have been observed by both sides. Additionally, this should hold true even is the updates are applied in a different order. *Strong Eventual Consistency* allows both the other properties in the CAP-theorem to exist together, while having a desirable consistency model. A network can be partitioned, but *Strong Eventual Consistency* still holds since both network partitions have seen different sets of operations and can later, when the networks merge, merge the two sets of operations to achieve convergence.[3]. In the next section, two families of algorithms/datastructures that incorporates *Strong Eventual Consistency* will be introduced.

2.4 CRDT and OT

The two families are Conflict-free Replicated Data Types and Operational transformation, where the later will be the basis of the primary datastructure used in the file-syncing system. A desired property in any syncing application is to ensure that the same replicas for any user converges. This can be visualized in Figure 5 where two users, Alice and Bob, are concurrently modifying a shared folder stored on their local computer. Alice adds a new file under "folder1" and Bob renames "folder1" to "animals". After applying their own operation and getting their desired outcome locally, both operations need to be applied on the other end. It is at this point a conflict occurs. On Alice's end, "folder1" gets renamed to "animals" and on Bob's end the new file is added under another folder. In short, concurrent updates to replicas can result in inconsistencies between the replicas that need to be solved to achieve consistency. To solve such conflicts there are two families of algorithm: the older *Operational Transformations* and the newer *Conflict-free Replicated Data-types.*

2.4.1 Operational transformation

Operational transformation(OT) was first introduced in 1989[4]. The first iteration did not require a centralized server architecture. But as more research was invested in OT, it soon became obvious that most of the methods did not work as they were expected, and in the end, required a centralized server to work. Currently, the only OTs that work are either too complex to implement or require a central server architecture. OTs work in a fashion where an external maintainer detects structural changes and modifies the incoming update so that the update is correct relative to the local state. This family of techniques is commonly used in collaborative real-time text editing to ensure eventual consistency between the editors. In our previous example, that would correspond to

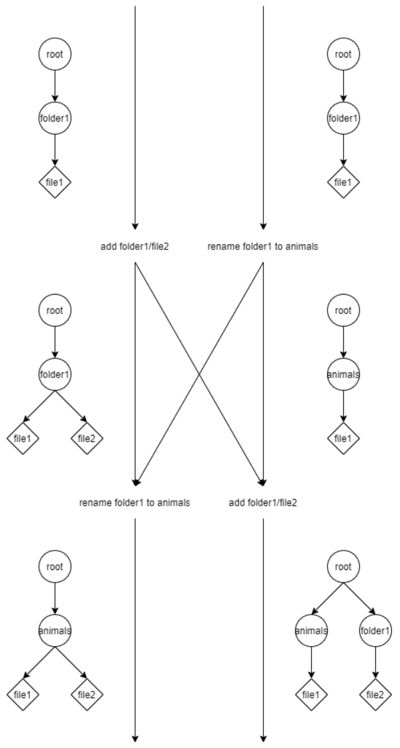


Figure 5: Diverged systems

changing the incoming update on Bob's end to add a new file under the renamed folder instead. As shown in Figure 6.

2.4.2 Conflict-free Replicated Data Types

For CRDTs, the way to solve consistency issues is built into the data type itself. The data type is designed in a way that allows for commutative execution of operations such that the CRDT results to the same state(SEC). This means that the operations can be executed immediately on arrival, instead of being executed in a chronological sequence. There are two main classes of CRDTs: state-based CRDTs, and operation-based CRDTs. The difference between these two lies in how the updates are propagated to each other.

Operation-based: Operation-based CTDTs send the update directly in form of metadata containing the descriptor of the operation and the data associated with the operation. For example to add the value "apple" into a set, an "ADD" operation descriptor is accompanied with the value "apple". The operations in operation-based CRDTs are commutative, order of operations applied does not matter, but depend on a reliable communication protocol to ensure that the operations are broadcast to other peers. All updates have to be received by all nodes for convergence to be achieved. Consequently, there must be an operation deduplication mechanism to handle duplicated operations.

State-based CRDT are the other class of CRDT, which instead of only sending the update, i.e. the data necessary to change the old state to the new state, whole local state of the CRDT is sent to the other replicas. The other replicas then merge the incoming state with their local state. The merge operation in state-based CRDTs must satisfy the commutative, associative, and idempotent criteria. The order of states that get merged is irrelevant (commutative" and "associative), and merging the same state twice has no effects (idempotent).

Both classes of CRDT can be converted to the other class, and which class to choose depends on the properties of the classes. State-based CRDTs are easier to implement as they only require a best-effort communication protocol to transmit the local-state to the other replicas, but have the disadvantage of potentially high cost of transmission size. Operation-based CRDTs on the other hand, only send the required data to the other replicas. However, depends on a reliable communication protocol to ensure SEC as it relies on all replicas having seen the same set of operations. State-based CRDTs can drop transmissions since the state itself implicit contains the whole local set of operations.[2]

Continuing on the previous example, a possible CRDT implementation to resolve the consistency issues in Figure 5 could be done:

- Leave a tombstone if a folder gets removed/renamed/moved. For renamed or moved folders, the tombstone points to the new location.
- If an operation involves a tombstone, check the new location if it exists and perform the operation there.

An example of the difference in how OTs and CRDTs resolve conflicts can be seen in Figure 6.

2.4.3 Examples of CRDTs

Several general-use CRDTs have already been implemented[2][5][6], and this report will focus on the Last-write-wins-element-set(LWW-element-Set) to build the file-syncing system in section 3. For short, a LWW-element-Set is functionally an extension to the classic hash-table, where an additional timestamp parameter is required in the "add", "remove", and "update" operation, and the values in the hash-table are associated with the timestamp during their insertion. The LWW-element-Set, as its name suggest, follows the rule of last-write wins. As an example below, when the "update" operation is called, a comparison between the passed timestamp and the stored timestamp is made to determine the outcome of the operation. Only when the passed timestamp

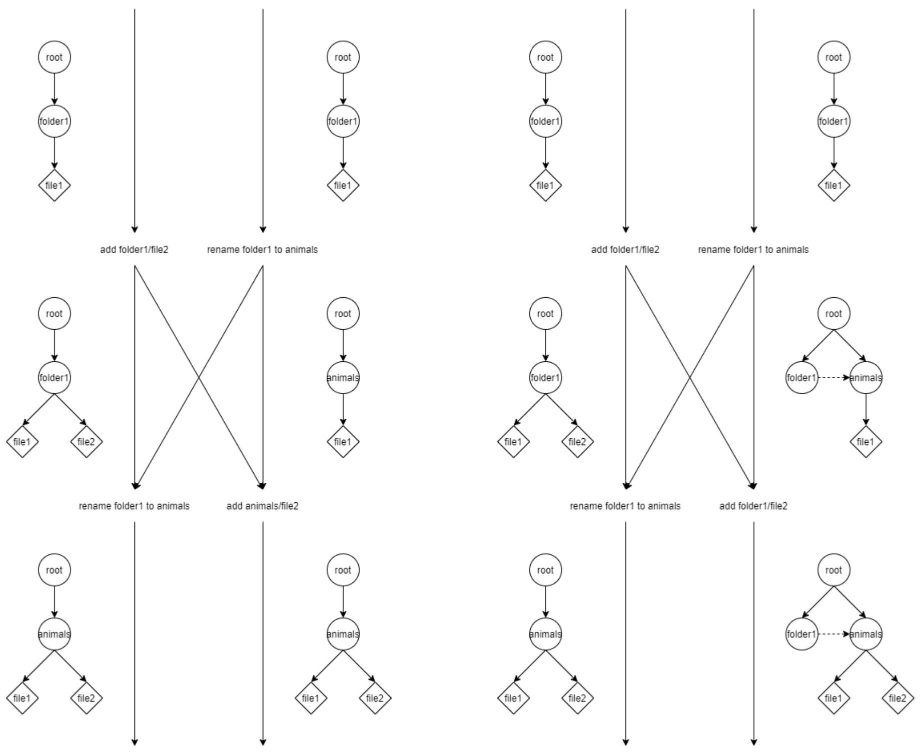


Figure 6: Convergence with OT(left), and CRDT(right)

is "later" than the stored timestamp, will the operation be completed. The pseudo-code of the new update, and merge function is shown below. In the next chapter, a CRDT will be designed from the LWW-element-Set, due to the similarity to the hash-tables used as cache in database systems.

```
function update(key, value, hash_table, new_timestamp) {
    old_timestamp = hash_table.get_timestamp(key)
    if new_timestamp > old_timestamp:
        hash_table.update(key, value)
        return true
    return false
}

function merge(local_hash_table, remote_hash_table) {
    for key, value, time_stamp in hash_table2
        update(key, value, local_hash_table, time_stamp)
}
```

2.5 Conflict resolving

On the topic of consistency, when a conflict happens, it needs to be resolved at two levels, system-level and application-level. Resolving the conflict at system-level means that the system itself is at a state where it can continue working. Whereas an application-level resolve means that the resolved state is desirable for the user. Sometimes resolving at one level automatically resolves both. This can be seen on systems where only the latest data is important, like the LWW-Element-Set previously described. An example where the resolve mechanism is not the same for system-level and application-level would be a "git merge-conflict". To briefly explain, a git merge conflict happens when two users concurrently commits the same code file, but with different content, and the system is unable to know if the parts should be kept or throw away. At that point the system has two version of the same file, and to solve at the system-level, Git keeps changes from both version. But even if the conflict is resolved at system-level and Git can continue working, the resolved state is not what either of the users want. Therefore, Git temporary halts, and forces the users to manually select the changes they want, thus solving at application-level. Another example of an application specific conflict resolve is the case in Amazon Dynamo where two different shopping carts were stored on the servers, and both were kept until the user had to manually selected the wanted items from both[7]. A detailed description of the different conflicts that can occur in a file-syncing system can be found in section 3.3.2.

2.6 Types of timestamps

In a distributed network, the matter of ordering event is no longer trivial. There needs to be a mechanism for to determine a deterministic ordering that can be reproduced in any of the peers in the network. One family of techniques commonly used are different type of timestamps. Two type of timestamp will be of great importance for the problem of determining an deterministic ordering in a distributed network.

2.6.1 Lamport timestamp

The simplest form of ordering is with the use of Lamport timestamps[8]. Lamport timestamps is a logical clock, which is only capable of tracking the causal ordering between events and not the time in-between. It accomplish that by requiring each process have an interval counter which increments for each event. In Figure 7 the process P0 gives the first event an 1 and the next a 2.

New event at the local process:

1. Retrieve the current counter.
2. Increment the counter.
3. Assign the the new value of the counter to the event.

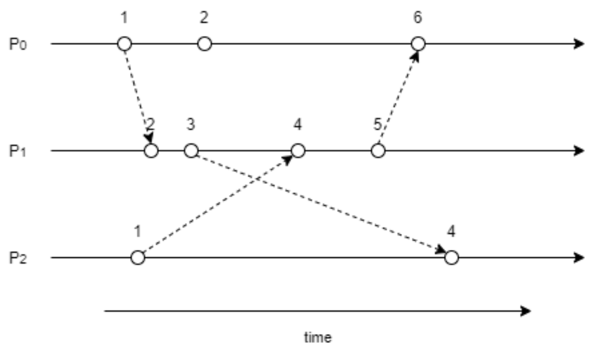


Figure 7: Causal order with Lamport timestamp.

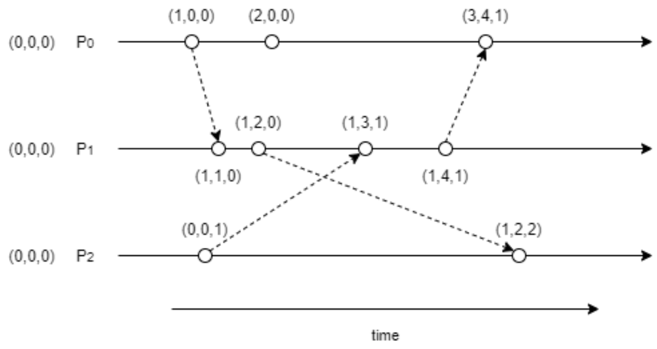


Figure 8: Causal order with vector clock.

When an event is sent to another process, the process compares the arrived event's Lamport timestamp with its own counter. The process then fast-forwards its counter to the highest of the two, increments the counter, and at last assign the newly incremented value to the event. In Figure 7, P1's event with timestamp 5 is sent to P0, and the P0's internal counter is 2. After P0 compares its 2 with the event's 5, P0 fast-forward its counter to 5 and increments the counter to 6. The event from P1 which was sent to P0 has a Lamport timestamp of 6.

Event from another process:

1. Retrieve the current counter and the incoming event's timestamp.
2. Fast-forward the process' interval counter to the maximum of the two value.
3. Increment the counter.
4. Assign the the new value of the counter to the event.

While this time stamping scheme is simple, it does not provide a sufficient ordering capability. It is known that if an event A happens before event B then the Lamport timestamp of A (LA) is strictly lower than the timestamp of B (LB). In short, $A \rightarrow B \Rightarrow L(A) < L(B)$. But it not possible to determine if A happened before B by their timestamps. $L(A) < L(B) \Rightarrow A \rightarrow B$ is undetermined. An example from Figure 7, P1's event 5 and P2's event 4 happens concurrently. Therefore Lamport timestamps can not be determine the ordering. But vector clock, which is an extension of it the Lamport timestamp can resolve the ordering. The benefit of Lamport timestamps compared to vector clocks is that the size of the timestamp is independent from the the number of processes.

2.6.2 Vector Clocks

While Lamport timestamp is only a single value associated with an event, the vector clock[9] is an array of values where each element in the array correspond to a process. Each process have its counter, just like in Lamport timestamps, but only uses its own position in the array. Process 0 only increments the first element, process 1 increments the second element, and process n increment the n-th element. Example from Figure 8, P0's (1,0,0) increments to (2,0,0).

New event at the local process:

1. Retrieve the current counter.
2. Increment the counter.
3. Copy the last vector clock seen and update the process' position in the vector clock with the new value.
4. Assign the new vector clock to the event.

Following the same example from Lamport timestamps, where the event from P1 is sent to P0. This time, the incoming event's vector clock and the vector clock last seen in P0 are compared. P0 merges the two vector clock by an pair-wise maximum to get a new vector clock. For example, the event (1,4,1) and (2,0,0) gives (2,4,1). After incrementing the process' counter, the new event becomes (3,4,1).

