PEER-TO-PEER SYNCHRONIZED STREAMING FILESYSTEM

Stepan Usatiuk

Bachelor's thesis
Faculty of Information Technology
Czech Technical University in Prague
Department of Software Engineering
Study program: Informatics
Specialisation: Software Engineering 2021
Supervisor: Ing. Tomáš Vondra, Ph.D.
May 15, 2025

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	Peer-to-peer synchronized streaming filesystem
Student:	Stepan Usatiuk
Supervisor:	Ing. Tomáš Vondra, Ph.D.
Study program:	Informatics
Branch / specialization:	Software Engineering 2021
Department:	Department of Software Engineering
Validity:	until the end of summer semester 2025/2026

Instructions

Due to the increase of the amount of data and the number of devices an individual possesses, it is necessary to synchronize the data between devices so that it is always available where the user needs it and also to create backups in case some of the storage devices were to fail. The current solutions to the problem are either proprietary, require a central server, or can only synchronize whole files, which makes remote access to a large file slow as the user has to wait for it to be completely downloaded.

1. Analyze the landscape of distributed filesystems and synchronization tools.
2. Design a file system that is able to synchronize files between devices in streaming mode as they are being accessed.
3. Implement the said file system.
4. Test the solution mainly with the focus of preventing loss of data in adverse situations.

Electronically approved by Ing. Michal Valenta, Ph.D. on 18 December 2024 in Prague.

First, I would like to thank my supervisor, Ing. Tomáš Vondra, Ph.D., for his invaluable guidance and feedback on this thesis.
I would also like to extend my gratitude to my colleagues, friends, and family, who have been supporting me not only during the work on the thesis but throughout the rest of my studies as well.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I declare that I used artificial intelligence tools during the preparation and writing of this thesis. I verified the generated content. I hereby confirm that I am aware of the fact that I am fully responsible for the contents of the thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Section 2373(2) of Act No. 89/2012 Coll., the Civil Code, as amended, I hereby grant a non-exclusive authorization (licence) to utilize this thesis, including all computer programs that are part of it or attached to it and all documentation thereof (hereinafter collectively referred to as the "Work"), to any and all persons who wish to use the Work. Such persons are entitled to use the Work in any manner that does not diminish the value of the Work and for any purpose (including use for profit). This authorisation is unlimited in time, territory and quantity.

In Prague on May 15, 2025

Abstract

This thesis deals with the design, development, and testing of a peer-to-peer streaming filesystem. Analysis of existing file synchronization solution shows that none of them have these two properties: working in a peer-to-peer manner and being able to stream files. Then, possible algorithms for file synchronization are discussed, and the algorithm to be used is selected and modified to fit specific needs of the filesystem. The software architecture of the solution then is designed, focusing on modularity and extensibility, and the design is implemented. The solution is then tested, with emphasis on keeping the user's data safe under all conditions.

Keywords CRDT, filesystem, peer-to-peer, distributed systems, file synchronization, file streaming

Abstrakt

Tato bakalářská práce se zabývá návrhem, vývojem a testováním peer-to-peer streamovacího souborového systému. Analýza existujících řešení synchronizace souborů ukazuje, že žádné z nich nesplňuje tyto dvě vlastnosti: fungování peer-to-peer a schopnost streamovat soubory. Následně jsou diskutovány možné algoritmy pro synchronizaci souborů a zvolen algoritmus, který bude použit, a upraven tak, aby vyhovoval konkrétním potřebám souborového systému. Pak je navržena softwarová architektura řešení se zaměřením na modularitu a rozšiřitelnost a návrh je implementován. Řešení je poté testováno s důrazem na zajištění bezpečnosti uživatelských dat za všech podmínek.

Klíčová slova CRDT, souborový systém, peer-to-peer, distribuované systémy, synchronizace souborů, streamování souborů

Contents

List of Figures

List of Code listings

List of abbreviations

CRDT	Conflict-free Replicated Data Type
HTTP	Hypertext Transfer Protocol
ACID	Atomicity, Consistency, Isolation, and Durability
API	Application Programming Interface
MVCC	Multiversion Concurrency Control

Chapter 1

Introduction

The amount of data in our lives has been increasing rapidly over the past few years, especially for those working with media: recording high-quality videos, capturing photos in RAW format, handling large datasets and virtual machine disks.

But where should all this data be stored?

Local storage on a single device is, of course, one solution. However, it is an inconvenient one, as manual copying of data is required to access it from other devices. Additionally, to protect this data, for example, if one of the devices were lost, manual backups are necessary.

While many solutions to this problem already exist, I will demonstrate in this thesis that none of them adequately address the volume of data we now encounter. I will then propose my own solution, designed to handle large amounts of data while maintaining convenience for the end user.

1.1 Goals

The primary goal is to create an application that users can install on their devices (multiple peers connected together), that will synchronize files in a specified folder between them.

The synchronization will happen in such a way that, when a file is accessed, only the parts of the file requested by the application are downloaded, rather than the entire file.

The application should also function without the need for a centralized server, operating on a peer-to-peer basis.

This design will empower users to maintain greater control over their personal data while also ensuring better reliability and performance under poor network conditions.

Chapter 2
Analysis

In this part, I will analyze the existing solutions to the problem of synchronizing and accessing files between devices, the advantages and disadvantages of existing approaches.
I will also take a look at existing distributed file systems. While not designed for the exact use case that file synchronization solutions are, their designs can still be useful to study.
Then, I will specify my requirements for a "perfect" file synchronization tool, and analyze the use cases I will have to support from the point of its user.

2.1 Analysis of Existing Solutions

Many file synchronization solutions already exist, but none offer all of my desired features. I had found that the existing solutions could be categorized roughly into the categories below.

1. Classic file synchronization
2. On-demand file synchronization
3. File streaming
4. Decentralized file synchronization
5. Cloud storage mounting via 3rd party filesystem

2.1.1 Classic file synchronization

This is the "original" approach to file synchronization between personal devices, employed by Dropbox and others. Here, a user selects a folder, or multiple folders on their device that are synchronized with a central server. It

can be possible to further limit file synchronization to a subset of specific files or subdirectories.

Unfortunately, this can be quite cumbersome to manage.

Assuming that the user does not want to keep everything downloaded locally all the time, for example, if total volume of data being larger that what can be stored locally, accessing previously unselected files will require the user to select them for synchronization, wait until the data is downloaded, and only then the files can be accessed.

And, of course, the files should be unselected from synchronization again, once the user no longer needs them and wishes to free up disk space.

2.1.2 On-demand file synchronization

As a straightforward improvement over the aforementioned ”naive” approach, to free the user from having to manually select what to sync all the time, ”on-demand file synchronization” solutions have emerged. In this category of solutions, files still can be only either ”downloaded” or ”not downloaded”, but the ”non-downloaded” files are replaced with a placeholder file, trying to access which causes the file to be automatically downloaded in its entirety.

According to my best knowledge, the implementation of this feature was pioneered by Dropbox in 2017 with their ”smart sync” feature [1].

Today, both Windows and macOS have built-in system APIs to help developers implement this feature in their cloud file storage clients (Windows Cloud Filter API [2] and macOS File Provider API [3] accordingly).

While a significant improvement over the naive approach, there are still drawbacks, which stem from the simple fact that a file, if accessed, still needs to be downloaded completely. So it does not provide the best user experience if user’s internet connection is slow, or the file is big, or both.

2.1.3 File streaming

Google Drive, with its File Stream technology [4] provides exactly what is desired as far as file streaming is concerned. Opening a file does not require downloading it completely, and reading the file downloads only the requested parts. Yet it is still possible to mark files to be downloaded locally, and internet connection is not required to access already downloaded files.

Unfortunately, it is completely proprietary, and as of now works only on Windows, with macOS support having been removed in place of a macOS File Provider API implementation [5].

2.1.4 Decentralized file synchronization

There are multiple solutions for decentralized, peer-to-peer file synchronization.

What is probably the most popular solution to this problem is Syncthing [6], and proprietary Resilio Sync [7] which share a similar feature set.

There is no central server required, all the synchronization happens purely between peers in a purely peer-to-peer fashion, and otherwise the synchronization model follows the "Classic file synchronization" model, with ability to select individual files/folders for complete synchronization.

2.1.5 Cloud storage mounting via 3rd party filesystem

There is a variety of 3rd-party tools that allow mounting arbitrary cloud file storage.

Some of them are proprietary, such as ExpanDrive [8], Mountain Duck [9], and CloudMounter [10]. They allow mounting and working with an arbitrary cloud storage, and offer a similar feature set of mixing file streaming with having an option of keeping some files downloaded locally. Unfortunately, they all are closed source, and obviously also not peer-to-peer.

RClone [11] is an open source solution that allows mounting cloud storages and streaming files. But it does not allow for any offline work.

In addition to that, I have found no information on how any of these solutions handle file conflict situations.

2.2 Distributed filesystems

In my research, most of the distributed file systems I had found are designed for a very different use case than what file synchronization applications are designed for.

Filesystems like Ceph [12], GoogleFS [13], Lustre [14], Hadoop [15], and others are designed primarily with high-performance computing in mind, the likes of supercomputer clusters and Google-scale search indexing or other big data applications.

In particular, they still have a separation of a client and (sometimes many kinds of) servers, and while they are designed to handle individual node failures, operation when a node is completely separate from others is not a use case that they are intended for.

Of interest to me then are Ficus [16] and Coda [17] filesystems: these filesystems were specifically designed for disconnected operation: ability for clients to access and synchronize files, while maintaining ability to work with them while disconnected from the rest of the system.

Unfortunately, they are also not peer-to-peer, and don't have file streaming, the files are synchronized in their entirety.

Then, IPFS, the InterPlanetary File System [18], also exists. It is a peer-to-peer file sharing system, but is not designed to be a mountable, POSIX-

compatible filesystem.

ElmerFS [19] is a promising filesystem for my use case, but, as described in its paper, it still has some practical issues that could be improved in regard to conflict resolution, which I will discuss more in the following chapters.

2.3 Requirements analysis

With that, let me proceed with trying to specify requirements for my peer-to-peer filesystem prototype.

Requirements for a software system are often separated into two kinds: functional and non-functional. Functional requirements define what the system should do, what are its inputs and outputs, while non-functional requirements define how exactly it should achieve these tasks [20].

2.3.1 Functional Requirements

Mandatory requirements Now, I will list the required functionality for the filesystem, what I consider to be the minimum functionality for my system to be useful to its end users.

F1: Works as a filesystem The application should function as a filesystem, providing transparent access to synchronized files without requiring special handling by user applications.

F2: File synchronization The file and directory structure, and, of course, file contents, should be synchronized seamlessly across multiple peers.

F3: Peer-to-peer The application should not require a central server: a user should be able to install the application on two or more peers and they should be able to synchronize the files directly with each other, without any additional setup.

F4: Offline work In the event of lost connection to other peers, all already downloaded files should remain accessible.

Users should also be able to edit these files or create new ones without requiring a network connection, and the changes will be synchronized with the rest of the peers when connection to them is available.

F5: File streaming Files should be accessible on any peer even if they are not fully downloaded. In such cases, the requested file parts should be fetched on-demand from other connected peers.

F6: Conflict resolution If a file is modified concurrently on multiple peers, all versions of the file should be preserved. Other types of conflicts in the directory structure should also be resolved reasonably.

F7: Peer management User should be able to add new peers that the files should be synchronized with, or to remove such peers.

Additional requirements Then, additional requirements, that would be nice to have, but not required for a minimally viable product, and, as such, will not be implemented in this thesis.

F8: Offline download Users should be able to specify files or directories to be downloaded entirely and made available locally at all times.

F9: Space reclamation The filesystem should automatically remove downloaded files that aren't marked to be "available locally" from the local storage to free up space on the local disk. For example, it can be done based on a storage limit the user can set.

F10: Sharing It should be possible to share the files and directories easily. For a peer-to-peer system, it can be implemented as some peers acting as a "gateway" to the public internet, or sharing with other clusters.

2.3.2 Non-functional Requirements

N1: Cross-platform The application should support major desktop operating systems, including Windows, macOS, and Linux.

N2: Extensibility It should be straightforward to extend the application with new features.

N3: Data safety The application should be resilient to crashes, both of itself and the computer, ensuring no data loss.

N4: Encryption All communication between peers should be encrypted, ensuring no external peer can interact with the peers.

N5: Performance The filesystem's performance should be sufficient for practical use, with sequential read/write speeds reaching at least hundreds of megabytes per second.

2.4 Use case analysis

The use cases of the application should also be considered.

▶ **Definition 2.1** (Use case)**.** *A use case identifies the actors involved in an interaction and names the type of interaction. This is then supplemented by additional information describing the interaction with the system [21].*

In my system, there is only one actor: the user who will be installing the filesystem on their computer and using it. I have identified 5 use cases, describing how the user specifically might interact with my filesystem application.

UC1: Add a peer User should be able to add a peer for the filesystem to be synchronized with.

UC2: Remove a peer User should be able to remove a peer from the list of peers with which the filesystem is synchronized.

UC3: Set a peer's address A user should be able to manually specify an address that can be used to connect to a peer that can't be found automatically.

UC4: Starting/stopping the filesystem There should be a simple interface to start or stop the filesystem.

UC5: Updating the filesystem As the application will be distributed manually, without an app store, providing the user with a convenient way to update the application becomes my responsibility.

Figure 2.1 depicts these use cases in a diagram form.

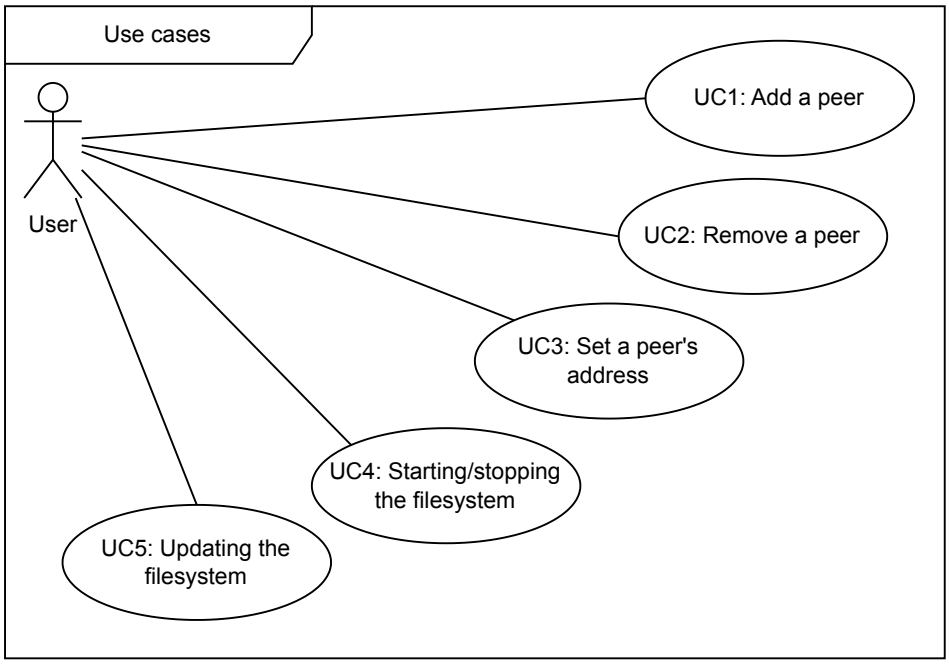

Figure 2.1 Use cases of the filesystem.

Chapter 3

Algorithm design

In this chapter, I will discuss the design of the algorithms that I will use to implement the filesystem.

3.1 CRDTs

The main thing to consider when designing the algorithms and data representation to be used in the filesystem is conflict resolution.

There is no single server that can be a source of truth for all operations, and the peers themselves can't be guaranteed to be always available either, as they can be, for example, mobile devices.

Yet the filesystem still has to remain completely functional, so the logical consequence of that is that all operations have to be executed completely locally, without any coordination with other peers. Then, these operations have to be synchronized with other peers, and, of course, the state of the filesystem on peers that had received the same operations has to be exactly the same.

This property of the system can be called *Strong eventual consistency*, and there exists a class of data types: *Conflict-free replicated data types*, that provide this property [22].

3.2 Filesystem structure

This section will discuss the algorithms to be used when implementing the filesystem tree representation, a problem that is notably separate from the storage and synchronization of file data itself.

3.2.1 Problem description

What is, exactly, a filesystem? Ritchie; Thompson consider that ”the most important role of UNIX” is to provide it [23]. Then, they go on to define three types of files in a filesystem: *ordinary files*, *directories*, and *symlinks*.

Of special interest are the directories, that in themselves contain another files and together build the structure of the filesystem. Importantly, the directory structure is constrained to be a tree: only directories can have other files in them, and a single directory cannot itself be a child of multiple directories.

