A peer-to-peer virtual filesystem for publicly accessible content

James Ravindran

Master of Computing in Computer Science with Honours
The University of Bath
2022-2023

A peer-to-peer virtual filesystem for publicly accessible content

Submitted by: James Ravindran

Abstract

Consumers are faced with fixed amounts of storage in their devices, which conflicts with their desire to access multimedia content that has increased in file size. The higher adoption and the faster speeds of Internet access have brought opportunities in the form of the on-demand streaming of content which more users are migrating to. However, existing solutions are restricted by using a client-server model that causes a bottleneck due to a dependency on a single content provider, or those that use peer-to-peer models experience file duplication or issues with discovery of content. None satisfies the requirements for a system that is able to distribute singular files between peers that can be identified using a singular identifier and which partial read requests are serviced in real-time. This project aims to resolve this by providing a comprehensive prototype solution that offers a custom peer-to-peer content addressable protocol to distribute files between peers, and a virtual filesystem that exposes these files to users and applications.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background of the problem

Users are likely to own a variety of devices that can access multimedia content, such as laptops, mobile phones and gaming consoles (Laricchia, 2022). Over the last decade however, there has been an exponential increase in the file sizes of multimedia content (Dinneen and Nguyen, 2021). Despite a reduction in the price of secondary storage (Kryder and C.S. Kim, 2009), many of the latest devices such as smartphones usually have a non-extendable and fixed amount of storage (H. Kim, Agrawal, and Ungureanu, 2012). This means that the amount of storage is restricted by the decision the consumer made at purchase, who according to research is also more likely to buy a device with a smaller storage capacity (Mohd Suki, 2013) due to the increase in price with higher storage models. Therefore, these factors compound the issue of growing file sizes for multimedia.

On the other hand, there is currently a prevalence in the adoption of high-speed Internet access (Tan and Teo, 1998), and therefore content providers have reacted by providing numerous on-demand streaming services to serve this growing audience of customers (Calboli, 2022). They usually utilise a client-server model to serve content from centralised servers, which has placed considerable strain on content providers to provide high-bandwidth connections and availability, which existing protocols such as the Hypertext Transfer Protocol or HTTP (and its successor HTTP/2) have not been designed for (Bishop, 2022).

In response, user communities have adopted alternative solutions that utilise peer-to-peer models (Cunningham, Alexander, and Adilov, 2004), therefore allowing end-users to participate and assist in the distribution of content and resulting in a substantial offload in bandwidth from the content provider. In terms of peer-to-peer file sharing protocols, BitTorrent and the InterPlantary File System (IPFS) are prominent ones that have seen significant usage. In academia, several authors have proposed their own solutions, such as the SFS read-only file system (Fu, Kaashoek, and Mazières, 2002) and the Andrew File System or AFS (John Howard, 1988). None of these solutions (as I have also seen with client-server protocols) are able to address all the issues faced as they only partially solve them, as I shall explore in the literature, technology and data survey chapter.

1.2 Proposed solution

I propose a solution to address this problem exhaustively via the creation of two separate parts. The first part will be the design and reference implementation for a custom peer-to-peer content-addressable file sharing protocol to address limitations with current existing protocols, while the second part is the creation of a virtualised filesystem built on top of the custom file sharing protocol to function as a front-end. Unlike conventional filesystems however, the filesystem will be read-only with the files being displayed determined by the implementation. The user will specify files they want to receive, and other peers will provide the clusters the user wants over the network. User applications will be able to read from the filesystem as normal - when the operating system sends a read request, the implementation will be instructed to download the part of the file the cluster belongs to and then respond with the necessary binary content for that file. The user can specify which files are to be made available via a command-line interface.

1.3 Aims

This project aims to provide an all-encompassing solution that addresses all the issues faced that no existing solutions have resolved entirely. This solution will attempt to allow users to stream multimedia content while meeting the following:

- Users have low and fixed amount of local storage.
- Users have high-speed and high-availability Internet connections.
- Users wish to access read-only publicly accessible content.
- Users should be able to download a file from other peers (via a peer-to-peer instead of client-server model).
- Files should be split into pieces for distribution.
- Individual files should be distributed so that duplication is eliminated.
- The files should appear to be mounted natively so that existing applications can read them with no changes required.

1.4 Objectives

- Create a distributed content-addressable file sharing protocol and corresponding implementation.
- Create a simple virtualised filesystem as a frontend to the file sharing implementation.
- Create a command-line interface to allow the user to select which publicly accessible files are to be made available.
- Implement pre-existing solutions' protocols (such as IPFS) as an alternative to the default file sharing protocol used in the implementation.
- Benchmark (in terms of network usage and speed) the implementation's file sharing protocol in contrast with pre-existing solutions such as IPFS.

1.5 Novelty

The problem this project addresses is worthy of study because it is a growing, inhibitive and multi-faceted issue, which can only be resolved with a new solution as existing ones are insufficient to fully address all the aims. This project is novel because it delivers all the following:

- The implementation is optimised for low storage - that is, if storage is full, older downloaded chunks will be deleted and if they needed again they will be re-downloaded given available space. This is contrast to some other solutions, which experience issues such as disk full errors (Gierth, 2017).
- The implementation's file sharing protocol is purely content addressable - individual file contents are hashed only. Some other solutions use an intervening representation which can lead to duplication (John, 2022).
- The implementation only supports read-only publicly accessibly content. Some other solutions unnecessarily introduce authentication or multi-user access which is redundant in this context (Morris et al., 1986).
- The implementation downloads a file from multiple peers which are also downloading and uploading a file, in contrast to some other solutions which only support an asymmetric dependency on a single provider (Nielsen et al., 1999).
- The implementation splits files into pieces for distribution, in comparison to some other solutions which relay the entire file over the network therefore creating a bottleneck (John Howard, 1988).
- The implementation mounts files natively on the user's filesystem so existing applications can read them as if they were stored locally on the user's disk. Some other solutions only download the existing files themselves, with no (or experimental and platform-specific (Magiera and Rataj, 2022)) support for streaming files directly to serve filesystem read requests.

1.6 Outline

Chapter 1 - Introduction In this chapter I introduce the context of the problem and I state the aims and objectives of the project.

Chapter 2 - Literature, technology and data survey In this chapter I have a deeper look into the literature surrounding the problem, evaluate existing solutions and determine the programming language, tools and libraries that will be utilised in the development of the introduction.

Chapter 3 - Requirements In this chapter I formulate a list of requirements based on the work done in the literature, technology and data survey.

Chapter 4 - Design In this chapter I convert the list of requirements into a design for the resulting implementation.

Chapter 5 - Implementation In this chapter I discuss important aspects of the code of the implementation.

Chapter 6 - Testing In this chapter I detail the testing of the implementation that has been performed in order to ensure that I have met the requirements. I also perform some benchmarking on multiple operating systems.

Chapter 7 - Conclusions In this chapter I reflect on the work done in the project and the contributions that have been made to the field.

Chapter 2

Literature, technology and data survey

2.1 Digital consumption of publicly accessible content

Due to the ubiquitous rise of the Internet, consumers are switching away from traditional forms of accessing content. Video and music streaming services have experienced growth in contrast to traditional services such as cable and telecommunication companies (Calboli, 2022). The file sizes of multimedia they deliver have dramatically increased usage of storage on user devices (Dinneen and Nguyen, 2021). This is a concern because although there has been a decrease in the price of secondary storage (Kryder and C.S. Kim, 2009) such as hard disk drives and flash memory, smartphones typically do not have removable or extendable storage unlike desktop computers (H. Kim, Agrawal, and Ungureanu, 2012). This means consumers are restricted by the amount of storage they choose at purchase, which is typically lower in the range available (Mohd Suki, 2013) due to the increased price of more storage deterring users from buying more expensive models.

2.2 Growing usage of peer-to-peer systems

The reliance on a client-server model (used in protocols such as HTTP) poses a problem on content providers, as it puts considerable strain on servers to provide high-bandwidth connections (Benet, 2014a). An alternative being increasing utilised by user communities involves replacing client-server systems with peer-to-peer ones instead. This results in a considerable reduction in bandwidth on the content provider, while users experience potentially higher download speeds (Cunningham, Alexander, and Adilov, 2004). Despite peer-to-peer systems having a reputation for illegally sharing copyrighted content (Envisional, 2011), content providers are increasingly turning towards these systems to share their content, including academic datasets (J.P. Cohen and Lo, 2014; 2016), operating system updates (Warren, 2015) and television programs (Solheim, 2008).

2.3 Existing solutions

2.3.1 HTTP(S)

HTTP remains one of the most used protocols on the Internet. A study by Schumann et al. (2022) found that for one particular link in the Americas during 2020, 60 to 70% of IPv4 traffic was HTTP(S), while reaching 90% for a link in Japan. Multiple streaming services such as Netflix and Spotify deliver content over HTTP, and are usually backed by infrastructure provided by cloud providers such as Amazon Web Services (Adhikari et al., 2012).

HTTP works by utilising a client-server model. First, a separate protocol called DNS is used to map domain names to IP address (Mockapetris, 1987). Next, a web browser (the client) contacts the server over a single TCP connection and can request content using a GET request (Nielsen et al., 1999) - this has been highlighted as a potential bottleneck in HTTP/1 and HTTP/2, and prompted a replacement using a new protocol (QUIC, which utilises UDP) for HTTP/3 (Bishop, 2022). HTTP also has support for partial content requests (Nielsen et al., 1999), in which byte ranges can be specified instead of downloading the entire file.

HTTP however is cited as a disadvantage in new scenarios, due to the increased demand for large files to be transported in real-time. Benet (2014a) gives examples of this such as "hosting and distributing petabyte datasets" and "high-volume high-definition on-demand or real-time media streams". Efforts such as HTTP/3 and WebSocket (Melnikov and Fette, 2011) have attempted to address the overhead of the protocol, but most distributors facing these issues have "given up HTTP for different data distribution protocols".

2.3.2 BitTorrent

The BitTorrent protocol is a popular peer-to-peer file sharing protocol (B. Cohen, 2008). BitTorrent has received significant adoption out of all peer-to-peer protocols, with Sandvine (2019) measuring 2.46% of downstream and 27.58% of upstream traffic being used for the protocol. Due to its popularity many content providers are using the protocol (J.P. Cohen and Lo, 2014; 2016; Solheim, 2008) or a modified version (Warren, 2015) as a form of distribution.

A torrent file can be generated for a collection of files and folders. The torrent file may include links to individual servers known as "trackers" (the only component that uses the client-server model and pre-existing protocols such as HTTP), which is used to help peers find each other. A BitTorrent client can get a list of peers for a torrent file and contact them to receive pieces of the files in question before reassembling them (which removes the single connection bottleneck as seen in client-server models such as HTTP, therefore leading to faster downloads on average). Pieces are of fixed-length, and the torrent file contains the SHA1 hash of every piece to ensure it has not been tampered with or corrupted during transit (B. Cohen, 2008).

BitTorrent v1 torrent files do not distinguish between individual files, leading to separate swarms of peers for each torrent file even if the files they describe are identical. BitTorrent v2 addressed this by splitting individual files into pieces instead of the entire torrent. These pieces are used to form a Merkle hash tree (or simply Merkle tree), which more efficiently splits and hashes individual files into tree-like structures so they can be represented by a single root hash (Norberg, 2020). BitTorrent also supports a distributed hash table (DHT) for peer discovery instead of using trackers, therefore allowing BitTorrent to be a fully peer-to-peer protocol.