New event at the local process:

1. Retrieve the current vector clock and the incoming event's vector clock.
2. Merge the two vector clocks by a pair-wise maximum.
3. Increment the corresponding counter of the merged vector clock.
4. Assign the new vector clock to the event.

This ordering scheme has the same property where for event A and B, $A \rightarrow B \Rightarrow V(A) < V(B)$. The $V(A) < V(B)$ is a pair-wise greater-than comparison. This time, the reverse also holds, $V(A) < V(B) \Rightarrow A \rightarrow B$. Therefore this scheme can determine the ordering from the vector clocks. When $V(A) < V(B)$ doesn't hold, event A and B happens concurrently, and a tiebreaker is needed for a deterministic ordering of concurrent events.

2.6.3 Physical clocks

While logical clocks such as Lamport timestamps and vector clocks are efficient in tracking the causal order of updates, the time between each event is lost. Physical clocks encapsulates this property by using a real-world representation of the time. One in particular is the Unix timestamp, which is the number of seconds elapsed since January 1st 1970. But using a physical clock requires the different users to synchronize their clocks in some way. This can lead to situations where two events A and B in the order "A then B" get registered as "B then A" because the clocks at location A and location B were out of sync. If the assumption is made that two clocks are relatively synchronized, i.e. the deviations between the two clocks are "small". Then, a mostly correct causal order can be achieved. But the most important aspect, a deterministic ordering is still valid. This kind of ordering is will be used in the later chapters as the tiebreaker for when vector clocks are unable to determine and ordering, i.e. concurrent events.

2.7 Related Works

CRDT on file systems[10] investigated the use of CRDTs in file systems, and laid the basis for this report. The paper described the general setup of a CRDT-based file system, and investigated the possible conflict which could occur in the Ficus Replicated File System[11]. To list the few conflicts: concurrent file updates, concurrent insertion of same-name files, and when the file get deleted and updates concurrently. It presented conflict resolve policies similar to section 3.3.2, and constructed automatic conflict resolves. For cases where a satisfiable state was unclear, the system opted for duplication of files for the user to then resolve manually. Furthermore, a conflict awareness mechanism was provided for the user. This report extends on the work of the paper. Where the paper designed the CRDT on top of a specific Ficus file system, with has limited conflicts, this report looks at application-level synchronization/replication support, which has inherent cross-platform capability, and is not dependent on a specific file-system.

Survey of data replication in P2P systems[12] investigated several p2p system, and constructed a list of properties which defined a proper p2p system. The different properties are the following:

- Data type independence: When sharing data, the system should be capable of replicating any type of data across the network.
- High level of autonomy: Coordination should be made optimistic and deterministic, so that minimal requirements are put on each peer. Peers should be able to safely perform operations with just the information they have and no additional network communication with other peers.
- Multi-master replication: Data should be replicated across multiple peers, and allow concurrent updates to the replicas.

- Semantic conflict detection: Concurrent operations can cause conflict, which need to be resolved to a desirable state.
- Eventual consistency: Global consistency is impossible in a proper p2p system. Replica divergence should be allowed, but should be converged once all peers are in a resting state after all changes are propagated through the network.
- Weak network assumptions: No assumption about the network infrastructure should be made. The system should work on all types of network infrastructures: Ethernet, Cellular, fast/slow, reliable/unreliable, etc.

File Synchronization Systems Survey[13] by Zulqarnain Mehdi and Hani Ragab-Hassen analyzed the different aspects of some of the most popular file-syncing services currently in the market. Including the most known ones like Google Drive, OneDrive, iCloud, and Dropbox. They looked at the different network architectures the services are built to work in, the advantages and disadvantages, how the files are stored, and whether or not file collaboration is supported. From the services they looked at, only the cloud-based(centralized server architecture) was capable of collaboration. At last they conclude that P2P-based, file-syncing systems are both preferred and better than cloud-based file synchronization systems.

3 Architecture

This chapter presents and discusses the overall architecture of the system, the roles and functions of the individual modules, the modifications to the concepts presented in the previous chapter and its trade-offs, and the general process diagram of the system under action.

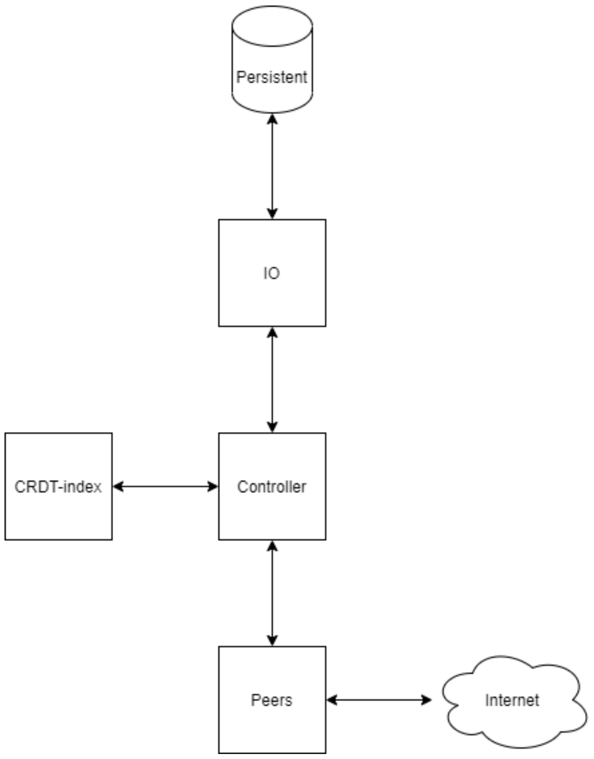


Figure 9: Conceptual architecture

3.1 Conceptual Architecture

To make the system as comprehensive as possible, it is split into four distinct parts, each responsible for one area. The conceptual architecture is shown in Figure 9.

CRDT-index is the CRDT data structure responsible for keeping track of all the files in the system, and merging all updates in a deterministic manner such that regardless which order the updates were applied, the end result after all updates have been applied, the state of the data structure is identical across the network. To be more specific, as long two nodes have applied the same set of updates, the state is the same on both ends.

IO is the module responsible for interacting with the persistent storage system, e.g. the local file system. This module will constantly watch all files in the designated folder, and will trigger the corresponding event based on whether it is a file or folder, and the type of action, file creation, file deletion, or file modification. All I/O-operations such as read and write operations are passed to this module.

Peers handles all the communications to the rest of the network. Such communications include, peer management, file updates, file retrievals, index synchronization, and etc. All remote file updates will be processed in a similar manner as the local file updates triggered by the I/O-module.

Controller is the brain of the system, that takes the file updates that were triggered by I/O-module, and sends it to the CRDT-index. After the index has finished processing an update, and have determined that the update is not outdated, it tells the controller to broadcast the Item to all known nodes. Consequently on the connected nodes that receive the update, the peers-module passes the update to the controller which then forwards it to the index. Just like the first node, after the index has finished processing, the update needs to be broadcast again due to the nature of P2P-networks that nodes are not guaranteed to be connected to all nodes. At last the controller is responsible for managing the user configurations.

3.2 Data object representation of a file

The most important part of the system is determining the data structure that should be the object representation of the individual files. The mapping between the file and the data object must be unique, and it must be unique in the context of the file. In other words, a file that has the same path as an deleted file are not related to each other, and thus not regarded as the same file[5]. Specifically an Item represent the a file at a given timestamp. Table 1 presents the core fields for the data object.

Path represent the relative path from the root folder where the system is watching. There are several advantages in only using the relative path instead of the full/absolute path. One being that the absolute path is longer than the relative path and can get unnecessary long depending on where the root folder is located. If the root folder is located deep within several folders in the file system, the memory footprint of the path field would be misspent. Another advantages is that the path field can be kept the same across all nodes in the system, as each individual node can get the absolute path of each file by joining the relative

[5]Overwriting a file with another file is considered a file modification not a pair of file creation and file deletion, therefore sharing the same data object.

Path	String
Id	String
VectorClock	Array([String, int])
LastModified	float
LastActionBy	String
LastAction	String
Tomb	Tomb

Table 1: Item object

path with the root path.

Id is the unique identifier assigned to the individual files. During the lifetime of the system, a file may have been created, deleted, and then created again, which in this case is assigned two different ids, on upon the first creation, and on the second creation. If the two peers concurrently creates the same file, i.e. the path is the same, then the two files are also considered as unrelated, and then assigned two different ids.

Vector clock is the main tool used to detect conflicts arose from concurrent updates. An empty vector clock is the same as a vector clock where all clock are set to 0, and each subsequent update increments the clock associated clock. By only tracking the clocks for the nodes that have performed some kind of change to the associated file, trades a longer processing time, by requiring the node's to find its clock before incrementation, for a smaller memory footprint. It can be argued that this change is in favor of the scalability of the network. By assuming a network with size n, the original vector clock's memory footprint would scale with n, with incrementation be of constant time. The modified vector clock's scalability in memory footprint and incrementation time will be of n', where n' is the average amount of users working on the same file, and this value can be argued to be much smaller than n. As for a large network, it is highly unlikely that all n nodes are modifying the same node and only a small subset do. For small networks its more likely that n' is close to n, but for small sized networks the linear scalability is not really an important issue. Additionally it could be said that the use of a resizable vector clock is more desirable as it is more important to accommodate a constant change of nodes in a network as new nodes can join the network and old nodes can permanently leave the network. In contrast to reallocating and recomputing the vector clock whenever the network grows or shrinks. But whether this modification truly is in favor during practical use is yet to be verified.

Last modified is a physical clock tracking when the last change was observed, and is used as a tie breaker together with **last action by** to determine a deterministic ordering when the vector clock observes a concurrent update. The ordering favors the more recent update as it is often more important.

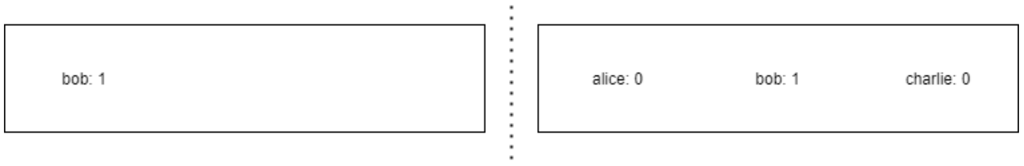


Figure 10: Unlisted peers are treated as 0.

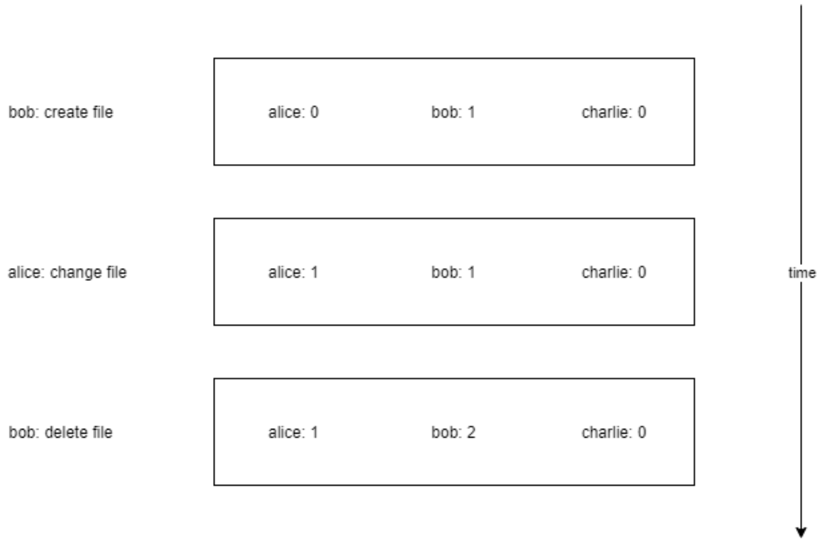


Figure 11: Incrementation of the vector clock.

Last action by is the globally unique id of the node that performed the last update, and is used as the final tiebreaker for the **last modified** comparison. While that scenario may be highly uncommon, depending on the physical clock used, but for the sake of correctness this last tiebreaker is required.

Last action is the identifier of the specific type of update. Update types include creation of new files, deletion of existing files, and modifications of existing files. Furthermore, the system can be tailored to include specific resolve mechanics on concurrent updates depending on the combination of the last update type, and the incoming update type.

Tomb is a optional field which indicate if the file has been deleted or rename/moved.

3.2.1 Representation of a Tomb

The **Tomb** object is shown in Table 2, and indicated when a file is deleted, renamed, or moved. For operations such as renaming a file, or moving a file, the object includes an additional field which contains the new path after renaming/moving.

Field name	Type
Type	String
MovedTo	String

Table 2: Tomb object

CRDT comes with its drawbacks. To ensure a convergence between replicas, an indication of deleted data must remain in form of tombstones. Any changes that removes, renames, or moves must leave behind a tombstone indicating what happened to the file. As the Item is connected to a file by the path, and any actions that changes the path, e.g. rename and move, will result in a new Item. These tombstones will remain permanently as there are no guarantee that all peers have observed the same tombstone, leaving the size of the CRDT to grow continuously.

Type is the the action that created the tomb. E.g. deleted, renamed, or moved.

Moved to is a value indicating where the file has been renamed to or moved to. For deleted files, this value is null.

3.3 Details: CRDT index

The module responsible for keeping a deterministic state of the files regardless of when a certain update is applied, is based on a CRDT version of the hash-table. The key used for each element in the hash-table is the file's relative path from the root. While only using the file's relative path keeps the process of retrieving the corresponding "Item" for the file simple, it does not accommodate the use of tombstones as discussed in section 3.2.1.

One simple solution to this is extending the key to not only require the relative path, but also the file's unique id. That way, the tombstones can exist alongside the regular "Items". But doing makes the act of retrieving a file's "Item" data problematic. This assumes that both the file's relative path and its id is trivial to get or at least there is an external mechanism that keeps track of the id of each existing file. This will pose a problem when the local file system is used for file persistence, in which only the file path is retrievable upon detecting a file update event. That is without writing a custom file update detector. The main problem with only having the file path to find the corresponding "Item" object it that the path is not enough as an unique identifier since over time multiple files might have had the same name. A common scenario where two unconnected ”Item” objects have the same file path is when an existing file gets deleted and sometime later a new file with the same name, or more accurately the same path, is created. Therefore, choosing this method puts an unnecessary restriction upon the choice of persistent storage system.

Another way could be merging the path and the id to one value, or just use the id as the path. But then, the files may be indistinguishable for the user, e.g. ”recipes - 0917...b30a.txt”. The id is a 64 characters long SHA-256 hash. Other applications such as the default Windows File Explorer may need to adapt to the new naming scheme by removing the id part to show a comprehensive list of files to the user.