Thus, the problem of synchronizing a filesystem can be reduced to the problem of synchronizing a tree.

3.2.2 Analysis of existing algorithms

Most mainstream cloud storage solutions use simple algorithms to handle conflicts, For example, in case of a conflict during file update or creation, both versions are kept and renamed, and the user is in some way asked to resolve the conflict manually later.

But file move operation is especially complicated. The ideal scenario after reconciliation of concurrent moves of the same file across different peers is that the file is moved exactly once, to one position, without being duplicated [24]. It is especially problematic if the moved file is a directory, as otherwise a cycle in the directory structure can be created, making the directory tree not a tree anymore.

Unfortunately, under more scrutiny, the algorithms used by common cloud file synchronization solutions might exhibit undesired behavior, as had been shown in various attempts to test the behavior of Dropbox, Google Drive, and others [24, 25, 26].

ElmerFS [19] is a filesystem implemented using CRDTs. To represent a directory a map CRDT is used, but for each directory a separate CRDT is used, and these separate directory CRDTs don’t ”communicate” with each other, so anomalies with move operations are still possible.

Thankfully, today, several designs of CRDT trees exist, that provide well-defined and proven solutions to this problem.

- A design by Kleppmann; Mulligan; Gomes; Beresford [24], which from now on will be referred to as the *Kleppmann tree*.
- ”Maram” tree by Nair; Meirim; Pereira; Ferreira; Shapiro [27].

After looking through both of these papers, the Kleppmann tree seemed to be clearly the more practical solution. Both file name conflict resolution and garbage collection algorithms are discussed in the paper and were easy to understand. That was not the case for me with the Maram paper, and, as far as garbage collection goes, its authors considered it ”non-trivial and out of scope of the paper”.

The idea of the Kleppmann tree is very simple: define a single operation, the move operation, that moves one tree node to be a child of another tree node, and a total order of these operations using logical timestamps.

There is no need for separate node creation and deletion operations: node creation happens implicitly when issuing a move operation on a non-existent node, and deletion can be simulated by moving a node to be a child of a special "trash" node.

In addition to that, a move operation also assigns a tree node some metadata, which can be used to store, for example, a file name or a reference to a file object containing its data.

Then, a combination of peer's unique ID and Lamport's timestamp can be used to create a global, total ordering of all the operations between the peers.

▶ **Definition 3.1** (Lamport timestamp)**.** *Lamport timestamp is a logical clock algorithm that allows to determine the order of events in a distributed system [28].*

Given a clock C_i algorithm works the following way:

1. *When an event happens, local to a given peer, the clock is incremented.*
2. *When an external event is received with a timestamp C_r, the clock is set to $max(C_i, C_r) + 1$*

Thanks to this algorithm, given an event with an associated timestamp, and another event, it is possible to determine whether one event could have happened before or after another (and therefore influenced it).

In case the clocks are the same, the ID of the peer where the event had originated can be used as a tiebreaker, thus creating a total order of all events in the system.

When a filesystem operation is executed on a peer, it is immediately applied to its local tree representation, and is put to a queue to be sent to other peers asynchronously. It is also appended to a totally-ordered operation log, together with the state of the moved node before the move was applied.

When a peer receives an operation, it checks whether the operation is strictly newer than the last applied operation.

If it is, then it is also applied and appended to the operation log.

If not, all operations newer than it are reverted, using the information about the previous state of the node saved in the log entry. Then, the received operation is applied, and all the reverted operations are reapplied on top of the new state. The reapplied operations might have different effects, as they are applied on top of a different initial state. But as all operations are deterministic, these effects will be the same on all the peers, so the states of peers that have received the same operations will converge. Before a move operation is applied it is also verified that it will not create a cycle, and if it would, the operation is skipped, so the directory structure will always be a tree.

3.2.3 Final design

Now, the Kleppmann tree should be extended with the needed features for my filesystem. Specifically, the following features need to be implemented:

- File information storage
- Conflict resolution
- Garbage collection
- Peer initial synchronization

File information storage In the paper, each filesystem tree node contains a set of children nodes, without concern for any metadata of the children.

Thus, is possible for one directory to contain multiple children with the same name, given that the name of the file is stored in the node's metadata.

My proposed solution is to replace the set of children nodes with a map, with file names as keys, that are automatically extracted from the children node's metadata, and the nodes themselves as values.

Then, two types of node metadata will be used for the filesystem:

File A file node metadata, containing a name, and a reference to a file object (what is a file object will be discussed in the following sections).

Directory A directory node metadata, containing only a name.

Conflict resolution The presence of unique filenames presents a need to resolve conflicts, in case files with the same name in one directory are created concurrently by multiple peers.

The solution to this problem is simple: when receiving a move operation from a peer that would result in creation of a child entry with a conflicting name, both entries should be renamed in a deterministic way.

For example, the file object ID can be appended to the file name, which would result in both conflicting files being renamed the same way independent of the peer that sends/receives the operation, ensuring the states always converge.

Unfortunately, this means that all the node creation operations will need to be recorded in the log (as now they can conflict with each other), while much of the effort in the paper was spent proving the fact that it is not necessary to do so.

ElmerFS solves the problem in a very interesting way: after such a conflict, each peer presents the file that was created locally as the file with the original name, while conflicting files have something appended to it (such as the name of the peer where the conflicting file was created). Then, after any of the conflicting files are renamed, and their names won't conflict, their real names will be presented. For simplicity, I've chosen to not implement this approach.

Garbage collection So far, as it has been described, the operation log is growing with each operation and never shrinks.

Thankfully, according to the paper, log truncation is possible using the following algorithm:

Notice that when receiving an external operation, only the operations with a timestamp newer than timestamp of the received the operation will need to be undone, the older operations are not relevant. Thus, if it can be determined that all new incoming operations will have a timestamp newer than a certain value t, the operation log can be truncated to have only operations with timestamp higher than this value. The timestamp t in this case can be called *causally stable* [29].

In practice, it means that each peer in a cluster has to remember the newest received operation from each other peer, and the causally stable threshold (the timestamp, before which the operations in the log can be deleted), is the minimum of these timestamps.

It also allows for removal of deleted files and other nodes: once an operation that has moved a node into the trash bin is causally stable and is truncated from the log, the node itself can be removed too, as then it is known no newer operations can refer to that node.

Peer initial synchronization It should be possible to add new peers at any time, and, of course, they should receive a full copy of the filesystem structure after being added.

As the log of operations is truncated, it is impossible to just send all of the operation history to newly created peers, so another way to introduce peers to the filesystem safely is needed.

Note that there is no lock on "adding a peer" or their initial sync, the filesystem should be completely available for all peers even when new peers are being added, or doing the initial synchronization with them.

Adding a peer should also be an operation that can be done on peers that are not connected to the rest of them. In that case, information about the new peer should be propagated to other peers when connected. This can be achieved by, again, using some kind of CRDT to store the list of connected peers. For implementation simplicity, the Kleppmann tree itself can be used, with peers being stored as tree nodes under the synchronized tree root (as will be described in subsection 5.3.5).

My proposed solution is to extend each tree node with an "effective operation" field, that contains the last operation that resulted in moving this tree node. Then, for initial synchronization, all that has to be done is reconstructing the "effective operation log", by visiting each node in the tree, and re-adding the respective "effective operations" to the log, and to queue of operations to be sent to the new peer. In effect, both new and old peer (and from the point of view of the new peer, the old peer is the new peer) share the entire effective operation logs with each other, merging them in a new total

order of operations and thus ensuring that the states will converge on both of them.

As a result of the log truncation, a node might end up being moved to a nonexistent parent, in case a parent was moved after adding a child to it. As operations are applied in sequential order, this will mean that the child moving operation will be applied first, and its new parent will not be found, as its ”effective operation” has a later timestamp.

This problem can be solved by ignoring the lack of a parent and creating a ”dummy” parent, as a child of a special ”lost+found” node. Then, it will be moved out of it to its correct location after processing the relevant move operation.

3.3 File data storage

Now, the format of the file storage itself, how file data will be represented in the system and synchronized between peers, should be determined.

As a possible implementation strategy, ElmerFS splits files into equally-sized blocks, each of which is a last-writer-wins CRDT data structure. As its name suggests, if multiple writers were to change the same block of file concurrently, only one of the writes will survive (determined to be the ”latest” using its timestamp), and the file itself will become corrupted.

For the filesystem to be of practical usefulness, in case of a conflict, there has to be a way to preserve both (or more) variants of a file.

This task is complicated by the fact that none of the peers participating in conflict might have the full copy of the file in their possession (due to streaming), therefore the filesystem must support copy-on-write copying of files.

Yet what happens when the peer that has the only copy of some chunk modifies it, and this modification conflicts with another peer, that had modified another chunk in the same file? In that case it will be impossible to recover the state of the files, and conflict resolution would be impossible. Figure 3.1 depicts such a problematic example of a conflict resolution attempt.

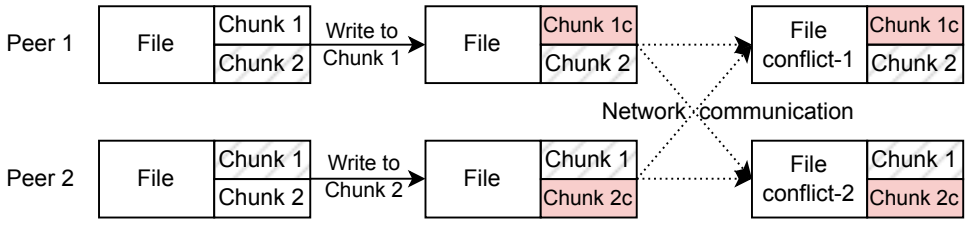

Figure 3.1 File corruption with LWWR for chunks.

Peer 1 only has Chunk 1 locally and writes to it. At the same time, peer

2 has only Chunk 2 locally and also writes to it.

A correct conflict resolution would be to create two copies of the file:

- Containing Peer 1's newly modified Chunk 1 and old Chunk 2
- Containing Peer 2's newly modified Chunk 2 and old Chunk 1

But it is impossible if the only copies of the chunks were already overwritten.

It then follows that the chunks themselves must be immutable, and must mot be deleted until it is known that no peer will ever reference it again. The garbage collection problem for file chunks will be described in detail in subsection 3.3.2.

Thus, a chunk is defined to be an immutable record, each assigned a unique random ID when created, and containing an immutable binary blob that is a part of one, or more files.

Then, a file will be a concatenation of these blobs, and a sorted map data structure can be used (using chunk's position in file as a key, and the chunk ID as a value), to effectively store and retrieve chunks in a given file (Figure 3.2).

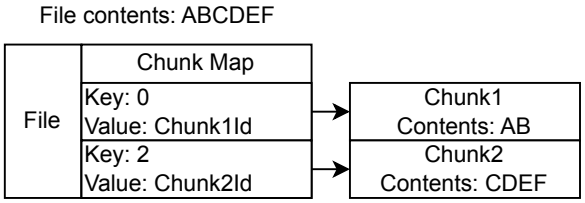

Figure 3.2 File representation using a map.

3.3.1 File conflict resolution

With ability to safely create copies of files, even without having their data completely at their disposition, detection and resolving file data conflicts themselves should be solved next.

To detect concurrent writes to a file by different peers, *version vectors* can be used, which are also used by Syncthing [30], Ficus filesystem [31], and Coda filesystem [17].

▶ **Definition 3.2** (Version vector)**.** *A version vector for a file f is a map of n entries, where n is the number of peers in the cluster. The ith entry $(P_i: v_i)$ gives the index of the latest version of f made at peer P_i. In other words, the ith vector entry counts the number v_i of updates to f made by peer P_i [32].*

Then, vectors are defined as *compatible*, if one of them has, for every key in the vector map, has a value that is the same or larger than in the other vector. Otherwise, they *conflict*.

A vector can also be defined to be *newer* than another, if it has at least one key in the vector map, for which the other vector has a lower value, and they do not conflict.

When a peer receives a newer vector, the version of a file present locally is simply updated to match the most recent information. If a file conflicts however, both versions of the file should be preserved.

In a manner similar to Syncthing, the file can be replaced with a version that has a more recent modification time, and then conflicting version can be saved in a new file, with the name containing a suffix with some information about the conflicting file [33].

For example, let's imagine a file created at peer A in a cluster of two nodes, A and B. Immediately after creation, it will have a version vector of $< A : 1, B : 0 >$. Then, let's imagine a peer B writes to this file, then it would update its version vector to $< A : 1, B : 1 >$. As for the all the keys in the vector, the values are higher than what peer A has, peer A accepts the update.

Then, let's imagine that A and B write to the file concurrently. Then, peer A will have a version vector of $< A : 2, B : 1 >$, and peer B will have a version vector of $< A : 1, B : 2 >$. From the point of view of peer A, the value of A in the vector is higher, but the value of B is lower (and, analogously, from point of view of peer B, the value of B is higher and of A is lower). Thus, the vectors conflict, and conflict resolution as described needs to be done.

3.3.2 Chunk garbage collection

File deletion should also be possible, and not only from the filesystem representation, but from the disk itself with all the unneeded data. The copy-on-write nature of conflict resolution makes this problem, unfortunately, not quite so straightforward as just deleting all the file chunks the file is referring to. Reference counting can be used on the chunks, but even that alone is not enough, as local reference counting cannot account for remote nodes possibly creating copy-on-write copies of files that could refer to any given chunk.

Let's consider a cluster with 3 peers, and a file consisting of two chunks (Figure 3.3).

In the beginning, Peer 1 and Peer 2 have locally downloaded only Chunk 1, and Peer 3 has all the chunks.

Peer 1 and Peer 2 write to the file concurrently, resulting in a conflict and its resolution. After that, let's imagine a situation where the contents of the original file corresponding to the second chunk are overwritten, or the original file is deleted.

It is then possible, because information about file structure and file contents is synchronized independently, that the information about the deletion (or the updated chunk list in case of overwriting the second chunk) of the file is received by the Peer 3 earlier than the information about File 2 contents (created as a

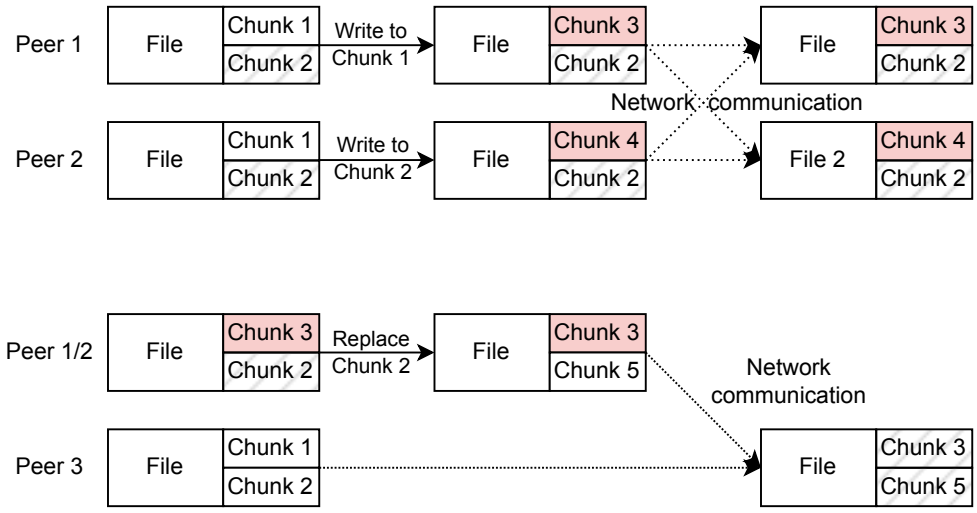

Figure 3.3 Deleting a referenced chunk with naive reference counting.

result of conflict resolution) is downloaded.

Therefore, locally, no references to Chunk 2 will remain, and it would be deleted, leaving File 2 corrupted, as Chunk 2 is no longer present anywhere in the system.

My proposed solution is verifying that it is safe to remove a given chunk by requiring all peers in the cluster to confirm that they have no references to a chunk, before a chunk can actually be deleted. Once a chunk has no incoming references on any of the peers, only then can it is possible to be sure that no other references to the chunk can appear, and only then can chunk be deleted (the details of the implementation of this will be described in subsection 5.3.3).

3.4 General object synchronization

As described in the previous section, version vectors can be used to detect conflicts between different versions of some file that was updated concurrently on different peers, and, in general, to track when a file has been updated by any of them, whether some version of the file is newer than another. This can be useful not only for files, but also to synchronize other kinds of data between peers.

For that, a ”remote object” abstraction will be introduced. In general, this can be any data type. In particular, it will be used to implement data types for files, file chunks, and information about a peer in the cluster. The object metadata, such as its version, can be also separated from the object data itself, allowing peers to be aware of an object's existence, without necessarily downloading its data, making this abstraction suitable for implementation of file chunk streaming. This is similar to approach used by, for example,

BitTorrent: information about the file chunks is shared first, and the chunks themselves can be downloaded later on demand [34].

