Faculty of Science and Technology

MASTER'S THESIS

Study program/Specialization: Computer Science Reliable and Secure Systems	Spring semester, 2021 Open
Writers: Eivind Mellemstrand Stavnes Daniel Urdal	Eivind Stavnes (Writer's signature) Daniel Urdal (Writer's signature)
Faculty supervisors: Hein Meling Racin Nygaard	
Thesis title: Extending the Snarl File Repair Component for Distributed Storage Systems	
Credits (ECTS): 30	
Key words: distributed storage systems erasure codes alpha entanglement codes file recovery IPFS Swarm	Pages: 86 + enclosure: Code on GitHub Stavanger, 15.06.2021 Date/year

Extending the Snarl File Repair Component for Distributed Storage Systems

Eivind Stavnes & Daniel Urdal

Tuesday 15th June, 2021

Acknowledgements

We would like to give our sincerest thanks to our supervisors Hein Meling and Racin Nygaard for their excellent guidance. Their feedback and discussions during our regular meetings have been invaluable. We also appreciate all their efforts in reviewing and providing detailed feedback for our drafts.

Abstract

This thesis extends the Snarl file repair component for distributed storage systems, and evaluates extensions. Snarl is an application using alpha entanglement codes to improve recovery rates of content stored in distributed storage systems. This work extends Snarl by adapting it such that it can be used for other systems than the Swarm network, which it was limited to in the original implementation. The extensions include an abstraction layer making it simple to extend Snarl to be used with other systems, and changes to Snarl's core algorithms to separate tasks into distinct processes. Additionally, an extension for the InterPlanetary File System (IPFS) is added. Finally, the newly added IPFS extension is evaluated. File recovery rates and network overheads are measured for several different percentages of data loss and peer loss in the network, demonstrating that Snarl performs well in these scenarios, and provides much better file recovery rates than simple data replication with the same storage overhead.

Contents

1 Introduction **3**
- 1.1 Motivation . . . 3
- 1.2 Challenges . . . 4
- 1.3 Contributions . . . 5
- 1.4 Outline . . . 5

2 Background **6**
- 2.1 Preliminaries . . . 6
  - 2.1.1 Hashing . . . 6
  - 2.1.2 Merkle Trees . . . 6
  - 2.1.3 Overlay Networks . . . 7
  - 2.1.4 Distributed Hash Table . . . 7
  - 2.1.5 Erasure Coding . . . 8
- 2.2 Decentralized Distributed Storage Systems . . . 8
  - 2.2.1 Overview . . . 8
  - 2.2.2 Swarm . . . 8
  - 2.2.3 InterPlanetary File System . . . 9
- 2.3 Alpha Entanglement Codes . . . 11
  - 2.3.1 Parameters . . . 12
  - 2.3.2 Strands . . . 12
  - 2.3.3 Entanglement . . . 14
  - 2.3.4 Repair . . . 15
- 2.4 Snarl . . . 16
  - 2.4.1 Overview . . . 16
  - 2.4.2 Entanglement Algorithm . . . 17
  - 2.4.3 File Recovery and Repair Algorithm . . . 21

3 Design **24**
- 3.1 Backend Requirements . . . 24
  - 3.1.1 Block DAG Requirements . . . 24
  - 3.1.2 Node Requirements . . . 25
  - 3.1.3 Requirements from the End User . . . 26
- 3.2 Abstraction Layer . . . 27
  - 3.2.1 Updated Architecture . . . 27
  - 3.2.2 Design Goals . . . 27
  - 3.2.3 The Node Abstraction . . . 29
  - 3.2.4 Entanglement . . . 31
  - 3.2.5 Uploading . . . 31
  - 3.2.6 Recovery . . . 32
  - 3.2.7 Repair . . . 34
- 3.3 IPFS Peer Management Layer . . . 35
  - 3.3.1 Motivation . . . 36
  - 3.3.2 Overview . . . 36

3.3.3 Requirements . . . 37

4 Implementation **38**
4.1 Backend API . . . 38
4.1.1 Node and DAG Representations . . . 38
4.1.2 Entanglement . . . 40
4.1.3 Uploading . . . 42
4.1.4 Node Retrieval and Deserialization . . . 43
4.1.5 Recovery and Repair . . . 44
4.2 Index Mappings . . . 47
4.2.1 Motivation . . . 47
4.2.2 The Index Mapping Data Structure . . . 47
4.2.3 Accessing Parity Blocks . . . 49
4.2.4 Swapping Internal Nodes . . . 49
4.3 Changes to the Lattice Type . . . 49
4.3.1 Implementing the Repairer Interface . . . 50
4.3.2 Updated Parity Block Retrieval Approach . . . 51
4.3.3 Updated Constructor Function . . . 55
4.4 IPFS Backend . . . 56
4.4.1 Node Implementation . . . 56
4.4.2 Node Getter . . . 59
4.4.3 Flattened Merkle DAG Constructor . . . 61
4.4.4 The Proxy Type . . . 61
4.5 Updates to the Swarm Backend . . . 62
4.5.1 Node Implementation . . . 62
4.5.2 Entanglement and Recovery . . . 63

5 Evaluation **65**
5.1 Experimentation Setup . . . 65
5.1.1 Simulation strategies . . . 65
5.1.2 Simulation procedure . . . 66
5.1.3 Runtime Metrics . . . 66
5.2 File Availability . . . 67
5.2.1 Experiment: Node Loss . . . 67
5.2.2 Experiment: Peer Failure . . . 68
5.3 Network Overhead . . . 69

6 Discussion and Insights **71**
6.1 File Availability Experiments . . . 71
6.2 Distribution Strategies . . . 72
6.3 Network Overhead Results . . . 73
6.4 Deadlocks . . . 73

7 Future Work **75**
7.1 Swarm using the Bee Client . . . 75
7.2 Retrieval Optimization . . . 75
7.3 Optimizing Memory Consumption . . . 76
7.4 Updated Repairer Implementation using CSP . . . 76
7.5 Updates to Parity File Layout . . . 76

8 Conclusion **78**

Chapter 1

Introduction

In this thesis we extend and evaluate the Snarl file repair component for distributed storage systems (DSSes). Snarl is an application which provides redundancy and repair opportunities for content stored in an underlying DSS, without requiring modification of the DSS itself [19]. It achieves this by using alpha entanglement (AE) codes; a family of erasure codes built on data entanglement, which can effectively repair damaged portions of data with few operations [5]. The contributed work extends Snarl by adding an abstraction layer to support different DSS backends, adds support for the InterPlanetary File System (IPFS) as a backend option, updates the original Swarm backend implementation, modifies Snarl's core algorithms, and conducts experiments to evaluate the utility of Snarl for the IPFS backend.

1.1 Motivation

In recent years there has been a surge of popularity in decentralized systems, such as blockchain applications and the decentralized web. Many blockchain applications, due to their replicative nature, depend on off-chain storage for cost reduction and efficiency (e.g. Ethereum smart contracts). Additionally, systems such as IPFS use fully decentralized models for storage and content distribution. Compared to centrally controlled platforms such as Amazon Web Services (AWS) and Microsoft Azure, these platforms have additional challenges when it comes to providing data availability and redundancy.

The decentralized storage systems generally have a mutual lack of trust between peers, and require mechanisms for ensuring correct operation. Firstly, when content is received from an untrusted peer, it must be possible to ensure that requested content has not been tampered with. Typically these systems use hash functions to validate content, and by extension use Merkle trees to efficiently prove that a block of data is part of the requested content. Secondly, the reliability and availability of other peers cannot be depended upon. Contrary to cloud services such as AWS which guarantee a certain number of nines for the reliability of their services (e.g. five nines is 99.999 % reliable), decentralized networks have fewer guarantees. Data stored by another peer could become corrupted due to hardware failure. This may also be more likely to occur with consumer hardware used by arbitrary peers than with the professional-grade hardware used by cloud providers. Additionally, the data may disappear due to peers leaving or losing connection to the network. Thirdly, there must be an incentive for peers to provide the content. There is a lot of active research on incentives in decentralized networks with technologies such as proof-of-space blockchains, e.g. Filecoin and Storj, but it is not discussed further here.

Due to the decentralized nature of DSSes, it is thus harder to achieve reliability. To achieve acceptable probabilistic reliability properties, the DSS must ensure a high level of replication of the content, such that it is most likely available when needed. Redundancy through replication is less storage-efficient than alternatives such as erasure coding, which

motivates research into implementing more efficient redundancy schemes in decentralized environments to improve reliability.

Furthermore, the reliability provided by a particular DSS may not be easily controllable by end users. The level of reliability depends on the feature set and implementation details of the DSS. Additionally, an end user cannot be certain of the replication level of their data at any given time, and cannot simply check whether all the data is available in the network. Even the loss of a single data block could render some content useless. Ad-hoc approaches are possible, such as uploading slightly modified content to increase the level of replication. However, such an approach relies on replication, which is less efficient than e.g. erasure codes.

Snarl is a possible solution to these problems. With the right amount of abstraction, Snarl can be used as a general purpose tool to add additional redundancy to any Merkle DAG (directed acyclic graph) [19]. By using data entanglement as an additional step, the original content can be uploaded to the DSS alongside redundant entangled data. If some blocks of the data are missing when the end user is retrieving content, the redundant data can be used to recover the missing blocks, and the recovered blocks can be re-uploaded to the DSS. AE codes, used by Snarl, are more storage efficient than replication, and competitive with other forms of erasure codes in terms of data recovery and storage overhead, while requiring fewer operations to repair common failures [5]. Using Snarl as a virtual layer on top of a DSS gives end users additional control over the level of reliability for content they upload to the DSS, regardless of the reliability and type of the redundancy scheme implemented by the DSS itself. To enable Snarl to be easily implemented for a variety of DSSes, a well-defined abstraction layer is desirable.

Another issue is that decentralized networks may be slow, particularly if data is not widely replicated or due to network congestion in peers storing the desired data. Retrieval and repair of data blocks should therefore be performed concurrently to reduce retrieval time of the requested content. The repair algorithm and backend abstraction layers should maximize the amount of concurrent work to minimize recovery times.

1.2 Challenges

Abstraction Layer for DSS Backends While large parts of Snarl relate to AE codes, entanglement and repair algorithms, other parts of the codebase were tied to implementation details of the Swarm backend. The original Snarl implementation only supported Swarm as the backend DSS, and in many places the code depended on details of this backend. This made it challenging to add new backends without refactoring, highlighting a need for an abstraction layer. Different DSSes expose different application programming interfaces (APIs), use different data structures or layouts of data structures, and may use different formats to identify content they store. Additionally, an abstraction layer for Snarl DSS backends should make it as simple as possible to add support for new backends. This makes it challenging to design an abstraction layer that is simple to use while covering many potential backends of different configurations.

Concurrent Repair A property of AE codes is that they permit repairing blocks in parallel under certain conditions. Also, when downloading content split into several data blocks, concurrent downloads and repairs reduce the time to recover the file. Despite this, concurrency is difficult for any sufficiently advanced algorithm, and Snarl's repair algorithm is no exception. For example, under certain conditions, in order to repair a block, the repair algorithm must first repair other blocks that the original block depends on, which is a recursive process. Doing this serially causes unnecessary slowdown of the system. Instead, the original process can pause until its dependencies are met, and proceed once it's notified of this. Dependencies (and recursive dependencies) can be processed concurrently. However,

a concurrent implementation requires synchronization, control of state and to avoid common problems such as data races and deadlocks.

1.3 Contributions

In this thesis, we make the following contributions:

- We implement a set of APIs, types and code adjustments which form an abstraction layer. This enables the Snarl application to use other backends than the original Swarm backend. Related to this work we contribute bugfixes and refactor much of the Snarl codebase to make it more idiomatic and readable.
- Using the implemented abstraction layer, we implement IPFS as an alternative Snarl backend DSS. The implementation is accompanied by a test library for running several IPFS nodes in a test configuration, and a large number of unit tests.
- The original Swarm backend is adjusted to work with the abstraction layer. This includes both adapting existing code to be used with the introduced abstractions, as well as implementing new algorithms from scratch.
- We update portions of the Snarl repair algorithm to work with the aforementioned abstraction layer. The updated code uses a concurrency model based on Hoare's communicating sequential processes (CSP) [6], which achieves concurrency without locks or shared memory.
- We develop experiments to evaluate various performance metrics for the contributed IPFS backend implementations. There are two types of experiments, evaluating file availability after certain percentages of failure. The first relates to the loss of nodes needed to retrieve a file, and is run as a local simulation. The second simulates failure of IPFS peers, and is run as several connected Docker containers.

1.4 Outline

The remainder of the thesis is structured in the following way:

- Chapter 2 gives the reader the background information necessary to understand the rest of the thesis. It describes preliminaries, DSSes, AE codes, and gives an overview of the original Snarl implementation.
- Chapter 3 defines requirements for Snarl backends, describes the introduced abstraction layer, and the IPFS peer management layer used for experiments related to the IPFS backend.
- Chapter 4 describes implementation details of the abstraction layer, updates to Snarl algorithms, the implementation of the IPFS backend, and changes made to the Swarm backend.
- Chapter 5 evaluates the file availability rate and network overheads when using Snarl with the IPFS backend. First it evaluates file availability in the presence of the loss of several nodes, then in the presence of the failure of several IPFS peers.
- Chapter 6 discusses the results presented in Chapter 5.
- Chapter 7 describes several approaches that could be interesting for future work on Snarl.
- Chapter 8 concludes the thesis.

Chapter 2

Background

This chapter describes the fundamentals of decentralized distributed storage systems (DSSes), data entanglement and Snarl. First, we look at some data structures and methods commonly used in DSSes. Secondly, an overview of the DSSes Swarm and IPFS is given. Furthermore, alpha entanglement codes — the redundancy mechanism used by Snarl — is presented. Finally, a high-level overview of the Snarl file repair component is given.

2.1 Preliminaries

In this section, we look at some selected concepts useful for understanding distributed storage systems. This includes Hashing, Merkle trees, overlay networks, distributed hash tables and erasure coding.

2.1.1 Hashing

Fingerprinting of large blocks of data can be performed using hashing [25]. Variable-length content can be used as input to a hash function, producing a fixed-length hash value. Hashing the same content multiple times should always produce the same hash, and using a good hash function should ensure collisions, which occur when different content produces the same hash, are close to non-existent in practice. This means a small hash can uniquely identify large pieces of content.

2.1.2 Merkle Trees

Verification of large amounts of data can be performed using a Merkle tree, also known as a hash tree. Introduced by Ralph Merkle in 1979 [17], this is a tree where each leaf node contains the hash of a block of data, and each non-leaf node contains the hash of its children's combined hashes. To verify if a data block is present using a Merkle tree, we no longer need all the data in order to recreate a hash. It can instead be recreated using a subset of intermediary hashes. The structure of a Merkle tree is shown in Figure 2.1.

2.1.2.1 Proof of Membership

To prove a certain block of data is present in a file, we need to recreate the root hash of the Merkle tree [18]. This can be performed by hashing the data block, and recreating hashes upwards in the tree until reaching the root. Only the hashes opposite the generated ones are needed in order to complete the proof.