2.3.3 InterPlanetary File System

The InterPlanetary File System (IPFS), is another peer-to-peer file sharing protocol (Benet, 2014a). Collections of files and folders are represented under a format called UnixFS (John, 2022) which is turn is represented by content identifiers (or CIDs, which are the single root hashes formed from the Merkle trees of the files) instead of torrent files. Similar to BitTorrent v2, IPFS uses a Merkle tree, where CIDs can represent folders, files, and their blocks. IPFS solely uses a DHT for peer discovery, and peers request a list of blocks from others which are verified by hashing them to determine the CIDs (Schilling, 2022).

The use of CIDs means that IPFS is content-addressable, which BitTorrent is not designed for (the8472, 2018), but this can still lead to duplication as they are not singular files. IPFS is also relatively new compared to BitTorrent, with few implementations except for Kubo (Protocol Labs, 2022) which is implemented in Go, and it only has experimental, unstable and platform-specific (only Linux and macOS) support (Magiera and Rataj, 2022) for servicing file cluster read requests (and most implementations of BitTorrent do not either). IPFS also is designed to focus on future-proofing, as for instance, CIDs are self-describing in what binary-to-text encoding and hashing algorithms they use (Rataj, 2022). This allows IPFS implementations to easily upgrade the cryptographic hash functions (i.e. as over time these may eventually be cryptographic broken) and any other adjustments in CIDs they may make without substantial difficulty (Benet, 2017).

2.3.4 SFS read-only file system

The SFS read-only file system proposed by Fu, Kaashoek, and Mazières (2002) suggests a federated alternative to client-server file sharing protocols. It re-uses some components from the Self-certifying File System by Mazieres (2000) to provide a distributed filesystem for users to access public, read-only data. Multiple servers offer a mirror of a database signed by the content provider via public-key cryptography, which contains a Merkle tree of blocks and inodes used by the files and dictionaries being represented. Blocks and inodes can requested by the client via identifiers called handles, which are similar to CIDs in IPFS.

The client-server protocol only contains two types of messages - "one to fetch the signed handle for the root inode of a filesystem, and one to fetch the data (inode or file content) for a given handle". The client also validates the authenticity using the provider's public key, meaning the protocol does not use in-transit encryption as it is unnecessary. However, despite SFS offering a virtual filesystem, it still encourages a client-server model - clients are not encouraged to share the blocks and inodes they receive (unlike BitTorrent and IPFS) unless they run a server separately.

2.3.5 Andrew File System

The Andrew File System (AFS) is a distributed multi-user read/write filesystem in which multiple servers offer a shared virtual filesystem for clients (John Howard, 1988). The use of multiple servers increases redundancy, and the reason for presenting the appearance of a normal filesystem was to reduce the need to "modify existing application programs". The designers also wished to "minimiz[e] network traffic" so the filesystem employs a method in which files are cached on the client's computer. When a file read request occurs, the file is downloaded and cached before being opened. When a write request occurs, the cached copy is updated and then pushed out to the server only once a closed file request is performed.

Originally designed for university usage, the AFS was stated by Benet (2014a) to have "succeeded widely and [to still be] in use today". An acknowledged disadvantage of the AFS however is that it copies whole files across the network, even if they have been partially modified (in contrast to BitTorrent and IPFS, which split files into chunks which are more efficiently transmitted across the network). A multi-user filesystem seems unnecessary in addressing the problem area of distributing publicly accessible content - end users do not need to authenticate nor modify the files they receive, they only need to retrieve them and validate their authenticity.

2.4 Examination of technical requirements

A study by Prechelt (2000) found that development time is quicker for Python compared to Java, C and C++, although C and C++ have the highest performance and lowest memory usage, with Java having the noticeably highest memory usage. The study describes scripting languages such as Python as a "reasonable" alternative to C and C++, stating that "relative runtime and memory-consumption overhead will often be acceptable". Google (Taylor et al., 2021) and Microsoft (Thomas, 2019) have stated that 70% of their vulnerabilities are due to memory safety issues, which places concern on using languages such as C and C++ for a prototype implementation. Therefore, it can be concluded that Python would be a suitable language to develop the implementation in due to its ability for quick development and memory-safety proving to be strong advantages despite its disadvantages.

In terms of whether a command-line interface (CLI) or a graphical user interface (GUI) should be used for the implementation, Unwin and Hofmann (1999) concludes it "depends on our needs and styles of working". Hultstrand and Olofsson (2015) also agrees, stating that the "CLI has a steeper learning curve but offers more control", whereas "the GUI is helpful and simple but contains a lot of auto-magic." They also discovered that 76% of a sample of Git version control users are using the CLI. Perez De Rosso and Jackson (2013) states that GUIs utilise Git's CLI underneath and aim to hide its complexity, therefore suggesting that development of a CLI also allows a base upon which a GUI may be developed. Given that I want to offer the user a precise level on control in where they place their files in comparison to current solutions, it appears a CLI is more suitable.

The version control system Git will be used to keep track of changes to the source code of the implementation and the dissertation. Compared to Mercurial and Subversion, Git is the most widely used version control system, being used by nearly 94% of those surveyed by Stack Overflow (2022) - GitHub and GitLab, both online version control platforms utilising Git, also comprise over 85% usage out of all version control systems surveyed. As a distributed version control system, Brindescu et al. (2014) stated that Git offers "higher quality commits" compared to a traditional centralised version control system, because commits are likely to be smaller with "cohesive changes". Baudiš (2014) also stated that "wide adoption and scale of use clearly indicates that the current [distributed version control] systems have reached appropriate scalability and reliability levels". Therefore, it appears Git is the most suitable choice for a version control system.

As a proof-of-concept and prototype, I have decided that user testing would be unnecessary because it is only needed to check that the functionality of the implementation is feasible. Unit and integration testing will be used, and I also expect to manually walk through the completed implementation as a form of system testing to ensure it works and the requirements have been met, along with some benchmarking to analyse performance and network usage.

Given that the implementation will use a CLI, a possible future extension of the concept would involve a GUI built on top of it (similar to Git GUIs, which utilise the CLI underneath).

Chapter 3

Requirements

Drawing from the exploration of relevant solutions from the literature, technology and data survey and additional brainstorming, I can reach several requirements for the proposed system. Requirements are prioritised to be either high, medium or low, which signifies how necessary they are towards the completion of the project.

3.1 Functional requirements

	Implementation-wide	**Priority**
1.1	The implementation will use the Python programming language.	High
1.2	The implementation will use a command-line interface (CLI).	High
1.3	The implementation will use a custom peer-to-peer file sharing protocol to retrieve files.	High
1.4	Files will be made available to the user and applications via a read-only filesystem.	High
1.5	The user can determine what files will be downloaded using the CLI.	High

	Content-addressable file sharing protocol	**Priority**
2.1	Files will be split into blocks which will be distributed among peers.	High
2.2	Files and their corresponding blocks will have content identifiers.	High
2.3	The blocks will be built up to form a Merkle tree representing the file.	High
2.4	Files will be distributed individually to reduce duplication.	High
2.5	The command-line interface allow users to add peers manually as an alternative to trackers.	High
2.6	Content identifiers will be future-proof (containing both the base encoding and the hashing algorithm used).	Low
2.7	Trackers will be used for peer discovery.	Low
2.8	The implementation will connect to existing BitTorrent trackers.	Low

	Read-only filesystem	Priority
3.1	The user can determine where files will be placed and their filenames using the CLI.	High
3.2	Clusters of files will only be downloaded when they are needed.	High
3.3	Clusters of files will be cached.	Medium
3.4	When the cache storage becomes full, the oldest downloaded clusters will be deleted.	Medium

3.2 Non-functional requirements

	Non-functional requirements	Priority
4.1	The implementation's source code will be tracked using the Git version control system.	High
4.2	The implementation may download and read files on par with or faster than existing similar solutions.	Low
4.3	The implementation may have a lower network usage compared to existing similar solutions.	Low

Chapter 4

Design

It was concluded that in order to fulfil the requirements, the implementation will consist of two large subsystems - a custom peer-to-peer file sharing protocol (which I shall in future refer to as "RAFDP protocol") to download and distribute files and a virtual filesystem to expose these files to end-users and applications with minimal disruption. I've highlighted areas where I found the design to require exquisite detail, therefore the following is not exhaustive of everything that will be included in the implementation.

4.1 Peer-to-peer file sharing protocol

4.1.1 UDP as transport layer protocol

I decided to base the RAFDP protocol on top of the UDP protocol. Due to protocol ossification (where middleboxes on the Internet block unrecognised protocols or modifications of existing ones), the only usable protocols are therefore UDP and TCP (McQuistin, Perkins, and Fayed, 2016). As for why UDP was chosen, it carries minimal overhead compared to TCP. I deemed it unnecessary to have a reliable, stream-orientated protocol as self-contained messages would offer better performance and can simply be resent at regular intervals if lost.

4.1.2 Splitting files into blocks

The files are split into blocks in the RAFDP protocol because this offers numerous advantages, such as quicker downloads (as multiple peers can share pieces instead of one source becoming a single bottleneck, and that other peers that have even only partially downloaded a file may begin sharing what they have with other peers as there is no need to wait for each peer to download the whole file) and better error handling (as corrupted pieces can be discarded rather than the whole file). An illustration of this is seen in figure 4.1.

4.1.3 Content identifier algorithm

I will describe the algorithm that produces content identifiers for files. A CIDv1 hash (used in IPFS) (Benet, 2016a) consists of the following four components:

- A "multibase" code stating what base encoding is being used (e.g. base64, base58 etc.).

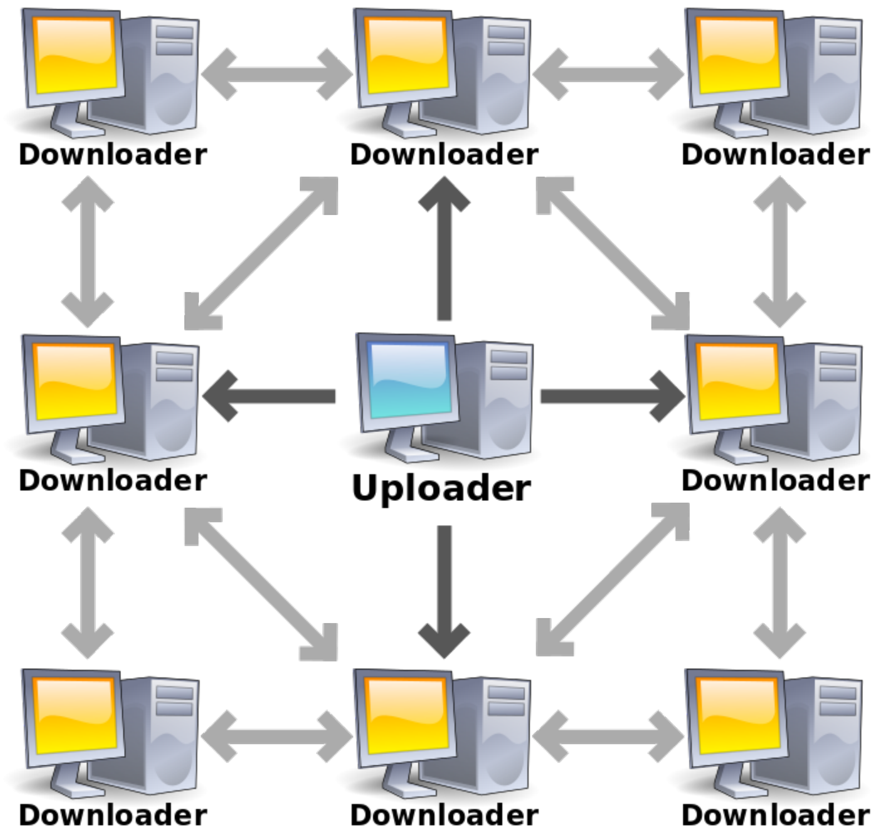

Figure 4.1: An illustration of the network structure in a peer-to-peer file sharing protocol (Martin, 2014)

- A "multicodec" value stating the version of CID used (in this case, 1).
- A "multicodec" value stating the type of data being hashed (e.g. a MerkleDAG protobuf which is used in IPFS to describe files).
- A "multihash" valve stating a self-describing hash containing the hash itself and an indication of the hashing algorithm used (e.g. sha256).