These remote objects can be thought of as a higher-level abstraction for a state-based CRDT. Unlike an operation-based CRDT, like the aforementioned Kleppmann tree, where updates between peers are sent in terms of operations, state-based CRDTs send the entire object states [22].

Then, the synchronization algorithm for these CRDTs is quite simple: a peer only needs to send the entire object to other peers if it's changed locally, and the receiving peer needs to merge the local and remote version using a commutative, associative, and an idempotent merge function. With such a merge function, multiple peers that have exchanged the same object states are guaranteed to have the exact same final local object state, satisfying the *strong eventual consistency* property of CRDTs.

What that means is that for any two object states, the merge operation should give the same result no matter what the order of these objects is (*commutative*), should give the same result if more than two objects are merged in any order (*associative*), and should give the same result when applied on the objects more than once (*idempotent*).

Commutative `merge(a, b) == merge(b, a)`

Associative `merge(merge(a, b), c) == merge(a, merge(b, c))`

Idempotent `merge(a, a) == merge(a)`

With the version vector approach, a merge function can simply replace the local version with the received one if the received version is newer than the local one, and run some data-type-specific conflict resolution function if they conflict.

This approach fulfills the criteria for a state-based CRDT merge function: selecting the newest version out of two merged objects is simply a `max` function, selecting the maximum version of all the known ones, that is indeed associative, commutative, and idempotent. Then, only the data-type-specific conflict resolution function should also have these three qualities.

3.5 Offline garbage collection

For both the chunk garbage collection, and the Kleppmann tree log truncation algorithms, all nodes have to be reachable and communicate with each other. Which, for a filesystem oriented at personal usage might be especially problematic. A user might have a device, a computer, which might not be able to come online regularly, like a computer in a cupboard that is not turned on often.

To handle this situation, it is possible to track which peers have not been seen in a while, and ignore them for purposes of garbage collection. Then,

when that peer is connected again, it can be resynchronized from scratch. The drawback is that, for reasons described in subsection 3.3.2, some files might become lost or corrupted, so it should be possible to disable this feature.

To do so, for each peer, its last-seen timestamp is recorded, that is periodically updated by every node. Updates of this timestamp are propagated between nodes: when a node receives a timestamp update, its local state is set to a maximum of the received timestamp and the local timestamp, ensuring convergence of the last-seen timestamp across nodes. In addition to that, a grow-only counter CRDT is kept for each node. In this data structure, each peer has its own counter that is incremented only by the peer itself. Then, a total value of the counter is the sum of all the individual peer counts [35].

To synchronize this information between peers, peer information will be stored in a ”remote” object as described in section 3.4.

When a peer notices that a node was not seen for some predetermined period of time, it will consider the node unavailable and will ignore its existence for the purpose of garbage collection. The grow-only counter is also incremented, and the update is sent to other nodes. When a node receives information about the counter being incremented, it will also consider the node unavailable. When the node is reconnected to a peer that had considered it unavailable, it will resynchronize with it. This approach (removing unavailable nodes and resynchronizing after it had been reconnected) is similar to the one used in, for example, MySQL Galera Cluster [36].

Chapter 4

Software design

In this chapter, I will discuss the high-level design of the filesystem, its inner components, their responsibilities and interactions.

4.1 Technology stack selection

Java, together with Quarkus framework was selected to be used as an implementation language of the system.

Why Java? There are a couple of reasons:

- Maturity: Java, as an established language, has many powerful, well-tested and well-documented libraries.
- Garbage collection: Garbage collection frees the developer from having to think about lifetimes of objects in many cases, which is especially helpful when writing multithreaded software.
- Cross-platform: Java is not normally compiled to native code, but to bytecode for a more abstract Java Virtual Machine. Therefore, the developer has to think about cross-platform compatibility explicitly only in rare cases, with most of the code and libraries being expected to work on any JVM out of the box.

Quarkus was chosen, as unlike other Java frameworks, such as Spring, it moves all the overhead of "Java magic", that Java is known for (automatic dependency injection, automatic boilerplate code generation) from runtime to build-time, therefore significantly improving the application's runtime footprint and start-up time [37].

In addition to that, it makes it easily possible to ahead-of-time compile the app into a single executable using GraalVM Native Image, which might be an interesting option to try to make the runtime footprint comparable to that of other natively compiled languages.

4.2 Module organization

To make the code easily understandable, modifiable, reusable, and testable, separating code into modules with well-defined responsibilities and interfaces is usually considered a good idea. With that, let's try to roughly define to which areas of interest the design can be divided.

As the Maven build system was used to build the Java code, its multimodule feature was used to organize the code [38]. The diagram of the modules and their dependencies can be seen on Figure 4.1.

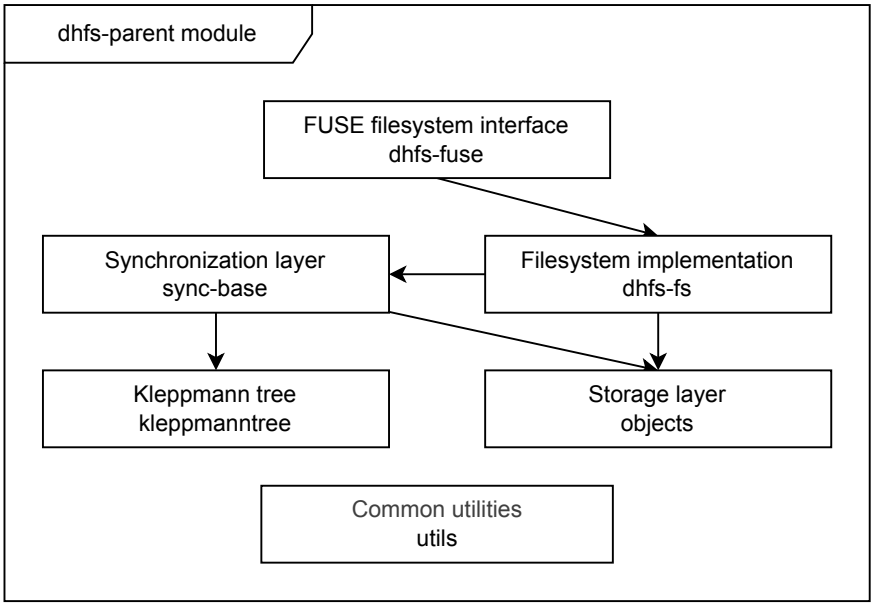

Figure 4.1 Maven module dependencies.
Dependencies on the `utils` module are not shown for clarity.

Kleppmann tree It seems to be a good idea to separate the algorithm of the Kleppmann tree to a module that is as separate from the rest of the system as possible. Then, it will be easier to test without relying on the specifics of the storage layer or the peer-to-peer communication. It also makes it possible to reuse this implementation in other applications, if someone wants to do so in the future.

Storage For the system to be of practical usefulness, the file structure, their data, and other metadata need to be persisted on disk.

Synchronization The files, other metadata should be synchronized between peers, reliably and with ability to handle conflicts that happen when using the filesystem with different devices. Ideally, this layer should know nothing about the filesystem layer on top of it, and should work only as kind of a

"framework" that the filesystem (or, possibly, other applications) can be built on.

Filesystem The filesystem itself. It should handle operations on the directory structure and operations with the files itself.

Filesystem interface User-facing interface for the filesystem. In the obvious case it can be a FUSE implementation, or other OS-specific interface. Yet, in future, it can also be, for example, Android's document provider API implementation, or a web interface.

In addition to the code of the filesystem itself, the user needs a way to run the application, add and remove its peers — some kind of a user interface will need to be implemented.

4.3 Class organization

Now, I will present a high level overview of the most important classes in the application (Figure 4.2).

Filesystem Starting with the user-facing interface, the `DhfsFuse` class implements the FUSE filesystem interface. The actual filesystem work is done by the `DhfsFileService` class. It itself is a filesystem implemented on top of the primitives implemented by the Synchronization layer.

To store filename and directory structure information, `JKleppmannTreeManager` is used, that provides an implementation of the Kleppmann tree, automatically synchronized between peers. Internally, it uses the Kleppmann tree implementation provided by the Kleppmann tree module (the `KleppmannTree` class).

To store file data itself "remote" objects are used — objects that are also automatically synchronized between peers, as described previously in section 3.4. To create, update, and read these objects the `RemoteTransaction` service is used. It handles updating the metadata of these objects, and downloading them, if needed, from other peers.

Notably, the contents of a file, the map of the chunks it is composed of, is not stored in the file object. If that were the case, on every read or write the entire list of chunks would have to be accordingly read or written, which would be slower and slower with increasing file sizes.

To address this issue, `JMapHelper` class provides a persistent ordered map implementation on top of the Storage interface. It allows to store and access a map of file chunks for each file, without the need to read/write the entire list of them every time.

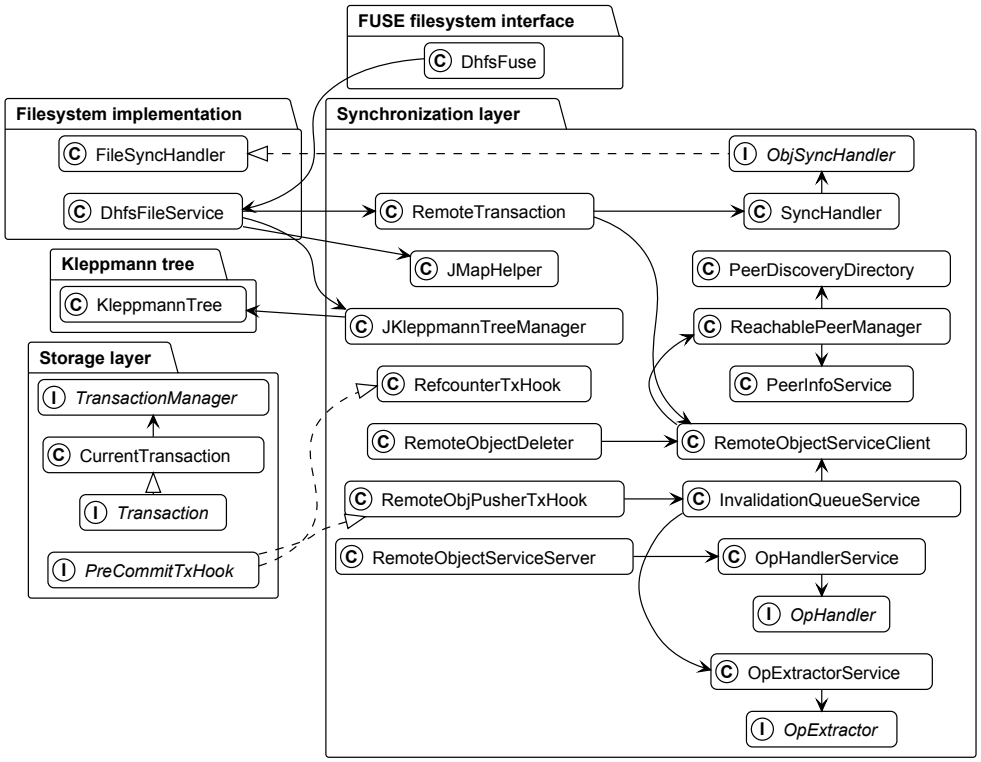

Figure 4.2 Overview of most important classes and most important dependencies between them.
Solid arows indicate dependencies between classes, dotted arrows indicate an implementation of an interface. Dependencies on `Transaction` and `TransactionManager` classes are not shown for clarity.

Synchronization Then, `RemoteTransaction` itself uses a `SyncHandler` service to process information received from other peers: the `SyncHandler` service processes updates sent by other peers and objects downloaded on demand. To handle conflicts, it can use an implementation of a `ObjSyncHandler` interface, such as `FileSyncHandler`.

To communicate with other peers, `RemoteObjectServiceClient` exists, which provides convenient interfaces to communicate with other peers. To know which peers are available, and how to connect to them, `ReachablePeerManager` keeps track of currently available peers. Itself, it uses `PeerDiscoveryDirectory` to know about the discovered peer addresses, and `PeerInfoService` to know which peers are a part of the synchronized cluster.

To keep track of updates that should be sent to other peers, `InvalidationQueueService` exists. It is notified about changes by the `RemoteObjPusherTxHook`, that is an implemenetation of the `PreCommitTxHook` interface, provided by the Storage layer, and is called whenever an object changes in the stor-

age, together with the information about the changes. Garbage collection and reference counting is also handled by these hooks, one of them is the `RefcounterTxHook`. Remote objects have a more complicated deletion procedure, that involves asking other peers whether the object to be deleted is safe to delete: this is handled by the `RemoteObjectDeleter`.

To know what to send to other peers, the `OpExtractor` interface can be implemented to "extract" operations that need to be sent from the objects that are put into the `InvalidationQueueService`. Then, the operations are sent to other peers, again using `RemoteObjectServiceClient` to communicate with them.

On the receiving side, `RemoteObjectServiceServer` handles the communication, and uses the `OpHandlerService` to process the received operations. This service dispatches the received operations to the correct implementation of an `OpHandler` based on the type of operation. In particular, it can be either using the `SyncHandler` to update the version of a known "remote object", or `JKleppmannTreeManager` to apply an external operation on some Kleppmann tree instance. It also handles sending data for requested remote objects.

Storage `Transaction` and `TransactionManager` interfaces (and, of course, their implementations) are provided by the storage layer, that are used by many other services. The transaction manager allows to start, commit, and abort transactions, and some convenience methods to run a function inside a transaction.

The `Transaction` interface itself allows one to create, update, remove and read objects in the storage layer. In particular, the implementation of the transaction that is exposed to automatic dependency injection is `CurrentTransaction`. It itself internally utilizes the `TransactionManager` to provide convenient access to the currently active transaction without the need to manually call it every time.

This layer also provides the `PreCommitTxHook` interface, that is used as described above. The details of the storage layer will be described further in section 4.6 and its implementation in section 5.2.

4.4 Filesystem

The Filesystem layer, built upon the Storage and Synchronization layers, will implement a simple filesystem with POSIX-like semantics. As it is primarily designed to be used with a FUSE frontend, it will closely mirror the required FUSE callbacks: such as opening file by path, reading, writing, creating symlinks, and so on [39]. Notably, at least for now, all files in the filesystem will be presumed to be owned by a single user, so not file ownership metadata will be stored.

To store directory structure and file name information, it will use the automatically synchronized Kleppmann tree provided by the Synchronization

layer, together with automatic file data synchronization, implemented on top of generic object synchronization support provided.

It will also handle conflict resolution of concurrent file updates, according to the previously specified algorithm, using the aforementioned `PreCommitTxHook` Implementation.

4.5 Filesystem interface

To present the filesystem as a "real" filesystem on the user's computer, FUSE (Filesystem in Userspace) [40] can be used. It is a Linux kernel interface allowing userspace applications to create their own filesystems, without needing to write a kernel module. It is also a name of a userspace library that presents its own interface, and communicates with the Linux kernel, `libfuse` [41].

For implementation of the filesystem this library will be used. The main advantage of using is its availability on platforms other than Linux. Specifically, both macOS and Windows have implementations: macFUSE [42] and WinFsp [43] accordingly.

As a Java FUSE wrapper, `jnr-fuse` [44] library was used. It was chosen as, unlike the other wrapper that I found, `jna-fuse` [45], it still seems to be maintained, providing out-of-the-box support for all major operating systems and processor architectures.

I had also considered using the new Foreign function and memory API [46]. But, as it is still a Preview API in the latest long-term-support release of Java at the time of writing, I had decided to not use it until it is more readily available.

The only significant class in the package is the implementation of the FUSE interface, `DhfsFuse`, implementing the required callbacks and the filesystem mounting/unmounting with the start and shutdown of the rest of the application.

4.6 Storage

Directory structure, the files and their data need to be persisted on disk, and this will be handled in the Storage layer.

In the initial prototype, a simple flat filesystem-based key-value storage (one object in the key-value storage stored in a single file in the filesystem) was implemented, with an additional caching and transaction layer being implemented incrementally on top of it. A file in the filesystem then was represented as a single object in the key-value storage, that in itself contained a complete map of a position in file to chunk ID entries.

Unfortunately, the overhead for filesystem operations has proved unacceptable, as writing/reading from a single file would necessitate reading and writing the entire list of its chunks with every operation.

Thus, a new requirement for the storage layer was to allow ordered iteration over the objects stored in it: an *ordered key-value store* is needed.

Then, this ordered key-value store could be used to implement a map, by, for example, using the unique file ID as a prefix and the position in the file as a suffix of a key, and storing a reference to the chunk as a value (the aforementioned `JMapHelper` service).