Using the tree in Figure 2.1 as an example, let's say we received a copy of block *L2* and want to verify that the data is correct. We would first hash *L2* to get *Hash 0-1*. *Hash 0-0* would have to be retrieved in order to be able to generate *Hash 0*, and *Hash 1* in order to generate *Top Hash*. This hash could then be compared to the stored root hash to verify that

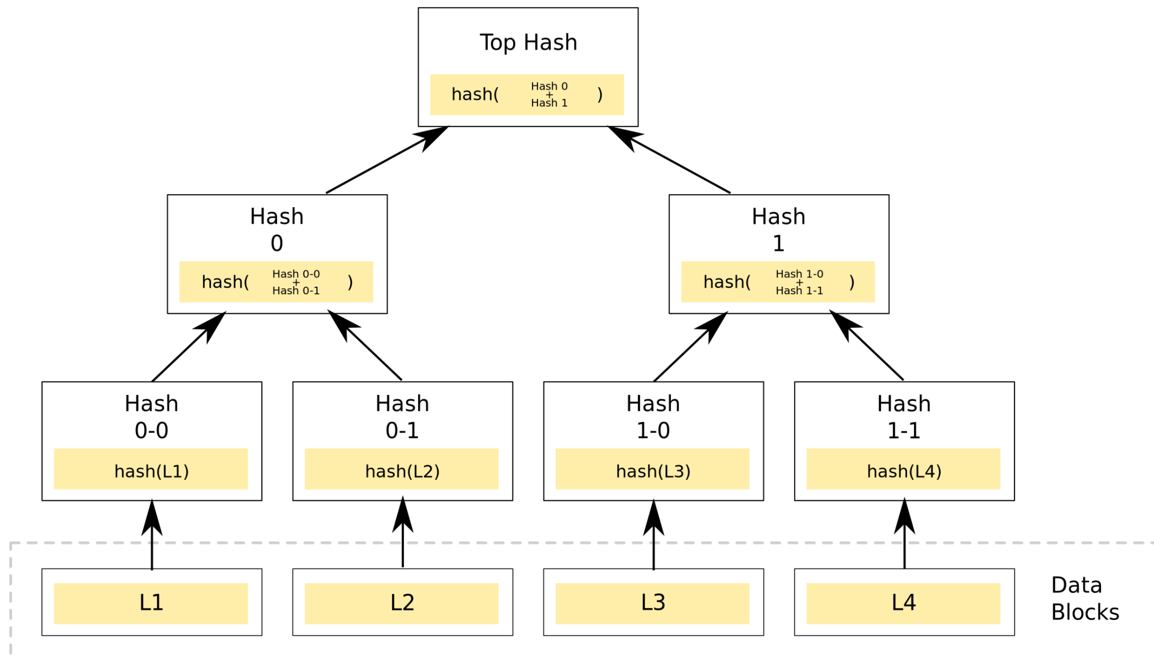

Figure 2.1: Merkle tree constructed from four data blocks. Illustration by David Göthberg. Source `https://upload.wikimedia.org/wikipedia/commons/9/95/Hash_Tree.svg`.

the data is correct. Using a Merkle tree of n nodes, only $log(n)$ items would be required to construct the proof.

2.1.3 Overlay Networks

Computer networks can exist as a layer on top of an already existing network [28]. Such a network is called an overlay network. The underlying network will provide the basic networking functionality, while the overlay network provides some utility not featured in the underlay network. In an overlay network, nodes can operate as if they are directly connected to other nodes (logical links), while they are indirectly connected through multiple links in the underlying network (physical links). Distributed systems are usually overlay networks, as they will in most cases run on top of existing networks such as the Internet.

2.1.4 Distributed Hash Table

Similarly to a regular hash table, a distributed hash table (DHT) provides a key-value lookup service. When trying to find some content in a distributed network, a DHT can be used to map a key to the peer that is storing the content. DHTs can scale to large numbers of nodes, as the responsibility of maintaining the table is distributed among the nodes. They form the basic infrastructure of many distributed file systems, peer-to-peer file sharing and content distribution systems.

2.1.4.1 Kademlia

One such distributed hash table is Kademlia, designed in 2002 by Petar Maymounkov and David Mazières [16]. It contains an address space where each peer can be identified by a unique node ID. To locate values, the algorithm uses this node ID.

To find some value, the associated key is needed. The network will be explored in multiple steps to gradually find nodes closer to the key. This will continue until the expected value is returned, or no closer node can be found. Kademlia allows for efficient lookup even through massive networks.

This works by treating each node as a leaf in a binary tree, with positions based on its (binary) ID prefix. Each node divides the binary tree into several subtrees not containing the node itself, each successively smaller closer to the node. Starting from the root, the first subtree is the half not containing the node (roughly 50 % of the nodes). Moving one step towards the node, the next subtree is the next half not containing the node (roughly 25 % of the nodes) and so on. The protocol ensures the node knows at least one other node in each of these subtrees.

To locate a specific node, the relevant subtree is located, and the known node in that subtree is queried. This node can help locate a node closer to the target, in a smaller subtree. Repeating this process until the target is found, assuming n nodes in the network, any node can be found after a maximum of $log(n)$ queries.

2.1.5 Erasure Coding

In every system, there is the possibility of data loss. The simplest technique to help prevent the loss of important data is replication. The issue with replication is the amount of additional storage required to store every replica. One alternative to simple replication is erasure coding [3].

With erasure coding, the original data is split into k data blocks, which are used to generate n encoded blocks, where $n > k$. Any combination of m encoded blocks can then be used to recreate the original data blocks, where m can be as low as k for certain codes. This means that any encoded block can be lost without loss of data, as long as at least m blocks remain. While introducing more complexity than simple replication, erasure coding can yield higher recovery rates using the same amount of additional storage.

2.2 Decentralized Distributed Storage Systems

In this section, we focus on decentralized storage systems. The general idea is presented in an overview, and two selected storage systems, Swarm and IPFS, are presented.

2.2.1 Overview

A *distributed storage system* (DSS) is a system where content is stored and made accessible through a network of nodes. We make the distinction between a DSS, such as Swarm, and a distributed file system (DFS), such as the Network File System (NFS). In a DFS, we use the file system interface of the host operating system to access and use the DFS. There are several transparency properties in place so that clients do not need to be aware that the file system is accessed through the network. In contrast, in a DSS we instead use a client application with a separate interface. Additionally, common DFSes such as NFS use a centralized client-server architecture.

2.2.2 Swarm

The Swarm project was created to build a permission-less storage and communication infrastructure [33]. Swarm is a system of peer-to-peer nodes forming a decentralized storage and communication service, using an incentive system enforced through Ethereum smart contracts. According to the Book of Swarm, Swarm is considered to have four separate layers, with the first two regarded as the core layers [33]:

- an overlay network with protocols powering a distributed immutable storage of chunks (fixed size data blocks)
- a component providing high-level data access and defining APIs for base-layer features
- a peer-to-peer network protocol to serve as underlay transport

- an application layer defining standards, and outlining best practices for more elaborate use-cases

2.2.2.1 Data Storage

Swarm's storage layer is built on the Kademlia overlay network. It uses an alternative interpretation of the DHT-model called *distributed immutable store for chunks (DISC)* [33, Chapter 2]. Instead of keeping a list of where files are found, DISC stores pieces of the file itself.

The basic storage unit in Swarm is called a *chunk*. It is a content-addressed, fixed-size chunk of data, identified by the 32 bytes Keccak256 hash of chunk content, referred to as the chunk's *BMT hash*. The content of the chunk may be part of a larger data blob, such as a file. The size is limited to four KiB, and smaller chunks are padded with zeros up to the maximum chunk size before hashes are calculated. In addition, chunks are prepended with a 64-bit little endian encoded span, which contains metadata to be able to differentiate between data chunks (leaf nodes) and intermediate chunks [33, Chapter 4]. Chunks may also be encrypted, but this is not be discussed here.

Swarm stores content using Merkle trees. The branching factor for the trees, i.e. the number of children nodes may have, is calculated as the chunk size divided by the reference size [33, Chapter 4]. For unencrypted content, the chunk reference is the chunk's BMT hash described above, while for encrypted content, it is the chunk's BMT hash catenated with its decryption key. Both the BMT hash and the decryption key are 32 bytes long. In other words, the branching factor is $4096/32 = 128$ for unencrypted content, or $4096/64 = 64$ for encrypted content.

Internal chunks (any non-leaf nodes) can refer to a number of child nodes equal to the branching factor. The content of internal chunks consists of the prefix span, followed by the catenation of 32-byte references to child nodes. As such, we can recover a data blob by recursively splitting internal nodes into a set of references, and downloading each child node referenced to by the references.

2.2.2.2 Accessing the Swarm Network

A Swarm client is required to communicate with the Swarm network, e.g. to upload or download data. There are two Swarm clients that we are aware of, the original Swarm client [20], and the new Bee client [26], which is under active development. Both clients are written in Go, and can be used as compiled binaries, or as a library. The two clients are incompatible with each other due to using different underlying network protocols. The original Swarm client, which Snarl supports as a backend, is no longer maintained as of January 2021, as of the v0.5.8 release [20, 29]. The Swarm team's nodes running on the public Swarm network have migrated to the new Bee client, however there may still be peers running on the old network as it is decentralized. It is however possible to self-host a Swarm network using the original client, which Snarl does for some of its experiments.

2.2.3 InterPlanetary File System

This section contains a brief overview of IPFS. IPFS is another peer-to-peer DSS, which Snarl is extended to support as a backend DSS in this thesis. The remainder of this section describes IPFS data storage and related matters briefly.

2.2.3.1 Overview

The IPFS is another peer-to-peer DSS with an initial release in 2015 [1]. It seeks to connect all computing devices with the same system of files. IPFS provides a content-addressed block storage model allowing for high throughput, with content-addressed hyperlinks. This

storage model forms a generalized Merkle DAG. The Merkle DAG is similar to a Merkle tree, but each node can have multiple parent nodes, and no loops can exist.

2.2.3.2 Data storage

All stored content in IPFS has a unique content identifier (CID) containing a cryptographic hash of the content [2]. The CID is used to locate content, as opposed to finding content by its location, like you would with URLs on the Web, with paths in regular file systems, etc. This means that the identifier always leads to the same content.

Content to be stored in IPFS is usually split into blocks. These blocks are used to build a Merkle DAG, where each node contains a hash of its contents. The Merkle DAG is constructed from the leaves, and parent nodes contain as their data their children's content hashes. Because all content in IPFS has a unique CID, content can be reconstructed by locating the desired node in the DAG, and then locating its children recursively.

The cryptographic hashes can be used to ensure the received content is correct, and every node in the DAG is immutable, as their CIDs depend on all of their children. Because of the globally unique CIDs, different content with identical parts (blocks), can reuse the same pieces of data. This means similar content can share much of the same data, reducing duplicate data in the network. Adding data to existing content simply means linking the new parts to the existing ones, removing the need for separate files.

An example is shown in Figure 2.2. In this example, a file is uploaded and split into three blocks. The file itself is identified by the CID *R1*, and its children by CIDs *L1*, *L2* and *L3*. In order to recreate the file, the CID *R1* is used to locate the correct node in the DAG, which contains references to its child nodes. These nodes contain the actual data, and can then be retrieved in order to recreate the file.

After adding some data to the file, the new version is uploaded. This new version is now identified by the CID *R2*, which references *R1* containing the old content, and a new block identified by CID *L4* containing the added data. In order to retrieve the data, we first find the node identified by *R2*, then we locate the children *R1* and *L4*, and then recursively locate *R1*'s children to obtain every data block.

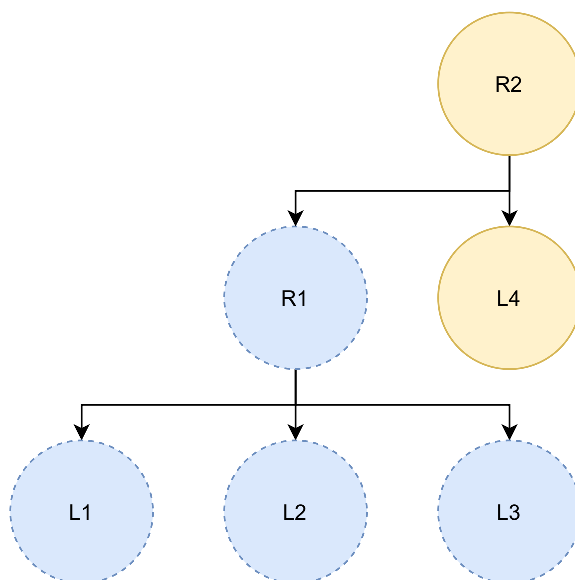

Figure 2.2: Example Merkle DAG. The blue (dashed) nodes represent the original uploaded file, and the yellow (solid) nodes represent an updated version uploaded later on.

2.2.3.3 Pinning

In IPFS, peers make nodes available through pinning [12]. When uploading content to IPFS, the related Merkle DAGs are by default pinned by the uploader. Pinning ensures that the content stays available from this peer by preventing it from being garbage collected. Other IPFS peers can retrieve pinned nodes by requesting them by CID.

To increase availability, other peers can pin the same content. Any peer can pin nodes by CID, and can choose whether to pin a single node (*direct pin*), or to recursively pin the node along with its subtree (*recursive pin*). Nodes pinned indirectly during recursive pinning are called *indirect pins*. An IPFS node, identified by CID, is available to IPFS peers from any peer that has pinned the node. Thus, popular content will be available from many peers, increasing availability and improving download times.

2.2.3.4 Data sharing

A DHT is used to locate content. One lookup in the DHT will identify the peers hosting the desired content (blocks). Another lookup will provide the location of those peers. This information can be used to request the desired blocks by sending a *wantlist*, and having the relevant peers send the requested blocks. Received blocks can be verified by hashing their content and comparing the hash with their CIDs.

2.2.3.5 Unix File System

IPFS has a special protocol-buffers-based format called Unix File System (UnixFS) which can be used when adding files to IPFS [9]. UnixFS adds metadata to each non-leaf node, representing files, directories or symlinks. Files too big to fit in a single block will be represented by a data object containing all their links and metadata, while single block files will contain their data directly.

When chunking a file into smaller parts, the leaf format can be either raw or UnixFS. UnixFS leaves add a wrapper used to determine whether objects are files or directories, while raw leaves (the default format) contain only raw data. The chunking strategy can be either *fixed size* or *rabin*. *Fixed size* chunking uses a specified size for every chunk, while *rabin* uses Rabin fingerprinting to determine where to split the data, resulting in variable chunk sizes. The constructed Merkle DAG can be either balanced with a specified max width, useful for random access, or a trickle DAG, useful for accessing data in sequence.

2.2.3.6 Go-IPFS and IPFS Daemon

Go-IPFS is a reference implementation of IPFS [10]. It provides both a command-line application, and exported APIs enabling its use as a Go library. Both approaches run an IPFS daemon process, and communicates with the daemon using an HTTP API. In the case of Snarl, the IPFS daemon is not embedded, but rather run expected to run as a separate process. During its runtime, Snarl uses IPFS APIs to communicate with the IPFS daemon to upload and download content. The end user is responsible for starting and configuring the IPFS daemon prior to using Snarl. Additionally, some tests and experiments may start or manipulate the IPFS daemon using the command-line interface through Go's `exec` package [23].

2.3 Alpha Entanglement Codes

In this section we look at alpha entanglement codes, which are the basis of the Snarl redundancy model. *"Alpha entanglement codes, AE(α, s, p), are a family of erasure codes built on data entanglement"* [4]. The entanglement algorithm creates a virtual storage layer above the input data, producing redundant blocks which connect fractions of the input data

and propagate redundancy properties. Storage overhead increases linearly with α, while the data recovery paths increase exponentially. Modifying s and p does not affect the storage overhead, but can further increase fault-tolerance at the cost of performance.

AE codes entangle *data blocks* with redundant *parity blocks*. Data blocks contain fractions of the data of the original file, such that the original file can be recovered when all data blocks are present. Whenever a new data block is added, it is entangled with α existing parity blocks by exclusive-or (XOR), which produces α new parity blocks. Parity blocks are used for recovery of other blocks. Since the entanglement uses a simple XOR operation, the cost to repair a single failure is a single XOR operation, regardless of the values of the α, s and p parameters.