A method which solves this problem, or at least solves to a degree, is to extend is to have the result of the hash-table lookup be a linked list or any data structures that is iterable and dynamically resizable. This way, a two-phase hash-table lookup is enough to get the corresponding ”Item” object for a given file. The first phase is to get the list of ”Item” objects by the file path and then the second phase is to find with the correct ”Item” by comparing with the id field. Although this method also rely on knowing the id before performing the lookup, the advantage is the ability to find ”Item” objects under the same path. Thus enables a way to infer the correct id and as a result the correct ”Item” object. One way to infer the correct ”Item” from the list is to always take the one with the latest modified timestamp that is not a tomb, since local updates such as ”remove” and ”change” assumes an existing ”Item” that correspond to the file, and add updates assumes either no ”Item” for the given path or no non-tomb ”Item” objects, thus is assigned a new id. This method is comparable to a range search in a B+ tree with (path, id) as the key.

The matter of assigning the corresponding id to a file is only present for the local node, as previously described in the file update event assumptions, and for incoming updates from remote nodes the id have already been assigned, thus skipping the pre-processing all together. The only step aside applying the update to the hash-table is to rebroadcast the update to its neighbors, due to the nature of P2P networks.

One important remark is that during the step of applying the update to the hash-table, only one update per path can be executed at the same time. Not doing so will cause race conditions within the update applying process to result in undesirable outcomes such as updates been overwritten instead, i.e. when two nodes makes changes to the same file on their local ends, and the timing works out such that on one node, or both, both the local update and the incoming remote update get applied at the same time. The desirable outcome is that both updates are kept, either by deterministic keeping one of the updates as an alternate file or merging both updates with a CRDT specific to the file type. Sequence CRDTs can be used to merge concurrent updates for text files.

3.3.1 Conflicts and Resolve Policies

The most important aspect of using CRDT is whether or not the state after merging is the desired state. What should happen if concurrent file updates occurred on the same file? Depending on the situation there could be several ways to solve such merge conflicts. As described before, concurrent file updates will result in the corresponding Item objects to have concurrent vector clocks, and then have to use the other information in the Item objects to resolve the tie and set a deterministic ordering between the two Items. In this case, the "last modified" value of the Items will be the tiebreaker. But this value can too end in a tie, and while this is highly unlikely due to the fact of how physical clocks work, a final tiebreaker is needed for correctness. Which in this case will be the user that performed the update, i.e. the "last action by" value. Since the each individual user id is globally unique, no further ordering schemes are necessary. Once the global ordering the two conflicting Items are determined, the latest Item will have its path changed to a new location. The system follows a "better safe than sorry" policy to make sure no information is lost.

- Last-write-wins: Default resolve policy. The latest update take precedence.
- Rename: Concurrent vector clocks. One file is deterministic renamed based on the ordering from "last modified".

While an automatic resolve policy may be practical, no need for the user to be involved, sometime situations will still occur where the users need to manually resolve the conflict. Like the previously mentioned "better safe than sorry" policy. Depending on the use-case, or the preference of the user, personalized resolve policies could be added to fully remove the manually part of the resolve policy. It is therefore important to at least give the users an conflict awareness tool to let the user determine which files are in conflict and where they are located.

3.3.2 Conflict Examples

This section provides an overview of the conflicts that can occur in this kind of file-syncing system, and shows that different resolve policies way be used on each conflict depending on the intended behaviors of the actions. This will be more comprehensive by showing a few examples of conflicts and how the context behind the conflicts may change the intended behavior. A selected subset of the conflicts are listed below and the complete set of conflict can be found in Appendix A.

Concurrent addition of files with the same name. The simplest conflict that can occur is when two or more users add a new file to the same folder in their local replica, but the files have the same name. If the conflicting files are not related to each other, i.e. the names were chosen just by coincidence, the system should then keep and deterministic rename the files. E.g. two users add their own picture of a cat, but by laziness both users name the file ”cat.jpg”, and the resolved state would then be two files ”cat.jpg” and ”cat (2).jpg”. This is the case duplicates are the intended outcome, but there exist another case where one would only like to keep one of the duplicate. E.g. two users want to add an instruction text file to the shared folder, but they did not decide who should do it, and in the end both added the instruction file. In this case a simple resolve policy would be last-write-wins. The file addition with the latest timestamp overwrites the older one. The two policies is illustrated in Figure 12.

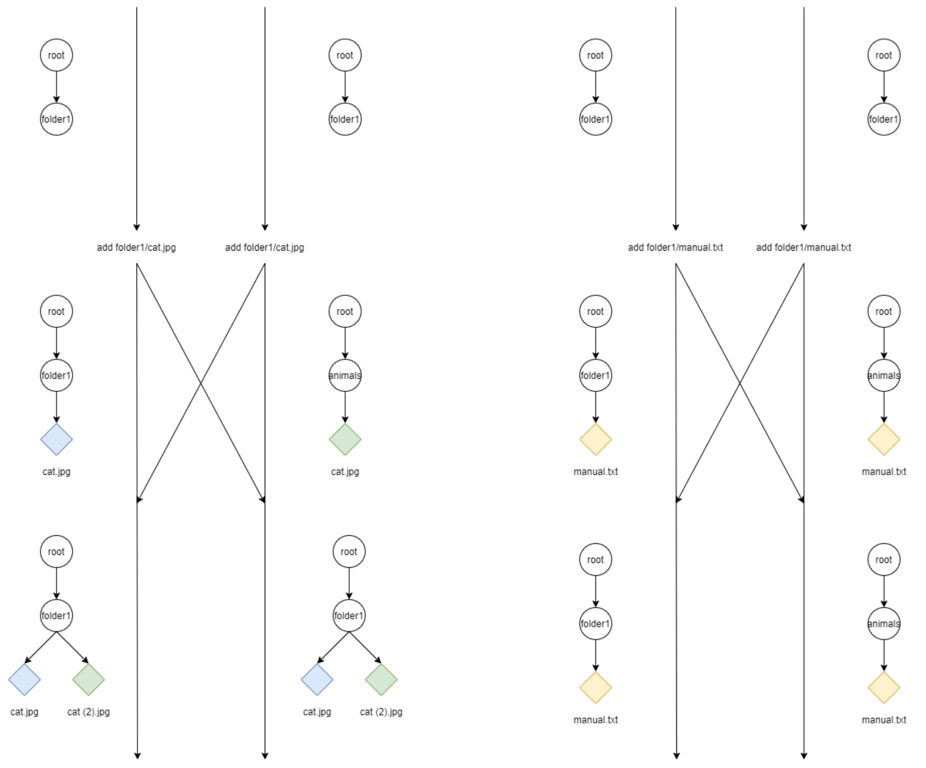


Figure 12: Two different resolve policies for concurrent addition of files with the same name. Left(deterministic rename) and right(last-write-wins).

Concurrent modification and move. This conflict involves an user modifying a file and another user moving the file to another location. An naive solution would be re-adding the modified file at the old location. Resulting the ”same” file appearing at two different locations. This is certainly not a state that the users desire. The modification needs be applied at the new location, and can be achieved with the use of tombstones like in Figure 6. In case when the parent folder was moved, tombstones of all the moved files must be created at the old location for the resolve.

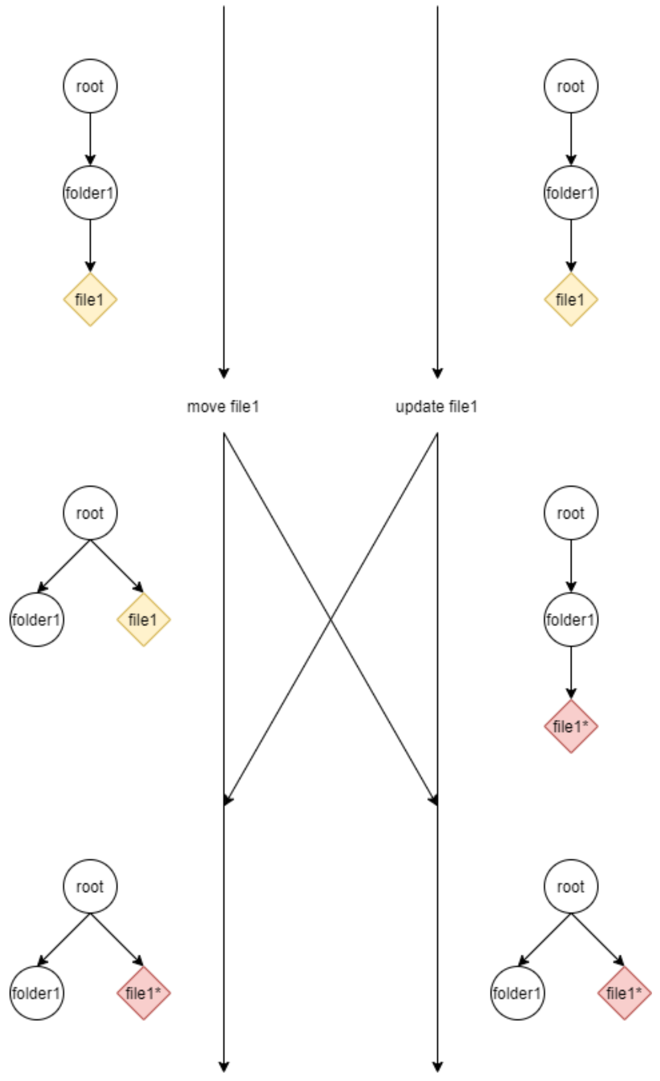


Figure 13: The file update is applied to the correct file for the other user. File before update (Yellow) and file after update (RED).

Folder cycle. There exist a special type of conflict which do not involve conflicting files. When one user moves a folder A under folder B, and another user moves folder B under folder A, a problematic state will occur. As shown in Figure 14, the concurrent moves could cause the inner folders to form a cycle, and thus inaccessible. Among all the conflicts this would be considered as the most dangerous as a large amount of files could be lost forever.

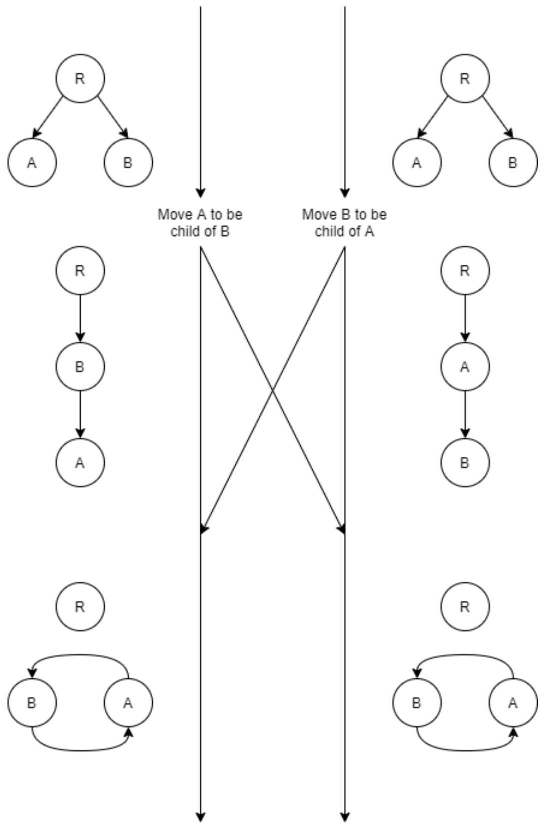


Figure 14: Right side moved A to be child of B, left side moved B to be child of A. A naive merge resolve will result in both A and B to be lost.

As a side note, depending on the use-case of the file-syncing system, some conflict might occur more frequently or less frequently, or maybe not occur at all. The creator of the shared folder could set up the shared folder in a way that only the creator can add, remove, or change files, then there will practically be no concurrent operation and thus no conflicts. If the shared folder is set up such that only files are allowed, no nested folder, then all conflicts involving moving files would not occur.

3.3.3 Nesting CRDTs for File Content Interleaving

To this point, the synchronization has only being applied to the files are the whole, but having an CRDT-based synchronization on the file content itself could allow for even less manual interaction from the user. An example for this is the use of CRDT for text. There exists many CRDT, that are tailored for the use for text collaboration, which as a reference would allow Google Docs collaboration. Such CRDTs for text are called "Sequence CRDTs", examples of this kind of CRDTs are RGA[14] and Woot[15]. Therefore, file types for which there exist an CRDT to enable content-based synchronization could integrate a CRDT nested within the Item object. This could resolve the concurrent modification conflict, where two user concurrently modifies the same file. Instead of resulting in two files for the two versions, a single with both modifications are merged in a deterministic manner. Of course not all file type would want this collaboration feature, e.g. JPEG, mp3, and etc, as there is no proper way collaborate on these in the first place.

3.4 Details: IO

The system should associate an update method to the three types of update events: creation of new files, deletion of existing files, modifications to existing files. The choice of having only these three base types of updates methods is due to the fact at these updates are the most trivial file events to detect. More advanced updates such as moving or renaming files can be inferred from pairs of file creation and deletion events. There should be two ways to execute the update events, the first is to bind these five update events to an off-the-shelf file watcher library to listen to changes on the local file system. This should be relatively easy to accomplish as most file watchers is able to differentiate between these three file events. The second, more general scenario, is to use the three update methods as the application programming interface(API) to the file-syncing system. This way, other developers could integrate this file-syncing system into their own applications.

The file-syncing system could implement the basic I/O operations read and write, to which the remote updates can be writing to local file system, and remote request for files can the retrieved and sent to others.

3.4.1 File Distribution System

The file update detection and file access should be split into two separate modules, such that in some use-cases one can decide to drop either one of them. If a node only wants to read the files, the node can drop the file detection module. In scenarios where the network is used as a file distribution system where nodes are either providers or consumers, the consumers do not need to keep tracking files for changes, and providers do not need to perform any incoming updates to the local persistent storage, since there would not be incoming updates at all. In conjunction with a permission module where an admin node can regulate the permissions the individual nodes have, the system can easily be turned into a file distribution system.

A slight remark on the notion of providers not requiring the file detection module, if the are multiple providers that have files with the same relative path, but of different context, then the providers need the file detection module to resolve the conflicted files.