The types "multibase", "multicodec" and "multihash" are defined by specifications created by Protocol Labs, the developer of IPFS, and are described further below:

- A "multibase" value has the format *<base-encoding-character><base-encoded-data>* (Benet, 2016b). The *<base-encoding-character>* is a value derived from the "multibase" table of pre-defined values (e.g. 0 for binary encoding, 9 for decimal encoding etc.).
- A "multicodec" is an unsigned multiformat varint used as a prefix to identify the data that follows (Benet, 2015). The values are derived from the "multicodec" table of pre-defined values (e.g. 0x70 for a MerkleDAG protobuf).
- A "multihash" is a self-describing hash. It has the format *<varint-hash-function-code><varint-digest-size-in-bytes><hash-function-output>* (Benet, 2014b) where *<varint-hash-function-code>* is derived from the "multicodec" table indicating which hashing

algorithm was used (e.g. 0x12 for sha2-256).

- The varint referred to before (which we will call the "unsigned multiformat varint") serialises unsigned integers 7 bits at a time, with the most significant bit of each output byte indicating if there is a continuation byte.

I can therefore conclude that a future-proof hash should self-describe the base encoding and hashing algorithm used so that any of these can be easily upgraded. It should also contain a version number which should support this upgrade. The RAFDP protocol will represent files as chunks forming a Merkle tree, therefore there should be three representations of data that could be transmitted - a single hash, a pair of hashes and the chunk of data itself. In addition, I have a magic header at the start ("RAFDP") in order to reduce ambiguity and confusion between CIDv1 and my custom hash format.

The hash will be in readable text instead of in a binary format (in comparison CIDv1 allows either) so it can be easily copied and transmitted. Due to this, the version number will not be encoded as an unsigned multiformat varint but using my own custom unsigned varint representation (and to avoid confusion between the two I shall refer to this as "RAFDP unsigned varint"). A RAFDP unsigned varint is serialised as follows: the first hex digit indicates how many more hex digits are remaining, and the remaining hex digits represent the number encoded as a big endian integer in hex (this therefore gives a possible range of 0 to $16^{15} - 1$).

Therefore, I plan to have RAFDP hashes in the following format:

- A magic header – the string "RAFDP".
- A varint representing the version of the RAFDP hash being used.
- A "multibase" code stating what base encoding is being used.
- A "multihash" value stating the hashing algorithm used (e.g. SHA256) and the hash itself.

4.1.4 Merkle tree

A Merkle tree splits data into blocks (Merkle, 1982). Hashes of these blocks are recursively computed pairwise (each pair of chunks are hashed, and these hashes are hashed in pairs in turn, and so on) until a single root hash is obtained (as shown in figure 4.2). This offers many advantages for a file sharing peer-to-peer protocol (Norberg, 2020), including the need to only share the root hash with another peer so they can initiate a download (as another peer can request the pairs of child nodes for the root hash, and then the child nodes for each received child node, and so on until the actual file data is obtained) and it allows quick verification of any malicious or corrupted blocks that may have been sent by peers so they can be easily discarded.

Similar to BitTorrent v2, I've decided to adapt a fixed 16 KiB piece size. A fixed piece size ensures there is no ambiguity and inconsistency (as trees would not have varying piece sizes encoded in them) so peers can reproducibly produce hash trees on files for verification by assuming this fixed 16 KiB piece size. It would also reduce overhead as the piece size does not need to be included as part of the RAFDP hash.

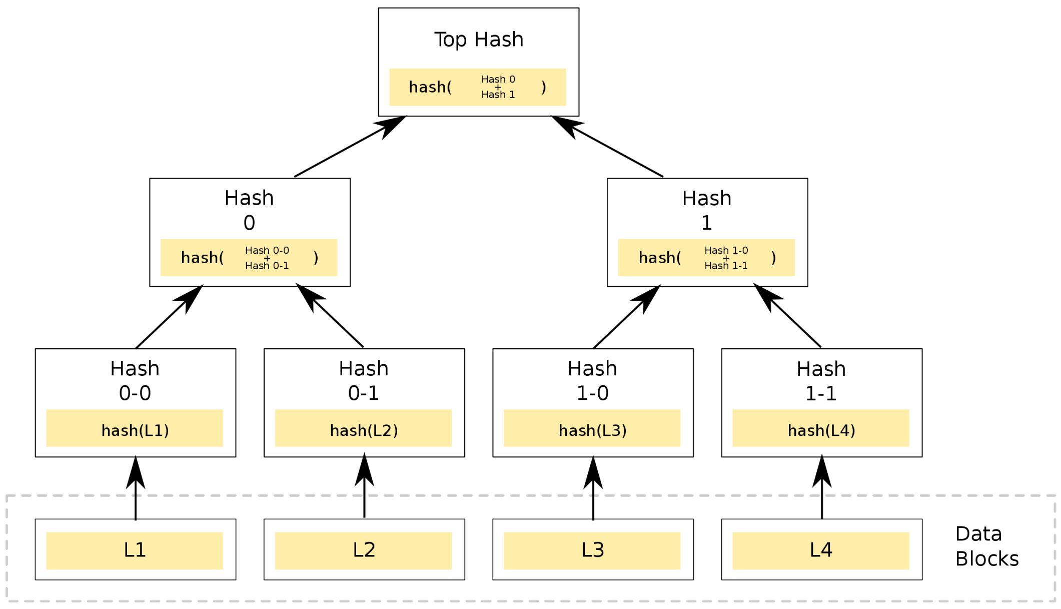

Figure 4.2: Example Merkle tree for a file consisting of 4 blocks (Azaghal, 2012)

4.1.5 BitTorrent trackers

In order to discover peers, BitTorrent uses centralised servers known as "trackers". I decided to make the RAFDP protocol compatible with existing BitTorrent trackers rather than creating my own custom tracker protocol because I felt it would be beneficial to utilise the vast number of existing BitTorrent trackers (Zhang et al., 2011).

The BitTorrent tracker protocol utilises HTTP. A client performs a GET request to a URL of the tracker with HTTP query parameters passed in containing several fields (B. Cohen, 2008). Through experimentation with public trackers, I found that the fields "info_hash" (which could be the RAFDP hash), "peer_id" (a unique identifier generated by the client), "port", "uploaded", "downloaded" and "left" (the latter three in number of bytes) are requisite, otherwise the trackers returned inconsistent or empty responses. As "uploaded", "downloaded" and "left" are designed for the tracker to log stats for BitTorrent torrents (Zelinskie, 2016), I decided it would be unnecessary for the RAFDP implementation to provide valid values for these fields, and the trackers seemed to request the same list of peers regardless. The tracker would in turn then respond to the client with the fields (B. Cohen, 2008) "interval" (the number of seconds before the client sends another request) and "peers" (list of dictionaries of peers).

I noted that public trackers also only responded when the hash value sent was truncated (which is also what BitTorrent v2 does as it handles 256-bit long SHA256 hashes (Norberg, 2020)) to a length of 20 bytes (as trackers only anticipated support for SHA1, which was utilised by BitTorrent v1 (B. Cohen, 2008)). This would instinctively be a concern because it would increase the chance of a collision between hashes so that a peer could be possibly be directed to download a conflicting RAFDP hash sharing other, possibly malicious, files (Cimpanu, 2017). However, as the RAFDP peer downloading the pieces has the full hash, it can validate each one via the recursive Merkle tree algorithm as belonging to the original

RAFDP hash, therefore this is not a concern.

4.2 Virtual filesystem

4.2.1 FUSE

First, I decided to attempt to use the popular software interface Filesystem in Userspace (FUSE) (Vangoor, Tarasov, and Zadok, 2017) to implement the virtual filesystem with. There are multiple third-party libraries to allow interfacing with FUSE in Python, including python-fuse, fusepy, pyfuse3 and python-llfuse (Zakharov, 2018). However, it appears all these libraries are suffering maintenance problems (mxmlnkn, 2022). I noted that fusepy (Honles, 2023) appears to have a high-level third-party library called fusetree (Costa, 2023), which would allow quicker development and integration of fuse into my implementation, so I decided to utilise that.

However, I later discovered that the low-level part of fusepy called fusell (which fusetree utilises) lacks support for Windows. I attempted to test the implementation in macOS but the library failed with a cryptic error message (as seen in listing 4.1). This therefore restricted my implementation's support to only Linux, but then I also discovered the same error message occurring inconsistently across different Linux distributions. I therefore decided to abandon FUSE for other options instead.

Listing 4.1: The output and error of the former *virtfilesystem.py* which used FUSE

```
Virtual filesystem RPC server listening on port 7274
Using selector: EpollSelector
Traceback (most recent call last):
  File "/home/james/Documents/rafdp/virtfilesystem.py", line 113,
     in <module>
    fusetree.FuseTree(rootNode, args.path, foreground=True)
  File "/home/james/Documents/rafdp/fusetree/fusetree.py", line 59,
     in __init__
    super().__init__(mountpoint, encoding=encoding)
  File "/home/james/Documents/rafdp/fusell.py", line 495, in
     __init__
    assert chan
AssertionError
```

4.2.2 SFTP

After issues with FUSE, I then looked at other possible ways of implementing a virtual filesystem. I noted that SMB seem to have consistent support across Windows, macOS and Linux (Ashcraft et al., 2021; Fleishman, 2020; Samba Team, 2023) but my supervisor advised me against reverse-engineering or implementing a server based on the protocol due to its complexity. I then looked at SFTP and noted there was also strong (although sometimes unofficial) support. Many Linux distributions feature native support for mounting SFTP file shares as virtual drives via SSHFS (Marakasov, 2023). There are some third-party applications that add support for SFTP for Windows (WinFsp (Zissimopoulos, 2022) and Dokany (Stark, 2022)) and macOS (FUSE for macOS (Fleischer, 2023a) or FUSE-T (Fishman, 2023b) coupled with their own SSHFS for macOS implementations (Fishman, 2022; Fleischer, 2023b)). I used the Python library paramiko (Forcier, 2023) in order to implement the virtual SFTP server that could present the virtual filesystem as a server share that could be mounted by third-party programs.