Various properties and functionality of AE codes are described below. We look at the parameters in depth, what a helical lattice is and how it is constructed based on the parameters, and finally an overview of the entanglement and repair processes. In this thesis, entanglements where $\alpha = 3$ — triple-entanglements — will be the primary focus.

2.3.1 Parameters

In this section we look at the parameters to alpha entanglement codes. The parameters determine parameters of a helical lattice, meaning its dimensionality and properties of its strands. Let us first look at these two terms.

Strands are chains of alternating data blocks and parity blocks. There are two categories of strands: (1) horizontal strands, which in three-dimensional space can be seen as moving horizontally; and (2) helical strands, which in three-dimensional space can be seen as spiralling around the horizontal strand. For $\alpha \geq 3$, there are more than two or more types of helical strands. Strands and are described in more detail in Section 2.3.2.

We define a *helical lattice* to be a series of nodes (data blocks) connected through edges (parity blocks) in a predictable manner. Pairs of input and output edges to and from each node belong to the same strand.

Furthermore, we look at the parameters. There are three parameters to alpha entanglement codes:

- α: Number of redundant blocks created per data block. The entanglement process computes α parities per data block, one for each type of strand, such that each data block becomes part of α strands. The redundant parity blocks increase storage overhead linearly while increasing data recovery paths exponentially. Thus, α determines the local connectivity of blocks [5].

- s and p: s indicates the number of horizontal strands, while p indicates the number of helical strands. This means that s and p define the dimensions of a helical lattice. Seen in two-dimensional space, as in Figure 2.3, the lattice has s rows and p distinct columns. In order to define a valid lattice, p must be greater than or equal to s. These parameters are said to determine the global connectivity of data blocks, i.e. which strands each data block becomes part of, and to which other data blocks it connects. Both parameters can be increased without impacting storage overhead. In a minimal erasure pattern (a pattern which causes irrecoverable loss of parity blocks and data blocks) involving y blocks, x of which are data blocks, tuning s and p can increase the ratio $\frac{y}{x}$, i.e. fewer of the lost blocks will be data blocks [5, 4]. In other words, by tuning these parameters we can reduce the number of data blocks that are entirely irrecoverable following a failure, meaning we can recover more of the original data. As a trade-off, this may increase recovery time.

2.3.2 Strands

Strands are chains of blocks, alternating between data blocks and parity blocks. In the entanglement function, a data block is XORed with the latest parity block on the strand,

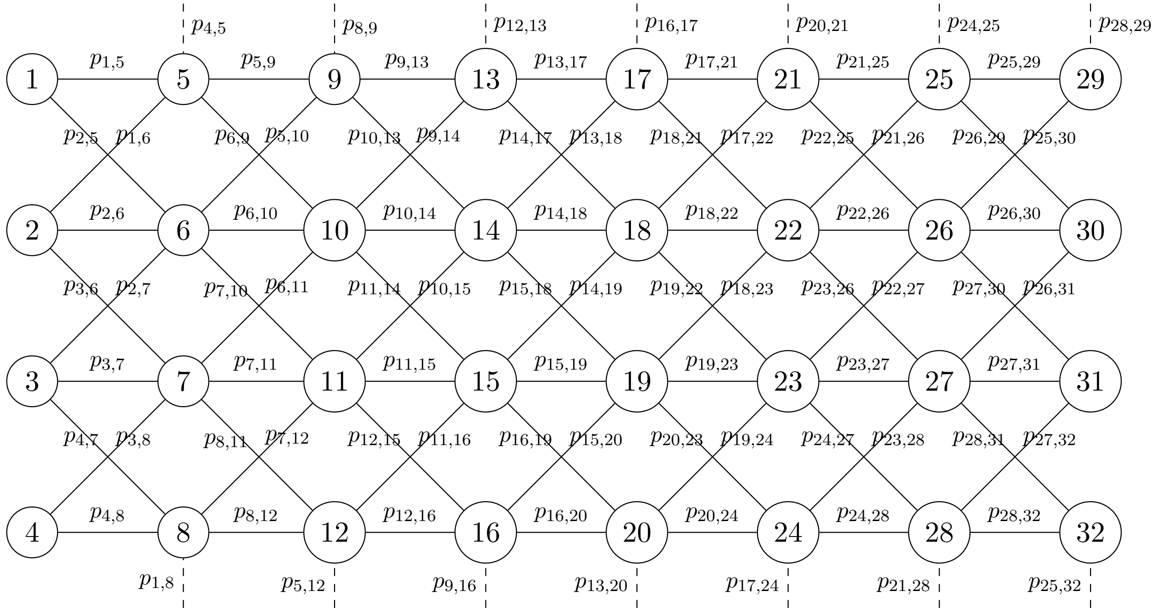

Figure 2.3: Helical lattice for the $AE(3, 4, 4)$ configuration. Vertices numbered i represent the corresponding data block d_i, while edges represent parity blocks. Dotted lines represent "wrapped" edges, the upper ones being wrapped from RH-strands and the lower ones being wrapped from LH-strands. While not displayed in the figure, the rightmost data blocks also generate α parity blocks each.

and the resulting parity block is inserted after the data block on the same strand. This process occurs for α strands for each inserted data block, such that each data block belongs to α strands, while each parity block belongs to a single strand. The damage of irreducible failure patterns on individual strands is limited, since data blocks can be reconstructed through any of the α strands they participate in.

For triple entanglements, there are three types of strands:

- Horizontal (H) strands. The number of horizontal strands is determined by s.
- Right-handed (RH) strands. In the helical lattice, RH strands connect data nodes on different H-strands from top to bottom, with a slope towards the right. The next data node connected to an RH-strand following a data node also connected to the bottom H-strand is a data node connected to the top H-strand. For example, in Figure 2.3, an RH-strand connects data nodes 8 (bottom H-strand) and 9 (top H-strand).
- Left-handed (LH) strands. In the helical lattice, LH-strands connect H strands going from bottom to top, in a similar manner to RH-strands. Following data nodes connected to the top H-strand, they connect data nodes from the top H-strand to the bottom H-strand.

More generally, the lattice contains s horizontal strands and $\alpha - 1$ helical strand classes. Each helical strand class (LH-strands and RH-strands for triple-entanglements) contains p helical strands, such that there are $s + (\alpha - 1) \cdot p$ strands in total. For example, for $p = 3$, there are three RH-strand classes RH_1, RH_2 and RH_3 which data blocks are connected to in a round-robin fashion with regards to the data block index. As in the referenced work [5, 4], we consider lattices with helical strands that connect horizontal strands with a diagonal of slope 1, and a balanced number of LH-strands and RH-strands. In this configuration, the helical strands meet on the same horizontal level, i.e. lattice row, after a period of p columns, at every $s \cdot p$ data blocks.

Table 2.1: Entanglement rules to determine which parity blocks are used for *input* to triple entanglement [4].

	d_i is tangled with $p_{h,i}$ and h index is		
node d_i location	**H strand**	**RH strand**	**LH strand**
top	$i-s$	$i-s\cdot p+(s^2-1)$	$i-(s-1)$
central	$i-s$	$i-(s+1)$	$i-(s-1)$
bottom	$i-s$	$i-(s+1)$	$i-s\cdot p+(s-1)^2$

Table 2.2: Entanglement rules to determine which parity blocks are *output* by triple entanglement [4].

	d_i entanglement creates $p_{i,j}$ and j index is		
node d_i location	**H strand**	**RH strand**	**LH strand**
top	$i+s$	$i+s+1$	$i+s\cdot p-(s-1)^2$
central	$i+s$	$i+s+1$	$i+s-1$
bottom	$i+s$	$i+s\cdot p-(s^2-1)$	$i+s-1$

Strands are the way redundancy propagation is achieved by AE codes. Since data blocks part of α strands, there are parallel paths to recover each data block. A data block can either be retrieved directly or regenerated using any of the α strands it belongs to. The data contained in the data block also becomes entangled into future parities on each strand, which enables recursive recovery of data.

Figure 2.3 provides an illustration of an $AE(3,4,4)$ lattice. There are $s=4$ H-strands, one on each row, and $p=4$ RH-strands and LH-strands. For example, the first RH-strand goes through the sequence $1 \to 6 \to 11 \to 16 \to 17 \to 22\ldots$, while another goes through the sequence $3 \to 8 \to 9 \to 14\ldots$. Similarly, LH-strands go upwards from the bottom, e.g. the sequence $4 \to 7 \to 10 \to 13 \to 20 \to 23\ldots$. When the helical strands reach the bottom or the top, they wrap to the opposite extreme for the next node they entangle with.

2.3.3 Entanglement

During entanglement the encoder produces parity blocks using the equation

$$p_{i,j} \leftarrow d_i \oplus p_{h,i}, \tag{2.1}$$

where $i>0$ is the data block's position in the lattice. For each data block d_i, α parity blocks are created following Equation 2.1. Table 2.1 is used to determine the indices h of α different input parity blocks. The node category is

$$\begin{aligned} \text{top} &\iff i \equiv 1 \mod s, \\ \text{central} &\iff i > 1 \mod s, \\ \text{bottom} &\iff i \equiv 0 \mod s. \end{aligned}$$

When the encoder produces parity blocks, pairs of blocks become related through *entanglement tuples*, of which there are two types:

- pp-tuples contain two consecutive parity blocks from the same strand. Each data block is associated with a pp-tuple for each strand class (i.e. H-strand and helical strands). The pp-tuples related to a central data block d_i is illustrated in Figure 2.4.
- dp-tuples contain a data block and an adjacent parity block, which are on the same strand. Each parity block is associated with two dp-tuples on the strand it belongs to.

More specifically, parity blocks are associated with dp-tuples consisting of the data blocks they are directly connected to on the same strand and their adjacent parity blocks, as illustrated in Figure 2.5.

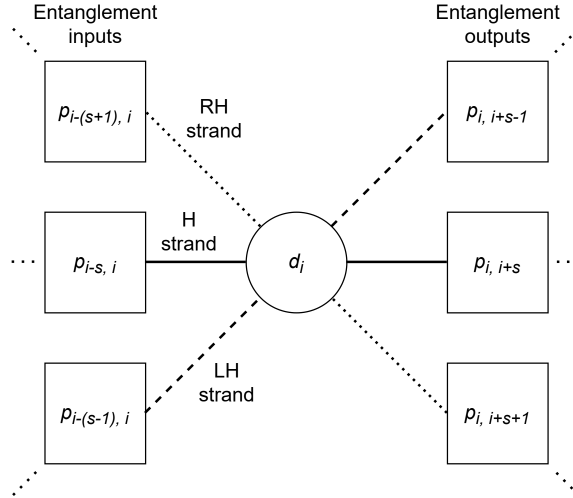

Figure 2.4: Illustration of the pp-tuples for a data block d_i, where $\alpha = 3$. d_i is considered a central node, and the entanglement rules from Table 2.1 and Table 2.2 are used. The full line illustrates the H-strand, the dotted line the RH-strand and the dashed line the LH-strand.

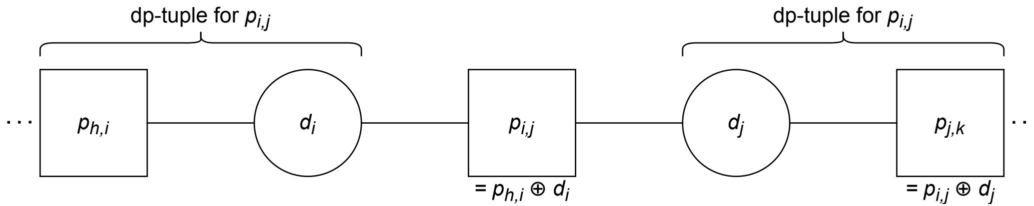

Figure 2.5: Illustration of the dp-tuples for a parity block $p_{i,j}$ on a strand. The XOR construction of parity blocks enables repair of $p_{i,j}$ through either of its related dp-tuples.

2.3.4 Repair

When we retrieve the data, no decoding is necessary unless some data blocks are damaged or missing. In the case of damaged or missing data blocks, the repair algorithm attempts to recreate the missing blocks to recover the structure of the helical lattice. We may also recover blocks using the repair algorithm for other reasons than damaged blocks, e.g. for load balancing or to avoid using slow connections in a network.

Both data blocks and parity blocks are repairable. Because of the XOR construction of parity blocks, both data blocks and parity blocks can be repaired with a single XOR operation taking the respective related pp-tuples or dp-tuples as inputs. A damaged data block d_i is repaired using the equation

$$d_i \leftarrow p_{h,i} \oplus p_{i,j}, \tag{2.2}$$

where the parity blocks $p_{h,i}$ and $p_{i,j}$ are from one of the α pp-tuples associated with d_i. Similarly, parity blocks are repaired using Equation 2.1, where the parameter values are from one of the two dp-tuples associated with $p_{i,j}$. In any case, the entanglement tuples required for repair are calculated following the rules in Table 2.1 and Table 2.2.

Estrada describes three groups of recovery scenarios, with varying numbers of steps to repair and varying rates of success [4]:

- *Small local damage:* Only affects a small region around a single data node which may or may not be available. No more than $\alpha - 1$ parity blocks may be missing, and at least one dp-tuple associated with each missing parity block must be available. When these conditions are met, repair completes in one round, using the method described at the start of this section. Multiple single failures, i.e. all elements in the small local damage, can be repaired in parallel during the single round of repair, without relying on recursive repairs.

- *Medium zonal damage:* Various consecutive elements on the same strand are damaged, where each damaged data node can be categorized as part of a small local damage. The data is entirely recovered by recursively performing local repairs. If no further faults occur during the repair of n missing parity blocks, the number of recursive calls is limited to at most $\lceil \frac{n}{2} \rceil + 1$.

- *Large global damage:* A wide area of close nodes are affected. Covers all erasure patterns not part of the previous categories and which require more than one round to repair. To repair such damage, an algorithm must combine local and zonal repairs. Depending on the presence of irreducible failure patterns, the success rate is in the range from 0 to 100 %.

2.4 Snarl

In this section we look at Snarl, a library and application for using alpha entanglement (AE) codes on top of distributed storage system (DSS) backends. Snarl is the foundation which this thesis builds and extends upon, so this section goes into some detail to provide context for the remainder of the thesis. The text in this section is based on the paper by Nygaard et al introducing Snarl [19], as well from the Snarl code itself, and discussions with Nygaard.

2.4.1 Overview

Snarl is a library and a command-line application which provides redundancy and file repair in distributed storage systems (DSSes) using AE codes. The motivation of Snarl is to enable file archival in unreliable storage environments, such as decentralized networks. Since it works on top of an underlying DSS, Snarl may be used by end users for additional protection of data stored in the DSS, without requiring changes to the DSS itself. This is beneficial since in a decentralized DSS the end user may not be able to determine the redundancy level of a file (usually achieved through replication), nor confirm the replication and level of availability of the file by querying the network.

The architecture of Snarl is illustrated in Figure 2.6. The user interacts with the Snarl command-line interface (CLI). This will trigger calls to various parts of the Snarl algorithm, depending on the given command. Snarl then interacts with the backend API to communicate with the backend, which may trigger network calls or calls to external applications related to the backend.

Snarl acts as an intermediary between the user and the DSS backend. Before uploading files to the DSS, Snarl entangles the file with AE codes in the $AE(3, 5, 5)$ configuration. When entangling files, it produces $\alpha = 3$ additional files containing AE parity blocks. When retrieving files, Snarl may use the parities to recover missing data blocks. These redundant files may be stored in the DSS in addition to the original file.