Next, to achieve reasonable performance, the filesystem, and therefore, the storage layer must support concurrent reads and writes.

There might be multiple threads reading, or writing different files, together with other threads doing background work, such as synchronization with other peers.

These threads then might access the key-value store, read objects, write objects. As these operations are not instant, the database might be left in an inconsistent state during the execution of the operations. For example, when writing something to a file, first a reference to the chunk might be written, and then the chunk itself. If some thread tries reading the file before the chunk is written then, it will see the file in an inconsistent state, with the chunk missing. This situation is obviously something that has to be avoided.

The solution to this problem are *transactions*, that can *isolate* different actors working on the database at the same time, and the exact guarantees that are provided by the transaction can be formalized by different *transaction isolation levels* [47].

In addition to that, if a transaction fails, for whatever reason, the database should not be left in an inconsistent state. For example, in the above case, if a chunk creation fails, the reference to it written in a file should not be kept, the entire transaction should be aborted, and no other thread should see that reference. Meaning that, the transactions should be *atomic*, either applied completely or not at all.

Additionally, when communicating with other peers, there has to be a way to guarantee that, the information that is sent to a peer will remain safe on disk. This property can be described as *durability*, meaning that once the transaction had been committed, it should survive all possible malfunctions [48]. That is, however, not normally the case for filesystems. To improve file performance, extensive buffering is performed by modern operating systems, of both reads and writes. Then, to ensure data durability an explicit operation, such as `fsync` is needed.

Thus, the storage layer needs to be able to handle both ”fast” commits, being able to return without data necessarily being flushed to the disk for the filesystem use case, and ”durable” commits, waiting for the transaction to be persistent on disk when communicating with other peers, so that the system isn't left in an inconsistent state.

As reference counting will be the strategy for garbage collection, to simplify it, it is also desirable to be able to create hooks that will be run when an object in the storage is created, deleted, or updated. Such an implementation could

guarantee *consistency* of the data in the storage layer.

In the initial implementation this was not the case: reference counting had to be done manually when modifying the objects, which had been the cause of many hard-to-debug bugs.

In these hooks, there should be access to both the old and updated object version, and it should be possible to modify other objects as well.

Then, reference counting can be implemented easily by extracting the outgoing references from both versions of the object, and calculating the difference between them.

The hooks can also be useful to automatically detect changes to objects for the purpose of sending updates to other peers.

These four properties, atomicity, consistency, isolation, and durability, commonly referred to as *ACID*, are the cornerstones of any data storage framework.

With all of that in mind, the key requirements for the storage layer can be summarized as follows:

- Ordered key-value storage
- ACID semantics
- Transaction hooks
- Allows non-durable commits

4.6.1 Motivation

Indeed, many existing Java libraries exist to handle the problem of persisting data on disk, utilizing existing databases as their backends. A solution using Hibernate, one of the most popular java persistence libraries, and one that is well-supported within Quarkus [49], was prototyped.

Unfortunately, its throughput as tested by file copying was significantly slower than the aforementioned naive filesystem-based key-value storage implementation.

In my understanding, the most important performance problem is write amplification caused by lack of non-durable commits. As will be described in section 5.1, writing to a file consecutively creates many chunks that are deleted almost immediately, as smaller write requests are combined into larger chunks, with the larger chunk being rewritten and recreated with every write. Thus, in the worst possible case, with, let's say, writing to a memory mapped file on a system with 4 KB page size, with file chunk size target of 512 KB, rewriting this 512 KB chunk page-by-page can result in 128 separate write operations, each of which creates a new 512 KB chunk with a 4 KB part of it replaced. Thus, in total, in a database engine with durable commits, each of these chunks will have to be completely persisted to disk before the write

operation completes, resulting in 65 MB of data being written, and a write amplification factor of 128, meaning that 128 times more data was actually written to the disk than to the filesystem itself.

Most database engines allow relaxing the durability requirement: for example, RocksDB [50], and LMDB [51] have options for that. But I had found no existing solution that would allow freely mixing durable and non-durable commits in one database.

Thus, it had been decided, for performance reasons, to write a custom storage layer with the specific requirements of the filesystem in mind.

4.6.2 Interface design

With that in mind, let's try to define what the interface of the storage layer will look like. I will describe the most important high-level parts, leaving more specific details for the upcoming implementation chapter.

In fact, in the first prototype of the filesystem the Storage layer was very tightly coupled with the rest of the system. Numerous deadlocks, race conditions, and general impossibility to change code without breaking one of the many special cases the storage layer was handling resulted in it being completely separated from the rest of the system, with a goal of making its outside interface as simple as possible.

First, types for the data that is going to be stored in the storage layer need to be defined. It is a key-value store, so let's define the data types that will be used for keys and the values in this layer.

Code listing 4.1 Proposed object key interface.

```
public interface JObjectKey
            extends Comparable<JObjectKey> {
    static JObjectKey of(String value);
    static JObjectKey random();
    static JObjectKey first();
    static JObjectKey last();
    String value();
}
```

The key interface, `JObjectKey` (Code listing 4.1) is, essentially, a simple wrapper for a String with a couple of helper methods. Every object in the storage layer is uniquely identified by its key. The static `of` method allows to create a key with a given string value, and keys created this way can also have their value read via the `value` method.

It can also be needed to create iterators that start from the beginning or the end of the database, so special implementations are also provided that are returned from special `first` and `last` functions, that accordingly compare less and more than any other normal key, and do not support reading their value.

Code listing 4.2 Proposed data object interface.

```
public interface JData {
    JObjectKey key();
    int estimateSize();
}
```

The `JData` interface provides a way for users of the storage layer to define objects that will be stored in the database. Anything that will be stored in a storage layer needs to implement this interface. The `key` method has to be implemented, and it will be used to define the object's key. Then, `estimateSize` method should return an estimated byte size of the object, and can be used, for example, to limit read or write cache size (Code listing 4.2).

Then, let's define an interface that will allow ordered iteration over the database (Code listing 4.3). While Java's `Iterator` class is enough for simple iteration over collections, for a more fully-featured key-value storage more functionality is desired. Reading some objects, but skipping over others based on their keys, for example, can be used when implementing a map, so a set of methods to read the key and skip over is needed. Reverse iteration sometimes can also be desirable, so all the methods are also duplicated with their inverted option. Also, the underlying database engine resources might need to be created or freed with each iterator, so it would also have to implement the `AutoCloseable` interface. The interface is generic, and can be used to define iterators of key-value paris of any types, with the constraint that the key implements the `Comparable` interface, meaning that the keys can be compared with each other.

Code listing 4.3 Proposed iterator interface.

```
public interface
CloseableKvIterator<K extends Comparable<? super K>, V>
            extends Iterator<Pair<K, V>>, AutoCloseable {
    K peekNextKey();
    void skip();
    K peekPrevKey();
    Pair<K, V> prev();
    boolean hasPrev();
    void skipPrev();
    CloseableKvIterator<K, V> reversed();
}
```

Then, the storage layer has to provide an interface to begin, start, and end transactions, and to access data in these transactions, so let's also define the interfaces for that.

Code listing 4.4 Proposed transaction interface.

```
public interface TransactionHandle {
    void onFlush(Runnable runnable);
}

public interface Transaction extends TransactionHandle {
    void onCommit(Runnable runnable);

    <T extends JData> void put(JData obj);
    <T extends JData> void putNew(JData obj);

    void delete(JObjectKey key);

    <T extends JData> Optional<T>
      get(Class<T> type, JObjectKey key);

    CloseableKvIterator<JObjectKey, JData>
      getIterator(IteratorStart start, JObjectKey key);
}
```

The `Transaction` interface (Code listing 4.4) contains methods for adding, removing, and reading objects, as well as a method to get an iterator, beginning with an arbitrary element. `IteratorStart` enum can be used to specify where the iterator will be placed: on an element less than, less or equal, greater, greater or equal relative to the provided key. The `get` method, in addition to the key of the object to read, has as a parameter the expected class of the read object: the object that is read is checked by the transaction to be an instance of the expected class. Otherwise, an exception would be thrown. Method to create new objects, that are known to not exist before is provided (for example, when creating file chunks that are known to have a unique random ID), `putNew`, it can provide a slight performance improvement as for these objects there will be no need to check their previous version when committing the transaction.

In addition to that, there is an `onCommit` method, which allows to add a callback which will be run once the transaction is committed (but might not have been yet persisted to disk).

As well as the `onFlush` method in `TransactionHandle` interface, which allows one to add callbacks that will be run once the transaction is persisted to disk. `Transaction` itself also implements this interface, allowing to add these callbacks from inside the transaction too.

Code listing 4.5 Proposed transaction manager interface.

```
public interface TransactionManager {
    void begin();
    TransactionHandle commit();
    void rollback();
    Transaction current();

    TransactionHandle run(Runnable fn, boolean nest);
    <T> T run(Supplier<T> supplier, boolean nest);
}
```

An implementation of the `TransactionManager` interface (Code listing 4.5) will handle the creation, rollback, and committing of the transactions, and should allow accessing the currently running transaction, if any.

It also provides some helper methods (`run`), that allow running a function inside an automatically managed transaction, that can be possibly nested inside another, already running, transaction, depending on the `nest` parameter. In case this parameter is `true`, and a transaction already is running, a new one will be created, and the function will run in it, and the new transaction will be completely isolated from the already running one. The transaction, in case of a conflict when committing, can also be automatically retried.

A function can either return some value of any type (`Supplier<T>`), or nothing (`Runnable`). In the latter case, the `run` method returns a `TransactionHandle`, that can be used to add callbacks that will be run after the transaction is flushed to disk.

Then, finally, the `PreCommitTxHook` interface (Code listing 4.6) is defined, that can be used to listen into what transactions are changing, and can be used to reference the mentioned automatic reference counting. It also has a `getPriority` method, that can be used to define the order in which the hooks run: for example, the hook that does the reference counting should run before the hook that deletes objects without incoming references.

Code listing 4.6 Proposed pre-commit hook interface.

```
public interface PreCommitTxHook {
    void onChange(JObjectKey key, JData old, JData cur);
    void onCreate(JObjectKey key, JData cur);
    void onDelete(JObjectKey key, JData cur);
    int getPriority();
}
```

4.6.3 Internal design

Thankfully, there is no need to write the storage layer completely from scratch, as there are many existing ordered key-value storage engines available. Some of the most popular alternatives considered were:

Berkeley DB One of the oldest key-value storage engines [52], not actively developed anymore.

LMDB A lightweight storage engine, created to be a faster and more lightweight alternative to Berkeley DB [53].

LevelDB A storage engine by Google [54], known for being prone to data corruption [55], also not actively developed anymore.

RocksDB LevelDB fork, created with an intent of improving its performance [56].

Berkeley DB and LevelDB both do not seem to be actively developed anymore, so they are immediately disqualified from being used. Then, LMDB and RocksDB were considered. Both have convenient Java wrappers, and storage backends using both of them were prototyped. LMDB was then chosen as having about 4× better throughput as measured by file copying.

As Java is used, serialization and deserialization of the stored objects will have to be done. With that in mind, to achieve reasonable performance, some cache of already deserialized Java objects will have to be implemented.

The requirement for mixing durable and non-durable commits also necessitates implementing a custom write cache, that will allow committed transactions to be written to disk in the background, with an option to wait for some transaction to be flushed to disk.

Thus, the implementation will have to combine the following storage layers:

Backing storage The persistent storage layer, provided by LMDB, or possibly something else. It is represented by the `ObjectPersistentStore` interface.

Read cache Cache of read and deserialized objects, for an additional performance improvement, it is implemented in the `CachingObjectPersistentStore` class.

Write cache Improves performance when writing data, allows multiple concurrent writers, and waiting on some transaction to be committed: it is implemented in the `WritebackObjectPersistentStore` class.

Transaction Reading objects inserted into the same transaction should also be possible.

A diagram of the classes implementing these layers can be seen on Figure A.1.

The main challenge that then follows is maintaining a consistent view of the database through all these layers, which I will talk about in the implementation chapter section 5.2.

To make the implementation more straightforward, all the layers of the storage stack will be able to provide a snapshot of themselves. A snapshot represents state of the layer at a consistent point in time at, and has a version, that can be read by the user of the snapshot. This version is global for all the layers, and is incremented when a transaction is committed. They also provide a way to read objects from this layer, either by a specific key, or by creating iterators. And, thanks to this standardized interface, none of the layers need to know what exactly the layers under it or above it are doing.

The layers also provide ways to propagate changes through them: `commitTx` method takes lists of objects that are written to or deleted, applies these changes, and forwards to the next layer. The committed transaction ID is assigned by the `WritebackObjectPersistentStore`, and then propagated down with the rest of the data.

For the write cache layer, the commit function also returns a callback function, that can be used to register other callbacks to be run when the transaction is actually committed to disk. For the same purpose, `asyncFence` function allows to register a callback to be run when a transaction with some specified ID is flushed to disk.

`TransactionService` class actually handles the work of creating, committing, and rolling back the transactions, while the `TransactionManagerImpl` provides a managed thread-local transaction stack. The `commit` method of `TransactionService` returns not only a `TransactionHandle` that allows adding callbacks to be run on transaction flush, but also a collection of callbacks that should be run immediately.

In between the read cache layer and the `ObjectPersistentStore`, there exists a `SerializingObjectPersistentStore`, which is a relatively thin layer that serializes written data, and deserializes read data. For data serialization, Java's built-in serialization API is used, as it required no effort to use, though for better performance, and to allow object schema evolution, some other library should be used in the future.

`ObjectPersistentStore` also provides `getFreeSpace` and `getUsedSpace` methods, that can be used to report the available space by the filesystem. `JDataVersionedWrapper` is a versioned wrapper for the `JData`, containing simply an additional `version` field, to know the version of the object in the storage, that is used during transaction commit for conflict detection. To improve performance, it also is lazy: the object is read and deserialized from storage only if the data of the object is actually accessed in the transaction.

4.7 Kleppmann tree

The Kleppmann tree module provides a storage-layer-independent version of the tree, which makes it possible to easily test, and possibly reuse in other projects.

For that, *dependency injection* will be used. The `KleppmannTree`, instead of using the Storage layer inside itself directly, will have in its constructor a dependency, a `StorageInterface`, and others, which allows abstracting away the specific storage implementation from the algorithm of the tree. The dependencies should provide a way to create, modify, and access tree nodes, operation log, and queues through which the operations are sent to other peers.

The implementation itself should provide simple ways to execute move operations, local and received from remote peers, and to traverse the tree to find files or directories by a given path.

4.8 Synchronization

In this section, I will describe the interface of the Synchronization layer from the point of view of its user. The details of its implementation will be described in the next chapter in section 5.3.

The synchronization module provides all the plumbing and the building blocks for building a filesystem on top of it.

Specifically, the parts of interest for building the filesystem are:

- Synchronization layer for file data, that is going to automatically synchronize files and their data across different peers.
- Synchronization layer for the Kleppmann tree, automatically taking care of sending and receiving operations from/to all the peers.
- Automatic reference counting for file chunks.
- Efficient persistent map implementation to store file chunk data.
- Peer management and their connectivity.

4.8.1 Remote objects

In the filesystem there are files, and the files consist of the chunks. Then, it should ”stream” these files when accessing them, thus files only have references to chunks, and when accessing a file, a chunk might have to be downloaded from some other peer.

To do so, it is required to know what peers have which chunks, and to have a way to easily download them, and a way for a peer to let other peers know which chunks it has.

As discussed before in section 3.4, this concept can be generalized to other data types, which I will call ”remote” objects.

A `JDataRemote` interface is provided, that can be used to define data types of objects that are automatically synchronized between peers. A remote object consists of a metadata part, and the data part, which allows for a peer to be aware of an object's existence without necessarily downloading all of its data.

The metadata, among other things, contains the version vector, that is used to track the latest version of the object, and to detect conflicts if it's modified on multiple peers concurrently.

For convenience, a `RemoteTransaction` service is provided, which can be used to conveniently create, read, and update remote objects. Reading a remote object will transparently try downloading it from remote peers if it's not available, even doing conflict resolution if needed. Creating and updating the remote objects with this interface should automatically handle incrementing the version vector, possibly updating other metadata, and notifying other peers of the changes.

For files, a ”data transfer object” also will need to be prepared for sending, as the file object does not have in itself data about the chunks: it is stored beside it in a map. An implementation of a `DtoMapper` can provide that functionality. In case of the files, it also reads the entire list of its chunks and adds that to the File data transfer object.

Not all objects need that though, so if the same class implements both `JDataRemote` and `JDataRemoteDto` interfaces then it could be sent right away without additional processing.

This abstraction will be used not only for chunks, but also for file nodes themselves, and this is the part where conflict resolution will be needed.