For Windows, it appears that WinFsp is currently still being maintained at the time of this project, while the authors of Dokany are currently looking for new maintainers (Stark, 2022). The author of WinFsp, Bill Zissimopoulos, also posted some benchmarks showing WinFsp being considerably faster than Dokany Zissimopoulos, 2016. For these reasons, I decided to go with WinFsp for Windows. For macOS, between the choices of underlying layers, it seemed that FUSE-T was the most suitable. This is because the alternative is macFUSE, which utilises a kernel extension and is therefore prone to breaking with each version of macOS (Fishman, 2023a) along with requiring the modification of security settings (Apple, 2021). I then therefore decided to use FUSE-T with SSHFS for macOS.

Chapter 5

Implementation

In terms of programming the implementation, I layered levels of code to successively met all the functional requirements and cover both subsystems (the RAFDP protocol and virtual filesystem). I first wrote underlying structures and functions that would be utilised (such as the Merkle tree implementation), then the class implementing the required functionality for each subsystem and then finally added a CLI and other interfaces (such as the RAFDP RPC library (*rafdplib.py*) to control the daemons implementing each subsystem when necessary. It's important to note that what I cover here is not exhaustive (for instance, details of CLI and daemon communication have been omitted). I've only chosen to cover areas which I particularly are influential and of value.

5.1 Overall system architecture

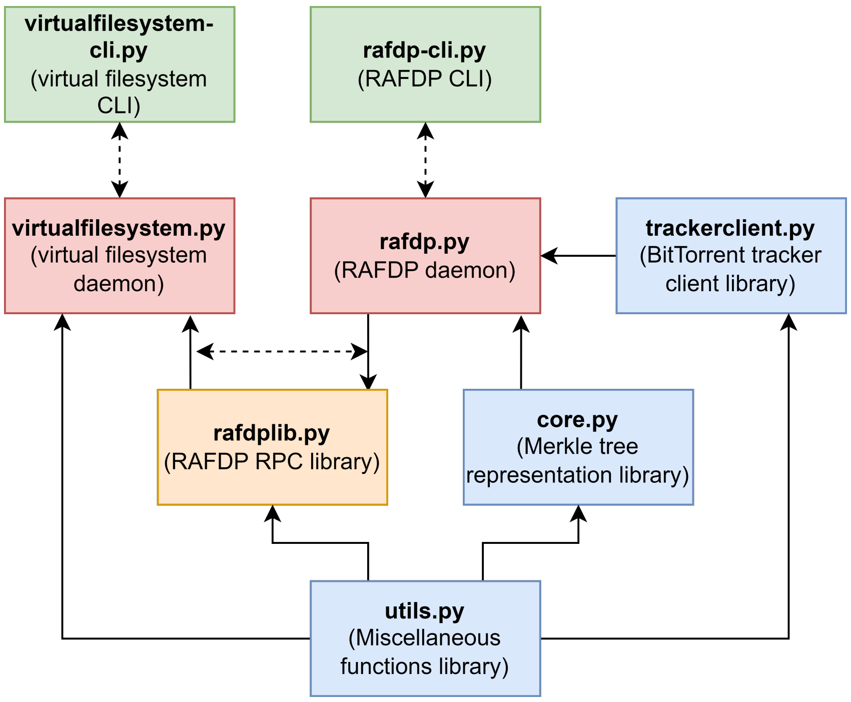

Figure 5.1: A diagram of the overall system architecture of the implementation

In the diagram shown by figure 5.1, the dotted lines indicate RPC communication (passing

JSON UDP messages to allow control of each daemon) while the solid lines indicate importing another module. All components in this diagram were written by me - some modules import functions from third-party libraries which will be described later on.

The overall system consists of two daemons - one for the virtual filesystem (*virtfilesystem.py*) and one for the RAFDP protocol (*rafdp.py*). The RAFDP daemon can run standalone, while the virtual filesystem daemon starts its own RAFDP daemon in order to utilise the protocol. The virtual filesystem daemon communicates with the RAFDP daemon utilising the same RPC protocol the RAFDP CLI uses but with the assistance of a RAFDP RPC library (*rafdplib.py*) which providers more direct control and some helper functions.

The RAFDP daemon utilises a Merkle tree representation (*core.py*) along with a BitTorrent tracker client (*trackerclient.py*). A miscellaneous functions library (*utils.py*) is utilised by all files shown except the CLIs and it contains commonly re-used code such as the custom varint implementation and SFTP object creation. Other Python files not included in the diagram are *test_multiple.py* which contain the unit and integration tests and *demo.py* which performs an automated run of the daemons for system testing.

5.2 RAFDP protocol

5.2.1 Merkle tree

First, I implemented the Merkle tree class which will be utilised by the RAFDP protocol in a file called *core.py*. As this was a prototype solution, I used some fixed parameters (specifically, a chunk size of 16 KiB, a hashing algorithm of SHA2-256 and a base encoding of base58). As the RAFDP hash is future-proof it can be upgraded to support any changes to these parameters so this should not be a problem. The algorithm for computing a RAFDP hash uses some components from the multiformats library (Gogioso, 2023).

The Merkle tree class generates a Merkle tree by first opening the file and hashing each chunk of 16 KiB, as seen in code listing 5.1. Note that the actual chunk content is not stored in-memory as shown by the line `tree[chunkhash] = (filename, len(chunkhashes), self.chunk_size)` - a pointer (containing the filename, the index of the chunk within the file, and the length of the chunk itself) is stored instead. This is because large files (such as those being gigabytes in size) may not fit in a user's RAM and it would also be unnecessary given the files are present on the user's hard drive.

Listing 5.1: Turning a file into a list of hashes

```
with open(filename, "rb") as file:
        while True:
                chunk = file.read(self.chunk_size)
                if not chunk:
                        break
                chunk = utils.tovarint(len(chunkhashes)) + chunk
                chunkhash = self.generate_hash(chunk)
                # pointer to offset in file stored in tree
                tree[chunkhash] = (filename, len(chunkhashes),
                    self.chunk_size)
                chunkhashes.append(chunkhash)
```

Generating the actual Merkle tree is done by recursively hashing pairs of hashes (or a single hash if a pair isn't possible) and adding them to the internal tree representation until a final root hash is obtained. This is seen in code listing 5.2.

Listing 5.2: Generating the Merkle tree by recursively hashing pairs of hashes

```
while len(chunkhashes) > 1:
        newchunkhashes = []
        for pair in itertools.zip_longest(*[iter(chunkhashes)] * 2):
                first, second = pair
                if second is None:
                        pair = first
                else:
                        pair = first + "," + second
                chunkhash = self.generate_hash(pair.encode("ASCII"))
                if chunkhash in tree:
                        raise Exception("Hash␣detected␣twice␣in␣
                           tree?")
                tree[chunkhash] = pair
                newchunkhashes.append(chunkhash)
        chunkhashes = newchunkhashes
roothash = chunkhashes[0]
```

However, when assembling a file streamed from another peer the chunks are stored in memory because they are only needed ephemerally when the parts of the file corresponding to the chunks need to be read. There is not an issue with memory usage here because the daemon detects that the amount of memory available is low and therefore randomly removes nodes from the Merkle tree class (as seen in code listing 5.3). If they are needed again (i.e. when the user loads a corresponding part of the file) they can always be re-downloaded from other peers.

Listing 5.3: Trimming the Merkle tree when running out of RAM

```
def reduce_tree_size(self):
        if psutil.virtual_memory().percent >= 95:
                choices = list(set(self.tree.keys()) -
                   set(self.roothashes))
                if len(choices) > 0:
                        del self.tree[random.choice(choices)]
```

5.2.2 Fragmentation and general communication

I realised that conveying binary data in response to requests for hashes would not fit inside a UDP packet that would successfully pass through the Internet (as gateways drop packets that are larger than the smallest MTU of the route (Moy, 1998)). I then decided to fragment the payload into multiple separate and smaller UDP packets (limiting the size to a maximum of 508 bytes (Beejor, 2016)) so that they could be delivered successfully (as seen in code listing 5.4).

Listing 5.4: The code to fragment a packet into multiple smaller prefixed subpackets to avoid exceeding the MTU

```
chunksize = 508
```

```
offsets = list(range(0, len(tosendhash), chunksize))
for index, i in enumerate(offsets):
        tosendhashpart = (1).to_bytes(1, "big") + (1).to_bytes(1,
           "big")
        tosendhashpart += utils.tovarint(index) +
           utils.tovarint(len(offsets))
        tosendhashpart +=
           utils.tovarint(len(wantedhash.encode("ascii")))
        tosendhashpart += wantedhash.encode("ascii") +
           tosendhash[i:i + chunksize]
        socket.sendto(tosendhashpart, peer)
```

Each fragmented packet is prefixed by the following: a varint for the fragment index, a varint for the number of fragments, a varint denoting the hash the data is represented by, the hash itself and then the binary data fragment. These parameters allow the receiving peer to reassemble the fragments into the entire binary data once all of them have been received. If any are lost, all the fragments will simply be recomputed and resent again once the peer re-requests the hash, as it does so at a regular interval in a background thread as seen in code listing 5.5.

Listing 5.5: Part of the background thread's loop code showing the regular sending of ping messages and requesting of data for missing hashes

```
for peer in peers:
    if not peers[peer]["valid"]:
        if (time.time() - peers[peer]["lastcontact"]) > 30:
            peers[peer]["lastcontact"] = time.time()
            socket.sendto(b"RAFDPPING", peer)
    else:
        for missinghash in overalltree.get_missing():
            if missinghash not in peers[peer]["missing"]:
                peers[peer]["missing"][missinghash] =
                   {"lastcontact": 0}
            if (time.time() -
               peers[peer]["missing"][missinghash]["lastcontact"])
              > 5:
                peers[peer]["missing"][missinghash]["lastcontact"]
                   = time.time()
                socket.sendto((0).to_bytes(1, "big") +
                   missinghash.encode("ascii"), peer)
```

In the RAFDP protocol (which is contained in *rafdp.py*, the daemon's code), apart from ping and pong messages (to establish that each peer exists and can communicate with each other), there are only two main types of messages - a **request** to retrieve to contents for the hash from another peer and a **response** of contents for the hash the other peer sends back. The peers identify the type by the first byte of the message - interpreted as a big-endian integer, the value is either 0 (for a request) or 1 (for a response). Drilling down for request messages the rest of the message is simply the hash requested, while for response messages the next byte denotes whether it is a hash/pair of hashes (which is then followed by the hash or pair of hashes themselves) or binary data, which is then fragmented (the former of which has been described previously). Each message's format is denoted in table 5.1.