At the cost of some storage overhead, the entanglement is beneficial for file recovery when data blocks are unavailable. When retrieving the file from the DSS, some data blocks

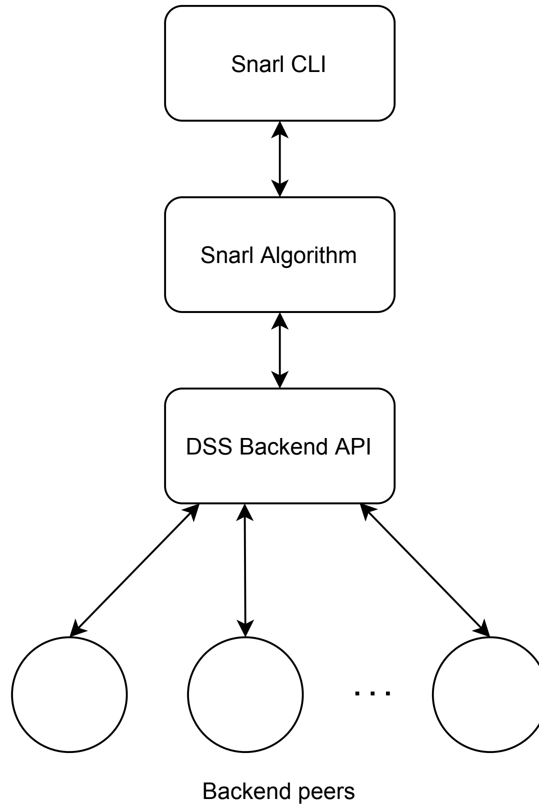

Figure 2.6: High-level overview of the architecture of Snarl.

may be missing. Snarl can then retrieve blocks from the parity files to recover the missing data blocks. If the block cannot be repaired in a single step, Snarl can recursively recover entangled blocks and perform repair, as described in Section 2.3.4. Recovered blocks can be used to recover the file locally, and can be re-added to the DSS.

The set of data blocks of a file is not arbitrary. Most decentralized DSSes use a Merkle tree or a similar data structure to represent the structure of a split file. The hashes of the Merkle tree nodes are used as the addresses of the blocks in the DSS during requests to the DSS. As such, Snarl must split the file in the same manner that the underlying DSS would do in order to be able to retrieve the blocks later on.

The related implementation details vary depending on the particular backend. In general, Snarl uses the same procedure as the backend for generating Merkle tree structures out of files during upload. The Merkle tree is then flattened into a list of data blocks, which are entangled in the same order, such that an item with index i in the flattened Merkle tree corresponds to the data block with index i in the helical lattice. Figure 2.8 and Figure 2.9 illustrate the Merkle tree flattening, which is further described in Section 2.4.2.2.

As a consequence of Snarl's approach to data block generation, it also protects the Merkle tree related to the file, not just the data stored in the leaf nodes [19]. By protecting all nodes in the Merkle tree, Snarl reduces the risk of cascading failures, which occur when available child nodes cannot be retrieved because the parent node containing their references is unavailable.

2.4.2 Entanglement Algorithm

In this section we look at Snarl's entanglement algorithm in more detail. Before starting the entanglement process, a file must be split and converted into a set of data blocks. Then, entanglement processes each data blocks, generating parity blocks in accordance with the procedure described in Section 2.3.3. In addition to the original file, the procedure generates

$\alpha = 3$ files consisting of parity blocks, which may be uploaded to the backend DSS. The remainder of this section summarizes the entanglement process as a series of steps.

2.4.2.1 Merkle Tree Generation

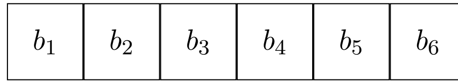

Figure 2.7: Illustration of the file being split into fixed-size blocks during entanglement.

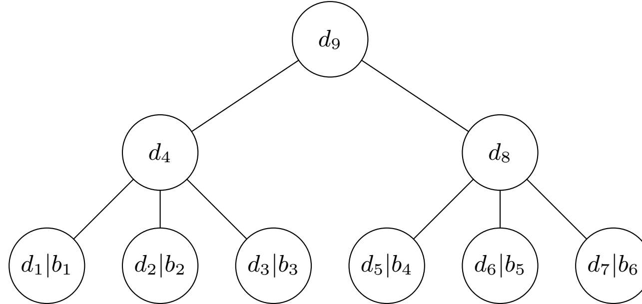

Figure 2.8: Illustration of the Merkle tree representation of the split file with nodes numbered in canonical order. Leaf nodes contain the blocks of data from the original file, while parent nodes contain links to child nodes and other metadata.

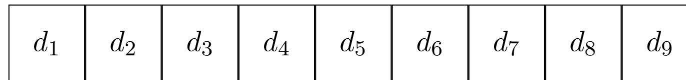

Figure 2.9: Illustration of the flattened Merkle tree with indices matching nodes in the Merkle tree from Figure 2.8.

In this step Snarl reads the file and generates a Merkle tree based on the file's contents in a backend-specific manner. All nodes of the Merkle tree are stored in an internal data structure which mirrors the structure of the Merkle tree. Using the API of the backend DSS, the Merkle tree is generated from the contents of the input file. Generally, the first step splits the file into fixed-size blocks of data per a provided or constant block size, as illustrated in Figure 2.7. The blocks are illustrated as b_i, each of the same block size, except the final block which may be incomplete depending on the size of the original file.

Furthermore, the Merkle tree is built from the blocks of data. The blocks from the original file become the leaf nodes of the Merkle tree, and the hash of each block becomes the CID of the related node. Additional nodes are added at higher levels of the tree whose contents are lists of CIDs of child nodes, and possibly some related metadata such as the size of the subtree. The *branching factor* determines the maximal number of child nodes a parent node may refer to. The process repeats until a single node exists at the highest level of the tree, which becomes the root node, and whose CID represents the file used to construct the Merkle tree.

Figure 2.8 illustrates the Merkle tree produced from the blocks of data illustrated in Figure 2.7. As explained above, the Merkle tree takes the blocks of data from the original file as leaf nodes, and produces parent nodes until there is a single root node. In this example the branching factor is three. Nodes d_4, d_8 and d_9 contain lists of CIDs to their respective child nodes, where each CID is called a *link*. The indices of the tree nodes are in the canonical order, which is explained in Section 2.4.2.2. In the illustration, the leaf nodes have two identifiers (aliases) d_i and b_j, denoted as $d_i|b_j$. The alias d_i identifies the data block with canonical index i, and the alias b_j identifies the block of data from the blocks in Figure 2.7.

The number of non-leaf nodes is inversely proportional to the branching factor. A low branching factor necessitates more parent nodes for groups of leaf nodes, which increases the overall size of the Merkle tree.

2.4.2.2 Flattening the Merkle Tree

In this step the Merkle tree is flattened in a *canonical order*. The canonical ordering of nodes in a Merkle tree is illustrated in Figure 2.8. Starting from the root node, the tree is traversed recursively in a children-first order, leftmost children being traversed first. While traversing the tree, there is a counter for the index, starting at one. The index of a node is set to the value of the counter when reaching a leaf node or a node whose children have already been visited. Whenever an index is set, the counter is incremented. For example, we can see that the leftmost leaf nodes d_1–d_3 are numbered first, then their parent node d_4, and so on, until the root node finally gets the index d_9 after processing all of its child nodes. The tree is flattened by appending to a list the content of each node in the tree in the canonical order. Resultingly we have a list with the content of each node from the Merkle tree in a predictable order, as illustrated in Figure 2.9.

Swapping Internal Nodes Additionally, after flattening the Merkle tree, Snarl swaps certain internal nodes with leaf nodes. It does this to prevent certain critical failure patterns, and may increase the recovery rate of data blocks. Pseudocode for the algorithm is given in Algorithm 1. The algorithm iterates over each intermediate node (non-root parent node) in ascending order. It then iterates other nodes of index, in increments of a window size $w = s \cdot p$. When it encounters a leaf node with index i that is outside of the window, it swaps the intermediate node with the leaf node. The indices of swapped nodes are registered such that a node is not swapped more than once.

Algorithm 1 Pseudocode for Snarl's internal node shift algorithm.

```
flattree ← flattened Merkle tree
swappedindices ← {}                                    ▷ set of swapped indices
inodes ← intermediate nodes
s, p ← AE code parameters
w ← s · p                                              ▷ window size
for all inode ← inodes do    ▷ iterate over each intermediate node in ascending order
    lowestchild ← inode_firstchild
    highestchild ← inode_lastchild
    for i ← w; i < |data blocks|; i ← i + w + s do
        inwindow ← i + 1 > lowestchild − w ∧ j + 1 < highestchild + w
        if isleaf(i + 1) ∧ ¬inwindow then
            if inode_index ∈ swappedindices ∨ i + 1 ∈ swappedindices then
                continue                               ▷ either node is already swapped
            flattree_{inode_index}, flattree_{i+1} ← flattree_{i+1}, flattree_{inode_index}
            swappedindices ← swappedindices ∪ {inode_index, i + 1}
            break                                      ▷ go to next internal node
```

2.4.2.3 Entanglement

In this step the blocks of the flattened Merkle tree from Section 2.4.2.2 are entangled. The data blocks in the flattened Merkle tree are entangled as described in Section 2.3.3, where the list index i (with the first index being 1) matches the data block position i in the AE helical lattice. For a flattened Merkle tree of length n, this results in α lists of n parity blocks, one for each type of strand (i.e. left-handed, right-handed or horizontal strands for

p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	p_9

Figure 2.10: Illustration of parity blocks produced from the flattened Merkle tree during entanglement. There are α lists of parity blocks, one for each strand class (H-strand, LH-strand, RH-strand).

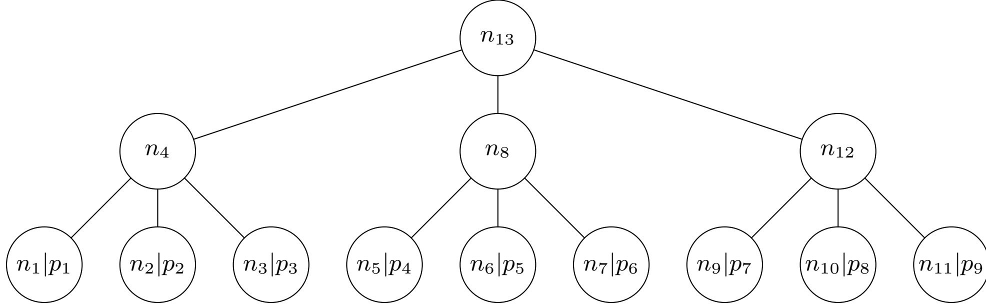

Figure 2.11: Illustration of the Merkle tree representation of a parity file with nodes numbered in canonical order. Only leaf nodes contain the parity blocks illustrated in Figure 2.10, where the *ith* leaf node (from left to right) corresponds to the *ith* parity block in the list of parity blocks.

triple-entanglements). One such list is illustrated in Figure 2.10. In the figure, the parity block p_i corresponds to the parity block $p_{i,j}$ resulting from an entanglement with data block d_i per the entanglement rules in Table 2.2. For each data block there is one parity block of each type of strand.

There are parity blocks for each data block in the Merkle tree produced from the original file. This includes non-leaf nodes which do not contain blocks of data from the original file as their content. As such, the parity files are slightly larger than the original file, which also causes the resulting Merkle trees for the parity files to be larger. Entangling every data block from the Merkle tree leads to an important property: The entire Merkle tree of the original file is protected, not just the data from the original file. Using the parity blocks it is possible to repair every data block, including the parent nodes which contain links to other data blocks.

Parity blocks are separated into α distinct subsets. Each subset consists of all parity blocks from one type of strand, e.g. all parity blocks on the H-strand. When the size of the content of a parity block is less than the block size, it is padded with zeros. This is necessary to predict the layout of the Merkle trees generated from parity files when uploading the files to the backend DSS. Finally, the padded parity blocks from each subset are written to a *parity file*, resulting in α parity files in total.

2.4.2.4 Uploading the Files

In the final step of entanglement, Snarl may also upload the files to the backend DSS. Using a backend-dependent implementation, Snarl uploads the original file as well as the parity files to the DSS. The original file will be stored in the backend equivalently to the Merkle tree generated during entanglement as described in Section 2.4.2.1. The parity files will have a different layout, since they contain more blocks of data.

An illustration of the Merkle tree used to stored each parity file in the backend DSS is provided in Figure 2.11, based on the parity blocks illustrated in Figure 2.10. For each type of strand, each parity block generated during entanglement becomes a leaf node in the Merkle tree representing the corresponding parity file. The leaf nodes are illustrated with two identifiers n_i and p_j, where n_i identifies the parity node with the canonical index i, and p_j corresponds to the matching parity block in Figure 2.10. Each parity block has been

padded such that its size is exactly the block size. Since the splitting process splits files into blocks of the predefined block size, after splitting, each block will contain only the content of a single parity block. When the Merkle tree is generated, each parity block will be stored in its entirety in its own leaf node. A number of parent nodes are generated based on the branching factor in the same manner as described in Section 2.4.2.1. The result is a Merkle tree for each parity file that has more nodes than the Merkle tree for the original file.

Only the leaf nodes of the parity file Merkle trees contain parity blocks. As such, only the leaf nodes can be used to repair data blocks, since only they contain portions of the parity file data. The parent nodes in the Merkle trees of the parity files are only used to acquire links to the relevant child nodes when traversing the trees.

2.4.3 File Recovery and Repair Algorithm

In this section we look at how Snarl recovers files from the backend DSS, and when and how repair occurs. First the general file recovery process is described, without going into detail on how repair works. Subsequently, the repair algorithm is described briefly.

2.4.3.1 File Recovery Process

File recovery is when Snarl attempts to recover the contents of a file that was previously entangled and uploaded to the backend DSS. This involves retrieving the data blocks that constitute the Merkle tree representing the file from the backend DSS, and rebuilding the file with the blocks of data contained in the leaf nodes.

Setup Some setup steps are necessary before the recovery process begins. Internally, Snarl uses a structure called `Lattice` to represent the AE helical lattice, which is used during recovery. To set up this data structure, knowledge of the layout of the Merkle tree of data blocks is necessary. Additionally, the size of each data block is needed, to be able to remove padding during any repairs that may occur.

For these purposes, Snarl must be provided a list of so called chunk metadata. The metadata is generated by creating a mirrored structure of the Merkle tree of data blocks. Snarl creates a temporary file with the same size as the original file, with arbitrary contents. Then, using the same procedure as described in Section 2.4.2.1, a Merkle tree is generated with the same branching factor and block size, and as such the same exact layout and size of nodes[1]. The nodes in the generated tree should only differ in the content of the leaf nodes, and the hashes or CIDs. The generated tree is then traversed, producing for each node an object which contains the following:

- *Size*: The cumulative size of the subtree of the node, including the size of the node itself.
- *Length*: The size of the node itself, which may not exceed the block size mentioned in Section 2.4.2.1. This value may be used when removing padding from parity blocks during repair.
- *Parent index*: The AE helical lattice index of the node that is the parent of this node.
- *Child node indices*: A list of AE helical lattice indices of the children of this node.

Furthermore, the `Lattice` must know the CIDs of the root blocks of the data block Merkle tree and of the Merkle trees of each parity file. Otherwise Snarl cannot find references to data blocks and parity blocks during recovery.

[1] In order for this to work, the algorithm that splits the file and produces a Merkle tree must only be based on the size of blocks of data, such that two files of the same size always produce the same Merkle tree layout.

Based on the provided metadata, Snarl generates empty blocks for each data block and parity block in advance for the `Lattice` data structure. Whenever a data block or parity block is downloaded or repaired, its content is stored in the corresponding block of the lattice. Additionally these blocks are used to determine related pp-tuples for data blocks and dp-tuples for parity blocks, which are necessary for repairs.