3.5 Details: Peers

The peers-module encompasses all communication between all other nodes. At the basic level, the module should maintain a collection of communication channels to all known. Each communication channel do not need to be of the same protocol, such that the individual nodes have a choice between which to use. While the ideal scenario is that regardless of which protocols a node chooses to use, the node may not be able to avoid mainstream protocols such as http(s). A globally default protocol needs to be assigned to be able to make sure proper propagation through the network. As seen in Figure 15, the are two distinct group of nodes, the bigger one that are communicate over **http** and the smaller one that communicates over a lesser known another protocol, to connect the two groups, node *d* from the smaller group communicates with node *b* from the bigger group through the designated default protocol, which in this case is **http**. The motivation for not limiting the communication over a single protocol is to allow seamless changes to the default protocol. In the future when new communication protocols that are deemed to be superior to older ones, and have gained a significant traction in the user base, the system can start to designate the newer protocol as the default protocol.

To make the implementation of newer protocols more developer-friendly, there needs to be a predefined set of the necessary remote procedure calls (rcp) to ensure each rcp are available on each peer. The set of remote procedure calls includes:

- a method to send updates
- a method to request remote files
- a method to request the state of the remote node's index

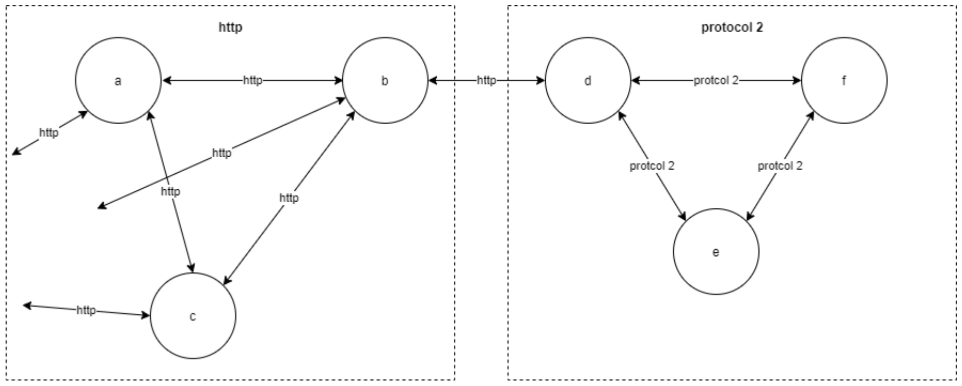


Figure 15: Two distinct group in the network.

- a method to request the permission to join the network when joining for the first time
- a method to add itself to the remote node's list
- a method to a specific permission like, e.g permission to add new files
- a method to quickly determine if the index between the two nodes are equal

After a local update have been processed and saved to the index. The update will be broadcast to all known nodes over the individual node's protocol, e.g. if a node is connected by http(s), then the update will be sent over http(s). When a remote update is received, its processed and save to the index, and then written to the local file system.

3.6 Extra Modules

This section lists a few optional modules with are not required for a working file-syncing system, but can enhance the system in different aspects.

3.6.1 Replication

Due to a p2p network never have a permanent population, a certain degree of replications is desirable. On the other end, having full replication over the network would create unnecessary storage space usage as the preference of full replication or partial replication is highly dependent on the peer. Therefore, a certain degree of replication is always required, and the each peer is given the choice between streaming files over the network or a full replication on the local computer.

3.6.2 Permissions

An important consideration in a file-syncing system is who can write or read the files. In the most basic configuration everyone has both the read and write privileges. But this can lead to a chaotic system where users can maliciously mess with the files. A type of privilege management mechanism should be included to restrict or grant specific privileges, depending on the use-case of the file-syncing system.

3.6.3 Availability

To ensure a high degree of availability of the P2P-network. Some nodes should play the role of seeds. Seeds form a complete graph between themselves, and normal nodes must be connected to one or more seeds. A certain amount of seeds should always be active to ensure high availability. When a seed leaves, a new seed is selected from the leaving node's list, and the seed replacement is propagated through the whole network. At the creation of a new P2P network, the creator becomes the first seed, and new node which joins the network automatically becomes a seed. This process repeats until a desirable number of seeds is reached. Figure 16 shows an example of seeds and normal nodes.

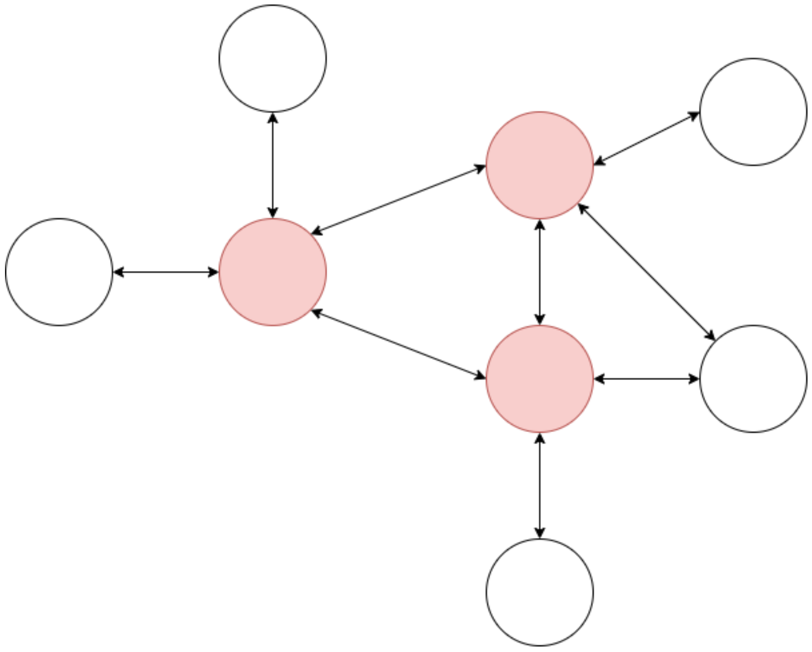


Figure 16: Seeds (red)

3.7 Process Flows Charts

This sections contains diagram to make the file-syncing systems program flow more comprehensive.

3.7.1 Node states

The states of a node/peer and the transition between the states are show below.

- off: The system is not running.
- idle: The CRDT-index should be initialized and ready,
- on: I/O-module is ready and is listening for file updates.
- online: The connection to the P2P network is established and can send and retrieve updates.

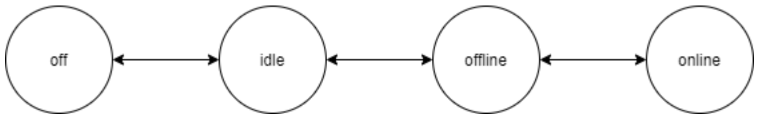


Figure 17: Node states

3.7.2 Startup/Exit Process

The system distinguishes between three ways a node can join the network. The first, is when a node creates a new network, i.e. the first node in the network. The second, is when a node first joins an existing network. The last, is when a node rejoins a network after being offline. Because the three ways have different prior configurations, i.e. what information is known before the actual method call to join the network, the process for three differs slightly. Figure 18 shows the process for the three scenarios.

New marks the start of a brand new sharing network, i.e. a node sets up a sharing token which identify this particular sharing network. In this scenario, the least amount of work needs to be done before transitioning from **off** to **online**. Before switching from **off** to **idle**, all local files in the root folder are added to the CRDT-index, or may be skipped if no such files exist. The next transition from **idle** to **on**, requires no addition work as there are only the default configuration that needs to be set up. At last, the transition from **on** to **online**, requires no additional configuration.

Joining a net network requires the joining node to ask for access and perform initial CRDT-index and file synchronization. In the transition from **off** to **idle**, if the network is private, the joining node must request access from an existing node that have the permission to add new nodes. If the network is public, then this step is skipped as public, in this case, imply that all nodes have both the permission to read and to write. Then, as the same as **New**, there may be pre-existing files that should be added to the network. From **idle** to **on**, the local CRDT-index needs to be synchronized with the ones that the local node are connected to. If the local node joins the network by connecting to multiple nodes, then the local CRDT-index must be synchronized with all the connected CRDT-indices. The transition from **on** to **online**, requires no additional configuration.

Rejoining a network implies that the node has previously been connected to the network and later disconnected. Previous configurations needs to be loaded.

3.7.3 File Update Process

Figure 19 illustrates the complete sequence of steps from when a new event is triggered. As previously mentioned in section , applying a remote update is significantly easier due to the Item preprocessing step is already done on the local end before transmitting the update.

Description of the local update step in Figure 19:

(L1) The I/O-module detects a file update and calls the corresponding method associated with the action and type.

(L2) A new **Item** object is created with the type, either file or folder, the file's path in the persistent storage, the specific action performed, the time, and who the action was performed by. Additionally a new vector clock is initialized and incremented.

(L3) Check whether if file or folder.

(L3.5) If file, then compute hash.

(L4) Check for existing **Item** object that corresponds to the same file. As discussed earlier in section 3.2, if the action is file creation then there is no need to search for an existing **Item** object. For other cases, the path is used to lookup a set of objects from the CRDT-index, and then iterated through to find the latest object that do not have a **Tomb** object.

(L4.5) There is an existing object, and the id and vector clock from the existing object are copied over to the new object and the vector clock is incremented.

(L5) The preprocessed **Item** object are sent to the CRDT-index to be applied.

(L6) Broadcast the update to connected peers.

(L7) Finished.

Description of the remote update step in Figure 19:

(R1) Incoming update is detected.

(R2) The incoming **Item** object are sent to the CRDT-index to be applied. If the object sent to the CRDT-index is discarded, i.e. the object's vector clock is older than the existing object in the CRDT-index, then no further action is needed.

(R2.1) Apply necessary I/O-operations which correspond to the **Item** object. Such as write, delete, rename, or move.

(R2.2) Broadcast the incoming update as the network may no be fully connected.

(R3) Finished.

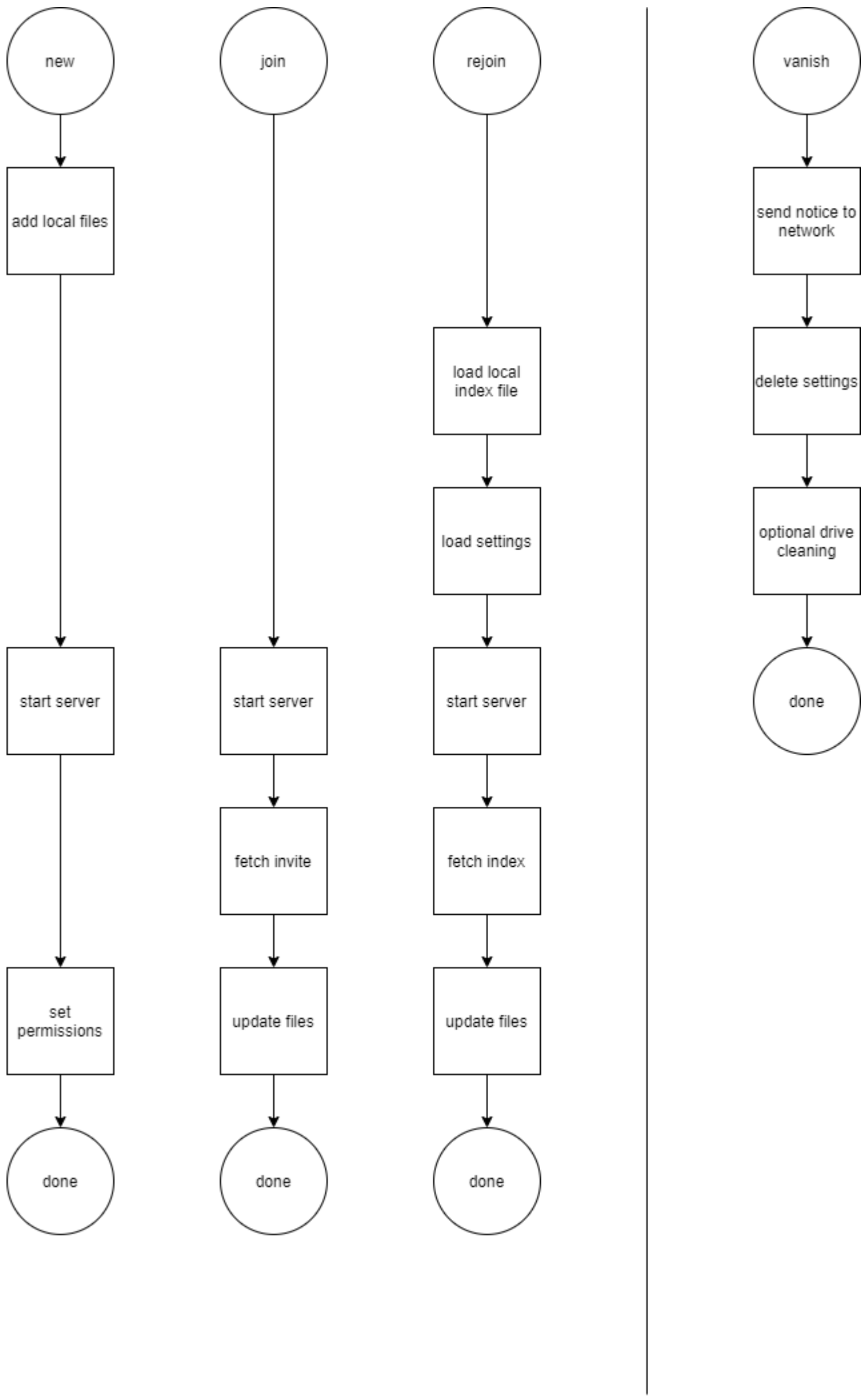


Figure 18: Startup procedure for each scenario.