Table 5.1: The formats of all the possible messages in the RAFDP protocol

Message description	Message format
A ping message	RAFDPPING
A pong message	RAFDPPONG
Request (for a hash)	0 (to denote a request for a hash, big-endian integer)
	The hash itself
Response (returning a hash or pair of hashes for a request message)	1 (to denote a response, big-endian integer)
	0 (to denote a response of a hash or pair of hashes, big-endian integer)
	The hash or pair of hashes
Response (returning binary content for a request message)	1 (to denote a response, big-endian integer)
	1 (to denote a response of binary content, big-endian integer)
	Fragment index (as a varint)
	The total number of fragments (as a varint)
	The length of the hash of the data (as a varint)
	The hash of the data
	The data fragment itself

5.3 Virtual filesystem

5.3.1 Prior FUSE work

Even though FUSE was later removed from the implementation due to discovered dependency maintenance issues and poor dependency multi-platform support, I was able to develop a proof-of-concept incorporation before discarding it. The fusetree library (Costa, 2023) was relatively high-level enough that this was quite simple. First, an empty dictionary is created called *rootNode*. Objects (when requested to do so by the CLI) initialised from classes (which inherit from `fusetree.nodetypes.BaseFile`) represent RAFDP and IPFS files and simulate the retrieval of bytes when requested by FUSE. The line `fusetree.FuseTree(rootNode, args.path, foreground=True)` initialises the FuseTree object which uses this dictionary to present the virtual filesystem. Code for all of this is seen in code listing 5.6 - note that much of the code for the virtual filesystem itself is omitted here (such as the RPC server code), along with repeated or similar instances of code between the *IPFSFile* and *RAFDPFile* classes.

Listing 5.6: The prior use of FUSE in *virtfilesystem.py*

```python
class RAFDPFile(fusetree.nodetypes.BaseFile):
    def __init__(self, cid, mode=0o444):
        super().__init__(mode)
        self.cid = cid
        rafdpdaemon.addhash(cid)

    async def open(self, mode):
        return self.__class__.Handle(self, self.cid)
    class Handle(fusetree.nodetypes.FileHandle):
        def __init__(self, node, cid):
            super().__init__(node, direct_io=True,
               nonseekable=False)
            self.cid = cid

        async def read(self, size, offset):
            logging.debug(f"{self.cid}␣(RAFDP)␣−␣requested␣bytes␣
               from␣{offset}␣of␣size␣{size}")
            result = rafdpdaemon.getsizeoffsetfromhash(self.cid,
               size, offset)
            while result is None:
                asyncio.sleep(0.1)
                result = self.gethash(hash)
            return result

class IPFSFile(fusetree.nodetypes.BaseFile):
    ...

    class Handle(fusetree.nodetypes.FileHandle):
        ...

        async def read(self, size, offset):
            logging.debug(f"{self.cid}␣(IPFS)␣−␣requested␣bytes␣
               from␣{offset}␣of␣size␣{size}")
            result = subprocess.run(["ipfs", "cat", self.cid, "−o",
               str(offset), "−l", str(size)],
               stdout=subprocess.PIPE)
            if result.returncode != 0:
                raise Exception(result)
            return result.stdout

class RPCHandler(socketserver.BaseRequestHandler):
    def handle(self):
        ...
        if data["method"] == "addrafdphash":
            rootNode[data["hash"]] = RAFDPFile(data["hash"])
        elif data["method"] == "addipfshash":
            rootNode[data["hash"]] = IPFSFile(data["hash"])
        ...

rootNode = {}
fusetree.FuseTree(rootNode, args.path, foreground=True)
```

5.3.2 SFTP and IPFS

Implementing the virtual SFTP server was somewhat more complex. The code for the file handle for SFTP (as seen in code listing 5.8) bears some resemblance to that for FUSE, but substantially more boilerplate is required elsewhere for purposes such as authentication and file permissions (which were redundant in this case as I am only presenting publicly available files on localhost to the user). I also had to provide the functionality of returning the SFTP attributes of files and folders, which I abstracted by implementing my own custom in-memory filesystem representation (parts of which are shown in code listing 5.7).

Listing 5.7: Parts of the *MemFS* in-memory filesystem representation

```
class MemFS:
    def __init__(self):
        self.files = {}
        self.addfile(8192, "40444", "/")

    def addfile(self, filesize, mode, path):
        ...
        SFTPobject = createSFTPobject(filesize, mode, time.time(),
           time.time(), path.name)
        self.files[str(path)] = (path, SFTPobject)

    def listdir(self, pathtolist):
        ...
        # Check if folder exists
        attrs = self.getattrs(pathtolist)
        if attrs is None or oct(attrs.st_mode)[2] != "4":
            return None
        # Find items in folder
        returnthese = []
        for filepath, SFTPobject in self.files.values():
            if pathtolist in filepath.parents:
                returnthese.append(SFTPobject)
        return returnthese
```

Even though it was not an explicit requirement, I was able to extend the virtual filesystem to support mounting IPFS files. The virtual filesystem calls the IPFS daemon's CLI (Matthews and Schilling, 2023), which is required to be installed on the user's computer beforehand. It allows requesting bytes for a length and offset in the same way that the RAFDP daemon also supports. The code listing 5.8 shows the function allowing the virtual SFTP server to present the range of bytes requested from either the file of the RAFDP hash or the IPFS hash.

Listing 5.8: The code for the file handle for SFTP (supporting RAFDP and IPFS)

```
def read(self, offset, length):
        filehash = self.filehash
        if filehash.startswith("RAFDP"):
                result =
                   rafdpdaemon.getsizeoffsetfromhash(filehash,
                   length, offset)
                while result is None:
                        time.sleep(0.01)
                        result =
                           rafdpdaemon.getsizeoffsetfromhash(filehash,
                           length, offset)
                return result
        else:
                result = subprocess.run(["ipfs", "cat", filehash,
                   "-o", str(offset), "-l", str(length)],
                   stdout=subprocess.PIPE)
                if result.returncode != 0:
                        raise Exception(result)
                return result.stdout
```

Chapter 6

Testing

Testing was done extensively enough such that the full test plan cannot be included in this chapter, but I will cover some notable test cases here while the rest are included in the appendices. Several test files were used, including an image (Voicu, 2018) for testing the Merkle tree and simple fetch-receive requests between RAFDP peers, a text book file (Dickens, 1998) for testing fetching random byte ranges for a hash from another RAFDP peer (as differences in text are more human-readable and therefore highlights better the cause of bugs in comparison to binary content such as an image or video) and finally a video (Perréard, 2020) for system testing as it allowed testing of seeking within a file via the video player to show that the RAFDP daemon and virtual filesystem combination can load parts of the file when requested to do so.

6.1 Unit testing

Unit testing involved testing individual functions to ensure that the basic building blocks of the implementation work as expected. However, as most of the code resided in either complex classes that relied on network communication or utilised third-party libraries, unit testing was minimal. Unit tests were written for the custom varint implementation, the Merkle tree implementation and the simulated in-memory custom filesystem class that I used for the virtual SFTP server that provided the custom filesystem implementation. All the unit tests are shown in table 6.1.

Table 6.1: All the unit tests

Test Number	**Test Case**	**Expected Result**	**Actual Result**	**Pass/Fail**

1.1	Test that the custom varint function can encode and decode integers properly	The varint function encodes and decodes a random integer while successfully separating and ignoring redundant randomised data	The varint function encoded and decoded a random integer while successfully separating and ignoring redundant randomised data	Pass
2.1	Test that a new Merkle tree can be built from a pre-built one via requesting values from hashes one at a time	The items in the new Merkle tree are the same as the pre-built one	The items in the new Merkle tree were the same as the pre-built one	Pass
3.1	Test that in-memory filesystem representation for the virtual SFTP server works correctly	Existing files and folders (some added during the test) return the correct values, while non-existing ones are also handled	Existing files and folders (some added during the test) returned the correct values, while non-existing ones were also handled	Pass

6.2 Integration testing

Integration testing seemed suitable for testing whether two RAFDP processes communicated correctly with one another. It was critical to test this as the implementation rests on the functionality of the underlying peer-to-peer file sharing protocol working correctly, and it would infeasible for unit testing to be performed on this as functionality is not isolated here. More specifically, the testing involved one RAFDP peer retrieving content for hashes - either for a single hash, an entire Merkle tree or multiple possible ranges of the file. A subset of the integration tests are shown in table 6.2.

Table 6.2: A subset of the integration tests

Test Number	**Test Case**	**Expected Result**	**Actual Result**	**Pass/Fail**

4.1	Test that a RAFDP peer is able to obtain the contents for a hash from another RAFDP peer	The first RAFDP peer successfully receives the expected bytes for a hash from another RAFDP peer	The first RAFDP peer successfully received the expected bytes for a hash from another RAFDP peer	Pass
5.1	Test that a RAFDP peer is able to obtain the entire Merkle tree for a hash from another RAFDP peer	The contents of the assembled received file from the other peer are the same as when the file is read locally	The contents of the assembled received file from the other peer were the same as when the file is read locally	Pass
6.1	Test that a RAFDP peer is able to fetch a random byte range **smaller** than the chunk size, **starting in** the file and **ending in** the file, that is represented by a hash from another RAFDP peer	The other RAFDP peer returns an identical byte range to that which would be returned if the file was read locally (in this case, some bytes less than 16 KiB in size)	The other RAFDP peer returned an identical byte range that matched the one read from the file locally	Pass

I noted that there were performance issues when spinning up each process and requesting hashes for satisfying each test. I decided to re-use the same RAFDP processes for a selection of some tests as I felt they were idempotent - the functionality being tested remained the same (so the test results would not be compromised) and the performance advantage of utilising the caching capabilities of each RAFDP process was considered advantageous.

I also discovered that on macOS, the Maximum Transmission Unit (MTU) of UDP is limited to 9,216 bytes (jusx, 2016). This causes RPC communication to the RAFDP daemon to fail when the latter sends back the file content (which exceeds the MTU in size) it has received. This issue does not appear to occur on Windows or Linux, which I speculate that both operating systems do not impose a lower UDP size limit than the theoretical maximum of 65,507 bytes over localhost (Zac67, 2022). It was resolved by using the command `sudo sysctl -w net.inet.udp.maxdgram=65535` (jusx, 2016) to increase the UDP max datagram size. As this change does not survive a reboot, I made my RAFDP implementation when booting detect it is running on macOS and prompt for admin rights in order to run the command before starting up the RAFDP daemon (as seen in code listing 6.1). This highlighted the overall importance that testing holds in uncovering bugs in different environments.

Listing 6.1: Increasing the UDP max datagram size on macOS

```
def fix_udp_macos():
    if platform.system() == "Darwin":
        maxdatagram = subprocess.check_output(["sysctl",
            "net.inet.udp.maxdgram"])
        maxdatagram =
            int(maxdatagram.decode("ascii").strip().split("␣")[1])
        if maxdatagram != 65535:
            os.system("""osascript -e 'do shell script "sudo sysctl
                -w net.inet.udp.maxdgram=65535" with administrator
                privileges'""")
```

The fetching of byte ranges of a file (even validating all possible reads such as those outside the size of the file) representing a hash from another RAFDP peer was thoroughly tested because I believed it would be critical that correct byte ranges are returned that are identical to those returned if reading the file normally as if it were stored locally on the user's secondary storage because the virtual filesystem needs to satisfy the requirement of serving the correct byte ranges as applications will see RAFDP virtual files as normal files.