Recovery The recovery process is the following:

- The root data block is retrieved. If the data block is not available in the backend DSS, it is repaired using a related pp-tuple.
- Starting from the root node, the data block Merkle tree is traversed. Parent nodes are retrieved and contain links to their child nodes. Each child node is retrieved concurrently. Whenever a node (data block) is missing, an attempt is made to repair it using related pp-tuples. Parity blocks may fail to download as well, and can be repaired using dp-tuples. The repair process is described in more detail in Section 2.4.3.2. The recovery process fails if any node is irretrievable both through download and repair.
- When all nodes are retrieved successfully, the reproduced data block Merkle tree is traversed again, and the contents of each leaf node is catenated to reproduce the content of the original file. The recovered data is written to an output file.

2.4.3.2 Repair Algorithm

In this section we look at a high-level overview of the Snarl repair algorithm. Snarl's repair algorithm is based on the AE repair concepts described in Section 2.3.4. The algorithm is used to repair missing data blocks using related pp-tuples, and may also need to repair parity blocks if they are also irretrievable, by using dp-tuples related to parity blocks. Using the properties provided by alpha entanglement, missing data blocks and parity blocks can be repaired recursively in several rounds, leading to high availability. In-depth descriptions of different parts of the algorithm are provided by Nygaard et al [19].

The repair algorithm is triggered whenever a data block or parity block b is not directly retrievable from the backend DSS. In simple terms it consists of the following steps:

1. Get pp-tuples (for data block) or dp-tuples (for parity block) related to b.
2. Repeat the following steps until the algorithm terminates:
   - For each tuple, recover data blocks or parity blocks the tuple consists of.
     - Data blocks are requested by CID.
     - Parity blocks are requested by sending requests to a middleware for communicating with the backend. The requests contain the CID of the root node of a parity file Merkle tree and the AE helical lattice index i of parity block $p_{i,j}$. The root CID is determined by the type of strand the parity block is requested from. The requested parity block $p_{i,j}$ is that which was generated from data block d_i, per rules in Table 2.2. The middleware retrieves and stores necessary blocks of the parity file Merkle tree to recover requested parity blocks.

     If required blocks are not locally available and not retrievable from the backend DSS, go to step 1 for the required block.
   - When both blocks of the tuple are available, recover b using Equation 2.2 for data blocks or Equation 2.1 for parity blocks. In this case the repair algorithm completes successfully.

- Snarl tries to repair tuples related to b until it detects a critical failure pattern. Such a failure pattern occurs when there is no way to reproduce any tuple necessary to repair b. In this case, the algorithm fails.

Chapter 3

Design

In this chapter we look at the design of contributions to Snarl made in this thesis. While Snarl was a functional application originally, it was in many places heavily tied to the Swarm backend. Large parts of this thesis are concerned with introducing abstractions and alternative algorithms to enable alternative backends for Snarl, and to make it simple to add more backends in the future.

Firstly, we formalize requirements for a backend DSS to be a viable Snarl backend. Secondly, we describe the abstraction layer, designed to enable alternative Snarl backend implementations, and to simplify usage of Snarl. Finally, we look at the IPFS peer management layer, used to conduct experiments to evaluate the performance of Snarl with the IPFS backend.

3.1 Backend Requirements

This section describes requirements for a DSS to be viable as a Snarl backend. The design of Snarl imposes a few requirements on the backend DSS to operate correctly. This is primarily due to the need to represent the data blocks and parity blocks being recovered in Snarl's internal data structures, including the number of blocks and size of each block. First, we define block DAGs, and describe required properties for compatibility with Snarl. Furthermore, we describe requirements of nodes in the block DAGs. Finally, we describe information the end user must provide to Snarl for successful file recovery.

3.1.1 Block DAG Requirements

In this section, block DAGs are defined, and various requirements of block DAGs for Snarl compatibility are described. The DSS is assumed to store files as blocks of data which are connected in a DAG in a way that makes it possible to recover the original file. We define this generic storage layout as a *block DAG*. We may refer to any entry in the DAG as a *node*, while *blocks* generally refer to data blocks or parity blocks.

The rest of the thesis discusses Merkle trees or Merkle DAGs as the storage layout of a backend DSS, but they are not necessary for Snarl to function, though they are perhaps the most convenient ways to achieve this functionality. Snarl only needs to know about the hierarchies of data blocks and parity blocks, and be able to recover necessary blocks from these hierarchies. To achieve this the DSS should fulfill the criteria described in the remainder of this section.

3.1.1.1 Predictable Block DAG Layout

The DSS produces block DAGs in a predictable manner. For two files of the same size, the layout of the block DAGs produced from each file must be identical. The layout being identical means that the same portions of each file (when looking at two byte offsets $i, j | i \neq$

j) must end up in the same block of data in the canonical order, and be of the same size $\max i, j - \min i, j$. Block DAGs are produced in a backend-specific manner which produces the same block DAG layout as when uploading or storing the same file in the backend.

During recovery, this enables Snarl to discover the layouts of the block DAGs for both data blocks and parity blocks related to the requested file, the mapping of parent indices to child indices in the canonical order, as well as metadata such as the size of each data block. These properties can be generated by only taking the size of the requested file, the block size, and the branching factor as inputs. In some cases, the block size and branching factor are constant in the backend DSS, and need not be provided by the end user.

This property implies that the layout of the block DAG of the file stored in the backend DSS is equivalent to the layout of a block DAG generated by Snarl locally. To discover the block layouts and related index mappings and metadata, Snarl generates a block DAG locally. When the size of the requested file is n bytes, Snarl generates a block DAG locally from a buffer of n bytes of arbitrary data. The local block DAG is traversed to generate index mappings and metadata. It is also flattened, and a parity file is generated from it, from which another local block DAG is generated, which is traversed to generate index mappings for the parity files. These values are used by Snarl during recovery and block repair.

3.1.1.2 Leaf Node Properties

There are two types of nodes in a block DAG. Firstly, *internal nodes* are nodes which have references to child nodes or outgoing edges. In contrast, *leaf nodes* have no child nodes or outgoing edges. We consider leaf nodes as blocks which contain a portion of the content of the entangled file.

Recall the entanglement process from Section 2.4.2. When generating a block DAG from an input file, the file is split into blocks of data, which become the leaf nodes in the block DAG. Thus, to access portions of the file, we must access the leaf nodes. Due to Snarl's entanglement process each node in the block DAG — data blocks — of the entangled file are protected by parity blocks. For the parity files, only leaf nodes in the resulting block DAG are parity blocks. This means that only leaf nodes are used by the repair algorithm to repair other parity blocks from the same block DAG, or to repair data blocks.

The placement of parity blocks in the block DAGs of the parity files calls for a way to address leaf nodes in the block DAG. Specifically, when requiring a parity block $p_{i,j}$, Snarl requires access to the i-th leaf node in the block DAG of a parity file. The requirement is not for random access, but rather the possibility to recover the i-th leaf node by recursively traversing a branch of the block DAG, from the root node to the requested leaf node, by repeatedly requesting links to the next child node from the current position in the branch. Parity block retrieval is described in more detail in Section 3.2.3.2.

3.1.1.3 Number of Data Blocks

The entanglement process described in Section 2.4.2 requires a certain number of data blocks to function correctly. Snarl uses a variation of AE codes, with a thoroidal lattice [19, Section 3.4]. To better protect data at the end of strands, the lattice is closed, i.e. the first and last data block of each strand is connected through a parity block. In order to prevent closing a data block with itself, the lattice must contain at least two data blocks for each strand. This means at least $2 \cdot max(s, p)$ data blocks are required.

3.1.2 Node Requirements

This section describes requirements of the nodes of block DAGs produced by the backend DSS. Firstly, the requirements for node identifiers and retrievability are described. There-

after follows a description of requirements related to the size of data blocks in the block DAGs.

3.1.2.1 Node Retrievability

Nodes in the block DAG must be retrievable from the backend DSS by providing identifiers. Initially, Snarl is only aware of the identifiers of root nodes of the data block DAG and parity block DAGs. Requests for these nodes should not attempt to retrieve the entire files represented by the block DAGs related to the root nodes, or at the very least not cause the algorithm to fail when initially failing to retrieve more than the content of a root node itself.

Though dependent on the backend implementation, Snarl algorithms and interfaces generally assume that the block DAG of data blocks is recovered recursively. Data blocks are recovered or repaired starting from the root data block, then recursively recovering or repairing child data blocks, until all data blocks are available, at which point the requested file is reproduced. Parity blocks are recovered from the parity block DAGs in the same manner, though only leaf nodes from these block DAGs are of interest, as mentioned in Section 3.1.1.2. Generally not all leaf nodes are necessary to perform repairs. As such, recovered internal nodes should contain references (identifiers) to their child nodes, to enable recursive recovery.

3.1.2.2 Block Size

In addition to block DAG layout requirements described in Section 3.1.1, Snarl imposes certain requirements on the size of blocks. The Snarl configuration specifies a block size in number of bytes; this size cannot be exceeded by any data block in the data block DAG. During the entanglement process described in Section 2.4.2.3, every block in the data block DAG is entangled, producing parity blocks. Furthermore, the parity blocks are padded such that their size matches the block size, and the blocks of each type of strand (H-strands, etc.) are catenated in the canonical order into α parity files. When storing the parity files in the backend DSS, they are each split and turned into block DAGs. Since the size of each parity block matches the block size, the splitting ensures the content of each block is stored in its entirety in a separate leaf node in the produced block DAG.

If the size of any data block exceeds the block size, the parity blocks produced by entangling the data block also exceed the block size. This ultimately leads to the content of the parity block being split into several nodes in the block DAG of that parity file, which makes the parity block irrecoverable by providing a single identifier. It also increase the total number of nodes containing parity block content, preventing the recursive recovery of parity blocks mentioned in Section 3.1.1.2.

While Snarl could pad parity blocks such that the size of each matches the longest block, this would still not work due to the requirements of predictable block DAG layouts described in Section 3.1.1.1. If arbitrary blocks in the block DAG exceed the block size, Snarl cannot produce another block DAG with identical layout by only using the parameters file size, block size and branching factor.

3.1.3 Requirements from the End User

This section describes requirements from the end user in order to be able to use Snarl to recover a file which was entangled and uploaded to a backend DSS. During recovery, the user must provide the configuration of Snarl used during the entanglement and storage of a file, as well as identifiers for the related block DAGs. The remainder of this section describes information the user must provide in order to successfully recover a requested file.

3.1.3.1 Snarl Configuration

The user must provide the configuration of Snarl originally used when entangling and storing the requested file. The backend configuration must be mirrored, which includes the type of backend DSS used, block size, and branching factor. There may be additional configuration required by certain backends, which must be provided as well. Finally, the size of the requested file in bytes must be provided such that Snarl can predict the layout of the related block DAGs, as described in Section 3.1.1.1.

3.1.3.2 Identifiers

The user must provide identifiers of root nodes in the block DAGs related to the requested file. To recover the file without the possibility of repair, the identifier of the root data block must be provided. To use Snarl's repair algorithm, identifiers of root nodes in the block DAGs related to parity files must be provided as well. Parity blocks from each type of strand are only available by traversing the related block DAG from the root node, i.e. the identifier of the root node of the block DAG of each type of strand is necessary to make use of parity blocks from that type of strand in the repair algorithm. Technically, it would be possible to only provide identifiers for a subset of the block DAGs of parity files, only making the same subset of types of strands available during repairs, however this functionality is currently not provided by Snarl.

3.2 Abstraction Layer

In this section we look at the design of the abstraction layer for Snarl. First we look at the updated architecture of Snarl with regards to the abstraction layer. Then the design goals behind the abstraction layer are described. Thereafter follows a brief overview of the different domains of the abstraction layer, namely the node abstraction and related operations, entanglement, uploading, recovery and repair. Each abstraction was created to enable using several backends for Snarl, and to make it simple to implement new backends. In several places, the original code depended on Swarm APIs or details of the Snarl implementation for Swarm, neither of which translate directly to non-Swarm backends. As such, abstractions were needed, but the abstractions are designed such that the majority of the original code can be reused.

3.2.1 Updated Architecture

The updated architecture of Snarl is illustrated in Figure 3.1. It differs from the architecture previously illustrated in Figure 2.6 in that it includes the abstraction layer. The end user interacts with Snarl either through the command-line interface (CLI) or using Snarl as a library in another application. The Snarl algorithm uses the abstraction layer to support multiple backends.

Backend abstractions are implemented primarily by Go interface types. The underlying backend implementation implements the abstraction layer API, e.g. using the DSS backend's API to generate DAGs, store content and retrieve content. This may trigger network calls or calls to external applications related to the backend in order to produce the intended effect.

3.2.2 Design Goals

There are several design decisions that affect the design of the abstraction layer. The paragraphs below describe the design goals for our contributions to Snarl.

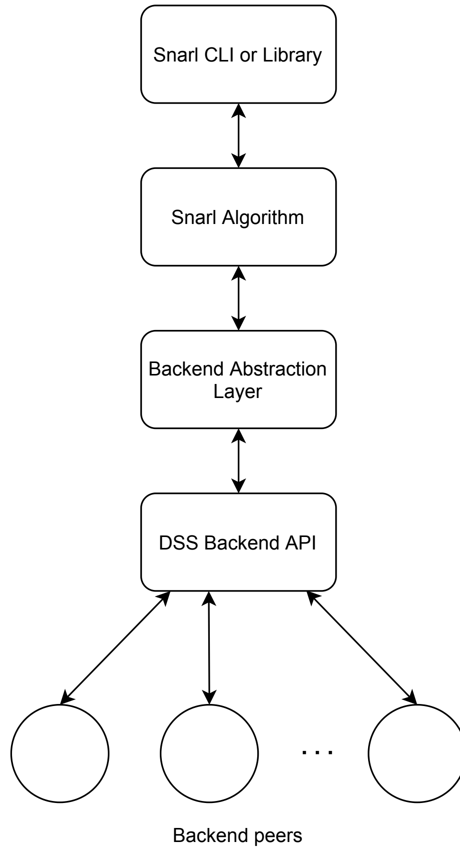

Figure 3.1: High-level overview of the architecture of Snarl, including the introduced abstraction layer.

Abstraction and simplicity The abstraction layer should provide a minimal and simple API with high-level abstractions. These properties should make the system easier to use and reason about, and simplify extensions. The abstractions should also be such that end users and developers do not require in-depth knowledge of alpha entanglement functionality to use Snarl. Two possible ways to implement abstractions in Go are interfaces and channels.

An interface type is a type which describes desired functionality of other implementing Go types[1]. It does not have to describe all possible operations implementing types can perform, just those we are interested in. An example is the `io.Writer` interface type whose single method `Write` accepts a slice of bytes to be written, and returns the number of bytes written and an error, if any [24]. This interface is widely used in the Go standard library and other Go code, e.g. for writing to buffers, files, network streams, and encrypted streams. Similarly, interfaces can represent operations in a backend-independent manner, while the interface implementations may vary in both layout and complexity. Additionally, interface types are composable, i.e. an interface can be the union of other interfaces, which makes it possible to make minimal interfaces and extend for cases that require more functionality.

Go channels on the other hand, operate as thread-safe FIFO queues, enabling the programmer to send any data across thread boundaries [30]. Channels are useful for abstractions when the source of a piece of data, or how it was produced, is irrelevant to the consumer. The consumer can simply receive the data from the channel and use it for its intended purpose.

[1] Go interface types [31] work similarly to interfaces in most object-oriented programming languages. A major difference in Go is that interfaces are implicitly implemented by any type which implements each method belonging to the interface.

Applicability and extensibility The abstractions should apply to many potential backend implementations and should be simple to extend. The applicability of the abstraction layer should be high: the abstractions should make it possible to add implementations of several new backends without friction. Particularly, the addition of a new backend should not require further changes to Snarl internals or the abstraction layer itself, nor provide limited functionality for the new backend.

Code reuse and refinement Whenever feasible, the abstraction layer should enable reuse, refinement or simplification of the existing codebase. The Snarl codebase already had thousands of lines of code before the start of this thesis. It is desirable to reuse as much code as possible, and add extensions or make simplifications where possible. In cases where the logic changes significantly, the existing code is adapted or used for inspiration.