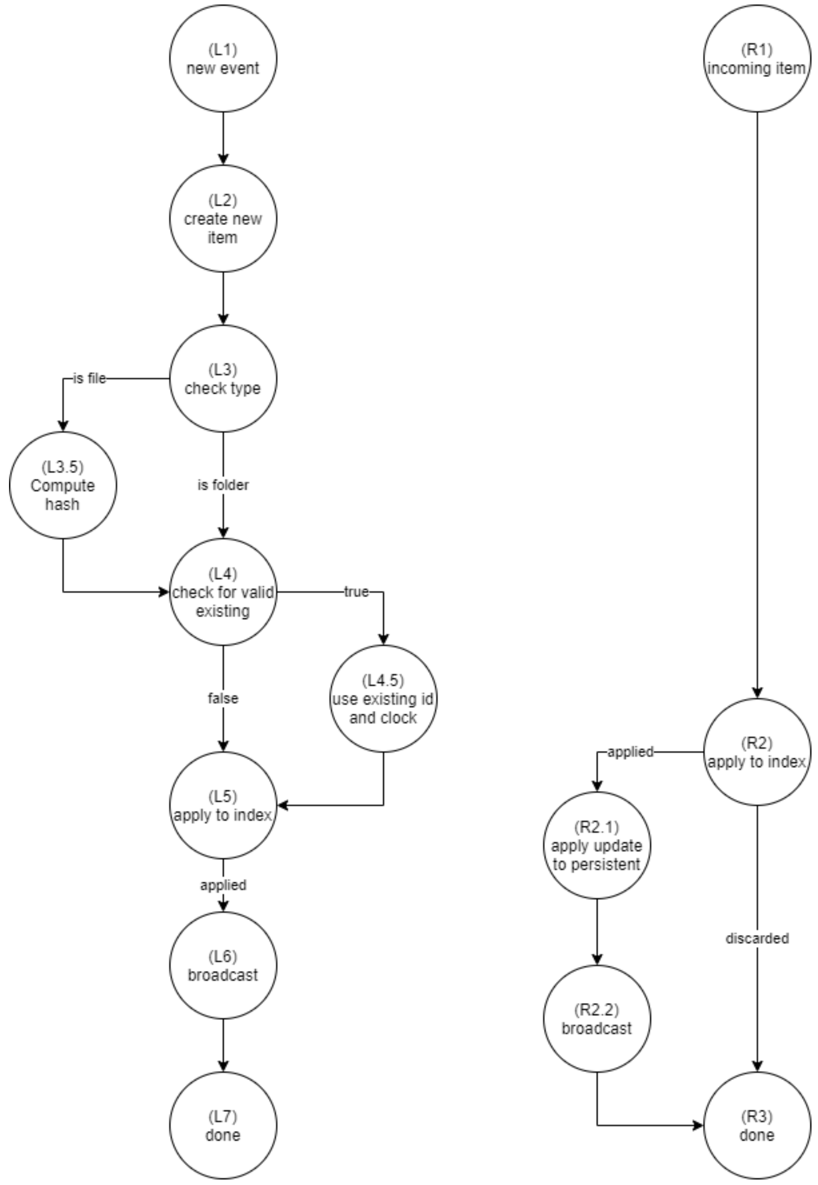


Figure 19: Processing steps for local updates (left) and remote updates (right)

4 Implementation

This chapter goes through the implemented architecture, shown in Figure 20, and discusses the various choices made during the development process.

The latest version is located at:
https://www.github.com/waffelroffel/Magellan

The state of implementation at the end of the project period:
https://github.com/waffelroffel/Magellan/commit/a4ea371ad83f48ad01dd1c35afcd6f8bca63b158

4.1 Choice of Language and Libraries

For the implementation of the file-syncing system, TypeScript was chosen. This file-syncing system is designed to run on most platform, and therefore required a cross-platform programming language. The reason languages like Python, Java, C#, C++/C, or any others were not considered is because of the author's proficiency in TypeScript is higher. Additionally, Typescript can be used on another platform, the browsers. This factor was considered because of ever increasing of web-based applications. Browsers also recently added access to the native file system, e.g. the Chrome browser added the File System Access API in version 86[6]. The system will be developed in the runtime environment Node.js[7].

- chokidar[8]: A highly customizable wrapper of the built-in file update watcher, with better event types and easy nested folder support.
- fastify[9]: Easy setup of a HTTP server.
- node-fetch[10]: A HTTP client for the node runtime environment.
- uuid[11]: Generate globally unique identifiers.

4.2 Item object

Figure 21 shows the interface of the Item object in TypeScript. In addition to the values mentioned in section 3.2, a new value **hash** is introduced. This value is used for multiple checks. As an example, when a remote peer requests

[6]https://developers.google.com/web/updates/2020/10/nic86
[7]https://nodejs.org/
[8]https://github.com/paulmillr/chokidar
[9]https://github.com/fastify/fastify
[10]https://github.com/node-fetch/node-fetch
[11]https://github.com/uuidjs/uuid

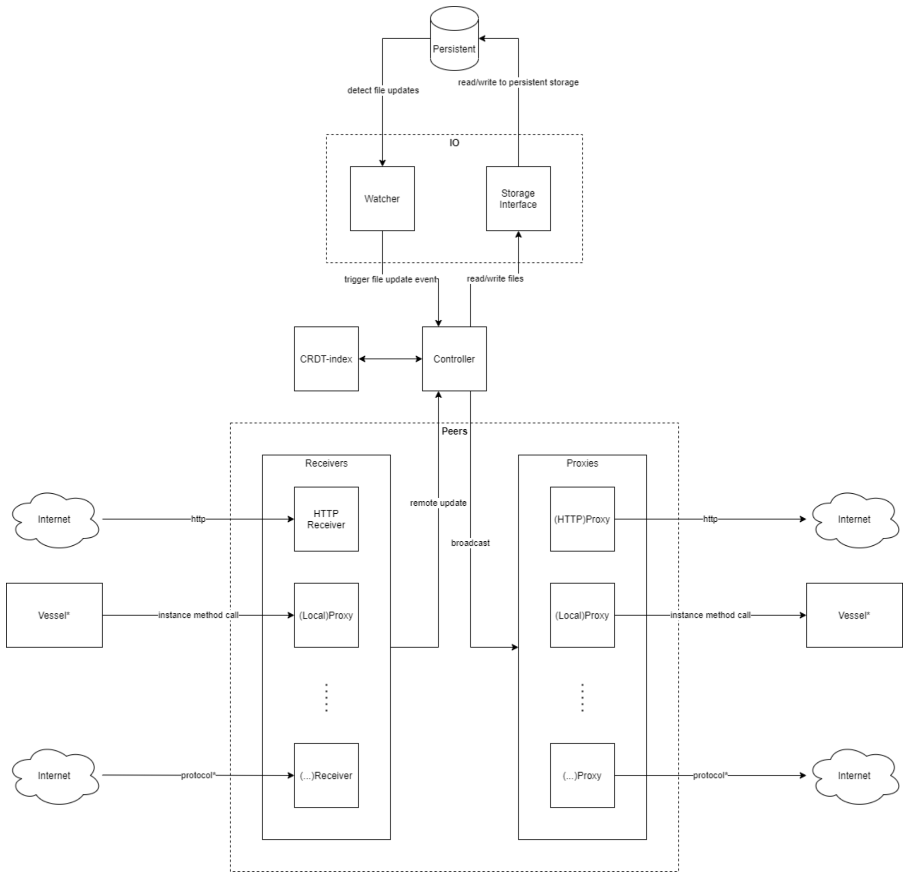


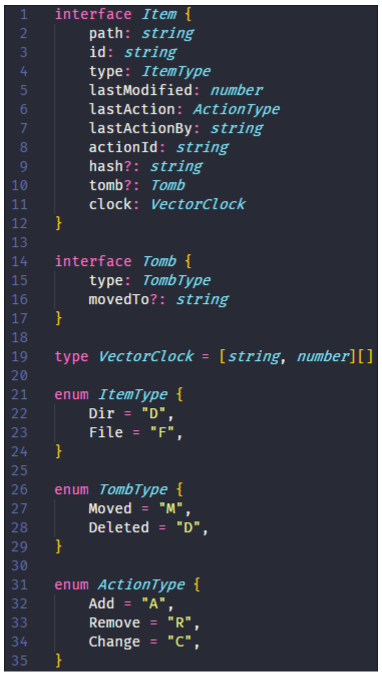
Figure 20: System Architecture

a file by the Item object, the local file's hash is computed and compared against the hash in the Item object. **ItemType**, **ActionType**, and **TombType** are enumerated types, which are data types that only contains a predefined set of values.

4.3 Controller: Vessel

Implementation: Vessel.ts

The controller component of the system is the Vessel class, which can be found in the aforementioned link, is the brain of the system. To simplify the use of the system, all interactions between the human operator or an external program have to pass through this component. The Vessel therefore provides an API, containing all the necessary functionality to access or modify the remaining components of the system. The Vessel provides five functions corresponding to



```
interface Item {
    path: string
    id: string
    type: ItemType
    lastModified: number
    lastAction: ActionType
    lastActionBy: string
    actionId: string
    hash?: string
    tomb?: Tomb
    clock: VectorClock
}

interface Tomb {
    type: TombType
    movedTo?: string
}

type VectorClock = [string, number][]

enum ItemType {
    Dir = "D",
    File = "F",
}

enum TombType {
    Moved = "M",
    Deleted = "D",
}

enum ActionType {
    Add = "A",
    Remove = "R",
    Change = "C",
}
```

Figure 21: Item Interface

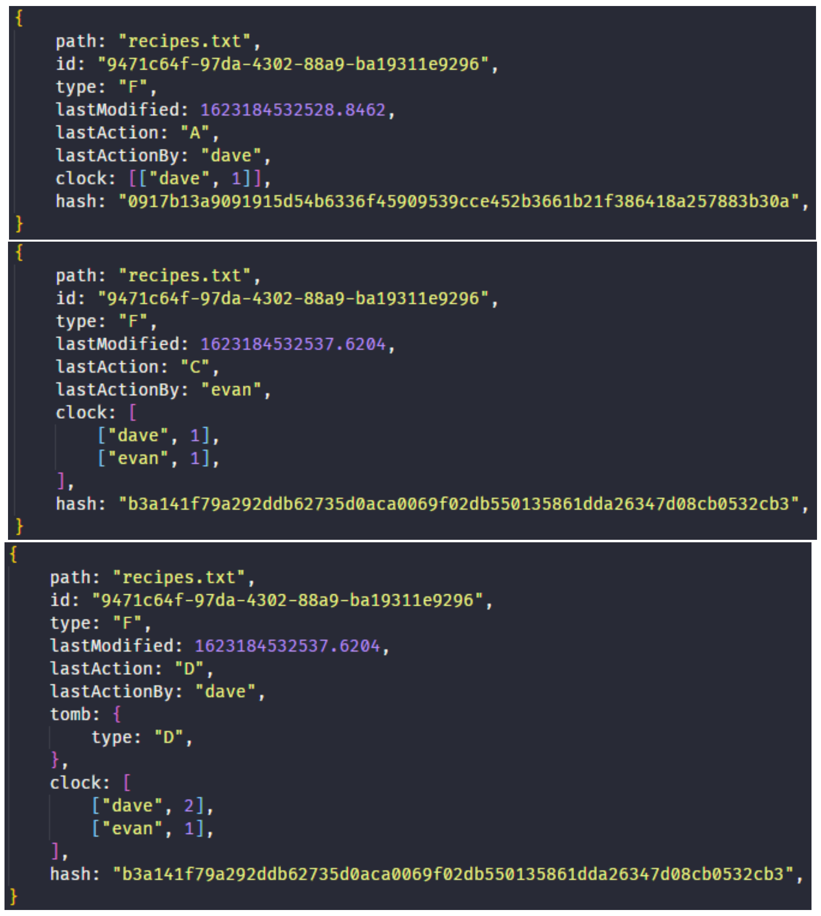
the three event types described in section 3.4.

Specific file updates implemented in Vessel:

- File added
- File removed
- File modified
- Folder added
- Folder removed

These five functions are used by the watcher component to pass the relative filepath to the Vessel, or can be used by other developers to create their own system.

It has an unique network identifier(NID) which correspond to how the peer can be reached in the network. The NID contains a set of values, the first denotes the protocol, and the subsequent values are the values the protocol's identification values. In this case the NID is an HTTP server identifier, the first value is then ”http” and the subsequent values are the IP address and the port number which the peer's HTTP server operates on. An example of a NID can



```
{
    path: "recipes.txt",
    id: "9471c64f-97da-4302-88a9-ba19311e9296",
    type: "F",
    lastModified: 1623184532528.8462,
    lastAction: "A",
    lastActionBy: "dave",
    clock: [["dave", 1]],
    hash: "0917b13a9091915d54b6336f45909539cce452b3661b21f386418a257883b30a",
}
```

```
{
    path: "recipes.txt",
    id: "9471c64f-97da-4302-88a9-ba19311e9296",
    type: "F",
    lastModified: 1623184532537.6204,
    lastAction: "C",
    lastActionBy: "evan",
    clock: [
        ["dave", 1],
        ["evan", 1],
    ],
    hash: "b3a141f79a292ddb62735d0aca0069f02db550135861dda26347d08cb0532cb3",
}
```

```
{
    path: "recipes.txt",
    id: "9471c64f-97da-4302-88a9-ba19311e9296",
    type: "F",
    lastModified: 1623184532537.6204,
    lastAction: "D",
    lastActionBy: "dave",
    tomb: {
        type: "D",
    },
    clock: [
        ["dave", 2],
        ["evan", 1],
    ],
    hash: "b3a141f79a292ddb62735d0aca0069f02db550135861dda26347d08cb0532cb3",
}
```

Figure 22: Example of states of the Item object. Dave creates new file (top). Evan modifies the file (middle). Dave deletes the file (bottom).

be found in Table 3. But as mentioned in section 3.5, a peer may be reached by multiple protocols. Thus requiring each peer to have a set of NIDs, where each NID correspond to the specific protocol that the peer actually uses. The peer do not need to artificially create NID for unused protocols. At the time of this report, only HTTP have been incorporated into the system, thus the incorporation of other protocols is then added to future works.

Type	http
IP	132.120.144.46
Port	8888

Table 3: Example of an NID object

4.4 CRDT-Index: CargoList

Implementation: CargoList.ts

As mentioned in section 3.7.3, the time when an Item object is been processed in the index should be considered as a critical region. The index should then set a lock on itself immediately before the index begins processing an Item, and the lock is released when the Item has been fully processed and stored in the index.

This is implemented in the way of an process queue which temporary holds the Item objects, and an interval timer which in each cycle, checks if the lock is set and retrieves the next Item to be processed.

After an Item object has been processed, the index constructs a Resolution object, which informs any deviations from the default processing behavior. The default behavior is the clean last-write-wins, where the new Item object's vector is strictly larger than the existing one's. The full set of values in the Resolution object are given in Table 4.

before	The Item object before processing.
after	The Item object after been processed.
ro	The specific resolve mechanism performed. E.g. last-write-wins, rename, etc.
new	Flag for indicating new files. The new Item's vector clock is later than the existing one,
rename	The conflicting file shall be renamed.
overwrite	Flag for the rename resolve mechanism denoting. The conflicting file shall overwrite the existing file.