6.3 System testing

System testing was important because it ensured that the software satisfied the requirements in the environment where it would be utilised in field conditions, so the tests were designed to emulate a plausible real-world scenario (adding a file via its RAFDP hash, opening it etc.). The sequence of tests required manual intervention in which I followed the steps of the testing procedure and were also ran for each operating system - Windows, macOS and Linux:

1. Start the virtual filesystem.
2. Start a RAFDP process.
3. Add the test video file to the RAFDP process via the CLI.
4. Add the IP address and port of the RAFDP process to the virtual filesystem via the CLI.
5. Add the RAFDP hash for the test video file to the virtual filesystem via the CLI.
6. Assign a location for the RAFDP hash file to appear via the CLI.
7. Open the virtual test video file from where it was assigned to in test 6.
8. Seek the virtual test video file to roughly halfway.

A subset of the systems tests are shown in table 6.3.

Table 6.3: A subset of the system tests

Test Number	**Test Case**	**Expected Result**	**Actual Result**	**Pass/Fail**

8.1	Start the virtual filesystem	A virtual drive should automount on macOS and Linux, but not on Windows. On all OSes the window should show messages indicating RAFDP + RAFDP RPC + virtual filesystem RPC is listening at port	A virtual drive automounted on macOS and Linux, but not on Windows. On all OSes the window showed messages indicating RAFDP + RAFDP RPC + virtual filesystem RPC is listening at port	Pass on Windows Pass on macOS Pass on Linux
8.2	Start a RAFDP process	A window should open with the message stating the RAFDP process is listening at a port	A window opened with the message stating the RAFDP process is listening at a port	Pass on Windows Pass on macOS Pass on Linux
8.3	Add a real file to the RAFDP process via the CLI	The CLI should print out the RAFDP hash of the file	The CLI printed out the RAFDP hash of the file	Pass on Windows Pass on macOS Pass on Linux

This collective sequence of actions provides a thorough test of the various components of the implementation interacting together. The virtual filesystem is tested to see if it can mount and allowing reading from an application while serving the right ranges of data in real-time without downloading the whole file at once (instead downloading the required pieces). The RAFDP daemons are tested to see if they can accept the request to add files, RAFDP hashes, other peers and locations to mount hashes via the CLI along with being able to communicate relevant information between themselves using the RAFDP protocol.

6.4 Benchmarking

To meet requirements 4.2 and 4.3, I shall now benchmark the implementation in terms of network usage and speed. I will be comparing the performance of the implementation in delivering the test video file through the virtual filesystem with the RAFDP and IPFS protocols. The video is chosen in particular because it has the largest file size in comparison to the other test files (19.86 MB, in comparison to 2.11 MB for the image and 1.04 MB for the book text file). Network usage will also be recorded via the tool Wireshark (via its Statistics tool). I was planning to perform a separate test over localhost but during tests I found that the difference in speed over LAN and localhost was negligible (less than 10 seconds difference in file transfer duration). This shows that network speed of the LAN used in the tests is not a bottleneck on the implementation.

In each test, the virtual filesystem is booted on each operating system. Another computer on the network is used to serve the file by running its own RAFDP daemon. The socket address of the daemon and the hash of the file are added to the virtual filesystem, the file is mounted at a specific location and OS-specific tools are used to measure the time it takes to read the entire file. To time file transfers, the command Measure−Command {**type** "PATHTOMOUNTEDFILE"> \$null} was used on Windows (in PowerShell), while time cat ''PATHTOMOUNTEDFILE''> /dev/null was used on Linux and macOS (in Bash and Zsh, respectively). The results are shown in table 6.4. Speed in MB/s was deduced by dividing the data transfer in MB by the time taken in seconds.

Table 6.4: All the virtual filesystem benchmark results

Protocol	**Operating system**	**Data transfer (MB)**	**Time (over LAN) in seconds**	**Time (from cache) in seconds**	**Speed (over LAN) in MB/s**	**Speed (from cache) in MB/s**
RAFDP	Windows	24.529	104.378	20.552	0.235	1.194
RAFDP	Linux	24.558	22.327	12.188	1.100	2.015
RAFDP	macOS	24.562	61.460	0.006	0.400	4093.723
IPFS	Windows	26.327	338.499	312.397	0.078	0.084
IPFS	Linux	26.381	21.748	21.045	1.213	1.254
IPFS	macOS	26.342	28.217	0.006	0.934	4390.415

From the results, I can reach several conclusions in regards to network usage and speed:

- macOS experiences extraordinary speeds for both protocols when utilising caching. I speculate that the caching is done either by fuse-t itself or by macOS in regards to the virtual NFS server fuse-t is leveraging (Fishman, 2023a).
- IPFS transfers slightly more data for the file compared to RAFDP (roughly an additional 2MB). I speculate IPFS performs additional encapsulation of packets compared to the more minimalistic RAFDP protocol.
- Over LAN for both protocols, the order in terms of speed for both protocols is the same. Linux is the fastest, followed by macOS, leaving Windows as the slowest (for caching, macOS overtakes Linux for both protocols).
- For IPFS, the difference in slowness for Windows is more pronounced (slowing down by a factor of around 3 in comparison to RAFDP), while the difference is slowness between Linux and macOS has narrowed (from a 3 times difference to merely a 30% difference).
- It's possible that Linux is faster than the other operating systems because FUSE (which sshfs utilises) has been partially incorporated into the Linux kernel (Nödler, 2005), while the libraries WinFsp (for Windows) and fuse-t (for macOS) run entirely in userspace and so would be slower.
- Overall, RAFDP is much faster on Windows compared to IPFS (3 times faster, and 16 times faster for caching), on par with Linux (but twice as faster for caching) and

somewhat slower than macOS (being approximately twice as slow over LAN, but identical timings in terms of caching).

Chapter 7

Conclusions

7.1 Contributions

This dissertation has contributed extensively to the field of peer-to-peer file sharing protocols and virtual filesystems. The following list is not exhaustive but highlights the major contributions this project has made to the field:

- The first simple and streamlined peer-to-peer content-addressable file-transfer protocol. Existing protocols such as IPFS are excessively complicated (such that there are few existing third-party implementations) while BitTorrent is not content-addressable.
- The first reliable combination of a content-addressable file-transfer protocol with a virtual filesystem that allows files to be read natively by the operating system and only serving the parts that have been requested instead of the entire file. Other solutions such as IPFS only have an experimental and platform-specific virtual filesystem implementation (unlike the solution proposed here which supports Windows, Linux and macOS).
- A thorough literature, data and technology survey which examines the current state of the art of the area and highlights the fact that none of the solutions can fully resolve the problems faced except the solution developed in this dissertation.

7.2 Future Work

Despite the project's success, I have identified a few limitations that could be addressed in the future.

As mentioned in the Testing section, I discovered that macOS limits the UDP max datagram size to 9,216 bytes, which interferes with RPC communication between the RAFDP daemon and CLI. Although a solution was implemented which temporarily (i.e. until reboot) raises the limit, future work could involve resolving the issue entirely by either performing my own custom fragmentation of the UDP datagrams in the RPC communication (re-using the same method as stated in the Implementation for the RAFDP protocol itself) or re-architecturing the RPC protocol to use a different transmission mechanism (such as using TCP for the protocol instead).

BitTorrent was not integrated into the implementation as a possible file transfer protocol.

This is because it is not natively content addressable so it will be difficult to integrate in as the implementation relies on CIDs. A representation of the torrent file known as an infohash (Boyd, 2020) does exist, but as torrents can contain multiple files, it is ambiguous which file the infohash is referring to. Possible future work if BitTorrent is integrated in would either be incorporating the path of a particular file in the torrent into the infohash to create a proper CID or allowing the user to specify which file they want to download from the torrent via the CLI.

7.3 Reflection

Overall, this project has gone extremely well. I was motivated to fully resolve the problem I described in the introduction and literature, technology and data survey through the development of a software implementation, and I feel this has been achieved as all requirements listed in the Requirements section have been met. Particularly commendable is meeting requirements 4.2 and 4.3, where contrary to my expectations the RAFDP protocol served files to the virtual filesystem much faster on Windows (particularly for caching), on par with Linux (but faster for caching) and somewhat slower than macOS compared to IPFS. This is because I had assumed that IPFS protocol has multiple developers and has been developed over a longer period of time so it would be engineered for consistent better performance, but it appears its complexity (as IPFS contains multiple subsystems for file transfer protocols, peer discovery etc. (Paisano and Schilling, 2023)) has diminished its performance on Windows in comparison to the simpler RAFDP protocol. This project has broken new ground in the area of peer-to-peer file transfer protocols and virtual filesystems and will revolutionise the way in which these activities are conducted.

(This dissertation has 10,019 words)

Chapter 8

Bibliography

Adhikari, V.K., Guo, Y., Hao, F., Varvello, M., Hilt, V., Steiner, M., and Zhang, Z.-L., 2012. Unreeling netflix: understanding and improving multi-CDN movie delivery. *2012 proceedings IEEE INFOCOM* [Online]. 2012 Proceedings IEEE INFOCOM. ISSN: 0743-166X, pp.1620–1628. Available from: `https://doi.org/10.1109/INFCOM.2012.6195531`.

Apple, 2021. *System and kernel extensions in macOS* [Apple Support] [Online]. Available from: `https://support.apple.com/en-gb/guide/deployment/depa5fb8376f/web`.

Ashcraft, A., Sharkey, K., Coulter, D., Batchelor, D., and Satran, M., 2021. *Microsoft SMB protocol and CIFS protocol overview - win32 apps* [Online]. Available from: `https://learn.microsoft.com/en-us/windows/win32/fileio/microsoft-smb-protocol-and-cifs-protocol-overview`.

Azaghal, 2012. *Diagram of a binary hash tree* [Online]. Available from: `https://commons.wikimedia.org/wiki/File:Hash_Tree.svg`.

Baudiš, P., 2014. *Current concepts in version control systems* [Online]. arXiv:1405.3496. arXiv. arXiv: `1405.3496[cs]`. Available from: `https://doi.org/10.48550/arXiv.1405.3496`.

Beejor, 2016. *Answer to "what is the largest safe UDP packet size on the internet"* [Stack Overflow] [Online]. Available from: `https://stackoverflow.com/a/35697810`.

Benet, J., 2014a. *IPFS - content addressed, versioned, p2p file system* [Online]. arXiv:1407.3561. arXiv. arXiv: `1407.3561[cs]`. Available from: `https://doi.org/10.48550/arXiv.1407.3561`.

Benet, J., 2014b. *Multihash* [Online]. Multiformats. Available from: `https://github.com/multiformats/multihash`.

Benet, J., 2015. *Multicodec* [Online]. Multiformats. Available from: `https://github.com/multiformats/multicodec`.

Benet, J., 2016a. *CID (content IDentifier) specification* [Online]. Multiformats. Available from: `https://github.com/multiformats/cid`.

Benet, J., 2016b. *Multibase* [Online]. Multiformats. Available from: `https://github.com/multiformats/multibase`.

Benet, J., 2017. *Multiformats* [Online]. Available from: `https://multiformats.io/`.

Bishop, M., 2022. *HTTP/3* [Online]. (Request for Comments RFC 9114). Num Pages: 57. Internet Engineering Task Force. Available from: `https://doi.org/10.17487/RFC9114`.

Boyd, I., 2020. *Answer to "what exactly is the info_hash in a torrent file"* [Stack Overflow] [Online]. Available from: `https://stackoverflow.com/a/60132513`.