3.2.3 The Node Abstraction

The node abstraction represents a node in the Merkle DAG of the backend[2]. The node contains the raw data stored by that node in the DSS, as well as other data such as links to child nodes or fields required by Snarl such as the lattice index. In effect, the node is a way to convert raw data retrieved from the backend DSS into Go types that can be used by Snarl. Different backend DSSes require different implementations, by implementing abstractions for common operations on top of the APIs and types of the backend. The Merkle DAG itself is represented as a sequence of nodes, in the canonical order as described in Section 2.4.2.

Previously, blocks were represented with the `TreeChunk` data structure, which is tied to the Swarm backend. The node abstraction provides the minimal interface to present nodes from any backend. `TreeChunk` remains as the node implementation for the Swarm backend.

For the purposes of Snarl, a node must provide the following operations:

- The node must be able to provide its unique identifier. This is an inherent property of content-addressing DSSes. However, location-addressed DSSes may also be used by storing the node's location address when it is retrieved, assuming the content at the location address is static.
- The node must be able to provide the raw data it contains. In most cases only leaf nodes contain data, while other nodes simply contain links to child nodes, as well as being used for Merkle proofs in the case of Merkle DAGs.
- The node must be able to indicate whether it is a leaf node or an internal node, based on the definitions given in Section 3.1.1.2.
- The node must implement the *walk* operation. It returns the subtree of the node, including itself, flattened in the canonical order. This operation is described in Section 3.2.3.1.
- The node must implement the *next path* operation. It returns a link to and index of the next node in the path to a requested leaf node. This operation is described in Section 3.2.3.2.

3.2.3.1 The Walk Operation

The *walk* operation, when called on a node, returns a sequence of nodes. The sequence consists of the subtree of the node *walk* was called on, including the node itself, flattened

[2]Note that for simplicity this section uses the term Merkle DAG interchangeably with the term block DAG defined in Section 3.1.

in the canonical order. The recursive definition of *walk* is

$$walk(x) = x, \qquad if\ |C_x| = 0, \tag{3.1}$$
$$walk(x) = walk(c_1)||walk(c_2)||\dots||walk(c_n)||x, \qquad if\ C_x = \{c_1, c_2, \dots, c_n\}, \tag{3.2}$$

where x is a node and C_x is the set of child nodes of x. In this way, starting at the root node, the operation can be recursively called on child nodes to recover the flattened DAG in canonical order. Leaf nodes return themselves, while a parent node returns the catenated results of calling the operation on its child nodes, finally catenated with itself.

The *walk* operation is a generalization of the approach used by Snarl initially to build and recover Merkle trees[3]. The initial approach would retrieve the root node, and then recursively recover and traverse child nodes until the complete Merkle tree was recovered. If a node was missing, the code would attempt to repair it. The *walk* operation essentially does the same, except that it may be implemented by any backend, and it defers recovery and repair of child nodes to a recoverer abstraction (described in Section 3.2.6), which reduces code duplication.

3.2.3.2 The Next Path Operation

The *next path* operation, when called on a node, returns a link to a node in the Merkle DAG. The operation takes as input an integer n which indicates the caller is requesting the n-th leaf node in the Merkle DAG. In response, the callee node returns the link to the next node in the path to the requested leaf node from the position of the callee in the Merkle DAG. For internal nodes, the return value must be the link to a child node of the callee, while for leaf nodes, the identifier of the callee itself is returned.

The purpose of this operation is to traverse a branch of the Merkle DAG from the root node to a requested leaf node. Contrary to the *walk* operation, which traverses the entire Merkle DAG, the *next path* operation visits as few nodes as possible, and is only used for the purpose of recovering leaf nodes. It is used to recover leaf nodes from the Merkle DAGs of parity files, i.e. for parity block recovery, whenever parity blocks are required by the repair algorithm (described in Section 3.2.7).

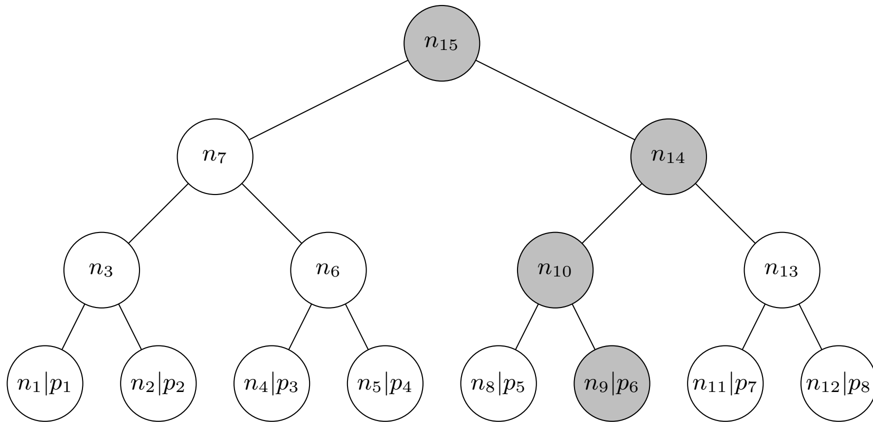

Figure 3.2: Illustration of calls to *next path* requesting the sixth leaf node p_6. Each node in the branch from the root node n_{15} to the destination node p_6 is highlighted.

An illustration of *next path* is provided in Figure 3.2. It contains a binary Merkle tree for a parity file, consisting of 15 nodes, with eight leaf nodes, i.e. eight parity blocks. The caller requires the sixth parity block p_6 for a repair operation. To retrieve this parity block, the caller recursively calls *next path* on nodes starting from the root node, each time with the

[3]Here we refer to Merkle trees, since originally Snarl only supported the Swarm backend, which uses Merkle trees to store files.

input $n = 6$, to traverse the branch to the leaf node. For each call, the caller retrieves the related block of data from the backend DSS, and deserializes the data into a node. When the caller retrieves a leaf node, in this case the node n_9 with the alias p_6, the algorithm completes.

Table 3.1: Results of calls to *next path* on nodes from the Merkle DAG illustrated in Figure 3.2. The call on the leaf node n_9 is not performed in practice, but the result is included for illustration.

Callee node	**Linked node**
n_{15}	n_{14}
n_{14}	n_{10}
n_{10}	n_9
n_9 (redundant)	n_9

The result of these calls to *next path* are provided in Table 3.1. Calls on internal nodes n_{15}, n_{14} and n_{10} returns links to their respective child nodes n_{14}, n_{10} and n_9 on the path to the destination node p_6 or n_9. In practice, the final call will be that made to n_{10}, which returns the link to the leaf node that is requested. The result of a hypothetical call to the leaf node n_9 is provided for the sake of illustration, which would return a link to the leaf node n_9 itself.

3.2.4 Entanglement

The entanglement procedure takes a file and produces α parity files each containing parity block content. Files in this context means sequences of bytes. The parity files are encoded as described in Section 2.4.2, such that all parity blocks from each type of strand — H-strands, LH-strands and RH-strands for triple-entanglements — are catenated into a single file for their corresponding type of strand. The code which produces parity files from the flattened DAG of the original file is a largely unmodified version of the original code.

Entanglement produces a flattened DAG and α parity files. The entanglement process is described in Section 2.4.2. The file is transformed into a Merkle DAG layout in the same way the backend DSS does for uploaded content. Furthermore, the Merkle DAG is flattened in the canonical order. Then, each node in the flattened DAG is treated as a data block, and entangled according to the rules in Section 2.3.3, producing α parity blocks for each node. To produce a flattened DAG, internally a backend implementation may split the file using the API of the backend, represent the blocks of data as nodes, then use the *walk* function from Section 3.2.3.1 to produce the flattened DAG.

3.2.5 Uploading

There are two scenarios where Snarl uploads data to the backend DSS. Firstly, Snarl may upload files containing entangled content. Both the original file and the α parity files are uploaded.

Secondly, Snarl may upload blocks to the backend DSS. When the repair algorithm fails to retrieve some block, and subsequently repairs it, it may optionally upload it. The repaired block is uploaded to the backend DSS for future availability.

Initially, uploading was tied to the APIs and implementation of the Swarm backend. The upload abstractions for files and blocks provide simple interfaces to implement this functionality for any backend.

3.2.6 Recovery

Recovery is defined as the retrieval of a requested node, either from the backend DSS or from local storage. There are two participants in recovery: the *requester* requests recovery of a node, and the *recoverer* performs recovery and returns the node to the requester. Requesters send requests containing the CID for a requested node in the backend DSS, the index of that node in the lattice, and a channel which the recoverer can use to respond to the particular request. In the simplest case, the recoverer simply downloads from the backend DSS the content pointed to by the CID sent by the requester, and ignores the index. This can occur regardless of AE codes, repairs, etc. In a more advanced case, the recoverer might need to repair blocks to recover the node, which is described in Section 3.2.7.

After retrieving the raw, serialized data related to the requested node, the recoverer must convert it to a node object. For this purpose, the abstraction layer defines a node deserializer, whose purpose is to convert raw, serialized data to nodes, in a backend-specific manner. The recoverer executes the deserialization code on the retrieved data, and forwards the result to the requester.

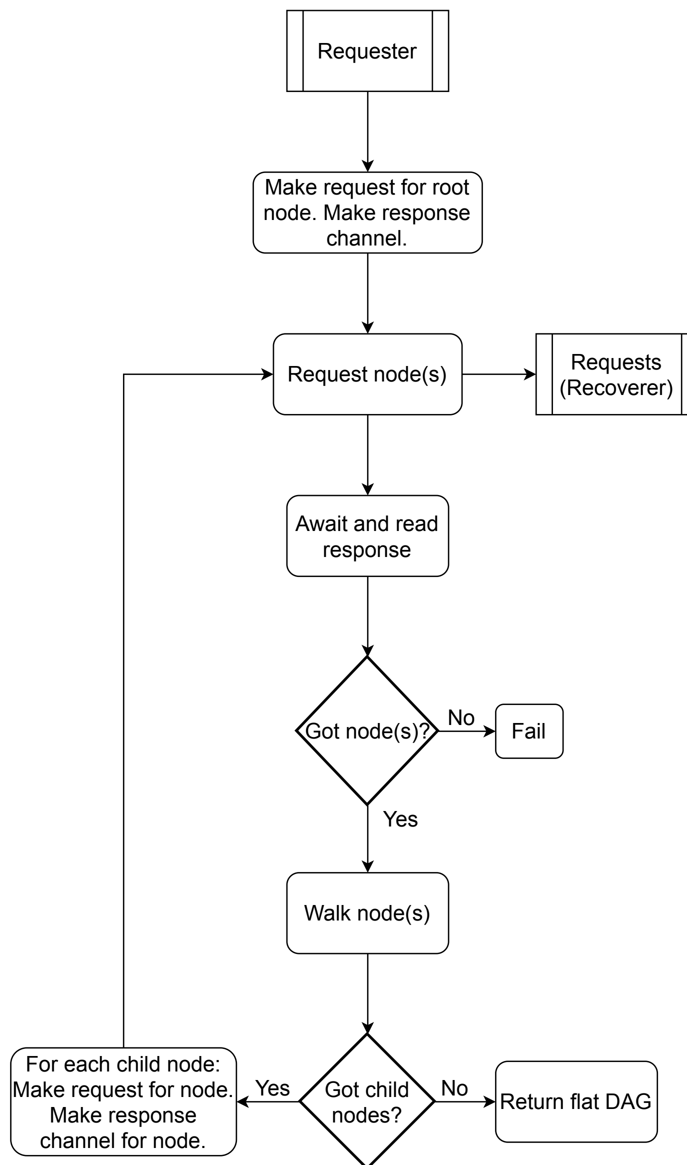

Figure 3.3: Flow chart of the requester in the recovery process.

The recovery process is illustrated with flow charts. The requester is illustrated in Figure 3.3, and is based on the provided interface implementation to recover a flattened

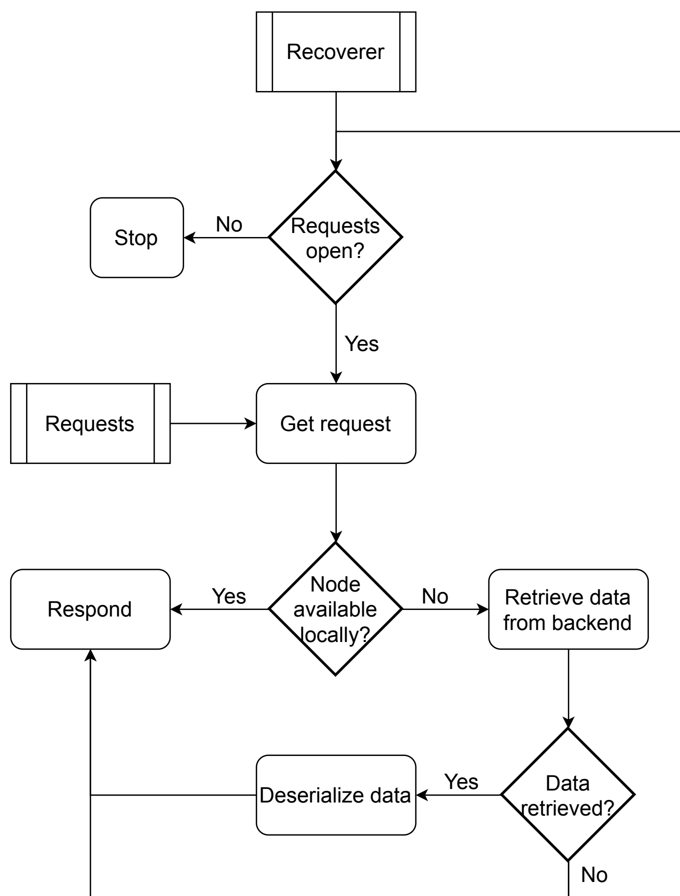

Figure 3.4: Flow chart of the recoverer in the recovery process.

DAG. The requester makes a request for the root node and requests it from the recoverer. If the node is returned without errors, the requester uses the *walk* operation described above to produce the flattened DAG rooted at the root node. The subsequent recursive requests for child nodes are performed by the backend implementation by sending further requests to the recoverer. Each recovery request provides a separate response channel.

The recoverer is illustrated in Figure 3.4. It simply receives requests, retrieves and deserializes the data, and responds to the request on the response channel provided in the recovery request. If an error occurs either during recovery or deserialization, the response includes the error message instead of the requested node.

The recoverer abstraction provides two important properties:

1. Requesters know nothing of the implementation details or capabilities of the recoverer. A requester simply has a reference to a channel where it can send recovery requests. The recoverer is responsible for producing the data, and deserializing it into a node. Whether the recoverer downloads the data, produces it by repairing, or already has the data is irrelevant to the requester. The requester simply looks at the response it receives, which either contains the requested node or an error message indicating failure.

2. The recoverer is independent of backend implementations. As mentioned, the recoverer accepts recovery requests, and produces nodes by downloading (or repairing) and deserializing data from the backend DSS. There are abstractions both for retrieving blocks by CID from a backend DSS, and for deserializing the data which is downloaded. These abstractions are implemented for each backend, and simply provided to the requester upon initialization, without specifying to the requester which backend

it operates on. It is assumed that recovery requests are only routed to a recoverer operating on the same backend, which is easily controlled by the implementation code. Since abstractions are used at all layers of the API, the exposed API is independent of any backend.

Comparatively, if the recovery and deserialization were performed without centralized recoverers, it would lead to code duplication. This is because each backend implementation would need to perform retrieval and deserialization themselves, and somehow communicate the need for repair, or also perform this themselves using exposed APIs. As mentioned, this was the case for the initial Snarl code. There, the Swarm backend implementation itself was responsible for retrieving blocks and attempting to repair them if they were irretrievable.