Table 4: Resolution object

Currently there are only five valid states for the Resolution object, which can be seen in Table 5. Two for the last-write-wins mechanism, and three for the rename policy. The first state "lww 1" is the default behavior, where the result is the old Item been updated with the new Item. The second state "lww 2" is where the new Item's earlier than the existing one, and thus can be ignored since it is an older Item that has already been applied. The rest of the states correspond to the new Item being concurrent with the existing Item, and have to be ordered by the last modified and user id value. The third state "rename 1", appears when the new Item is concurrent and later than the existing Item, by the "last modified" value, thus requiring the new Item to change its name/path. The last two states "rename 2" and "rename 3" occurs at the same time. This is when the new Item is concurrent and earlier than the existing Item, thus requiring the existing Item to change its name/path, and the new Item to overwrite the existing file's content. Which is why two Resolution objects are created when processing one Item.

	lww 1	lww 2	rename 1	rename 2	rename 3
before	-	-	Item	Item	Item
after	Item	Item	Item*	Item*	Item
ro	lww	lww	rename	rename	rename
new	true	false	true	false	true
rename	-	-	true	true	false
overwrite	-	-	false	false	true

Table 5: The five valid states for the Resolution object. * denotes the Item object where its path value has been changed.

The code for adding new Item objects and the function called by the timer is given below.

```
1  // CargoList instance methods
2  putInQueue(item, post) {
3      this.queue.push(item, post)
4  }
5
6  processNext() {
7      if (this.queue.length === 0) return
8      if (this.busy) return
9      this.busy = true
10     const qitem = this.queue.shift()
11     const resarr = this.apply(qitem.item)
12     qitem.post?.(resarr)
13     this.busy = false
14 }
```

Explanation:

(1) The method takes in two parameters. The first is the Item object, and the second is the post-processing actions which will be performed after the Item has been processed by the index. Such post-processing actions, are for example broadcasting the Item to other peers or I/O operations.

(2) The Item object and the post function are added to the internal queue as a single entry.

(7) Quit if no Item in queue to process.

(8) Quit if the index is processing another Item.

(9) Set the lock on the index.

(10) Get and delete the first Item in the internal queue

(11) Process the Item object and get the processing result.

(12) Call the post-processing function, if any, with the result.

(13) Release the lock on the index.

One particular observation is that, only Item objects with the same path will cause unexpected behaviors. As there are no shared resources between concurrent Item objects with different paths. Therefore, the global lock on the index can be modified to individual locks on the specific paths which are been accessed. Allowing more Item objects to flow through the index.

4.5 IO

Implementation: Vessel.ts#L265

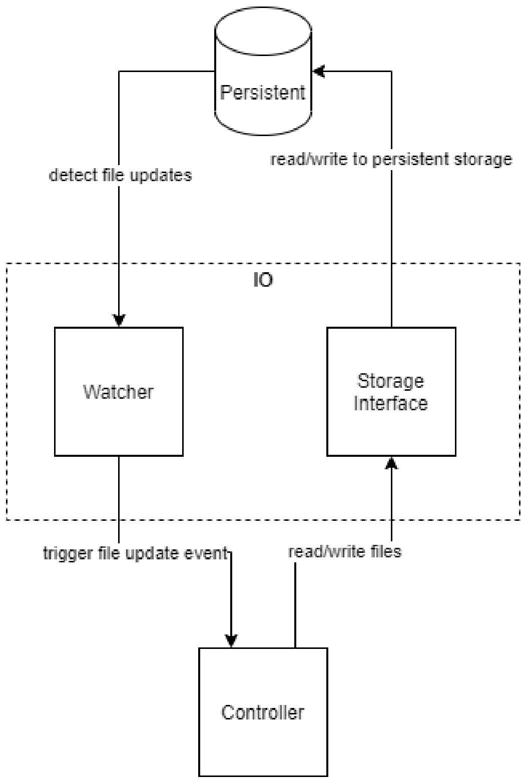


Figure 23: Close up view of the I/O part of the system

4.5.1 Watcher

The previously described responsibility for file update detection and handling I/O operations are split into two components. The Watcher component is the input of the system, detecting each file update event specified in section 3.4 and calling the corresponding Vessel method (section 4.3) are called. The watcher component listen to every file under the designated root folder, with exceptions of system configuration files, such as the configuration of the system and the list of known peers.

The implementation of the component is mostly accomplished by the Chokidar library, where the three basic file events are incorporated. One particular annoyance with the Chokidar library is the lack of advanced file events, such as file renaming and file moving. Both are represented by a **delete** event followed by a **create** event. Thus needs an additional mechanism to detect the advanced file events. One example to achieve the two advanced file events is to temporary keep the deleted Item in a list such that when the following create event is triggered and the file's hash matches the Item in the list, the rename/move event can be incorporated.

4.5.2 ABCStorage

Implementation: Storages

On the side, the way the system access or modify the local file system, or the persistent storage system of choice, is through the ABCStorage, an interface comprising the full the set of methods that are required for the system. The set of methods are given below, all methods takes requires an Item as the parameter. With the exception of the "Move" operation, which takes two parameters, the Item with the current path, and the Item with the destination path. Currently, only the local file system, LocalStorage (see Figure 24), are implemented. Access to other storage system, e.g. databases, can be accomplished by implementing the specified set of methods in ABCStorage.

- **Exists**: Check if the file exist locally.
- **Lastmodified**: Find the time for when the last file update occurred.
- **Computehash**: Calculate the hash of file with a predesignated hash function. Is not applicable to folders.
- **ApplyFileIO**: Perform write or delete on the file.
- **ApplyFolderIO**: Create or delete the folder.
- **GetData**: Read the whole file into memory.
- **CreateRS**: Create a readable stream of the file. For large files.
- **Move**: Move a file from one path to another. A combination of both rename and move, as both is a change in the path.

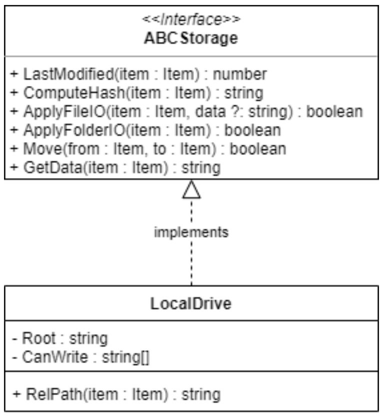

<<Interface>>
ABCStorage
+ LastModified(item : Item) : number
+ ComputeHash(item : Item) : string
+ ApplyFileIO(item : Item, data ?: string) : boolean
+ ApplyFolderIO(item : Item) : boolean
+ Move(from : Item, to : Item) : boolean
+ GetData(item : Item) : string
implements
LocalDrive
- Root : string
- CanWrite : string[]
+ RelPath(item : Item) : string

Figure 24: Relation of ABCStorage(interface) and LocalDrive(class)

4.5.3 Files Caching

As the I/O operations requires a significantly larger time to execute, a temporary file cache can be used to reduce the amount I/O operations. This caching scheme stores the file content when a file has been accessed, and updates whenever the Watcher detects a change in file content or an update from another peer. The cache of a file is then removed when the file event for file deletion is triggered. This way the concept of "temporal locality", an accessed file is more likely to be accessed again, is utilized. Each entry in the cache has a timestamp, and after a certain duration the entry will be removed. "TempDeleted" is a history of deleted files, and is used to make rename/move updates easier. While the rename/move is not explicitly implemented, the file is first deleted then refetched from another peer. The "TempDeleted" history, then skips the refetch part and used the local file instead. Temporary deleted files are stored in the "TempDeleted" folder. Figure 25 shows the types for each datastructure.

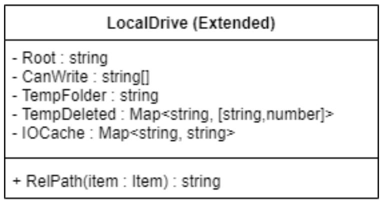

LocalDrive (Extended)
- Root : string
- CanWrite : string[]
- TempFolder : string
- TempDeleted : Map<string, [string,number]>
- IOCache : Map<string, string>
+ RelPath(item : Item) : string

Figure 25: LocalStorage with caching.

4.6 Peers

This module consist of sending(Proxy)/receiving(Receiver) messages to/from other peers. Due to time limitations, only the http protocol is incorporated.

Figure 26 shows a simplified version the of the peer module.

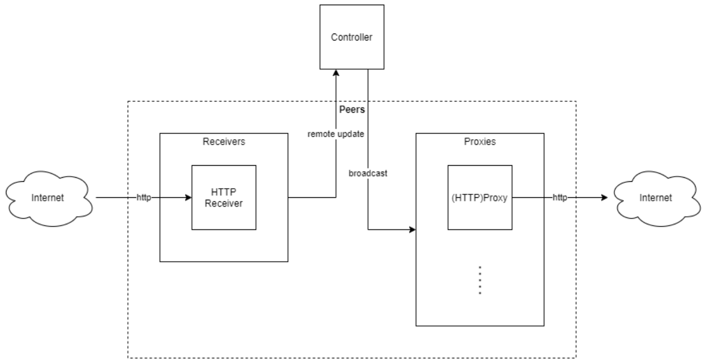


Figure 26: Peers module with only http

4.6.1 Proxy

Implementation: Proxies

The communication between peers is split into two components, one responsible for sending messages, and the other for receiving messages. By having two separate components to handle the communication, the overall structure of the components can be simplifies. On the sending side, each individual remote peer are represented as a Proxy object. A Proxy is the local representation of the remote peer, such that all the remote procedure calls are hidden as local methods defined in the Proxy interface. The default protocol used for all the communications is HTTP, and the implementation(HTTPProxy) hides the complexity of the HTTP-calls behind its methods, hiding the "remote" part of the procedure calls. All protocols must implements the base Proxy interface, such that broadcasting to all known peers is simplified to iterating through all Proxy instances.

Methods defined in the Proxy interface:

- Send: Transmit Item and file to peer.
- FetchItem: Request file from peer.
- FetchIndex: Fetch index from peer.
- GetInvite: Request access to the network.
- AddPeer: Add itself to the peer's list.
- ReqPerm: Request a specific permission.
- GrantPerm: Grant a specific permission to peer.

- CheckIndexVer: Check index version.

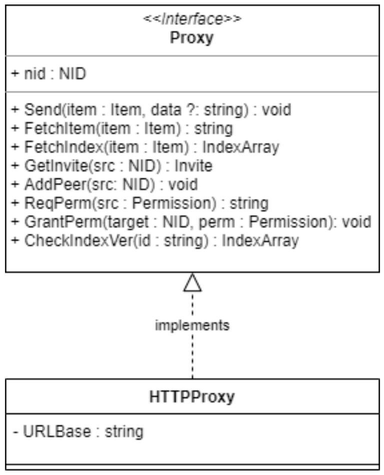

<<Interface>>
Proxy
+ nid : NID
+ Send(item : Item, data ?: string) : void
+ FetchItem(item : Item) : string
+ FetchIndex(item : Item) : IndexArray
+ GetInvite(src : NID) : Invite
+ AddPeer(src: NID) : void
+ ReqPerm(src : Permission) : string
+ GrantPerm(target : NID, perm : Permission): void
+ CheckIndexVer(id : string) : IndexArray
implements
HTTPProxy
- URLBase : string

Figure 27: HTTPProxy implements Proxy.

4.6.2 Receiver

Implementation: VesselServer.ts

Splitting the message sending and message receiving into its own components is advantageous because protocols have different ways to receive messages. For high-level protocols like HTTP, all incoming messages are funneled through a single destination, i.e. a single HTTP-server for all the peers. While low-level protocols like the Transmission Control Protocol, requires manual management of the individual connections between the peers. Thus splitting the sending and receiving part simplifies the implementation of each protocol, and puts no restriction on how each protocol should communicate. Currently, only the HTTP protocol is implemented in the system.

4.7 Extra: Permissions

Implementation: Permissions.ts

Some use-cases of the system, require a dynamic permission rights system, where an admin can grant or revoke certain permissions such as read or write.

There are two default permission states for the two scenarios described in section 2.1.1. In "All2All", every peer is immediately given the permission to write and read files. For "One2All", it starts with one admin that has permission to both write and read files, while other peers can only read files. The admin can then grant the permission to write to files, to whom the admin sees trustworthy. One observation is that the system deals with files stored on the local device, thus making the write and read permissions harder to restrict, in case of permissions revoked. Instead. the revoking of write or read permissions result in skipping local file events or incoming remote updates. No write permissions stops all file events triggered by the Watcher, and no read permission stops all remote updates received by the Receivers.

5 Testing and Evaluation

This chapter evaluates the intended behavior of the implemented system by executing specific tests that simulate the different conflict situations from A. Additionally, the overall code quality will be evaluated, and the implemented system will be compared to existing systems.

5.1 Setup

First of all, the specific version of both the runtime environment and the dependencies used are listed. The full dependency list can be found in the package.json file.

- Runtime environment:
  - Node.js version 14.15.0
- Dependencies:
  - chokidar: version 3.5.1
  - fastify: version 3.14.0
  - node-fetch: version 2.6.1
  - uuid: version 8.3.2

5.2 Test Cases

Four test sets was made in the development process. Two of these are unit tests for the different components. One set tests the system where the local file update, the Item processing, and the broadcasting happens in distinct separate phases, i.e. no file updates when Items are being processed. The last set tests the system for arbitrary file updates to assess the overall performance of the system. This section will look at the test sets for distinct phases and arbitrary file updates.

5.2.1 Distinct Phases

The implementation of the test set can be found in the tests/Conflicts.test.ts file.

This set of tests are responsible for verifying the resolve policy discussed in section 3.3.1 and are automatically executed. The first two tests check for the correct use of the vector clock, where the first checks if the logic non-concurrent updates are applied correctly by overwriting the existing file. The second test checks if concurrent files are ordered deterministic across the peers. The third test checks if the base updates in the composite updates such at rename or move are applied correctly regardless of the order which the base updates are broadcast. The separated phases are done by performing file updates while offline, and then reconnect for the updates to be broadcast to other peers. This ensures that only the conflict resolve mechanism are tested.

Test #	Description	Intended behavior
1	Add a line of text to an existing file.	The modification is propagated to the correct file on the other peers.
2	On two peers, create a file with the same name.	One of the files are renamed according to the specified resolve policy.
3	Test composite file updates such as rename/move.	The composite updates are propagated correctly regardless of the order of the base updates.

Table 6: Tests from the "Distinct Phases" set.

Due to the the advanced file updates, file renaming and moving, not being implemented. Both updates correspond to a remove followed by add. Therefore, conflicts which involve one or both of the updates, will result in duplicate files. This includes the folder cycle conflict.

5.2.2 Arbitrary Updates

The implementation of the test set can be found in the tests/arbitrary.ts file.