Brindescu, C., Codoban, M., Shmarkatiuk, S., and Dig, D., 2014. How do centralized and distributed version control systems impact software changes? *Proceedings of the 36th international conference on software engineering* [Online], ICSE 2014. New York, NY, USA: Association for Computing Machinery, pp.322–333. Available from: `https://doi.org/10.1145/2568225.2568322`.

Calboli, I., 2022. Legal perspectives on the streaming industry: the united states. *The american journal of comparative law* [Online], 70 (Supplement_1), pp.i220–i245. Available from: `https://doi.org/10.1093/ajcl/avac021`.

Cimpanu, C., 2017. *SHA1 collision attack can serve backdoored torrents to track down pirates* [BleepingComputer] [Online]. Available from: `https://www.bleepingcomputer.com/news/security/sha1-collision-attack-can-serve-backdoored-torrents-to-track-down-pirates/`.

Cohen, B., 2008. *The BitTorrent protocol specification* [Online]. Available from: `https://www.bittorrent.org/beps/bep_0003.html`.

Cohen, J.P. and Lo, H.Z., 2014. Academic torrents: a community-maintained distributed repository. *Proceedings of the 2014 annual conference on extreme science and engineering discovery environment* [Online], XSEDE '14. New York, NY, USA: Association for Computing Machinery, pp.1–2. Available from: `https://doi.org/10.1145/2616498.2616528`.

Cohen, J.P. and Lo, H.Z., 2016. *Academic torrents: scalable data distribution* [Online]. arXiv:1603.04395. arXiv. arXiv: `1603.04395[cs]`. Available from: `https://doi.org/10.48550/arXiv.1603.04395`.

Costa, P., 2023. *Fusetree* [Online]. Available from: `https://github.com/paulo-raca/fusetree`.

Cunningham, B.M., Alexander, P.J., and Adilov, N., 2004. Peer-to-peer file sharing communities. *Information economics and policy* [Online], 16(2), pp.197–213. Available from: `https://doi.org/10.1016/j.infoecopol.2003.09.009`.

Dickens, C., 1998. *Great expectations* [Online]. Available from: `https://www.gutenberg.org/ebooks/1400`.

Dinneen, J.D. and Nguyen, B.X., 2021. How big are peoples' computer files? file size distributions among user-managed collections. *Proceedings of the association for information science and technology* [Online], 58(1), pp.425–429. Available from: `https://doi.org/10.1002/pra2.472`.

Envisional, 2011. *An estimate of infringing use of the internet* [Online]. Available from: https://www.ics.uci.edu/~sjordan/courses/ics11/case_studies/Envisional-Internet_Usage-Jan2011-4.pdf.

Fishman, A., 2022. *Macos-fuse-t/sshfs* [Online]. Available from: https://github.com/macos-fuse-t/sshfs.

Fishman, A., 2023a. *FUSE-t* [Online]. Available from: https://www.fuse-t.org/.

Fishman, A., 2023b. *Macos-fuse-t/fuse-t* [Online]. Available from: https://github.com/macos-fuse-t/fuse-t.

Fleischer, B., 2023a. *Osxfuse/macFUSE* [Online]. macFUSE. Available from: https://github.com/osxfuse/osxfuse.

Fleischer, B., 2023b. *Osxfuse/sshfs* [Online]. macFUSE. Available from: https://github.com/osxfuse/sshfs.

Fleishman, G., 2020. *AFP is no longer supported in macOS big sur. here's the fix* [Macworld] [Online]. Available from: https://www.macworld.com/article/234926/using-afp-to-share-a-mac-drive-its-time-to-change.html.

Forcier, J., 2023. *Paramiko* [Online]. paramiko. Available from: https://github.com/paramiko/paramiko.

Fu, K., Kaashoek, M.F., and Mazières, D., 2002. Fast and secure distributed read-only file system. *ACM transactions on computer systems* [Online], 20(1), pp.1–24. Available from: https://doi.org/10.1145/505452.505453.

Gierth, L., 2017. *Fix storage limits · issue #3066 · ipfs/kubo* [GitHub] [Online]. Available from: https://github.com/ipfs/kubo/issues/3066#issuecomment-343036019.

Gogioso, S., 2023. *Multiformats: python implementation of multiformat protocols* [Online]. Hashberg. Available from: https://github.com/hashberg-io/multiformats.

Honles, T., 2023. *Fusepy* [Online]. fusepy. Available from: https://github.com/fusepy/fusepy.

Hultstrand, S. and Olofsson, R., 2015. *Git - CLI or GUI : which is most widely used and why?* [Online]. Available from: http://urn.kb.se/resolve?urn=urn:nbn:se:bth-10539.

John, S., 2022. *File systems | IPFS docs* [Online]. Available from: https://docs.ipfs.tech/concepts/file-systems/#unix-file-system-unixfs.

John Howard, 1988. An overview of the andrew file system. *USENIX winter technical conference* [Online], pp.23–26. Available from: http://reports-archive.adm.cs.cmu.edu/anon/itc/CMU-ITC-062.pdf.

jusx, 2016. *Answer to "UDP message too long"* [Stack Overflow] [Online]. Available from: https://stackoverflow.com/a/35335138.

Kim, H., Agrawal, N., and Ungureanu, C., 2012. Revisiting storage for smartphones. *ACM transactions on storage* [Online], 8(4), 14:1–14:25. Available from: `https://doi.org/10.1145/2385603.2385607`.

Kryder, M.H. and Kim, C.S., 2009. After hard drives—what comes next? *IEEE transactions on magnetics* [Online], 45(10). Conference Name: IEEE Transactions on Magnetics, pp.3406–3413. Available from: `https://doi.org/10.1109/TMAG.2009.2024163`.

Laricchia, F., 2022. *Average number of connected devices in UK households 2020* [Statista] [Online]. Available from: `https://www.statista.com/statistics/1107269/average-number-connected-devices-uk-house/`.

Magiera, Ł. and Rataj, M., 2022. *Kubo (FUSE)* [Online]. IPFS. Available from: `https://github.com/ipfs/kubo/blob/e4908a01633582e205b26ab36769830fe8490c10/docs/fuse.md`.

Marakasov, D., 2023. *Fusefs:sshfs packages dissection - repology* [Online]. Available from: `https://repology.org/project/fusefs:sshfs/information`.

Martin, S., 2014. *Illustration of a BitTorrent filesharing network* [Online]. Available from: `https://commons.wikimedia.org/wiki/File:BitTorrent_network.svg`.

Matthews, J. and Schilling, J., 2023. *Kubo CLI | IPFS docs* [Online]. Available from: `https://docs.ipfs.tech/reference/kubo/cli/`.

Mazieres, D., 2000. *Self-certifying file system* [Online]. AAI0802720. PhD thesis. USA: Massachusetts Institute of Technology. Available from: `https://dspace.mit.edu/bitstream/handle/1721.1/86610/48227669-MIT.pdf?sequence=2`.

McQuistin, S., Perkins, C., and Fayed, M., 2016. Implementing real-time transport services over an ossified network. *Proceedings of the 2016 applied networking research workshop* [Online], ANRW '16. New York, NY, USA: Association for Computing Machinery, pp.81–87. Available from: `https://doi.org/10.1145/2959424.2959443`.

Melnikov, A. and Fette, I., 2011. *The WebSocket protocol* [Online]. (Request for Comments RFC 6455). Num Pages: 71. Internet Engineering Task Force. Available from: `https://doi.org/10.17487/RFC6455`.

Merkle, R.C., 1982. *Method of providing digital signatures* [Online]. U.S. patent 4309569A. L.S.J. University. Available from: `https://patents.google.com/patent/US4309569A/en`.

Mockapetris, P., 1987. *Domain names - concepts and facilities* [Online]. (Request for Comments RFC 1034). Num Pages: 55. Internet Engineering Task Force. Available from: `https://doi.org/10.17487/RFC1034`.

Mohd Suki, N., 2013. Students' demand for smartphones: structural relationships of product features, brand name, product price and social influence. *Campus-wide information systems* [Online], 30(4). Publisher: Emerald Group Publishing Limited, pp.236–248. Available from: `https://doi.org/10.1108/CWIS-03-2013-0013`.

Morris, J.H., Satyanarayanan, M., Conner, M.H., Howard, J.H., Rosenthal, D.S., and Smith, F.D., 1986. Andrew: a distributed personal computing environment. *Communications of the ACM* [Online], 29(3), pp.184–201. Available from: `https://doi.org/10.1145/5666.5671`.

Moy, J., 1998. *OSPF version 2* [Online]. (Request for Comments RFC 2328). Num Pages: 244. Internet Engineering Task Force. Available from: `https://doi.org/10.17487/RFC2328`.

mxmlnkn, 2022. *Comment on answer to "which module is the actual interface to FUSE from python 3?"* [Stack Overflow] [Online]. Available from: `https://stackoverflow.com/questions/52925566/which-module-is-the-actual-interface-to-fuse-from-python-3#comment129584262_52943795`.

Nielsen, H., Mogul, J., Masinter, L.M., Fielding, R.T., Gettys, J., Leach, P.J., and Berners-Lee, T., 1999. *Hypertext transfer protocol – HTTP/1.1* [Online]. (Request for Comments RFC 2616). Num Pages: 176. Internet Engineering Task Force. Available from: `https://doi.org/10.17487/RFC2616`.

Nödler, J., 2005. *File-systems.fuse.devel - FUSE merged to 2.6.14! - msg#00021 - recent discussion OSDir.com* [Online]. Available from: `https://web.archive.org/web/20160420173822/http://osdir.com/ml/file-systems.fuse.devel/2005-09/msg00021.html`.

Norberg, A., 2020. *BitTorrent v2* [Online]. Available from: `https://blog.libtorrent.org/2020/09/bittorrent-v2/`.

Paisano, E. and Schilling, J., 2023. *Subsystems overview | how IPFS works | IPFS docs* [Online]. Available from: `https://docs.ipfs.tech/concepts/how-ipfs-works/#subsystems-overview`.

Perez De Rosso, S. and Jackson, D., 2013. What's wrong with git? a conceptual design analysis. *Proceedings of the 2013 ACM international symposium on new ideas, new paradigms, and reflections on programming & software* [Online], Onward! 2013. New York, NY, USA: Association for Computing Machinery, pp.37–52. Available from: `https://doi.org/10.1145/2509578.2509584`.

Perréard, J.-L., 2020. *Poetic ballad in the empty streets of paris* [Online]. Available from: `https://commons.wikimedia.org/wiki/File:Paris_lockdown_-_Vimeo.webm`.

Prechelt, L., 2000. An empirical comparison of seven programming languages. *Computer* [Online], 33(10). Conference Name: Computer, pp.23–29. Available from: `https://doi.org/10.1109/2.876288`.

Protocol Labs, 2022. *Ipfs/kubo* [Online]. IPFS. Available from: `https://github.com/ipfs/kubo`.

Rataj, M., 2022. *Content addressing and CIDs | IPFS docs* [Online]. Available from: `https://docs.ipfs.tech/concepts/content-addressing/`.

Samba Team, 2023. *Samba - opening windows to a wider world* [Online]. Available from: `https://www.samba.org/`.

Sandvine, 2019. *The global internet phenomena report (september 2019)*.

Schilling, J., 2022. *How IPFS works | IPFS docs* [Online]. Available from: `https://docs.ipfs.tech/concepts/how-ipfs-works/`.