For that, a user of this interface can provide an implementation of a `ObjSyncHandler` interface. That implementation will be called when receiving an object from a remote peer, and it can handle possible conflict resolution, or updating linked data types, for example processing information about the chunks of a file received from its DTO.

4.8.2 Synchronized Kleppmann tree

For the filesystem implementation, a way to store and synchronize filesystem structure information is needed, and, as described previously, for this purpose the Kleppmann tree will be used.

As it is implemented separately in the Kleppmann tree module, the job of the Synchronization layer will be to provide a wrapper over this existing implementation, providing the implementations of required dependencies and handling the communication and synchronization between different peers, which should be transparent for the user of this layer.

4.8.3 Map

A simple map implementation on top of the Storage layer is provided in the `JMapHelper` service. An object, such as `File`, can implement a `JMapHolder` interface, and then it holds an ordered key-value map of references to other objects. Its interface is similar to the `Transaction` interface: it should be possible to get, create, remove references to objects by a map key, and to create an ordered iterator.

A set of transaction hooks could guarantee consistency of the data, automatically deleting the map entries if the map holder is deleted, and maintaining the reference counts for the objects referenced from the map entries.

4.8.4 Peer management

My filesystem is a peer-to-peer filesystem after all, so a rather important, yet not terribly complicated part of the design is peer management.

All the peers in the cluster should be remembered, it should be known which ones are available, and an interface for other parts of the system to communicate with them should be provided.

An API for the user interface will also should be available, to conveniently add, remove, view, the peers in the system.

4.9 User interface

The user needs to have a way to add, remove, and view the peers in the system. For that, a web interface seems to be a nice fit. A web interface will be cross-platform, and will also make it easy to run the filesystem on a server without a graphical desktop, if needed.

The user will also need a way to run the filesystem, and update it. Ideally, something with a nice graphical user interface will be implemented, but it is not required for a functioning system now.

Chapter 5
Implementation

In this chapter, I will discuss some of the more specific implementation details, that had to be solved to make the system work, and the development process.

5.1 Filesystem

Data format To store the filesystem directory structure, the synchronized Kleppmann tree is used, provided by the Synchronization layer. File information is then stored in the tree node's metadata fields, with two types: `JKleppmannTreeNodeMetaDirectory` and `JKleppmannTreeNodeMetaFile` for a directory and a file accordingly (Code listing 5.1).

Code listing 5.1 Filesystem tree node metadata definitions.

```
public record JKleppmannTreeNodeMetaFile(String name,
                                         JObjectKey fileIno)
    implements JKleppmannTreeNodeMeta {...}

public record JKleppmannTreeNodeMetaDirectory(String name)
    implements JKleppmannTreeNodeMeta {...}
```

File data itself is then stored in two remote objects: `File` for the file itself, and `ChunkData` for the file chunk (Code listing 5.2).

Code listing 5.2 Filesystem remote objects definitions.

```
public record File(JObjectKey key, long mode,
                   long cTime, long mTime,
                   boolean symlink)
    implements JDataRemote, JMapHolder<JMapLongKey> {...}

public record ChunkData(JObjectKey key, ByteString data)
    implements JDataRemote, JDataRemoteDto {...}
```

A symlink is represented as a normal file with a `symlink` flag set. The symlink target is stored as a string inside the file data.

An important implementation detail of the filesystem, that significantly improves its performance is the usage of `ByteString` class provided by Protobuf.

This class implements a rope data structure, meaning that concatenating two byte strings does not create a copy of the data, but only a new object that contains the references to two already existing strings.

Together with the caching layer, this ensures that file data isn't copied needlessly in the filesystem layer when writing to them.

Reading When reading a file, the function call contains the file identifier (file object key), offset from which to read, and length of data to read from the offset, and the return value is, of course, the read data (Code listing 5.3).

Code listing 5.3 Read method definition.

```
ByteString read(JObjectKey fileUuid, long offset,
                int length);
```

For that, first it is required to find the first chunk with its position smaller or equal to the offset. Then, the offset from which to start reading in the chunk is calculated, `offset - chunkStart`, and the contents of the chunk are read, and then the following chunks, until the requested amount of data is read, or the end of file is reached (Figure 5.1).

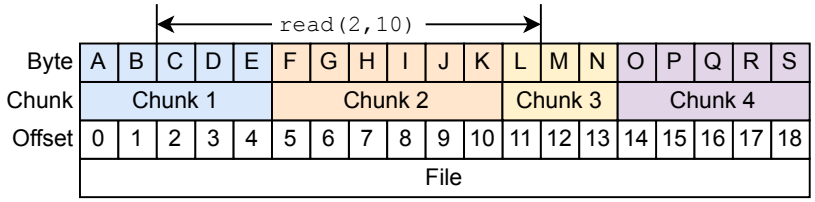

Figure 5.1 Reading a file with chunks

Writing The write command takes in the offset of the file from which writing should begin, the data that should be written itself (Code listing 5.4), and returns the number of bytes that were actually written.

Code listing 5.4 Write method definition.

```
long write(JObjectKey fileUuid, long offset,
           ByteString data);
```

Although writes can be arbitrarily small, creating too many small chunks is not desirable: the overhead for storing them, processing them, and especially synchronizing them would be quite noticeable.

With that in mind, the sizes of individual chunks should be kept consistent. For ease of implementation, the desired size can be limited to being a power of two.

When writing, the write start offset is taken and aligned down to the selected power of two. Then, the first chunk in which the new offset is located is found.

For first chunk, it is checked whether the offset that is being written from is further in the file than the start of the chunk itself. If it is, it means it is required to keep the prefix of the first chunk. Because of the power-of-two alignment, it might also happen that there will be multiple chunks between the first chunk and the actual write offset. In this case, all of the data in these chunks needs to be preserved.

Then, all the chunks in between the first chunk and the last chunk (containing the end of written data) are simply removed: their contents are not needed.

The last chunk through, like the first one, might still contain some information that should be kept, so if its end is past the position the writing ends, the last non-overwritten bytes of the chunk also need to be remembered.

Then, all the chunk entries that have been read are deleted, and replaced with the written data (including the possibly kept data from first and last chunks), split into new chunks of desired size.

An example of what happens to a file when executing a write command (without the rounding of the chunk sizes and the beginning to the power of two) can be seen on Figure 5.2.

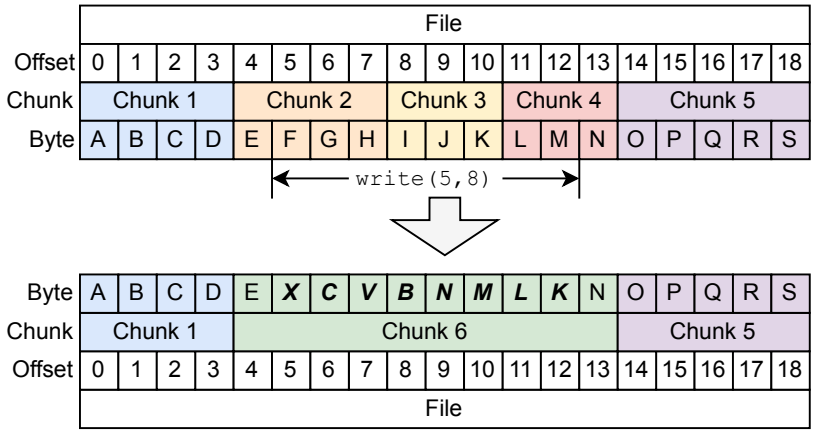

Figure 5.2 Writing to a file

An important thing to notice is that, if, for example, a file is being simply copied, by appending new data at its end, then new chunks will be constantly created until the last chunk of the file reaches the target size. This will require some consideration in the implementation, so that too much computer power is not wasted simply reading, copying, and writing the same data all over again.

5.2 Storage

Now, I will discuss some of the implementation details of the Storage layer.

5.2.1 Transaction isolation

The highest form of transaction isolation, according to the ISO SQL standard, is *serializable* [47], meaning that all transactions have such effect as if they were executed in some order, one after another.

A less strict isolation level is *snapshot isolation* [57, 58], which means that each transaction has a view of a consistent snapshot of a database, consisting of a view of the effects of all transactions, committed before it. This is the approach that many modern databases and database engines choose, being the default in PostgreSQL [59], MySQL [60], and others.

And this is the isolation level that LMDB provides, and this is the isolation level that all the layers on top of LMDB will have to also provide.

A simple way to solve this problem is to use persistent data structures at each of the caching levels. A persistent data structure is such a structure, that allows access to not only the newest version of itself, but also to the older versions, at any time [61].

For example, when a new item is added to a regular map, for example in Java, the map is modified in place. There is only one `TreeMap` object, for instance, that is modified, and there is no way to access the map before the item was added.

A persistent implementation, for example, would not have a method that modifies the object in place. Instead, like in the persistent data structure library that I will be using, `PCollections` [62], the map might have something like a `plus` method, that would return a new version of the map containing the new item, while keeping the old version intact.

Effectively, this is also how LMDB works under the hood: a write transaction doesn't overwrite existing data, but only creates new nodes in the tree, and then changes the root pointer to point to the new root. This way, the old readers can still use the old tree, and have a consistent view of the storage.

If persistent data structures are used for the caches, the implementation of snapshotting is trivial. No locking is needed, either. All that has to be done is creating snapshots (i.e. reading the persistent data structure) of each of the layers, and then checking if they have the same (or compatible) version.

5.2.2 Layer merging

Once the snapshots of each of the layer have been created, they need to be merged.

For simple read-by-key operations, this is trivial: just checking each layer from top to bottom. If there is an entry for the requested key, then it is

returned, otherwise, it is delegated to the layer below.

But ordered iteration is less trivial. My design is inspired by RocksDB design [63]: to take iterators of the layers, and then merge them together using a merging iterator. Although, while RocksDB utilizes a ”flat” iterator design: all iterators are equal, and contain multiple versions of the same object, identified by the transaction number that wrote them, my design is hierarchical.

Most of the work is handled by the `TombstoneSkippingIterator`, which given multiple iterators, will combine their views into one consistent view. As the snapshots are created using persistent data structures, and the updates of the database are propagated from the transaction layer down to the backing storage layer, the iterators from the above layers have an effective ”priority”. Meaning that if an iterator with higher priority and an iterator with lower priority have an object with the same key, the higher priority item will ”shadow” the lower priority item.

For example, if an object was deleted, and the information about that is still only in the write cache layer (the object's ”tombstone”), and the object still exists in the layers below, the tombstone of the upper layer will take priority, and in the merging iterator the object will not be visible.

5.2.3 Conflict detection

Snapshot isolation often goes hand in hand with *multiversion concurrency control*, and, indeed, this will be my approach too. Instead of locking the keys and limiting only one transaction at a time being able to access a given key, there can be multiple versions of the object in the system, with multiple transactions working with them at the same moment in real time.

It follows then, that this might generate some *anomalies*, if all the transactions were allowed to commit their changes without any verification that they don't conflict with each other.

In snapshot isolation, to solve that, when a transaction is being committed it is verified for all the objects it had written, that none of them have been updated by other transactions in the meanwhile. If such updates occurred, the transaction is aborted and can be retried.

Because it is not known whether any of the read objects have changed, *write skew* is possible. It is an anomaly when two transactions reading an overlapping set of objects, and make non-overlapping writes to these objects, can create an inconsistent database state [64].

A classic example of that is a problem with two doctors [65]. Imagine a hospital where at least one doctor must be always available on-call.

Then, let's imagine a situation where Alice and Bob are the only doctors on-call right now, and want to leave home, and do so concurrently.

- Both Alice and Bob check each other's state, and see that one another are on-call.

- As they see that at least one doctor will still be on-call if they leave, both of them still leave.
- As they have not written to the same rows (assuming that the ”on-call” state is stored in a single row per doctor), the transaction is committed successfully.
- Now, there are no doctors on-call, and the constraint of always having one is violated.

To avoid these problems, a stricter isolation is needed. *Serializable snapshot isolation* is one of the solutions, and it can be implemented by checking, when committing the transaction, whether it depends on any stale data, and abort if it does.

A simple way to achieve that is to check if not only the written-to rows have been updated, but also to check all the read objects [66].

5.2.4 Caching implementation

Storage layer Implementation of the underlying storage layer is also modular and replaceable: the `ObjectPersistentStore` interface is provided.

There are two implementations, the LMDB one, and an in-memory storage for testing, and the presence of this interface allows for easy implementation of other storage backends, if one were to desire so.

Implementation of the LMDB storage layer is quite straightforward: the snapshot and commit facilities are already provided by the LMDB itself, they only need to be wrapped around to fit the proposed interface.

Then, the backing storage layer is wrapped by a serialization layer, which simply deserializes the data that is being read from the underlying layer, and serializes data that will be written. For simplicity, Java's built-in serialization mechanism is used.

Read caching layer To implement snapshotting, the read caching layer is implemented using a persistent tree map. When a snapshot of the cache is taken, it creates a snapshot of the underlying storage layer, then reads its cache map. If their versions match, then it is possible to proceed with returning a merge of them. If not, reading them is retried until compatible versions are read.

Reading an object and then accessing its data, either through the `readObject` calls, or via an iterator, causes an attempt at caching the object. A global cache is read, modified, and then an attempt is made to atomically replace it. If the versions of the cache match, it succeeds. If not, an attempt at caching is skipped. Additionally, the snapshot version of the global cache is checked (that is incremented during commit). If a commit had happened since the cache snapshot had been created, no more caching attempts will be

made to the global cache by this snapshot, and the current caching attempt will not succeed.

The cache size is limited, to estimate object's size the `estimateSize` method is used. If caching an object would cause the cache to go over the size limit, objects are removed from the cache in its iteration order. As most of the objects have random keys, this is similar to a random eviction policy.

While I believe that least-recently-used eviction policy would be better, as reading/writing files has a much higher probability of accessing the same chunks consecutively, I had not found a persistent data structure library that would allow for a simple implementation, and this "random" eviction policy had proven to be adequate in my testing.

Write caching layer Write caching layer is asynchronous, meaning that the transactions committed to it are written back to the underlying storage after its `commit` function returns to the caller. The size of the write cache is also limited, and the commit function waits for the already-committed transactions to be flushed to disk, if the cache is completely filled already.

To know when a transaction is actually persisted to the disk, `commit` returns a function that allows to register a callback that runs when it happens.

An important optimization is that when a transaction is committed, its effects are merged with all the transactions that weren't flushed to disk yet. Which means that, for example, if a file chunk was created and then immediately deleted, it will never get written to disk, if the first transaction that had created it hasn't yet actually been flushed to disk.

It is also need to keep track of all the objects that aren't committed yet, so that new transactions see them. This is also simply stored in a persistent sorted map, that is read during the creation of a snapshot and merged with the underlying cache layer.

Transaction layer Every transaction also has a sorted map of all written objects, that is then used to ensure that inside a single transaction, all objects that were put into it can be read back.

5.2.5 Data objects

To make MVCC work, the data objects themselves are expected to be immutable: all the data objects that extend the `JData` interface should never change after they've been constructed. For that, Java's `record` feature is used, together with persistent data structures provided by the `PCollections` library.

5.2.6 Transaction commit

Hooks An important part of the transaction commit procedure is running all the registered pre-commit hooks.

Every hook should see the changes made to every object not only by the transaction, but by other hooks too. For that, the pending list of changes is tracked for each hook individually, together with the versions of the objects each of them should see. As transactions are expected to be small, this solution has proven to be adequate so far.

Conflict detection Then, the transaction, when committing, should be checked for conflicts. To do so, locks are taken on the keys of all the objects written or read in the transaction, and this is done in a well-defined order to avoid deadlocks — in natural order of the keys.

Then, after all the relevant objects are locked, it is known that no other transaction can change them, so another snapshot of the underlying storage is taken.

If the snapshot version is the same as the version of the transaction, this means that no other transactions have been committed in the meanwhile, and the checks on object versions can be skipped.

If not, it is needed to verify that none of the objects that the transaction being committed had read or written to, have been updated since it had been started, using the aforementioned write timestamp. If some object in the database is newer, or has been deleted/created compared to the state of the transaction being committed, the transaction is aborted.

This is a simple implementation of serializable snapshot isolation, which is simplified by the fact that the key-value storage itself is not distributed, and taking locks on all the objects during commit is not a particularly expensive operation.

It is also important to note that in my implementation, write skew can still be possible when using iterators. Only individual objects are checked, not ranges of objects, which is, for now, adequate for the purposes of my filesystem. The iterators are used only when accessing file chunks, and any conflict there would materialize itself with the fact that the file object will also be updated with an incremented version vector.

5.3 Synchronization

The main purpose of the Synchronization module is to provide higher-level synchronized data structures to be used by the Filesystem module: namely the Kleppmann tree (provided by the `JKleppmannTreeManager`), and the remote objects as described in subsection 4.8.1.