3.2.7 Repair

Repair is defined as recovering a requested node that was unavailable in the backend DSS by using the AE repair algorithm described in Section 2.3.4. The repair algorithm uses pp-tuples of two parity blocks, retrieved from the parity files described earlier in this section, to repair data blocks. Parity blocks might also need to be repaired, in which case the repairer uses a dp-tuple. All intermediate results are stored locally by the repairer such that the same block is not downloaded more than once during the recovery of a single file.

In the abstraction layer, a *repairer* is simply an extension to the recoverer. In addition to responding to recovery requests received on its request channel, it will attempt to perform repair whenever a data block (serialized node) is missing. Since lattice indices are necessary to determine dp-tuples and pp-tuples during repair, each recovery request must include the lattice index of the requested node.

The repairer is illustrated in Figure 3.5. Similarly to the recoverer in Figure 3.4, the repairer receives requests, produces the serialized data blocks, and deserializes them into nodes. The result of each request — the requested node or an error message — is sent as a response to the requester. Repair occurs when the node is not available locally, and cannot be retrieved from the backend. The repair process checks if any of the α possible pp-tuples are available to repair the data block. If not, the repairer recursively retrieves and repairs dependent blocks until any of the pp-tuples are available. If the pp-tuples are recovered successfully, the repairer repairs the data block, and proceeds with deserialization as if the block was retrieved regularly. Additionally, repaired blocks can optionally be uploaded to the backend DSS by using the block uploader abstraction, illustrated with a dashed line in the flow chart. If any intermediate step fails, the response contains an error message instead of the requested node.

Meanwhile, the requester exactly the same as illustrated in Figure 3.3, and explained in Section 3.2.6. As mentioned, the requester is ignorant of the implementation of the recoverer it sends requests to; it only considers the responses it receives to its requests. Initialization of the repairer is not related to the implementation of the requester.

3.2.7.1 New Approach to Parity Block Recovery

The abstraction layer specifies a new approach to retrieve parity blocks from the backend DSS. As described in Section 2.4.3.2, originally Snarl retrieved parity blocks by sending requests to a middleware for communicating with the Swarm backend. The repairer sent requests containing the CID of the root node of a parity file Merkle tree, and the AE helical lattice index i of parity block $p_{i,j}$. The middleware was then responsible for retrieving the parity block from the backend, and storing any nodes necessary from the parity file Merkle tree to retrieve the parity block. With this approach, the backend implementation does most of the work related to retrieving parity blocks, while the repairer simply sends requests. This is fine when there is only a single backend implementation, but requires each new backend implementation to duplicate the work related to retrieving parity blocks

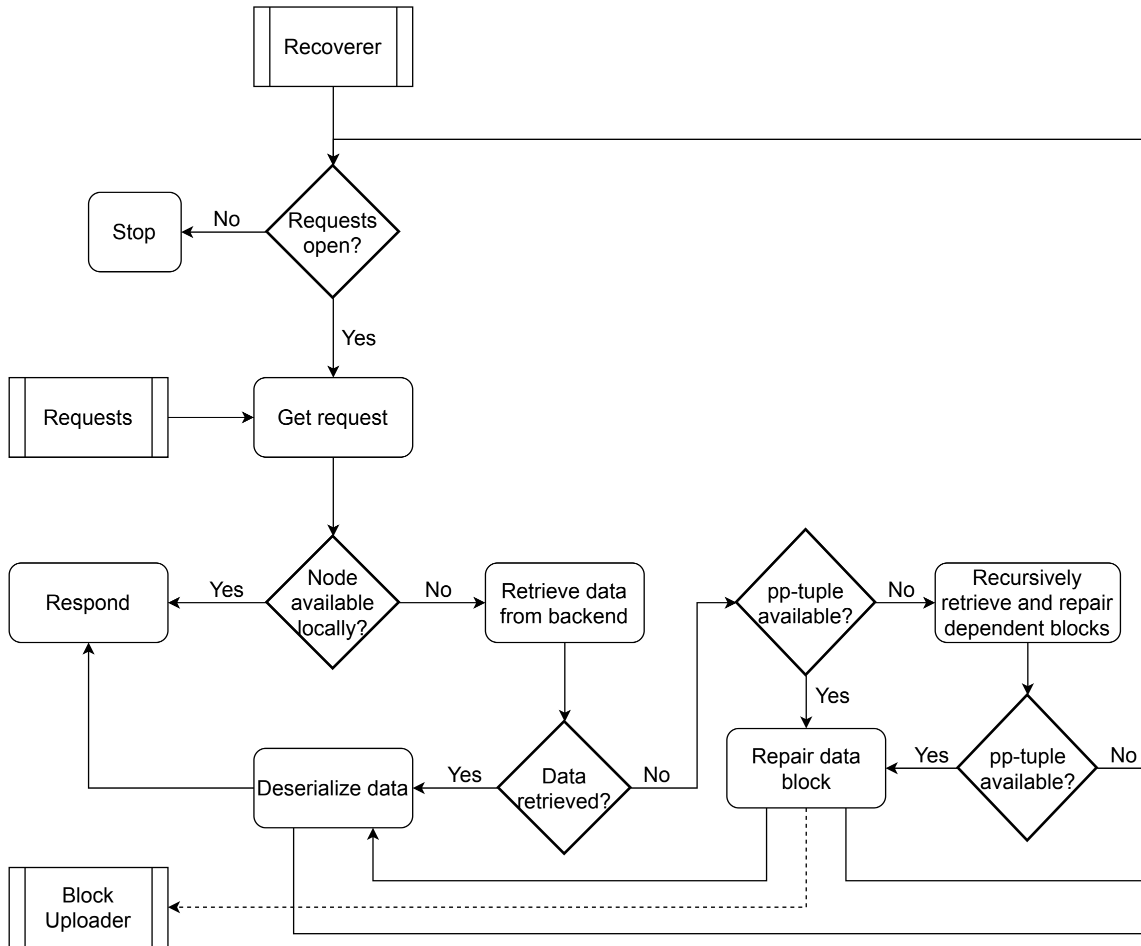

Figure 3.5: Flow chart of the recovery process of the repairer.

specifically, as well as thinking of a strategy to temporarily store internal nodes necessary for the retrieval of other potential parity blocks.

The abstraction layer instead uses the *next path* operation for parity block retrieval. Since parity blocks are leaf nodes of the Merkle DAGs generated from parity files, the operation is used to retrieve parity blocks (leaf nodes) from those DAGs. The repairer, when requiring a parity block for repair, recursively calls *next path* starting from the root node of the corresponding parity file Merkle DAG, continuing until it encounters a leaf node. Each retrieved parity node is stored by the repairer in a map data structure such that they are not downloaded more than once. The algorithm is simple to implement for new backends, and the majority of the logic is now performed by the repairer, rather than by each backend implementation.

3.3 IPFS Peer Management Layer

In this section we look at the design of the IPFS peer management layer (PML). The PML is used to control a set of network-connected peers. The purpose is to conduct experiments to test the operation of the IPFS backend implementation for Snarl. There already exists an experimental framework for the Swarm backend using Helm and Kubernetes [19]. We found it more convenient to implement a separate one for the IPFS backend. The remainder of this section looks at the motivation for the PML, an overview of how it operates, and requirements for the desired functionality.

3.3.1 Motivation

In this section we look at the motivation for creating a PML for conducting experiments for the IPFS backend for Snarl. To evaluate Snarl for the newly implemented IPFS backend, we need a way to run experiments involving several IPFS peers. The experiments should be controlled from a single peer, which controls the operation of other peers in the network to suit the experiment. For example, peers should be able to simulate crashing, losing data. It should also allow for distributing blocks of data in various ways.

There are existing solutions for IPFS peer and block management. For example, one solution is IPFS Cluster, a distributed application for controlling a cluster of IPFS peers and allocating and pinning items among them [8]. It does not match our needs for a few reasons. Firstly, the way the Snarl IPFS backend is implemented make them infeasible. The IPFS backend uses a proxy for communicating with the backend, using low-level IPFS APIs for uploading and downloading blocks. We would need to reimplement the IPFS proxy for the IPFS Cluster as well. Secondly, to allow for a broad range of potential experiments, we prefer direct control over peer operation.

There is also the experimental framework for the Swarm backend of Snarl. This was not adapted for a few reasons. Firstly, it is implemented with specifics of the Swarm backend in mind, and we would have to replicate most of the functionality for IPFS. Secondly, we do not have previous experience with Helm and Kubernetes, and prefer to use other frameworks we are more experienced with.

3.3.2 Overview

In this section we look at an overview of the communication flow between peers during experiments. There are two types of peers. The *controller* controls the operation of other peers, distributes nodes, and uses Snarl for entanglement and recovery. All other peers simply execute commands received through the PML, and run the IPFS daemon during experiments.

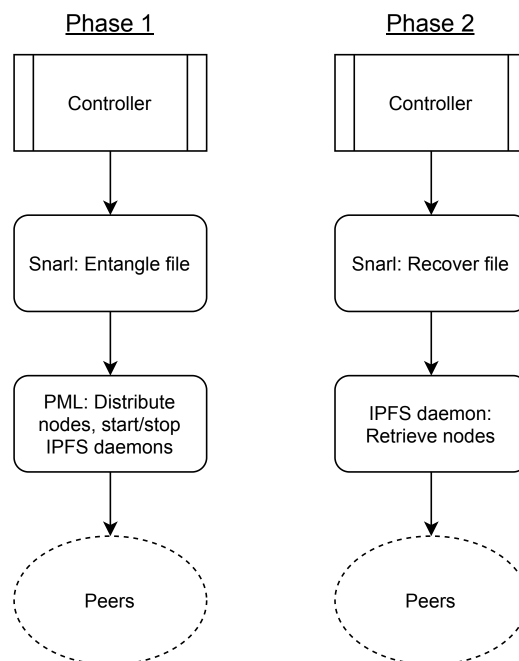

Figure 3.6: Overview of operation of controller, Snarl and PML during experiments.

The operation of the controller and PML during experiments is illustrated in Figure 3.6. During experiments, there are two phases. In the first phase, the controller uses Snarl to entangle an input file, and communicates with peers directly through the PML to set up

the experiment. This involves starting or stopping IPFS daemons on other peers and distributing nodes from the entangled file to various peers.

In the second phase, the controller uses Snarl to recover the file. The Snarl repairer, using the abstraction layer, retrieves nodes from other peers through the IPFS daemon. Based on the scenario, node and peer loss may have occurred, necessitating the retrieval of parity nodes for the repair algorithm.

3.3.3 Requirements

In this section we examine the requirements for the PML. To enable the functionality described in Section 3.3.2, and to enable various scenarios, certain requirements must be met. The following paragraphs describe requirements related to the operation of peers, node distribution, and operations needed to fulfill the requirements.

Controller The PML must enable a single peer, the controller, to control operation and failure scenarios of other peers. This makes it simple to conduct experiments and obtain results. The controller runs Snarl to entangle files and distribute nodes, and to recover uploaded files.

Failure scenarios There are two types of failure scenarios to simulate. The first is *node loss*. This occurs when one or more DAG nodes are unavailable from one or more peers where they should have been available. Corrupt node content is not considered — we assume IPFS confirms downloaded content based on CID. If node content is corrupt at the sender, the node is considered unavailable altogether.

The second type of failure is *peer loss*. This occurs when a peer that stores a node is unavailable, e.g. due to crashing, leaving the network, or other network issues.

In both cases, the node cannot be retrieved from the peer in question. It must be retrieved in other ways, such as from another peer storing the same node, or through repair.

The PML must grant the ability to seamlessly inject both node loss and peer loss. The loss should be able to be injected in any peer participating in the network, and to any node that can be identified by CID.

Operations The PML must support certain operations to meet the above requirements. The following operations are considered essential:

- *Pinning nodes:* To distribute nodes through the network, the controller orders peers to pin nodes identified by CID. Pinning may be recursive, i.e. including the subtree of the identified node. Otherwise only the identified node is pinned.
- *Deleting nodes:* To simulate node loss, the controller orders peers to delete nodes identified by CID. The peer removes the node from their local storage, making it unavailable through requests to IPFS.
- *Start and stop IPFS daemon:* To simulate peer loss, we must be able to start and stop the IPFS daemon of peers. While peers are accessible through the PML at any time, the IPFS daemon is initially not running. The controller orders the peers involved in an experiment to start their IPFS daemon. Peers in a peer loss scenario are ordered to stop their IPFS daemon.
- *Diagnostics:* The PML should provide certain diagnostics, such as whether nodes are pinned or available, peer's network ID, etc.

Chapter 4

Implementation

This chapter presents API for the abstraction layer and implementation challenges. The first section gives a detailed overview of most APIs defined by the introduced abstraction layer. The information in that section is quite detailed, and complete understanding of it is not necessary to understand the remainder of the thesis. The other section describes challenges encountered during updates to Snarl internals, the implementation of the IPFS backend, and updates to the original Swarm backend.

4.1 Backend API

In this section we look at the Go API which provides the functionality described in Section 3.2. The API description is divided into several domains of related functionality to make relations more apparent. The domains that are described are the node representation, entanglement of data, uploading to the backend DSS, retrieving and deserializing nodes from the backend DSS, and finally file recovery and repair.

4.1.1 Node and DAG Representations

Listing 4.1: API for node and flattened DAG representations.

```
type Node interface {
  ID() []byte
  ToRaw() []byte
  Walk(ctx context.Context, recoverer chan<- RecoveryRequest)
      (FlatDAG, error)
  NextPath(index uint) (link []byte, linkIndex uint, err error)
  IsLeaf() bool
}

type FlatDAG []Node

func (dag FlatDAG) ToRaw() [][]byte

func (dag FlatDAG) ToRawData() []byte

func (dag FlatDAG) Root() Node
```

The API for representing nodes and flattened DAGs is shown in Listing 4.1. The `Node` interface is used to represent nodes as described in Section 3.2.3. Meanwhile the `FlatDAG` type represents a flattened Merkle DAG as a sequence of nodes. The remainder of this section describes these types in more detail.

4.1.1.1 The Node Interface

In this section we look at the ToRaw, Walk and NextPath methods of the Node interface. The ID and IsLeaf methods are self-explanatory.

ToRaw method The ToRaw method returns the serialized, raw data contained in the node, i.e. the same data stored by the backend DSS. This data has two purposes. Firstly, the data from the leaf data blocks is used to reconstruct the original file during recovery. Secondly, raw data of any data block and of leaf nodes from the parity file Merkle DAGs (parity blocks) is used for the repair algorithm as described in Section 2.3.4.

Walk method The Walk function returns the subtree of the node, including the node itself, in flattened, canonical order, as described in Section 3.2.3.1. While not enforced by the interface, it is intended to be called recursively on child nodes linked from the origin node. In this way, Walk can be called on the root node to recover every node in the Merkle DAG, flattened in canonical order. When encountering missing child nodes in the path, recovery requests for the missing nodes are sent to the input recoverer channel. Retrieval of child nodes and recursive calls to Walk can be concurrent, since nodes are arranged hierarchically, i.e. child nodes do not depend on sibling nodes, uncle nodes, or similar, only the branch from the root node.

NextPath method The NextPath method implements the *next path* operation described in Section 3.2.3.2. The input argument specifies which leaf node is requested. As a result, the node which NextPath is called on returns the CID (link) and canonical index (linkIndex). Both of the aforementioned return values are necessary to retrieve and deserialize the next node in the path. The caller recovers the linked-to node and calls the method again, repeating the process until it recovers the requested leaf node.

4.1.1.2 The FlatDAG Type

In this section we look at the FlatDAG type. It is used to represent flattened Merkle DAGs in the canonical order. Additionally, it is used for intermediate results, i.e. flattened subtrees in canonical order from the result of a call to Walk on a non-root node.