This set consist of a test where an arbitrary amount of file updates are performed with a stochastic interval. These updates is a random selection between three actions, creating a new file, modify an existing file, and deleting an existing file. The random selection is more weighted towards modifying existing files as it is the most likely action that a peer may perform. After file modification, the test assumes that the peer is more likely to create a new file than to delete an existing file. The time between each action is a stochastic value, and is to separate the time when each peer performs a file update. A lower bound to the stochastic value is set to simulate a realistic scenario. This test is manually executed because of the latency of I/O operation.

Due to the latency of I/O operations, a minimum of 2 seconds a set as the lower bound between each update. This lower bound is artificially high as all the peers are executed on the same process. Thus is not a limiting under real use since a peer will have a whole process designated to it, and the fact that a peer performing multiple file updates on the same file is highly unlikely.

Test #	Description	Intended behavior
4	Set up a network of 3 peers, and assign each peer an arbitrary amount of arbitrary file updates to perform.	The files and indices of all peers should converge regardless of the sequence of local updates and remote updates.

Table 7: Test from the "Arbitrary Updates" set.

5.3 Memory Usage

As mentioned in section 3.2.1, the main concern with CRDTs is the inability to remove tombstones. Therefore, an analysis of the memory footprint needs to be made to determine whether an eviction mechanism needs to be implemented to avoid extensive memory footprint. The footprint will only include the index, since it is the only component that is strictly increasing. Before even the beginning of the calculation, one can already see that the overall memory footprint will not be of significant size. Mainly due to the index only containing the metadata of the files and not the file content itself. Needless, it is still rewarding to do the estimation due to the implication of nested CRDT mentioned in section 3.3.3. Where each index entry can have its own CRDT to enable content-based synchronization, and the size of the nested CRDT is usually larger than the file itself.

A rough estimate can be calculated by analyzing the primitive values in the Item object. The size in bytes for each primitive will be calculated by the ECMA-262[16] standard. For simplicity, special primitives such as *undefined* will be ignored.

- **String**: length $\times$ 2 bytes
- **Number**: 8 bytes

The process to determining the memory foot will first start by determining the average Item objects in the index for each use-case. Then, determine an average size of each field, including nested objects. Again for simplicity, the overhead of objects and list will be ignored since the main concern is the amount of Item objects.

Key	Size	Description
Path	$25 \times 2 = 52$ bytes	Assume average file path of 25 characters
Id	$36 \times 2 = 72$ bytes	36 character id from uuid library.
Type	$1 \times 2 = 2$ bytes	Single character.
LastModified	8 bytes	Number
LastAction	$1 \times 2 = 2$ bytes	Single character.
LastActionBy	$36 \times 2 = 72$ bytes	36 character id from uuid library.
Hash	$64 \times 2 = 128$ bytes	64 character from build-in crypto library.
TombType	$1 \times 2 = 2$ bytes	Single character.
MovedTo	$36 \times 2 = 72$ bytes	Assume average file path of 25 characters
Clock	$36 \times 2 = 72$ bytes	36 character id from uuid library.
	8 bytes	Number
	n'	Vector clock entries

Table 8: Breakdown of individual sizes values in Item objects

$$
\begin{aligned}
size_{Item} =& Path + Id + Type + LastModified + LastAction + & (1)\\
& LastActionBy + Hash + TombType + MovedTo + Clock & (2)\\
=& 52 + 72 + 2 + 8 + 2 + 72 + 128 + 2 + 72 + (72 + 8) \times n' & (3)\\
=& 410 + 80n' & (4)\\
size_{total} =& size_{Item} \times n_{items} & (5)\\
=& n_{items}(410 + 80n') & (6)\\
n_{items} =& \text{\# of items in index} & (7)\\
n' =& \text{average \# of users modifying a file} & (8)
\end{aligned}
$$

With Equation 6, an estimate for each scenario can be calculated. Starting with **Team collaboration** from section 2.1.1.

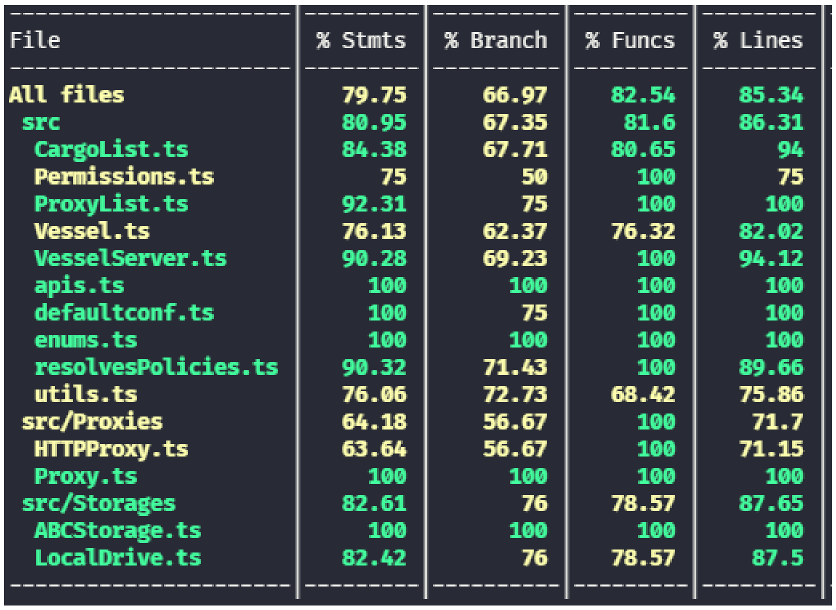
Use-case	n	n'	n_{items}	Size (mB)
Team - small	10	10	30	0.06
Team - large	100	20	1000	2.01
Sharing	-	1	10000	4,90

Table 9: Estimates of memory footprint in different use-cases

Acknowledgments The estimate of the total index size in bytes ignored many aspects of the index, e.g. overhead, but to provide a comprehensive estimate, these aspects had to be ignored. The choice of values for n, n', and n_{items} was not based on any empirical observations, but solely on subjective assumptions. A future analysis with empirical data is highly encouraged, but should include the use of nested CRDTs for the result to be of any significance.

5.4 Coverage Criteria

Apart from evaluating the functionality of the system, the code itself should be evaluated. A common method is to examine the code coverage. Code coverage is a measure of how many of the statements, branches, functions, and lines were covered by all the tests. Usually in a percentage. The measure also shows all the lines that was not tested. To have a high code coverage means that there is a lower probability of a bug occurring since most of the ways the code could run are covered in the tests. The most desirable is to have the code coverage as high as possible, testing the last percentage of the code is much harder due to parts of the code being hard to reach, i.e. hard to artificially create a situation where the code gets executed. Examples of this could be a multiple level fail-safe, where a certain condition is checked in multiple parts of the code, but only the first check could trigger the fail-safe. For this implementation, the aim is to reach a minimum code coverage of 70%. Figure 28 shows that the percentage of code branches did not reach the goal, but with a average coverage of 70%+, its considered good enough.



```
----------------------|---------|----------|---------|---------|
File                  | % Stmts | % Branch | % Funcs | % Lines |
----------------------|---------|----------|---------|---------|
All files             |   79.75 |    66.97 |   82.54 |   85.34 |
 src                  |   80.95 |    67.35 |    81.6 |   86.31 |
  CargoList.ts        |   84.38 |    67.71 |   80.65 |      94 |
  Permissions.ts      |      75 |       50 |     100 |      75 |
  ProxyList.ts        |   92.31 |       75 |     100 |     100 |
  Vessel.ts           |   76.13 |    62.37 |   76.32 |   82.02 |
  VesselServer.ts     |   90.28 |    69.23 |     100 |   94.12 |
  apis.ts             |     100 |      100 |     100 |     100 |
  defaultconf.ts      |     100 |       75 |     100 |     100 |
  enums.ts            |     100 |      100 |     100 |     100 |
  resolvesPolicies.ts |   90.32 |    71.43 |     100 |   89.66 |
  utils.ts            |   76.06 |    72.73 |   68.42 |   75.86 |
 src/Proxies          |   64.18 |    56.67 |     100 |    71.7 |
  HTTPProxy.ts        |   63.64 |    56.67 |     100 |   71.15 |
  Proxy.ts            |     100 |      100 |     100 |     100 |
 src/Storages         |   82.61 |       76 |   78.57 |   87.65 |
  ABCStorage.ts       |     100 |      100 |     100 |     100 |
  LocalDrive.ts       |   82.42 |       76 |   78.57 |    87.5 |
----------------------|---------|----------|---------|---------|
```

Figure 28: Code Coverage

5.5 Resolve Policies of Existing Services

This section will look at the conflict resolve policies in other file-syncing systems, specifically Google Drive.

As previously mentioned, **Google Drive** was chosen as the comparison against our implementation since it is one of the most well known applications for file-syncing. Three interesting cases are presented below, and the remaining cases can be found in Appendix B.

Add file A || Add file A: For concurrent insertions of files with the same name, but different content, were added into the system. One through the local folder with *Google Backup and Sync*, and the other though the web interface. The result was that the local file, diverged from the remote file. The content depended on which interface were used to access the file. It was only after one of the replicas were manually modified afterwards that both files converged.

Rename file A to B || Rename file A to C: Two users concurrently rename the file. The result was two diverged file names. For the user who renamed the file to B got C instead, and the user who renamed to file to C got B. Both was still the file internally in the system as deleting the file on one side removed both.

Remove/Rename/Change file A || Replace file A: When one user renames, removes, or changes a file, and concurrently another user replaces the file, resulted in a system error which did not get resolved. An interesting observation from the **Remove file A || Replace file A** case was that it showed a different error message than the others. It said that the synchronization failed due to pending updates that have yet to be synchronized, and thus hints that under the hood a causal order mechanism is used.

The *Folder cycle* conflict is not tested, but it is known that Google Drive encounters a system error[17].

6 Conclusion

This paper has presented the current state of file-syncing services, where most of them choose to use a cloud-based network architecture. The viability of a CRDT-based P2P file-syncing has been discusses. It is shown that with a carefully designed CRDT index to keep track files, the system can resolve any concurrent file updates in a deterministic manner, with no additional coordination with the other peers. This paper choose to design a hash-table CRDT due to the similar use of hash-tables as cache in database systems.

This paper has presented a list of possible file conflicts that can occur in daily use, and proposed several reasonable resolve policies to each individual conflict. It is shown that the "best" resolve policy of each conflicts is determined by the context of the file updates, i.e. what the user intended to achieve with the file update. For the proof-of-concept, the resolve policy is designed to keep conflicting file updates as separate files. This ensures that no information is lost. For conflicts that can not be resolve by vector clocks, one of the updates are kept as a separate file. Unfortunately, only the basic file updates were accounted for. Other updates such at file renaming or moving required an external mechanism to identify. Even without the advanced updates, the amount of conflicts is still huge, and to ensure the resolve policies worked through the whole development process, several tests were created to verify the correctness of the system.

An extension of the CRDT-index has been proposed to further streamline the conflict resolves to merge concurrent file updates into the same file. Much like how text collaboration services like Google Docs functions. This is a hard feature to implement correctly as this essentially stores the whole file into CRDT-index, which will drastically increase the memory footprint. A further note in chapter 7.

7 Future Work

Stream-based file transfers: Currently the whole file are read into memory before sending the file to other peers. The same when files are received, before writing to persistent storage. This would be ineffective with large files, thus should use streams file transmissions. The current implementation did not use streams, due to this being a proof-of-concept, and the asynchronous nature of streams caused an additional source of bugs.

Security: As for now, the updates and files are send over the internet in plaintext. Which is not acceptable in today's standards. An encryption scheme should be implemented, in the form of end-to-end encryption for all communication. In addition to forcing all communication over HTTPS.

Alternative CRDT-based index As mentioned in section 3.3.3 and 5.3, the inclusion of nested CRDTs for content-based synchronization may drastically increase the memory footprint. At that point, an eviction scheme will be needed to temporary store unused Item objects (with the nested CRDT) to the persistent storage. A possible scheme is to is to introduce a bloom filter to manage infrequently accessed files, but has the cost of periodically reconstructing the bloom filter. There is an alternative structure that would be interesting to mention, a tree-structured CRDT. Which could be more appropriate, as the file system is similarly structured like a tree. And an similar eviction strategy like the one in Leanstore[18] could then be adopted.

References

[1] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *SIGACT News*, vol. 33, no. 2, p. 51–59, Jun. 2002. [Online]. Available: https://doi.org/10.1145/564585.564601

[2] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "Conflict-free replicated data types," in *Stabilization, Safety, and Security of Distributed Systems*, X. Défago, F. Petit, and V. Villain, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 386–400.

[3] V. B. F. Gomes, M. Kleppmann, D. P. Mulligan, and A. R. Beresford, "Verifying strong eventual consistency in distributed systems," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, p. 1–28, Oct 2017. [Online]. Available: http://dx.doi.org/10.1145/3133933

[4] C. A. Ellis and S. J. Gibbs, "Concurrency control in groupware systems," *ACM SIGMOD Record*, 1989.

[5] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "A comprehensive study of Convergent and Commutative Replicated Data Types," Inria – Centre Paris-Rocquencourt ; INRIA, Research Report RR-7506, Jan. 2011. [Online]. Available: https://hal.inria.fr/inria-00555588

[6] M. Kleppmann and A. R. Beresford, "A conflict-free replicated JSON datatype," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 10, pp. 2733–2746, Apr. 2017.

[7] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 6, p. 205–220, Oct. 2007. [Online]. Available: https://doi.org/10.1145/1323293.1294281

[8] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, p. 558–565, Jul. 1978. [Online]. Available: https://doi.org/10.1145/359545.359563

[9] C. J. Fidge, "Timestamps in message-passing systems that preserve the partial ordering,"," in *Proc. 11th Austral. Comput. Sci. Conf. (ACSC '88)*, 1988, pp. 56–66.

[10] M. Ahmed-Nacer, S. Martin, and P. Urso, "File system on CRDT," *CoRR*, vol. abs/1207.5990, 2012. [Online]. Available: http://arxiv.org/abs/1207.5990

[11] R. G. Guy, J. S. Heidemann, W. Mak, T. W. Page, Jr., G. J. Popek, and D. Rothmeier, "Implementation of the Ficus replicated file system," in *USENIX Conference Proceedings*. Anaheim, CA: USENIX, Jun. 1990, pp. 63–71. [Online]. Available: http://www.isi.edu/%7ejohnh/PAPERS/Guy90b.html

[12] V. Martins, E. Pacitti, and P. Valduriez, "Survey of data replication in P2P systems," INRIA, Research Report RR-6083, 2006. [Online]. Available: https://hal.inria.fr/inria-00122282

[13] Z. Mehdi and H. Ragab-Hassen, "File synchronization systems survey," 2016.