This section will describe the supporting infrastructure used to implement these data structures.

5.3.1 Operation sender

For both remote objects, and the Kleppmann tree, there has to be some way to send updates of the state of these data structures to other peers: updates of version vectors for remote objects, tree operations for the Kleppmann tree.

Specifically, it is possible to abstract away the specifics of what is being synchronized in an `Op` class, which represents ”something” that can be sent to another peer. The operations are then ”generated” by objects: for the Kleppmann tree, there is a root object in the database, that stores all the operation that have to be sent to other peers. Thus, when this object is changed, it is known that something probably has to be sent to other peers. Analogously, when a file is modified, the change is also bound to a specific object in the storage.

A viable abstraction then seems to be to create a queue of database objects, from which the operations will be ”extracted” to be sent to other peers. The `InvalidationQueueService` will provide that functionality.

Then, `pushInvalidationToOne` and `pushInvalidationToAll`, are its outside interface: these functions can be called when some object changes, and that will asynchronously send the required operations extracted from the objects to other peers.

On filesystem shutdown it also writes the list of objects, that were not sent yet to the disk, and reads them back on startup. If a crash is detected, all the objects in storage are put into the invalidation queue on startup.

Crash detection is provided by the `ShutdownChecker` service, which creates a file on startup and deletes it on shutdown. If a file is found on startup, it means that the last shutdown was irregular (a crash).

To ”extract” the operations from objects, the `OpExtractor` interface (Code listing 5.5) can be implemented for any given object type.

Code listing 5.5 Op extractor definition.

```
public interface OpExtractor<T extends JData> {
    Pair<List<Op>, Runnable> extractOps(T data,
                                        PeerId peerId);
}
```

Given a data object and a peer ID for which the operations should be sent, it will return a list of operations extracted from the object, and a callback to be run when the operations have been sent and confirmed received by the peer.

Accordingly, `OpHandler` (Code listing 5.6) interface also exists to implement the operations on the receiving side.

Code listing 5.6 Op handler definition.

```
public interface OpHandler<T extends Op> {
    void handleOp(PeerId from, T op);
}
```

The implementations of these interfaces are then collected and dynamically dispatched to by the `OpExtractorService` and `OpHandlerService` accordingly. A similar approach is also used for other generic interfaces that can be implemented by the users of the Synchronization layer (such as `DtoMapper`).

The operation interface itself is defined in Code listing 5.7. The method `getEscapedRefs` should return all the references to objects a given operation contains, so that they can be marked as "seen by other peers" for the garbage collection algorithm, which will be described in subsection 5.3.3.

Code listing 5.7 Op definition.

```
public interface Op extends Serializable {
    Collection<JObjectKey> getEscapedRefs();
}
```

5.3.2 Kleppmann tree

For the Kleppmann tree to be able to trim the log, it needs to know that all operations will be newer than a certain timestamp, and it does so by tracking the latest timestamp of operation received per peer.

It follows then, if there's any single peer that never sends any operations, garbage collection will never happen. For this purpose, an additional "dummy" operation type is used as an acknowledgment of receiving operations. When a peer had received operations from another peer, and has no operations of its own to send, it will reply with an operation that only contains its timestamp. Then, it is known that this peer will newer send operations with a timestamp lower than that, and log trimming can be possible.

Otherwise, the implementation of the Kleppmann tree is rather straightforward, using the Kleppmann tree module and providing the required implementations of the interfaces to communicate with the storage layer, and defining the required data types for it and the operation sender to use.

For simplicity, all the operation logs and queues for operation sending are stored in a single object: `JKleppmannTreePersistentData`. Though, for better performance, in future this should use the map as described in subsection 4.8.3, or something similar.

5.3.3 Reference counting

`JDataRefcounted` interface is provided in the module as an extension of the `JData` interface. Using transaction hooks, it can automatically update the references to other objects, and to delete them when no reference to an object exist.

Remote object deletion is handled specially. As discussed earlier, to delete a remote object, confirmation from all the peers that it is indeed safe to delete is required.

Thus, if local reference count of a remote object is dropped down to zero, it is put into asynchronous deletion queue.

Chunks are remote objects, and when writing to a file many chunks that are deleted almost immediately can be generated. A simple optimization then, is not to require a confirmation for deletion of objects that themselves, or any references to them, were sent to other peers. This fact is tracked in remote object metadata's `wasSeen` field. It is updated not only when sending an update to that object itself, but also when sending other operations that might expose the reference to a remote object.

With that, all objects that are put into deletion queue and haven't been seen by other peers before are immediately deleted. If an object has been seen by other peers however, it will be subject to a possibly lengthy deletion mechanism.

For a given object, all other peers are asked if it's possible to delete it, and the responses are recorded. If on some remote peer the object is not considered to be a deletion candidate, it sends a list of object keys that refer to the object, and these objects are downloaded locally.

This is done to avoid an infinite cycle of asking other peers for deletion information. Otherwise, for example, a chunk object that is referred to from a file that isn't downloaded locally will have a local reference count of zero, so the file chunk list needs to be downloaded to know the chunks shouldn't be garbage collected. These state transitions can be described in a UML *state diagram*, that is shown in Figure 5.3.

▶ **Definition 5.1** (State diagram)**.** *A state machine diagram models the behavior of a single object, specifying the sequence of events that an object goes through during its lifetime in response to events [67].*

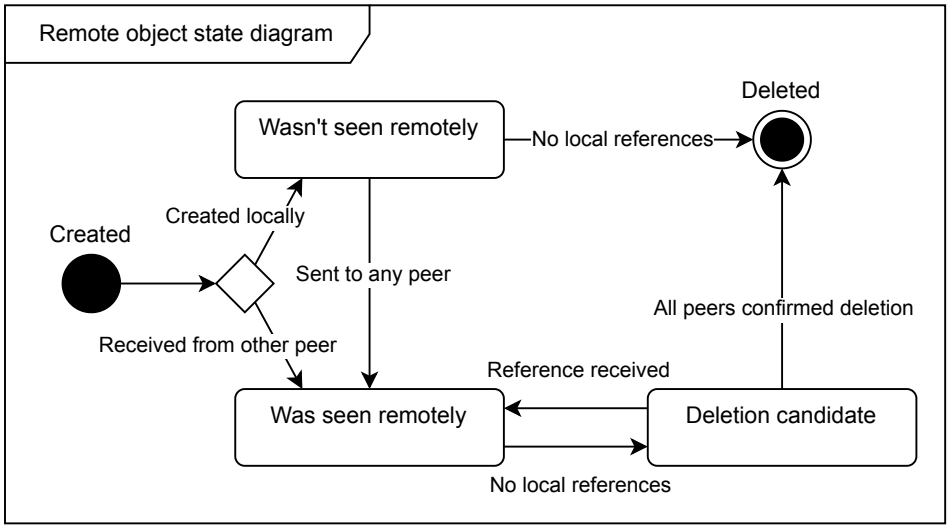

Figure 5.3 Remote object deletion state diagram.

5.3.4 Automatic downloading

A simple way to download files in the background is also implemented. The `AutosyncProcessor` is responsible for that.

It is used for the aforementioned automatic downloading of deletion candidate references, and can be used for automatic downloading of all the files and their data, if the user so desires (for example, on a dedicated backup server).

5.3.5 Peer communication

For peer communication, I've chosen `gRPC` framework.

It has good support in Quarkus, and, has a very convenient solution for defining the request and response types using Protobuf serialization format: implementing remote functions and calling them is not much harder than implementing local calls. Support for sending binary data, such as file chunks, is also native to the protocol and can be mixed arbitrarily in the message types, which would not be the case if I used, for example HTTP with JSON.

To ensure security of the communication, to avoid someone listening in to the transmitted data or impersonating a known peer, mutual TLS is used. Each peer has a self-signed certificate that is used to verify its identity, which is shared with other peers.

To keep track of all the peers, and synchronize information about them, the Kleppmann tree is reused. The peers then are stored in the root of the tree, identified by their IDs, and the tree nodes, using a custom metadata format (`JKleppmannTreeNodeMetaPeer`), then refer to a `PeerInfo` object Code listing 5.8.

Code listing 5.8 PeerInfo data definition.

```
public record PeerInfo(JObjectKey key, PeerId id,
                       ByteString cert,
                       PMap<PeerId, Long> kickCounter,
                       long lastSeenTimestamp)
    implements JDataRemote, JDataRemoteDto {...}
```

For the purposes of detecting unavailable peers and ignoring them during garbage collection, `kickCounter` and `lastSeenTimestamp` are used to implement the algorithm described in section 3.5, together with a special `ObjSyncHandler` implementation: `PeerInfoSyncHandler`.

`PeerId` right now is a simple wrapper for a `JObjectKey`, to ensure only real peer IDs are passed to functions that expect it.

The communication format is then defined by four remote calls (Code listing 5.9).

Code listing 5.9 gRPC calls definition.

```
service DhfsObjectSyncGrpc {
rpc OpPush (OpPushRequest) returns (OpPushReply) {}

rpc GetObject (GetObjectRequest) returns (GetObjectReply) {}
rpc CanDelete (CanDeleteRequest) returns (CanDeleteReply) {}

rpc Ping (PingRequest) returns (PingReply) {}
}
```

Their roles are as follows:

OpPush Contains a list of serialized `Op`s, used to send updates to other peers about local object changes.

GetObject Used to download remote objects. The request contains the key of the remote object to download. The reply contains the serialized object, together with its version vector.

CanDelete Used in the garbage collection algorithm of remote object. The request contains the key of remote object that is considered a deletion candidate locally. The reply contains whether it also is considered a deletion candidate by the called peer, and if it is not, a list of object keys that refer to it on that peer.

Ping Empty message with an empty response, used to verify peer connectivity.

5.3.6 Peer discovery

There needs to be a way to find peers that user can add to the cluster, and to find addresses for communicating with already added peers.

For that, the `PeerDiscoveryDirectory` service exists. It receives messages of known peer addresses, and keeps track of them, removing stale addresses after a configurable timeout.

Then, for the purpose of finding peers to be added to the cluster, it supports giving out all the addresses it remembers. And, for the purpose of connecting to an existing peer, it can return addresses for specific peers too.

There are three peer discovery methods are implemented right now: static discovery, manual discovery and LAN discovery over broadcast.

Static discovery Expected to be mostly used for testing purposes, a peer's address and port for communication can be specified manually as a command-line option.

Manual discovery It is possible to specify a peer's address manually, through the user interface.

LAN discovery Via local network broadcast, it is possible to automatically detect peers on the same local network.

Internet discovery Though not implemented right now, in the future, a proxy or some kind of NAT traversal can be used for the filesystem to easily work over the internet.

It is also required not only to keep track of not only known peer addresses, but also of the peers that are actually connected right now. For that, `ReachablePeerManager` service exists.

It periodically collects the best addresses it can get from the `PeerDiscoveryDirectory`, and tries connecting to them. If it does, it remembers the connection, and then regularly sends a heartbeat message to verify that the peer is still connected.

The "best" address is chosen by the following order: LAN address is better than an Internet address, which is better than an address through a proxy. The type of the address is provided by the discovery source.

On the connection and disconnection event, it also runs registered event listeners. For example, they can be used by the `InvalidationQueueService` to stop trying to send updates to a disconnected peer, and start retrying when a peer is connected.

5.4 User interface

A simple HTTP API was implemented that allows the following:

- Reading information about the peer itself: its unique ID and its TLS certificate.
- Reading a list of peers that are known and can be added to be synchronized with.
- Adding, removing, and listing peers that are synchronized with already (known peers).
- Adding, removing, and reading communication addresses for known peers (the aforementioned "Manual discovery").

For the user interface itself, I've chosen to create a web interface using React [68] with React Router [69].

A web interface was chosen as it is cross-platform, will allow to install the filesystem on a server without a graphical desktop, and because I was already familiar with the chosen technologies.

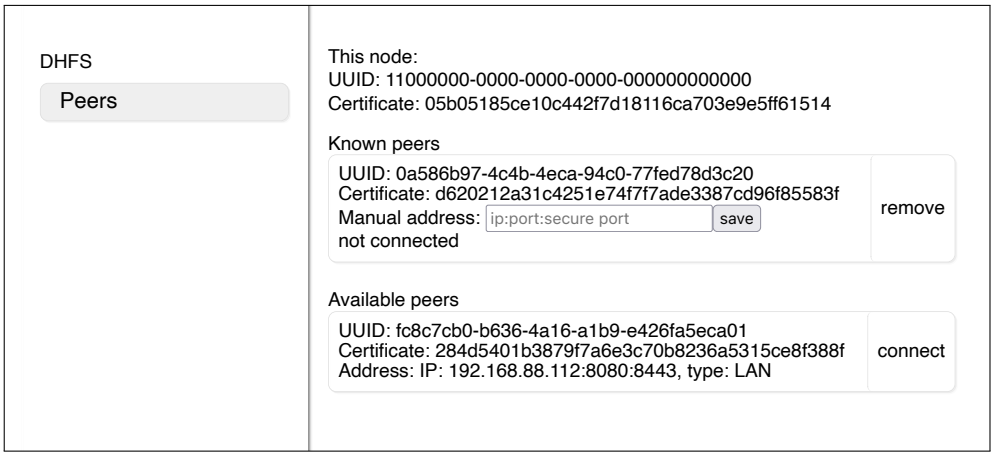

Figure 5.4 DHFS Web interface.

5.5 Packaging

To let the user start, stop, and update the application, a set of simple shell scripts were created.

The `run` script starts the java application with all the required arguments, and creates a file with the PID of the started application, so that it knows it has already been started, and can avoid starting the application multiple times.

The `stop` script reads the PID file created by the run script, and sends a signal to it to stop the application.

The `update` script checks what is the latest build of the filesystem available on GitHub using `https://nightly.link` service. If the available version is newer than what's downloaded, it will download it and replace the local installation with the updated version.

These scripts are also duplicated in PowerShell to be used on Windows.

5.6 Development process

During development, of course, version control system was used, specifically Git, arguably the most popular and the most used one. The repository for the filesystem is public and is available at `https://github.com/usatiuk/dhfs`.

To improve confidence in my code, and to produce usable artifacts continuously, CI/CD pipeline using GitHub Actions was created, automatically running the tests and producing usable binaries on every push to the `main` branch.

The interfaces of the Java objects are documented using Javadocs, a format that allows the documentation to be in code, and automatically extracted to a human-readable HTML form.

Chapter 6
Testing

In this chapter, I will discuss the testing methods used to test the filesystem, with a focus on how user data safety is tested.

For a filesystem, the most important thing is to keep user's data safe and free of corruption under any circumstances. In the case of a peer-to-peer filesystem, it is also very important to handle conflicts reasonably, again, without any data loss. This will be the main focus when testing the filesystem.

6.1 End-to-end tests

End-to-end tests verify that the filesystem, as shipped to the end user, works as intended. Complete instances of the filesystems are started in separate Docker containers, connected with a shared network. Then, the interaction with the filesystem is the same as an end user would interact with it: using the FUSE filesystem and the web API to add and remove peers.

This can also be used to test proper conflict handling and process crash recovery, various problematic scenarios can be created, and it can be verified that the filesystem is in a consistent state after, on all of the peers.

Conflicts Let's remember the conflict scenarios should be tested. A conflict happens when two nodes create, move, or update the same files concurrently.

Thus, testing for their correct resolution is quite straightforward.

Multiple instances of the filesystem are started, the peers are introduced to each other, and then disconnected from network.

Then, on each peer, independently of each other, filesystem operations are executed, which specifically depends on which conflict is being tested.

For file-level conflict, a file is created before the peers are disconnected. Then, while disconnected, each peer writes a unique string to the file. After peers are reconnected again, it is verified that all versions of the file are pre-

served, and that the contents of the filesystem are exactly the same for every peer.

For directory-level conflict, a file with the same name and different contents is created on each peer after they are disconnected. After reconnection, again, consistency level and the preservation of each file version is verified.

Process crash To test filesystem consistency in an event of the process crash, Docker can also be used. Then, a process crash can be simply simulated by killing the container.

First, two peers are brought up into a cluster. Then, multiple scenarios are tested.

In one, a file is being written to continuously by some process, writing an increasing sequence of numbers to the file. While it is being written to, one of the peers crash. Then, after the crashed peer is restarted, the file consistency is verified: all files that were created should still be readable, and their states should be the same on every peer.

In another scenario, consecutively numbered files are being created in the filesystem by a concurrent process. Like in the first scenario, one of the peers killed and restarted, and the filesystem consistency is verified.

Computer crash Testing a computer crash is less straightforward than testing a process crash. The most obvious way to test it would be to run the filesystem on multiple physical computers and then physically turn them off by, for example, disconnecting them from power.