Schumann, L., Doan, T.V., Shreedhar, T., Mok, R., and Bajpai, V., 2022. *Impact of evolving protocols and COVID-19 on internet traffic shares* [Online]. arXiv:2201.00142. arXiv. arXiv: `2201.00142[cs]`. Available from: `https://doi.org/10.48550/arXiv.2201.00142`.

Solheim, E., 2008. *Thoughts on BitTorrent distribution for a public broadcaster* [NRKbeta] [Online]. Available from: `https://nrkbeta.no/2008/03/02/thoughts-on-bittorrent-distribution-for-a-public-broadcaster/`.

Stack Overflow, 2022. *Stack overflow developer survey 2022* [Stack Overflow] [Online]. Available from: `https://survey.stackoverflow.co/2022/#version-control-version-control-system`.

Stark, L., 2022. *Dokany* [Online]. Dokan. Available from: `https://github.com/dokan-dev/dokany`.

Tan, M. and Teo, T.S., 1998. Factors influencing the adoption of the internet. *International journal of electronic commerce* [Online], 2(3). Publisher: Routledge, pp.5–18. Available from: `https://doi.org/10.1080/10864415.1998.11518312`.

Taylor, A., Whalley, A., Jansens, D., and Oskov, N., 2021. *An update on memory safety in chrome* [Google Online Security Blog] [Online]. Available from: `https://security.googleblog.com/2021/09/an-update-on-memory-safety-in-chrome.html`.

the8472, 2018. *Would a v2-aware DHT allow individual file search? · issue #77 · bittorrent/bittorrent.org* [GitHub] [Online]. Available from: `https://github.com/bittorrent/bittorrent.org/issues/77`.

Thomas, G., 2019. *A proactive approach to more secure code – microsoft security response center* [Online]. Available from: `https://msrc-blog.microsoft.com/2019/07/16/a-proactive-approach-to-more-secure-code/`.

Unwin, A. and Hofmann, H., 1999. *GUI and command-line – conflict or synergy?* University of Augsburg.

Vangoor, B.K.R., Tarasov, V., and Zadok, E., 2017. To {FUSE} or not to {FUSE}: performance of {user-space} file systems [Online]. 15th USENIX Conference on File and Storage Technologies (FAST 17), pp.59–72. Available from: `https://www.usenix.org/conference/fast17/technical-sessions/presentation/vangoor`.

Voicu, A., 2018. *Tabby cat with blue eyes* [Online]. Available from: `https://commons.wikimedia.org/wiki/File:Tabby_cat_with_blue_eyes-3336579.jpg`.

Warren, T., 2015. *Microsoft to deliver windows 10 updates using peer-to-peer technology* [The Verge] [Online]. Available from: `https://www.theverge.com/2015/3/15/8218215/microsoft-windows-10-updates-p2p`.

Zac67, 2022. *Answer to "the maximum length of a datagram packet is 65,536 or 65,535?"* [Super User] [Online]. Available from: `https://superuser.com/a/1697822`.

Zakharov, M., 2018. *Answer to "which module is the actual interface to FUSE from python 3?"* [Stack Overflow] [Online]. Available from: `https://stackoverflow.com/a/52943795`.

Zelinskie, J., 2016. *Tracker protocol extension: scrape* [Online]. Available from: `https://www.bittorrent.org/beps/bep_0048.html`.

Zhang, C., Dhungel, P., Wu, D., and Ross, K.W., 2011. Unraveling the BitTorrent ecosystem. *IEEE transactions on parallel and distributed systems* [Online], 22(7). Conference Name: IEEE Transactions on Parallel and Distributed Systems, pp.1164–1177. Available from: `https://doi.org/10.1109/TPDS.2010.123`.

Zissimopoulos, B., 2016. *Add information "comparing" WinFsp and dokany · issue #19 · winfsp/winfsp* [GitHub] [Online]. Available from: `https://github.com/winfsp/winfsp/issues/19#issuecomment-265368159`.

Zissimopoulos, B., 2022. *WinFsp · windows file system proxy* [Online]. WinFsp. Available from: `https://github.com/winfsp/winfsp`.

Appendix A

Full test plan

A.1 Unit testing

Test Number	Test Case	Expected Result	Actual Result	Pass/Fail
1.1	Test that the custom varint function can encode and decode integers properly	The varint function encodes and decodes a random integer while successfully separating and ignoring redundant randomised data	The varint function encoded and decoded a random integer while successfully separating and ignoring redundant randomised data	Pass
2.1	Test that a new Merkle tree can be built from a pre-built one via requesting values from hashes one at a time	The items in the new Merkle tree are the same as the pre-built one	The items in the new Merkle tree were the same as the pre-built one	Pass

3.1	Test that in-memory filesystem representation for the virtual SFTP server works correctly	Existing files and folders (some added during the test) return the correct values, while non-existing ones are also handled	Existing files and folders (some added during the test) returned the correct values, while non-existing ones were also handled	Pass

A.2 Integration testing

Test Number	**Test Case**	**Expected Result**	**Actual Result**	**Pass/Fail**
4.1	Test that a RAFDP peer is able to obtain the contents for a hash from another RAFDP peer	The first RAFDP peer successfully receives the expected bytes for a hash from another RAFDP peer	The first RAFDP peer successfully received the expected bytes for a hash from another RAFDP peer	Pass
5.1	Test that a RAFDP peer is able to obtain the entire Merkle tree for a hash from another RAFDP peer	The contents of the assembled received file from the other peer are the same as when the file is read locally	The contents of the assembled received file from the other peer were the same as when the file is read locally	Pass
6.1	Test that a RAFDP peer is able to fetch a random byte range **smaller** than the chunk size, **starting in** the file and **ending in** the file, that is represented by a hash from another RAFDP peer	The other RAFDP peer returns an identical byte range to that which would be returned if the file was read locally (in this case, some bytes less than 16 KiB in size)	The other RAFDP peer returned an identical byte range that matched the one read from the file locally	Pass

6.2	Test that a RAFDP peer is able to fetch a random byte range **smaller** than the chunk size, **starting in** the file and **ending beyond** the end of the file, that is represented by a hash from another RAFDP peer	The other RAFDP peer returns an identical byte range to that which would be returned if the file was read locally (in this case, some bytes less than 16 KiB in size)	The other RAFDP peer returned an identical byte range that matched the one read from the file locally	Pass
6.3	Test that a RAFDP peer is able to fetch a random byte range **smaller** than the chunk size, **starting beyond** the end of the file and **ending beyond** the end of the file, that is represented by a hash from another RAFDP peer	The other RAFDP peer returns an identical byte range to that which would be returned if the file was read locally (in this case, some bytes less than 16 KiB in size)	The other RAFDP peer returned an identical byte range that matched the one read from the file locally	Pass
6.4	Test that a RAFDP peer is able to fetch a random byte range **larger** than the chunk size, **starting in** the file and **ending in** the file, that is represented by a hash from another RAFDP peer	The other RAFDP peer returns an identical byte range to that which would be returned if the file was read locally (in this case, some bytes more than 16 KiB in size)	The other RAFDP peer returned an identical byte range that matched the one read from the file locally	Pass

6.5	Test that a RAFDP peer is able to fetch a random byte range **larger** than the chunk size, **starting in** the file and **ending beyond** the end of the file, that is represented by a hash from another RAFDP peer	The other RAFDP peer returns an identical byte range to that which would be returned if the file was read locally (in this case, some bytes more than 16 KiB in size)	The other RAFDP peer returned an identical byte range that matched the one read from the file locally	Pass
6.6	Test that a RAFDP peer is able to fetch a random byte range **larger** than the chunk size, **starting beyond** the end of the file and **ending beyond** the end of the file, that is represented by a hash from another RAFDP peer	The other RAFDP peer returns an identical byte range to that which would be returned if the file was read locally (in this case, some bytes more than 16 KiB in size)	The other RAFDP peer returned an identical byte range that matched the one read from the file locally	Pass
6.7	Test that a RAFDP peer is able to fetch a random byte range the **same size** as the chunk size, **starting in** the file and **ending in** the file, that is represented by a hash from another RAFDP peer	The other RAFDP peer returns an identical byte range to that which would be returned if the file was read locally (in this case, some bytes 16 KiB in size)	The other RAFDP peer returned an identical byte range that matched the one read from the file locally	Pass

6.8	Test that a RAFDP peer is able to fetch a random byte range the **same size** as the chunk size, **starting in** the file and **ending beyond** the end of the file, that is represented by a hash from another RAFDP peer	The other RAFDP peer returns an identical byte range to that which would be returned if the file was read locally (in this case, some bytes 16 KiB in size)	The other RAFDP peer returned an identical byte range that matched the one read from the file locally	Pass
6.9	Test that a RAFDP peer is able to fetch a random byte range the **same size** as the chunk size, **starting beyond** the end of the file and **ending beyond** the end of the file, that is represented by a hash from another RAFDP peer	The other RAFDP peer returns an identical byte range to that which would be returned if the file was read locally (in this case, some bytes 16 KiB in size)	The other RAFDP peer returned an identical byte range that matched the one read from the file locally	Pass
7.1	Test that two RAFDP peers can find each other using a BitTorrent tracker	Each peer has the other peer in its tracker list	Each peer had the other peer in its tracker list	Pass

A.3 System testing

Test Number	**Test Case**	**Expected Result**	**Actual Result**	**Pass/Fail**

8.1	Start the virtual filesystem	A virtual drive should automount on macOS and Linux, but not on Windows. On all OSes the window should show messages indicating RAFDP + RAFDP RPC + virtual filesystem RPC is listening at port	A virtual drive automounted on macOS and Linux, but not on Windows. On all OSes the window showed messages indicating RAFDP + RAFDP RPC + virtual filesystem RPC is listening at port	Pass on Windows Pass on macOS Pass on Linux
8.2	Start a RAFDP process	A window should open with the message stating the RAFDP process is listening at a port	A window opened with the message stating the RAFDP process is listening at a port	Pass on Windows Pass on macOS Pass on Linux
8.3	Add a real file to the RAFDP process via the CLI	The CLI should print out the RAFDP hash of the file	The CLI printed out the RAFDP hash of the file	Pass on Windows Pass on macOS Pass on Linux
8.4	Add the IP address and port of the RAFDP process to the virtual filesystem via the CLI	The CLI should print out the IP address and the port of the peer added	The CLI printed out the IP address and the port of the peer added	Pass on Windows Pass on macOS Pass on Linux
8.5	Add the RAFDP hash for a test video file to the virtual filesystem via the CLI	The CLI should state the file has been added successfully	The CLI stated the file had been added successfully	Pass on Windows Pass on macOS Pass on Linux
8.6	Assign a location for the RAFDP hash file to appear via the CLI	The virtual test video file should appear at the assigned location	The virtual test video file appeared at the assigned location	Pass on Windows Pass on macOS Pass on Linux

8.7	Open the virtual test video file from where it was assigned to in 8.6	The virtual test video file should open successfully and start playing	The virtual test video file opened successfully and started playing	Pass on Windows Pass on macOS Pass on Linux
8.8	Seek the virtual test video file to roughly halfway	The virtual test video file should continue playing from the new position after a delay	The virtual test video file continued playing from the new position after a delay	Pass on Windows Pass on macOS Pass on Linux