[14] H.-G. Roh, M. Jeon, J.-S. Kim, and J. Lee, "Replicated abstract data types: Building blocks for collaborative applications," *Journal of Parallel and Distributed Computing*, vol. 71, no. 3, pp. 354 – 368, 2011. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731510002716

[15] G. Oster, P. Urso, P. Molli, and A. Imine, "Data consistency for p2p collaborative editing," in *Proceedings of the 2006 20th Anniversary Conference on Computer Supported Cooperative Work*, ser. CSCW '06. New York, NY, USA: Association for Computing Machinery, 2006, p. 259–268. [Online]. Available: https://doi.org/10.1145/1180875.1180916

[16] ECMA International, *Standard ECMA-262 - ECMAScript Language Specification*, 5th ed., June 2011. [Online]. Available: http://www.ecma-international.org/publications/standards/Ecma-262.htm

[17] M. Kleppmann, D. P. Mulligan, V. B. F. Gomes, and A. R. Beresford, "A highly-available move operation for replicated trees and distributed filesystems," 2020. [Online]. Available: https://martin.kleppmann.com/papers/move-op.pdf

[18] V. Leis, M. Haubenschild, A. Kemper, and T. Neumann, "Leanstore: In-memory data management beyond main memory," in *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, 2018, pp. 185–196.

[19] M. Letia, N. Preguiça, and M. Shapiro, "Crdts: Consistency without concurrency control," 2009.

A List of conflicts

A.1 Intra-folder conflicts

Operations such as "rename", "move", and "replace" can be decomposed into a "remove" followed by "add".

Add file A || Add file A: Users concurrently create a file with the same name.

- Overwrite: Last write wins solution where the resolved file is determined by the creation timestamp of each file.
- Keep all: Keep all files and deterministic rename. First file gets the original name while the others get "userid" or "(n)" appended to the name.

Remove file A || Add file A: Users concurrently create and remove a file with the same file.

- Overwrite: New operation overwrites the tombstone based on timestamp. As mentioned in subsubsection 3.2.1, tombstones must be preserved for correctness, and overwriting the tombstone in this way may lead to unexpected behaviors.
- Extend: In case a new file gets created with the same name. A tactic similar to extended hashing can be used to preserve both the old tombstone and the new file.

Rename file A to B || Add file A: An user renames a file from A to B, and another user creates a new file A.

- Follow: Go to the new location of the file and apply concurrent add resolve logic. Depending on the intention and the context of the add file operation, one user could be not synchronized with the others, and therefore the file is still at the old location. The user then plans to replace the old file with the new file. In this case treated as **Add file B || Add file B**.
- Extend: On other circumstances new file could be unrelated to the old file, and should appear at the former location.

Rename file A to B || Add file B: An user renames a file from A to B, and another user creates a new file B. Treat as **Add file B || Add file B**.

Change file A || Add file A: Users concurrently create and modify a file with the same file.

- Overwrite: Overwrite: Last write wins solution where the resolved file is determined by timestamps.

- Rename: There is a possibility where a user have been offline for a while, and adds a new file with name A. But on the rest of the network a file A has already been created, and modified a couple of times. In this situation, this should be treated as a duplicate.

Remove file A || Remove file A: Trivial case.

Rename file A to B || Remove file A: An user renames a file from A to B, and another user removes the file A.

- Follow: Treat as **Add file B || Remove file B**
- No follow: Treat as **Remove file A || Remove file A**

Change file A || Remove file A: Users concurrently remove and modify a file with the same file.

- Timestamp: The operation with the latest timestamp wins. In cases where the modify operations "wins", the file will reappear on users where the file has been deleted.
- Remove-priority: If the timestamp difference sufficiently small, this could be an situation where the intended outcome is to delete the file. If the difference between the timestamps are larger than a threshold, then the logic could default to *Timestamp*.

Rename file A to B || Rename file A to C: Users concurrently rename the same file to different names.

- DAG: The operation timestamp will serve as a deterministic consensus, the one with the latest timestamp determine the final location of the file. The file tombstone in its previous locations (in all replicas) must point to the final location, and the tombstones need to be added on all peers to ensure SEC.

Rename file A to C || Rename file B to C: Users concurrently rename different files to the same names. Treat as **Add file C || Add file C**.

Change file A || Rename file A to B: A new file gets deleted, but another user moved it to a new location. Treat as **Change file A || Add file A**.

Change file A || Change file A: Users concurrently modify the same file.

- Timestamp: Only the changes with the latest timestamp gets applied. If the peers are not communicating when they try to modify the same file, some peers might get annoyed that their edits suddenly disappeared, and replaced with someone else's. Here the importance of an awareness mechanism is a necessity, to quickly tell the peer that someone else is modifying the file. In cases where the system functions as a read-only archive, the frequency will be greatly reduced.

- CRDT-based files: A way to keep the concurrent changes is to implement a kind of CRDT-based mechanism for the file content. Then, in the case of text files, existing CRDT implementations can be used.[19][14][15]

Add file A $\|$ **Replace file A:** Users concurrently creates and replace the same file. Treat as **Add file A** $\|$ **Add file A**.

Remove file A $\|$ **Replace file A:** Users concurrently remove and replace the same file. Treat as **Remove file A** $\|$ **Add file A**.

Rename file A $\|$ **Replace file A:** Users concurrently rename and replace the same file. Treat as **Rename file A** $\|$ **Add file A**.

Change file A $\|$ **Replace file A:** Users concurrently modify and replace the same file. Treat as **Change file A** $\|$ **Add file A**.

Replace file A $\|$ **Replace file A:** Users concurrently replace the same file. Treat as **Add file A** $\|$ **Add file A**.

A.2 Inter-folder conflicts

Add file A || Move file A: Users concurrently create and move the same file. As "move" is functionally the same as "rename", treat as **Add file A to B || Rename file A**.

Change file A || Move file A: Users concurrently modify and move the same file. As "move" is functionally the same as "rename", treat as **Change file A to B || Rename file A**.

Remove file A || Move file A: Users concurrently remove and move the same file. As "move" is functionally the same as "rename", treat as **Change file A to B || Rename file A**.

Replace file ... to ... || Move file ... to ...: Users concurrently replace and move the same file. As "move" is functionally the same as "rename", treat as corresponding **Replace file ... to ... || Rename file ... to ...**.

Rename file ... to ... || Move file ... to ...: Users concurrently rename and move the same file. As "move" is functionally the same as "rename", treat as corresponding **Rename file ... to ... || Rename file ... to ...**.

Move file ... to ... || Move file ... to ...: Users concurrently move the same file. As "move" is functionally the same as "rename", treat as corresponding **Rename file ... to ... || Rename file ... to ...**.

Add folder A || Add folder A: Users concurrently add a folder with the same name.

- Merge: In most cases, the two folders could just be merged together as the content in both folders are of the same context. This assumes that the folder name is not a commonly used name like (misc, temp, pictures, etc.). Conflicting files inside the folders will follow **Add file A || Add file A**.
- Rename: In case of generic folder names, The same logic as *Duplicate* in **Add file A || Add file A** should be followed.
- Hybrid: Instead of either just Merge or Rename, a both methods can be used at the same time. With *Merge* prioritized over *Rename*.

Remove folder A || Add folder A: Users concurrently delete and add a folder with the same name. Analogous to **Remove file A || Add file A**.

- Overwrite: New operation overwrites the tombstone based on timestamp, but overwriting the tombstone in this way may lead to unexpected behaviors. Applies for all sub-files.

- Extend: In case a new file gets created with the same name. A tactic similar to extended hashing can be used to preserve both the old tombstone and the new file. Applies for all sub-files.

Move folder A to B || Add folder A: Users concurrently move and add a folder with the same name. Analogous to **Move file A || Add file A**, and follows the same logic.

- Follow: Go to the new location of the file and apply concurrent add resolve logic. Depending on the intention and the context of the add folder operation, one user could be not synchronized with the others, and therefore the file is still at the old location. The user then plans to replace the old folder with a new folder. In this case treated as **Add folder B || Add folder B**.
- Extend: On other circumstances the new folder could be unrelated to the old folder, and should appear at the former location.

Move folder A to B || Add folder B: Users concurrently move and add a folder with the same name. Analogous to **Move file A to B || Add file B**, and treat as **Add folder B || Add folder B**.

Rename folder A to B || Add folder A: An user renames a folder from A to B, and another user creates a new folder A. Treat as **Move folder A to B || Add folder A**.

Rename folder A to B || Add folder B: An user renames a folder from A to B, and another user creates a new folder B. Treat as **Move folder A to B || Add folder B**.

Change folder A || Add folder A: Users concurrently change the folder structure and add a folder with the same name. Analogous to **Change file A || Add file A**.

- Overwrite: Overwrite: Last write wins solution where the resolved file is determined by timestamps.
- Rename: There is a possibility where a user have been offline for a while, and adds a new folder with name A. But on the rest of the network a folder A has already been created, and changed a couple of times. In this situation, this should be treated as a duplicate.

Remove folder A || Remove folder A: Trivial case.

Move folder A to B || Remove folder A: Users concurrently move and remove a folder with the same name. Analogous to **Move file A to B || Remove file A**.

- Follow: Treat as **Add file B** || **Remove file B**
- No follow: Treat as **Remove file A** || **Remove file A**

Rename folder A to B || **Remove folder A:** Users concurrently rename and remove a folder with the same name. Treat as **Move folder A to B** || **Remove folder A**.

Change folder A || **Remove folder A:** Users concurrently change and remove a folder with the same name. Analogous to **Change file A** || **Remove file A**.

- Timestamp: The operation with the latest timestamp wins. In cases where the modify operations ”wins”, the folder will reappear on users where the folder has been deleted.
- Remove-priority: If the timestamp difference sufficiently small, this could be an situation where the intended outcome is to delete the folder. If the difference between the timestamps are larger than a threshold, then the logic could default to *Timestamp*.

Move folder ... to ... || **Move folder ... to ...:** Users concurrently move the same folder. Apply the same DAG logic as **Move file ... to ...** || **Move file ... to ...:**.

Rename folder A || **Move folder A:** Users concurrently rename and move the same folder. Treat as **Move folder ... to ...** || **Move folder ... to ...:**.

Change folder A || **Move folder ... to ...:** Users concurrently change and move the same folder. Analogous to the corresponding **Change file A** || **Move file ... to ...**.

Rename folder ... to ... || **Rename folder ... to ...:** Users concurrently rename the same folder. Treat as the corresponding **Move folder ... to ...** || **Move folder ... to ...**.

Change folder A || **Rename folder ... and ...:** Users concurrently change and rename the same folder. Treat as the corresponding **Change folder A** || **Move folder ... to ...**.

Change folder A || **Change folder A:** Users concurrently change the same folder.

- Merge: Merge and apply **Change file A** || **Change file A** to all conflicting files.
- Rename: Have the two diverged folder as separate folder.

Add folder A || **Replace folder A:** Users concurrently add and replace a folder with the same name. Treat as **Add folder A** || **Add folder A**.

Remove folder A || Replace folder A: Users concurrently remove and replace a folder with the same name. Treat as **Remove folder A || Add folder A**.

Rename folder A || Replace folder A: Users concurrently rename and replace a folder with the same name. Treat as **Rename folder A || Add folder A**.

Move folder A || Replace folder A: Users concurrently move and replace a folder with the same name. Treat as **Move folder A || Add folder A**.

Change folder A || Replace folder A: Users concurrently change and replace a folder with the same name. Treat as **Change folder A || Add folder A**.

Replace folder A || Replace folder A: Users concurrently replace and replace a folder with the same name. Treat as **Add folder A || Add folder A**.

Folder cycle: This is a special case where a peer moves folder A to be under folder B, and another peer moves folder B to be under folder A. If not treated carefully, a situation will arise where both folders are removed from the root. See Figure 14. In the best case we get a system error, which need to be recovered from, and in the worst case both folders are inaccessible (equivalent to being deleted). A resolve method involving a deterministic procedure of undos and redos exist.[17]

Folder operation || file operation: All folder operations will recursively perform a corresponding file operation, and there is no need for extra logic involving both folder and file operations.

B Conflict resolves in Google Drive

Add file A || Add file A: For concurrent insertions of files with the same name, but different content, were added into the system. One through the local folder with *Google Backup and Sync*, and the other though the web interface. The result was that the local file, diverged from the remote file. The content depended on which interface were used to access the file. It was only after one of the replicas were manually modified afterwards that both files converged.

Rename file A to B || Add file A: One user creates a new file A and quickly changed the name to B, and concurrently another user creates a different file A. The end result was that both files were preserved, the file renamed to B and the new file A. Follows the ”extend” resolve policy.

Rename file A to B || Add file B: One user creates a new file A and quickly changes the name to B, and concurrently another user creates a different file B. The end result was that both files were preserved, the file renamed from A to B kept its name, and the new file B was renamed to ”B (2)”. Follows the ”keep all” resolve policy.

Change file A || Add file A: One user creates a new file A and quickly changes the content, and concurrently another user creates a different file A. The end result was that both files were preserved, a ”A” and ”A (2)”. Follows the ”keep all” resolve policy.

Remove file A || Remove file A : Trivial case.

Rename file A to B || Remove file A: One user renames file A, and concurrently another user deletes the file. The end result was the file being deleted regardless of the time-order of the operation, but that may be a result of time-deviation due to the aforementioned synchronization issue in section 2.6.

Change file A || Remove file A: One user updates the file, and concurrently another user deletes the file. The end result was the file being deleted regardless of the time-order of the operation, but that may be a result of time-deviation due to the aforementioned synchronization issue in section 2.6.

Rename file A to B || Rename file A to C: Two users concurrently rename the file. The result was two diverged file names. For the user who renamed the file to B got C instead, and the user who renamed to file to C got B. Both was still the file internally in the system as deleting the file on one side removed both.

Change file A || Rename file A: One user updates the file, and concurrently another user renames the file. The end result was the update being applied to the renamed file, and stays true to the user’s intention.

Change file A || Change file A: Two users concurrently update the file. The latest change was applied while the other did not. Follows last-write-wins policy.

Remove/Rename/Change file A || Replace file A: When one user renames, removes, or changes a file, and concurrently another user replaces the file, resulted in a system error which did not get resolved. An interesting observation from the **Remove file A || Replace file A** case was that it showed a different error message than the others. It said that the synchronization failed due to pending updates that have yet to be synchronized, and thus hints that under the hood a causal order mechanism is used.