Unfortunately, this is non-trivial to automate, and, fortunately, a better solution exists that can be easily run automatically without dedicated physical hardware.

LazyFS [70] is a FUSE filesystem specifically designed to test databases. It presents a POSIX-compatible filesystem that the filesystem can normally run on, but with ability to inject various types of failures that a real power failure can produce.

It can test not only simple power failures when all written but not explicitly flushed data is lost, but also scenarios where only a part of data written was persisted on disk.

Lost writes Data was written, but not flushed to the disk is lost.

Linear torn writes Multiple pages of data were written, and some of the later writes were not persisted.

Non-linear torn writes Multiple pages of data were written, and an arbitrary subset of them was not persisted.

Tests for all three of the types of failures are similar to the tests simulating a process crash. But instead of a process crash, LazyFS is asked to crash instead, and to corrupt the data in one of the specified ways.

These tests, like the process crash tests, are also integrated into the CI test suite.

Testing of crash testing To verify that crash tests work, I intentionally left a race condition in the peer-to-peer communication code.

Specifically, it was possible that a peer would send, for example, the information about a newly created file, that wasn't flushed to the disk yet, to another peer. If the first peer were to crash, it would lose the data about the file, and an unreadable corrupted file would appear on the second peer.

As expected, this bug was found by the crash testing and then fixed (by waiting for all of the referred objects of a sent operation to be flushed to disk before actually sending it).

No data corruption bugs were found, as expected, as LMDB is already known to be a robust data storage library.

Deletion testing In addition to the previous tests, multiple deletion tests exist, that test that the garbage collection mechanisms do, indeed, work. A file is created and then deleted, and it is verified through the log messages that the objects representing this file were deleted from the object storage.

Not only the happy scenario with all nodes online is tested, a test scenario where one node is unavailable for a period of time exists, and it is expected that in that case garbage collection will still happen, and the nodes will resynchronize after being connected again, as described in section 3.5.

6.2 Integration testing

Integration tests for the filesystem and object storage layers are also present in the codebase. They test a larger part of the system, interaction between its different components, but not a complete system as end-to-end tests do.

Filesystem tests A set of integration tests test the Filesystem layer, testing basic filesystem functionality and various edge cases I had found. FUSE layer is also tested, by mounting the filesystem into a temporary directory and verifying some basic functionality as well.

Storage tests Storage layer is also extensively tested by the integration tests: creating, deleting, reading, and iterating over objects, including various multithreaded scenarios.

6.3 Unit testing

Unit tests are also used, testing various smaller parts of the system (isolated units) on their own.

Kleppmann tree The Kleppmann tree has a couple of simple tests in its module. *Test doubles* of the required dependencies are provided (independently implementing the required functionality in a simpler way compared to the real implementation), and simple conflict resolution and synchronization tests are performed.

Storage layer In the storage layer, one of the important things to test are the merging iterators. For that I've used property-based testing library jqwik [71]. It allows to generate random testing data, and to verify that for all of it, the iterator conforms to some property.

For example, I will describe a test of the merging iterator. The property that is being tested is the fact that it merges multiple provided iterators in a correct way, taking into account their priority. This is achieved by creating a simpler model of the iterator, in this case, a single map iterator containing manually merged data. Then, random actions are performed on both the "ideal" model of the iterator, and the real merging iterator, and it is verified that their outputs are exactly the same. This specific testing method can be called model-based testing [72].

Chapter 7
Evaluation

In this chapter, I will evaluate the practical usefulness of the filesystem, its performance, day-to-day usability.

7.1 Performance

I have tested the read/write performance of the filesystem on a local device and when streaming files from a peer over local network.

Local performance I have tested the performance of the filesystem locally using the `iozone` [73] benchmark suite.

On Figure 7.1, the results of measuring write/read speeds with different operation sizes are shown (working with a total of 1 GB of data)

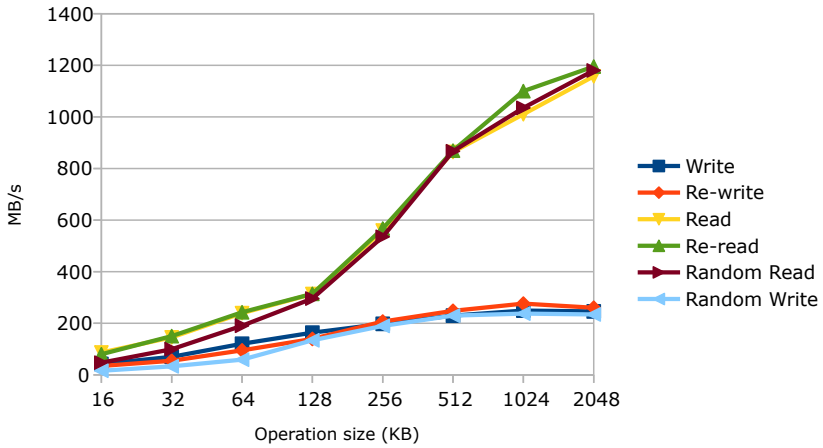

Figure 7.1 Benchmark results with different operation sizes.

As expected, performance with small chunk sizes is relatively slow — the

overhead of individual filesystem operation calls is quite high, and this is something that could be definitely improved in future iterations of filesystem. The read performance scales well with the chunk sizes, while write performance eventually stops increasing. My assumption is that instead of the `write` call, the process of serialization of data and writing to the disk becomes the bottleneck (which is done only by a single thread).

On Figure 7.2 results of testing with increasing parallel process counts (each one working with a separate 64 MB file, and using 16 KB operation sizes) are shown.

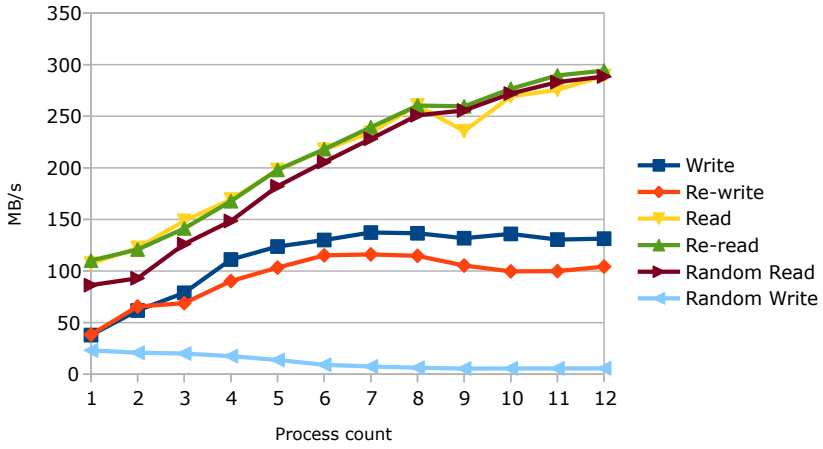

Figure 7.2 Benchmark results with different process counts.

The performance increases quite steadily with multiple processes, though not linearly, which I assume is due to the fact that transaction commit is still only single-threaded, so eventually write operations reach a performance ceiling at about 7 threads. Interestingly, random writes only get slower with increasing numbers of threads, which I assume is due to the write amplification of the random writes (a 16 KB write must rewrite an entire 128 KB chunk), and the write cache becomes the bottleneck quickly.

The benchmarks were done on a Lenovo IdeaPad 16ACH6 laptop with an AMD Ryzen 5800H CPU, 64 MB read cache size, 16 MB write cache size, 512 MB JVM heap size, 128 KB file chunk alignment (default settings). Both benchmarks had 3 runs for each data point, and the results were averaged. The laptop's disk is a Samsung `PM981a` NVMe SSD, with sequential read and write speeds of about 2 GB/s, and random read/write speeds using 4 KB chunks of about 60 MB/s as tested by the CrystalDiskMark benchmark suite [74].

Network performance I have tested network performance of the filesystem by copying a roughly 380 MB set of photos to the filesystem on a remote peer, and copying the entire directory to a local folder on another (Code listing 7.1).

Code listing 7.1 Copy over network speed test.

```
Transferred:  383.604 MiB / 383.604 MiB, 100%, 21.662 MiB/s
Transferred:      161 / 161, 100%
Elapsed time:    17.9s
```

The resulting performance definitely could be improved, and is probably the result of a lack of any kind of read-ahead mechanism from the chunks, each chunk has to be downloaded synchronously as the file is being read.

Streaming performance I had tested the file streaming by copying a large video file to the filesystem on a remote peer, and trying to play it locally. As expected, the file had opened and playback had started instantly, and seeking through the video was responsive, indicating that the streaming aspect of the filesystem does indeed work.

Day-to-day usage I had tried to use the filesystem to synchronize the git repository used to write this document between two laptops, and switching between them while writing it. I had not encountered any problems doing so.

7.2 Future work

In my opinion, the results are rather promising, and show that it is indeed possible to build a peer-to-peer streaming filesystem. Though, of course, there are still some unsolved problems and possible improvement on the implementation side.

Garbage collection The most obvious algorithmic weakness is the garbage collection algorithms used. In particular, the requirement for all peers to be online and responsive for any garbage collection to happen at all, or risking data corruption if done without all peers' confirmation. I think it could be possible to extend the "seen" object tracking to track not only if it had been seen by any peer, but also by what peers it had been seen. And as such, require only confirmation of these peers to collect the object.

Performance Performance of the filesystem can definitely be improved: by not requiring deserializing chunks when reading files, the read performance could be improved, and streaming performance could be improved by implementing some kind of read-ahead mechanism.

Interface While a web interface and a simple script set to run the filesystem is enough for it to work, it would certainly be a nicer user experience to have a "proper" application-style packaging of the filesystem.

Chapter 8
Conclusion

The main goals of this thesis were to analyze the existing solutions to the problem of sharing files between devices (distributed filesystems and file synchronization tools), to design a filesystem that will be peer-to-peer and that will stream files, implement, and test it with a focus on keeping user data safe. I think that these goals have been successfully achieved.

Analysis of existing solutions I have examined the existing solutions to the problem, and, while some of the solutions were quite close to being perfect, such as Google Drive File Stream having very well working file streaming, Syncthing having practically proven peer-to-peer synchronization model, and ElmerFS using less-than-perfect CRDT data structures, none of them have all the properties that I wanted. Existing distributed filesystems were also researched, with some useful learning from their implementation and design.

Yet this analysis, in my opinion, had shown, that the properties of being peer-to-peer and allowing file streaming are desirable, as both of them are selling points of multiple separate products, and that trying to combine them could be a worthy endeavor.

Design, Implementation I believe my software design is up to the standards of modern software engineering. The design is modular and extensible: the Kleppmann tree, Storage layer, and Synchronization layer, I believe, are quite independent and well tested on their own, and could be used by other people in future projects.

In fact, most of the work was done in these layers - the filesystem and filesystem interface layers are only 1200 lines of code, compared to roughly 8000 lines of code in the other layers (excluding tests).

Testing The solution is extensively tested, both by unit tests, integration tests, and end-to-end tests. A variety of hostile scenarios is also tested, includ-

ing process crashes, computer crashes, synchronization conflicts, and all found bugs were promptly corrected.

Then, I had evaluated my solution, and, while there is definitely room for improvement, I am more than happy with the results. I believe that the direction of peer-to-peer streaming filesystems is something that is worth researching and iterating on in the future.

Appendix A

Storage layer class diagram

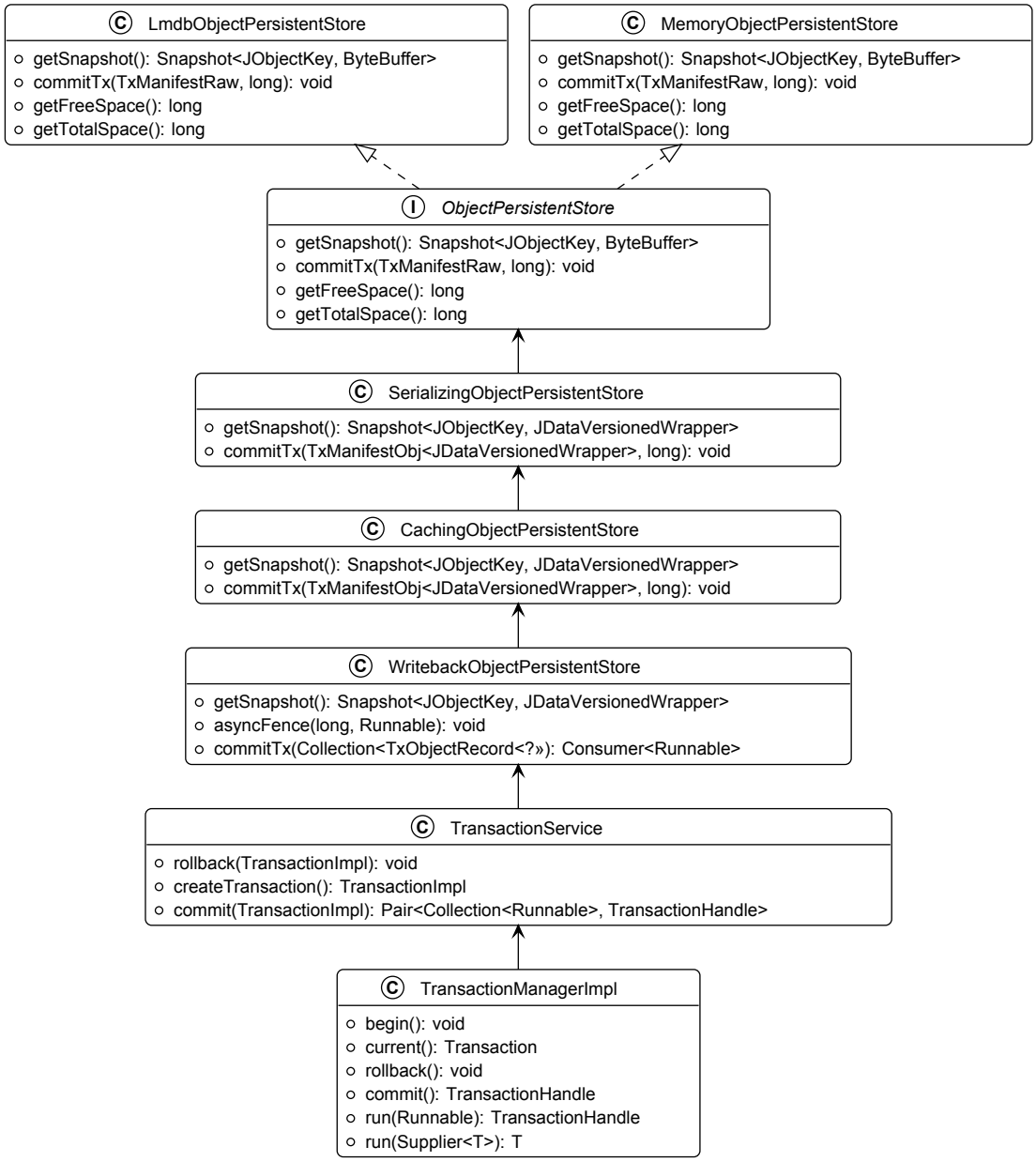

Figure A.1 Overview of the storage layer classes.
Solid arows indicate dependencies between classes, dotted arrows indicate an implementation of an interface.

Bibliography

1. *Dropbox Smart Sync: tips and tricks* [online]. 2018. [visited on 2025-04-20]. Available from: https://assets.dropbox.com/documents/en-us/marketing/SmartSync_tips_and_tricks_March_2018.pdf.
2. ALVIN ASHCRAFT; SAISANG CAI; BLAKE MADDEN; ANDREW MITTEREDER; DAVID COULTER; MCLEAN SCHOFIELD; JOHN LUEDERS; GORDON RUDMAN. *Build a Cloud Sync Engine that Supports Placeholder Files - Win32 apps* [online]. 2021. [visited on 2025-04-20]. Available from: https://learn.microsoft.com/en-us/windows/win32/cfapi/build-a-cloud-file-sync-engine.
3. *Synchronizing files using file provider extensions* [online]. [N.d.]. [visited on 2025-04-20]. Available from: https://developer.apple.com/documentation/fileprovider/synchronizing-files-using-file-provider-extensions.
4. *Stream & mirror files with Drive for desktop - Google Drive Help* [online]. [N.d.]. [visited on 2025-04-20]. Available from: https://support.google.com/drive/answer/13401938.
5. *Use Drive for desktop on macOS - Google Drive Help* [online]. [N.d.]. [visited on 2025-04-20]. Available from: https://support.google.com/drive/answer/12178485.
6. *Syncthing* [online]. [N.d.]. [visited on 2025-04-20]. Available from: https://syncthing.net/.
7. *Resilio Sync | Personal file sync and share powered by P2P* [online]. [N.d.]. [visited on 2025-04-20]. Available from: https://www.resilio.com/sync/.
8. *Amazing Software for Cloud Storage* [online]. [N.d.]. [visited on 2025-04-20]. Available from: https://www.expandrive.com/.
9. *Mountain Duck* [online]. [N.d.]. [visited on 2025-04-20]. Available from: https://mountainduck.io/.

10. *Cloud storage manager – Manage multiple cloud services | CloudMounter for Mac & Win* [online]. [N.d.]. [visited on 2025-04-20]. Available from: https://cloudmounter.net/.

11. *Rclone* [online]. [N.d.]. [visited on 2025-04-20]. Available from: https://rclone.org/.

12. WEIL, Sage A.; BRANDT, Scott A.; MILLER, Ethan L.; LONG, Darrell D. E.; MALTZAHN, Carlos. Ceph: a scalable, high-performance distributed file system. In: *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*. Seattle, Washington: USENIX Association, 2006, pp. 307–320. OSDI '06. ISBN 1931971471.

13. GHEMAWAT, Sanjay; GOBIOFF, Howard; LEUNG, Shun-Tak. The Google file system. In: *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*. Bolton Landing, NY, USA: Association for Computing Machinery, 2003, pp. 29–43. SOSP '03. ISBN 1581137575. Available from DOI: 10.1145/945445.945450.

14. GEORGE, Anjus; MOHR, Rick; SIMMONS, James; ORAL, Sarp. *Understanding Lustre Internals. Second Edition* [online]. 2021-09. [visited on 2025-04-20]. Tech. rep., ORNL/TM-2021/2212, 1824954. Available from DOI: 10.2172/1824954.

15. SHVACHKO, Konstantin; KUANG, Hairong; RADIA, Sanjay; CHANSLER, Robert. The Hadoop Distributed File System. In: *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*. 2010, pp. 1–10. Available from DOI: 10.1109/MSST.2010.5496972.

16. GUY, Richard George. *FICUS: a very large scale reliable distributed file system*. USA: University of California at Los Angeles, 1992. PhD thesis. UMI Order No. GAX92-06681.

17. SATYANARAYANAN, M.; KISTLER, J.J.; KUMAR, P.; OKASAKI, M.E.; SIEGEL, E.H.; STEERE, D.C. Coda: a highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*. 1990, vol. 39, no. 4, pp. 447–459. Available from DOI: 10.1109/12.54838.

18. BENET, Juan. *IPFS - Content Addressed, Versioned, P2P File System*. 2014. Available from arXiv: 1407.3561 [cs.NI].

19. VAILLANT, Romain; VASILAS, Dimitrios; SHAPIRO, Marc; NGUYEN, Thuy Linh. CRDTs for truly concurrent file systems. In: *Proceedings of the 13th ACM Workshop on Hot Topics in Storage and File Systems*. Virtual, USA: Association for Computing Machinery, 2021, pp. 35–41. HotStorage '21. ISBN 9781450385503. Available from DOI: 10.1145/3465332.3470872.

20. SOMMERVILLE, Ian. Software engineering. In: 9th ed. Boston: Pearson, 2011, pp. 84–91. ISBN 978-0-13-703515-1. OCLC: 462909026.

21. SOMMERVILLE, Ian. Software engineering. In: 9th ed. Boston: Pearson, 2011, pp. 106–107. ISBN 978-0-13-703515-1. OCLC: 462909026.

22. SHAPIRO, Marc; PREGUIÇA, Nuno; BAQUERO, Carlos; ZAWIRSKI, Marek. Conflict-free Replicated Data Types. In: DÉFAGO, Xavier; PETIT, Franck; VILLAIN, Vincent (eds.). *Lecture Notes in Computer Science*. Grenoble, France: Springer, 2011, vol. 6976, pp. 386–400. Lecture Notes in Computer Science. Available from DOI: 10.1007/978-3-642-24550-3_29.

23. RITCHIE, Dennis M.; THOMPSON, Ken. The UNIX time-sharing system. *Commun. ACM*. 1974, vol. 17, no. 7, pp. 365–375. ISSN 0001-0782. Available from DOI: 10.1145/361011.361061.

24. KLEPPMANN, Martin; MULLIGAN, Dominic P.; GOMES, Victor B. F.; BERESFORD, Alastair R. A Highly-Available Move Operation for Replicated Trees. *IEEE Transactions on Parallel and Distributed Systems*. 2022, vol. 33, no. 7, pp. 1711–1724. Available from DOI: 10.1109/TPDS.2021.3118603.

25. HUGHES, John; PIERCE, Benjamin C.; ARTS, Thomas; NORELL, Ulf. Mysteries of DropBox: Property-Based Testing of a Distributed Synchronization Service. In: *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. 2016, pp. 135–145. Available from DOI: 10.1109/ICST.2016.37.

26. TAO, Vinh; SHAPIRO, Marc; RANCUREL, Vianney. Merging semantics for conflict updates in geo-distributed file systems. In: *Proceedings of the 8th ACM International Systems and Storage Conference*. Haifa, Israel: Association for Computing Machinery, 2015. SYSTOR '15. ISBN 9781450336079. Available from DOI: 10.1145/2757667.2757683.

27. NAIR, Sreeja; MEIRIM, Filipe; PEREIRA, Mário; FERREIRA, Carla; SHAPIRO, Marc. *A coordination-free, convergent, and safe replicated tree*. 2022. Available from arXiv: 2103.04828 [cs.DC].

28. LAMPORT, Leslie. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*. 1978, vol. 21, no. 7, pp. 558–565. ISSN 0001-0782. Available from DOI: 10.1145/359545.359563.

29. BAQUERO, Carlos; ALMEIDA, Paulo Sérgio; SHOKER, Ali. Making operation-based CRDTs operation-based. In: *Proceedings of the First Workshop on Principles and Practice of Eventual Consistency*. Amsterdam, The Netherlands: Association for Computing Machinery, 2014. PaPEC '14. ISBN 9781450327169. Available from DOI: 10.1145/2596631.2596632.

30. *Block Exchange Protocol v1 — Syncthing documentation* [online]. [N.d.]. [visited on 2025-04-12]. Available from: https://docs.syncthing.net/specs/bep-v1.html.

31. REIHER, Peter; HEIDEMANN, John; RATNER, David; SKINNER, Greg; POPEK, Gerald. Resolving file conflicts in the Ficus file system. In: *Proceedings of the USENIX Summer 1994 Technical Conference on USENIX Summer 1994 Technical Conference - Volume 1*. Boston, Massachusetts: USENIX Association, 1994, p. 12. USTC'94.

32. PARKER, D.S.; POPEK, G.J.; RUDISIN, G.; STOUGHTON, A.; WALKER, B.J.; WALTON, E.; CHOW, J.M.; EDWARDS, D.; KISER, S.; KLINE, C. Detection of Mutual Inconsistency in Distributed Systems. *IEEE Transactions on Software Engineering* [online]. 1983, vol. SE-9, no. 3, pp. 240–247 [visited on 2025-04-12]. ISSN 1939-3520. Available from DOI: 10.1109/TSE.1983.236733.

33. *Understanding Synchronization — Syncthing documentation* [online]. [N.d.]. [visited on 2025-04-26]. Available from: https://docs.syncthing.net/users/syncing.html.

34. COHEN, Bram. *bep_0003.rst_post* [online]. [N.d.]. [visited on 2025-05-09]. Available from: https://www.bittorrent.org/beps/bep_0003.html.

35. SHAPIRO, Marc; PREGUIÇA, Nuno; BAQUERO, Carlos; ZAWIRSKI, Marek. *A comprehensive study of Convergent and Commutative Replicated Data Types*. 2011-01. Research Report, RR-7506. Inria – Centre Paris-Rocquencourt ; INRIA. Available also from: https://inria.hal.science/inria-00555588.

36. *Node Failure & Recovery — Galera Cluster Documentation* [online]. [N.d.]. [visited on 2025-05-05]. Available from: https://galeracluster.com/library/documentation/recovery.html.

37. *Quarkus Performance* [online]. [N.d.]. [visited on 2025-04-20]. Available from: https://quarkus.io/performance/.

38. *Guide to Working with Multiple Modules – Maven* [online]. [N.d.]. [visited on 2025-04-25]. Available from: https://maven.apache.org/guides/mini/guide-multiple-modules.html.

39. *libfuse: fuse_operations Struct Reference* [online]. [N.d.]. [visited on 2025-04-25]. Available from: https://libfuse.github.io/doxygen/structfuse__operations.html.

40. *FUSE — The Linux Kernel documentation* [online]. [N.d.]. [visited on 2025-04-26]. Available from: https://www.kernel.org/doc./html/next/filesystems/fuse.html.

41. *libfuse/libfuse* [online]. libfuse, 2025 [visited on 2025-04-26]. Available from: https://github.com/libfuse/libfuse. original-date: 2015-12-19T20:27:34Z.

42. *Home - macFUSE* [online]. [N.d.]. [visited on 2025-04-26]. Available from: https://macfuse.github.io/.

43. *WinFsp* [online]. [N.d.]. [visited on 2025-04-26]. Available from: https://winfsp.dev/.

44. TSELOVALNIKOV, Sergey. *SerCeMan/jnr-fuse* [online]. 2025. [visited on 2025-04-20]. Available from: https://github.com/SerCeMan/jnr-fuse. original-date: 2015-05-27T21:42:28Z.

45. PEROT, Etienne. *EtiennePerot/fuse-jna* [online]. 2025. [visited on 2025-04-20]. Available from: https://github.com/EtiennePerot/fuse-jna. original-date: 2012-01-15T01:15:28Z.

46. *Foreign Function and Memory API* [online]. [N.d.]. [visited on 2025-04-20]. Available from: https://docs.oracle.com/en/java/javase/21/core/foreign-function-and-memory-api.html. Publisher: October2024.

47. ISO CENTRAL SECRETARY. *Information technology – Database languages SQL — Part 1: Framework (SQL/Framework)*. Geneva, CH, 2023. Standard, ISO/IEC 9075-1:2023. International Organization for Standardization. Available also from: https://www.iso.org/standard/76583.html.

48. HAERDER, Theo; REUTER, Andreas. Principles of transaction-oriented database recovery. *ACM Comput. Surv.* 1983, vol. 15, no. 4, pp. 287–317. ISSN 0360-0300. Available from DOI: 10.1145/289.291.

49. *Using Hibernate ORM and Jakarta Persistence - Quarkus* [online]. [N.d.]. [visited on 2025-05-06]. Available from: https://quarkus.io/guides/hibernate-orm.

50. *Write Ahead Log (WAL)* [online]. [N.d.]. [visited on 2025-05-06]. Available from: https://github.com/facebook/rocksdb/wiki/Write-Ahead-Log-(WAL).

51. *LMDB: Environment Flags* [online]. [N.d.]. [visited on 2025-05-06]. Available from: http://www.lmdb.tech/doc/group__mdb__env.html.

52. OLSON, Michael A.; BOSTIC, Keith; SELTZER, Margo. Berkeley DB. In: *Proceedings of the Annual Conference on USENIX Annual Technical Conference*. Monterey, California: USENIX Association, 1999, p. 43. ATEC '99.

53. CHU, Howard; CORP, Symas. *MDB: A Memory-Mapped Database and Backend for OpenLDAP*. Tech. rep. Available also from: https://www.openldap.org/pub/hyc/mdm-paper.pdf.

54. *LevelDB: A Fast Persistent Key-Value Store* [online]. [N.d.]. [visited on 2025-05-07]. Available from: https://opensource.googleblog.com/2011/07/leveldb-fast-persistent-key-value-store.html.

55. PILLAI, Thanumalayan Sankaranarayana; CHIDAMBARAM, Vijay; ALAGAPPAN, Ramnatthan; AL-KISWANY, Samer; ARPACI-DUSSEAU, Andrea C.; ARPACI-DUSSEAU, Remzi H. All file systems are not created equal: on the complexity of crafting crash-consistent applications. In: *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*. Broomfield, CO: USENIX Association, 2014, pp. 433–448. OSDI'14. ISBN 9781931971164.

56. *facebook/rocksdb* [online]. Meta, 2025 [visited on 2025-05-07]. Available from: `https://github.com/facebook/rocksdb`. original-date: 2012-11-30T06:16:18Z.

57. BERENSON, Hal; BERNSTEIN, Phil; GRAY, Jim; MELTON, Jim; O'NEIL, Elizabeth; O'NEIL, Patrick. A critique of ANSI SQL isolation levels. *SIGMOD Rec.* 1995, vol. 24, no. 2, pp. 1–10. ISSN 0163-5808. Available from DOI: `10.1145/568271.223785`.

58. *Snapshot Isolation* [online]. [N.d.]. [visited on 2025-04-21]. Available from: `https://jepsen.io/consistency/models/snapshot-isolation`.

59. *13.2. Transaction Isolation* [online]. 2025. [visited on 2025-04-21]. Available from: `https://www.postgresql.org/docs/17/transaction-iso.html`.

60. *MySQL :: MySQL 8.4 Reference Manual :: 17.7.2.1 Transaction Isolation Levels* [online]. [N.d.]. [visited on 2025-04-21]. Available from: `https://dev.mysql.com/doc/refman/8.4/en/innodb-transaction-isolation-levels.html`.

61. DRISCOLL, J R; SARNAK, N; SLEATOR, D D; TARJAN, R E. Making data structures persistent. In: *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing*. Berkeley, California, USA: Association for Computing Machinery, 1986, pp. 109–121. STOC '86. ISBN 0897911938. Available from DOI: `10.1145/12130.12142`.

62. HAROLD. *hrldcpr/pcollections* [online]. 2025. [visited on 2025-04-26]. Available from: `https://github.com/hrldcpr/pcollections`. original-date: 2014-03-20T09:28:01Z.

63. *Iterator Implementation* [online]. [N.d.]. [visited on 2025-04-25]. Available from: `https://github.com/facebook/rocksdb/wiki/Iterator-Implementation`.

64. *A5B (Write Skew)* [online]. [N.d.]. [visited on 2025-04-21]. Available from: `https://jepsen.io/consistency/phenomena/a5b`.

65. KLEPPMANN, Martin. Designing Data-Intensive Applications. In: Sebastopol, CA: O'Reilly Media, 2017, pp. 246–251. ISBN 978-1-4919-0310-0.

66. YABANDEH, Maysam; GÓMEZ FERRO, Daniel. A critique of snapshot isolation. In: *Proceedings of the 7th ACM European Conference on Computer Systems*. Bern, Switzerland: Association for Computing Machinery, 2012, pp. 155–168. EuroSys '12. ISBN 9781450312233. Available from DOI: 10.1145/2168836.2168853.

67. *State Machine Diagram - UML 2 Tutorial | Sparx Systems* [online]. [N.d.]. [visited on 2025-04-15]. Available from: https://sparxsystems.com/resources/tutorials/uml2/state-diagram.html.

68. *React* [online]. [N.d.]. [visited on 2025-04-25]. Available from: https://react.dev/.

69. *React Router Official Documentation* [online]. [N.d.]. [visited on 2025-04-25]. Available from: https://reactrouter.com/.

70. RAMOS, Maria; AZEVEDO, João; KINGSBURY, Kyle; PEREIRA, José; ESTEVES, Tânia; MACEDO, Ricardo; PAULO, João. When Amnesia Strikes: Understanding and Reproducing Data Loss Bugs with Fault Injection. *Proc. VLDB Endow.* 2024, vol. 17, no. 11, pp. 3017–3030. ISSN 2150-8097. Available from DOI: 10.14778/3681954.3681980.

71. *jqwik* [online]. [N.d.]. [visited on 2025-04-26]. Available from: https://jqwik.net/.

72. JOHANNES LINK. *Model-based Testing* [online]. [N.d.]. [visited on 2025-04-26]. Available from: https://johanneslink.net/model-based-testing/.

73. *Iozone Filesystem Benchmark* [online]. [N.d.]. [visited on 2025-05-09]. Available from: https://www.iozone.org/.

74. *CrystalDiskMark - Crystal Dew World [en]* [online]. 2018. [visited on 2025-04-26]. Available from: https://crystalmark.info/en/software/crystaldiskmark/,%20https://crystalmark.info/en/software/crystaldiskmark/.

Contents of the attachment

```
/
├── readme.txt ..... short description of the contents
├── declaration.pdf ..... signed thesis declaration from KOS
├── exe ..... directory with a packaged implementation of the filesystem
├── docs ..... Javadoc documentation of the system
├── src
│   ├── impl ..... source code of the implementation
│   └── thesis ..... source code of the thesis in LaTeX format
└── text ..... text of the thesis
    ├── thesis.pdf ..... text of the thesis in PDF format
    └── thesis-print.pdf ... text of the thesis in PDF format for printing
```