There are three convenience methods for the type. ToRaw returns the result of calling the ToRaw method on each node in the DAG, and Root returns the last node in the DAG. Most useful is the ToRawData method, which catenates the results of calling the ToRaw method on each leaf node in the DAG. This is used to recover the original file from the flattened data block Merkle DAG.

4.1.2 Entanglement

Listing 4.2: API for entanglement.

```
type FlatDAGConstructor interface {
  Construct(ctx context.Context, r io.ReadSeeker) (FlatDAG, error)
}

type ShiftingConstructor interface {
  FlatDAGConstructor
  Shift(FlatDAG) (FlatDAG, error)
}

type EntangleHelper struct {
  // unexported fields
}

func NewEntangleHelper(aeConf config.AEConfig,
                       constructor backend.FlatDAGConstructor)
                      *EntangleHelper

func (e *EntangleHelper) WithOutputDirectory(outDir string)
                                            *EntangleHelper

func (e *EntangleHelper) WithParityWriters(parityWriters []io.Writer)
                                          *EntangleHelper

func (e *EntangleHelper) Entangle(ctx context.Context, r io.ReadSeeker)
                                 error

func (e *EntangleHelper) FlatDAG() backend.FlatDAG
```

The API for entanglement is shown in Listing 4.2. It consists of an interface for constructing flattened Merkle DAGs, which is implemented for each backend, and helper type used to perform entanglement. In the remainder of this section we look at them in more detail.

4.1.2.1 The FlatDAGConstructor Interface

This is a high-level interface for constructing a flattened DAG from the contents of a file. The configuration (block size, branching factor, etc.) of nodes in the flattened DAG matches that which the backend would use when uploading the same file. The `Construct` function constructs a flattened DAG from the contents of the `r` parameter. `r` is an `io.ReadSeeker`, which means in addition to providing the `Read` method, it provides the `Seek` method which can be used to determine the size of the contents of `r`[1], which is necessary for some backends, such as Swarm. The implementation of `Construct` may vary, but both Swarm and IPFS implementations use the `Walk` method to produce the flat DAG, using a simple, internal recoverer, recovering nodes already locally available after splitting and processing the contents of `r`.

An extended interface `ShiftingConstructor` provides a generic interface for the internal node swapping described in Section 2.4.2.2. The interface extends the `FlatDAGConstructor` interface by providing a method `Shift` to swap internal nodes in the flattened DAG according to Algorithm 1. The interface is optional to implement, and internal node swapping can be disabled in the configuration for backends that do not implement this interface.

[1]Note that the original purpose of `Seek` is to change the offset of subsequent read or write operations. It returns the new offset. By seeking to an offset 0 from the current position, then switching to an offset 0 from the final position, finally seeking back to the original position, we can discover the size by subtracting the initial position from the final position.

4.1.2.2 The EntangleHelper Type

The `EntangleHelper` type, with its `Entangle` method, is used for entanglement as described in Section 3.2.4 It is adapted from the algorithm described in Section 2.4.2.3, additionally having some configuration possibilities for where to write parity file outputs. In the following paragraphs we look at implementation details of the entangler.

Constructor and additional settings An `EntangleHelper` instance is initialized with the constructor function `NewEntangleHelper`, and one or both of the `WithOutputDirectory` and `WithParityWriters` methods. We use a builder pattern, initializing an `EntangleHelper` instance with obligatory fields with the constructor function, and then using the with-methods to modify internal fields of the instance. By modifying unexported fields of the helper type, the with-methods modify the behavior of its `Entangle` method. At least one of the with-methods must be called — it is pointless to generate parity files from the constructed data Merkle DAG without storing them anywhere. E.g. we could initialize it by calling `NewEntangleHelper(...).WithOutputDirectory(...)`. Beneficially, the with-methods enable three different configurations: with output directory, with parity writers, or with both.

The Entangle method The `Entangle` method generates a flattened Merkle DAG from an input reader, then generates α parity files from its contents. It uses the internal `FlatDAGConstructor` to generate the flattened Merkle DAG of data blocks. It then uses the entanglement algorithm to produce α parity files. The entanglement algorithm operates as described in Section 2.4.2.3, but is adapted to not necessarily write results to files.

When the helper is configured using the `WithOutputDirectory` method, parity files are written to the specified output directory. The output directory provided as the argument to the method is created if it does not exist, and the α parity files are written to the directory, named $0, 1, \ldots, \alpha$. Using this configuration produces the same results as the original implementation did.

When the helper is configured with the `WithParityWriters` method, parity files are written to the provided writers. This has many potential use cases, such as writing the results to existing files, in-memory buffers, network streams, or encrypting the content.

As mentioned above, both of the with-methods may be called to produce both effects when `Entangle` is called.

The FlatDAG method The helper also provides the `FlatDAG` method. It can be called after the `Entangle` method is done, returning the data blocks produced by the `FlatDAGConstructor` as part of the entanglement process. This is mostly used for testing purposes, but could potentially have other use cases.

4.1.2.3 Summary

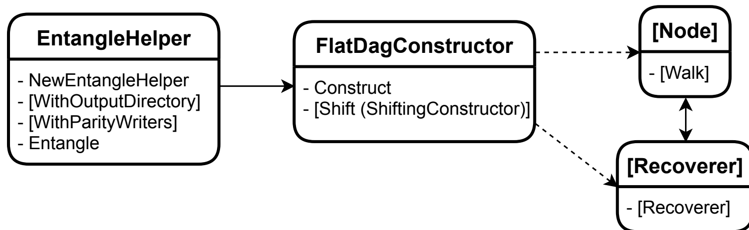

Figure 4.1: Overview of type relations in the entanglement process. Optional elements have names surrounded by brackets (`[]`), and are pointed to by dotted lines.

Figure 4.1 illustrates relationships between the abstraction layer types in the entanglement process. The `EntangleHelper` type is the entrypoint to the entanglement process. After initialization, the `Entangle` method is called to perform entanglement. To construct a flattened DAG of data blocks from the input data, it calls the `Construct` method implemented by the backend. While the implementation of the `FlatDAGConstructor` interface is backend-dependent, a reasonable approach may use the `Walk` method of the `Node` interface to communicate with a local recoverer implementing the `Recoverer` interface. After constructing the flattened DAG, if the `ShiftingConstructor` interface is implemented by the backend, the entangler calls the `Shift` method to shift internal nodes according to Algorithm 1. Finally, it entangles the data blocks to produce α parity files, written to the configured output directory, parity writers, or both.

4.1.3 Uploading

Listing 4.3: API for uploading blocks and files.

```
type BlockUploader interface {
  UploadBlock(ctx context.Context, data []byte) (id []byte, err error)
}

type FileUploader interface {
  UploadFile(r io.Reader) (rootID []byte, err error)
}
```

The API for uploading data to the backend DSS is shown in Listing 4.3. Upload in this case means storing blocks of data or files in the background DSS, to be able to retrieve the content later on. Uploading does not have to be a networked operation — a backend could define uploading as anything, such as storing the content in the local file system. The upload API consists of two interfaces, `BlockUploader` and `FileUploader`, which are described in the remainder of this section.

4.1.3.1 The BlockUploader Interface

The `UploadBlock` method uploads a block of raw data to the backend. On success, the CID of the uploaded block is returned. Otherwise, an error is returned describing the cause of the error. Snarl uses this method to upload repaired blocks that were missing from the backend, to make them available once again, as mentioned in Section 3.2.7. This interface can optionally not be implemented. If an implementation of `BlockUploader` is not provided to the Snarl repairer, it ignores the uploading step of the repair algorithm.

4.1.3.2 The FileUploader Interface

The `UploadFile` method uploads the content of a "file" (an `io.Reader` implementation) to the backend. The implementation reads all content from the current position of the `r` parameter until the end, and uses the mechanism of the backend DSS for uploading the contents. Large files are probably split into blocks and turned into a DAG by the backend, e.g. the Merkle DAG in IPFS. If successful, the method returns the CID of the root node in the DAG representing the uploaded file. Otherwise, an error is returned describing the cause of the error.

This method is used to upload the data file and the parity files produced by entanglement as described in Section 4.1.2. The end-user must store or remember the CID, both for the uploaded data file and related parity files. The CIDs must be provided to the repairer to recover and repair the file.

4.1.4 Node Retrieval and Deserialization

Listing 4.4: API for retrieving and deserializing nodes.

```
type BlockGetter interface {
  GetBlock(ctx context.Context, id []byte) (data []byte, err error)
}

type NodeDeserializer interface {
  Deserialize(rawNode, id []byte, index uint, isParity bool) (Node, error)
}

type NodeGetter interface {
  BlockGetter
  NodeDeserializer
}
```

The API for retrieving and deserializing nodes is shown in Listing 4.4. Node retrieval means retrieving the raw, serialized node from the backend DSS, typically by downloading it. Deserialization means converting the raw, serialized node into a `Node` instance for use by Snarl algorithms. Some nodes, such as leaf nodes, may be stored as raw data depending on the backend; still the backend `Node` implementation may add additional information to deserialized leaf nodes. The API contains interfaces for these operations, which are described in the remainder of this section.

4.1.4.1 The BlockGetter Interface

`GetBlock` is the most basic recovery method. It is used to retrieve a block from the backend, which may be available locally or need to be downloaded from the backend DSS. In this context, block means block of raw data, or a raw, serialized node. The CID of the block is provided to identify the block. If successful, the method returns the serialized block. Otherwise, an error is returned.

The implementation should only retrieve the requested block, not its subtree. The *walk* and *next path* operations rely on using this method to retrieve single blocks while traversing all or parts of a DAG. If the method tries to retrieve the subtree of the requested block and fails if parts of the subtree is unavailable, the recovery and repair algorithms may not perform as intended.

The contents of retrieved blocks depends on the backend and type of node. `data` may be the raw data of the block (e.g. for leaf nodes), or may contain additional data such as links to child nodes or other metadata. Since the serialized node is simply represented as a sequence of bytes, the content can be arbitrarily complex (as long as the block size is not exceeded). E.g. it would be possible to store nodes as JSON-serialized structures, and for the deserializer to turn the JSON-structure into an instance of a Go type. In most cases the data must be deserialized into a `Node` using the `NodeDeserializer` interface, in order to be used by other algorithms.

4.1.4.2 The NodeDeserializer Interface

The `Deserialize` method takes a serialized node retrieved from the backend DSS, and converts it into a `Node` instance. In addition to the serialized data, the CID, canonical index, and a parity block predicate are provided. This information should be sufficient for the deserializer to convert the serialized data into a node. The index is needed for calls to `Walk` and `NextPath` on the node. It may be necessary to differentiate between data blocks and parity blocks, e.g. because their DAG layout is different. As such, the `isParity` field enables the use of a single deserializer instance to deserialize both data blocks and parity blocks. The deserializer might track relationships between nodes and their indices, and more depending on the backend.

4.1.4.3 The NodeGetter Interface

This interface is simply a composition of the `BlockGetter` and `NodeDeserializer` interfaces. Accepting an implementation of this interface, a recoverer can retrieve and deserialize nodes without any knowledge of the operation or storage layout of the backend DSS. It can also be used for more than just file recovery. For example, in the `FlatDAGConstructor` implementations for IPFS and Swarm, the interface is used by a local recoverer when producing the flattened Merkle DAG.

4.1.5 Recovery and Repair

Listing 4.5: API for recovery and repair.

```
type Recoverer interface {
  Recoverer() chan<- RecoveryRequest
}

type Repairer interface {
  Recoverer
  RepairBlock(ctx context.Context, index int) (rawBlock []byte, err error)
}

type RecoveryRequest struct {
  Ctx      context.Context
  ID       []byte
  Index    uint
  IsParity bool
  Result   chan RecoveryResult
}

type RecoveryResult struct {
  Node Node
  Err  error
}

func Recover(ctx context.Context, id []byte, index uint,
              isParity bool, recoverer chan<- RecoveryRequest)
             (node Node, err error)

func (r *RecoveryRequest) Respond(node Node, err error)

func RecoverWrite(ctx context.Context, dataRootID []byte, rootIndex uint,
                  recoverer Recoverer, w io.Writer) (FlatDAG, error)

func RecoverDAG(ctx context.Context, dataRootID []byte, rootIndex uint,
                recoverer Recoverer) (FlatDAG, error)
```

In this section we look at the API for recovering blocks. The API is displayed in Listing 4.5. It consists of interfaces for the recoverer and repairer abstractions, primitives for sending and responding to recovery requests, and functions built on these abstractions that recover flattened Merkle DAGs.

4.1.5.1 The Recoverer Interface

This interface represents a type which is capable of recovering nodes. The `Recoverer` method returns a write-only channel accepting `RecoveryRequest` instances. In this way callers can request nodes identified by `ID` (CID) and `Index` (AE helical lattice index), and have the result returned on the `Result` channel they provide.

The `Recoverer` interface is simple, and there are many ways to implement it. A simple implementation of the recoverer already has nodes locally, and responds with them when requested. Another possible implementation is the recoverer described in Section 3.2.6.

Note that a recoverer does not necessarily implement a repair algorithm for missing nodes, and can be implemented for more trivial use cases, such as for use during flattened DAG construction.

The `Index` and `IsParity` fields of recovery requests can be used to give hints to the recoverer about how it should handle the recovery request. Lattice indices are numbered starting from 1. With this in mind, for data blocks, the requester can set the `Index` field of a recovery request to 0 to indicate to the recoverer to not attempt repair of blocks, only recovery. Other than that, for parity nodes (when the `IsParity` field is set to true), the recoverer may handle recovery different, e.g. by using the *next path* operation. Note that the requester in the recovery process as described in Section 3.2.6 is not intended to request parity blocks; the repair algorithm may send recovery requests to its recoverer internally when parity blocks are required. The interface does not guarantee that these hints are respected.

The requester and recoverer communicate only by using channels, which provides several benefits:

- The requester requires no knowledge about the implementation or capabilities of the recoverer; either the recoverer provides the requested node or not, and the requester acts accordingly.
- There is no need to duplicate recovery logic in the implementations of requesters. Every requester implementation (in practice implementations of the `Walk` method of the `Node` interface) recovers data blocks in the same way by sending recovery requests to the recoverer.
- Mode of operation can be configured through channel initialization. Go channels can be unbuffered, in which case the send-receive operations are synchronized (the sender cannot proceed until a receiver takes the sent item, and vice versa). Otherwise Go channels can be buffered, in which case send operations are only blocking while the buffer is full [30]. In other words, depending on how we initialize the channels, we can control whether the requester and recoverer communicate synchronously or asynchronously, and limit the number of concurrent requests by using a buffered channel.
- When using channels for requests it is simple to create mock types for testing the implementation of requesters. The mock types can respond to recovery requests in a predictable manner depending on the test scenario, e.g. refusing to provide certain blocks to test how the requester handles errors.

4.1.5.2 The Repairer Interface

This interface extends the aforementioned `Recoverer` interface, additionally providing repair capabilities for irretrievable nodes. A possible implementation is the repairer described in Section 3.2.7. When retrieving a block with `GetBlock` fails, the `RepairBlock` function is used to repair the block using the AE repair algorithm, which may involve recursive repairs. The `index` parameter is available from the `Index` field of `RecoveryRequest`, and represents the AE helical lattice index of a data block. Repairing parity blocks is considered an implementation detail, and not exposed by the `Repairer` interface.

When successful repairing blocks, the result of `RepairBlock` is equivalent to that which `GetBlock` would have produced. It will similarly need to be deserialized using a `NodeDeserializer` implementation. Repaired blocks may be uploaded to the backend DSS if an implementation of the `BlockUploader` interface is provided to the repairer, however this is not exposed by the interface either, thus making it optional.

As previously mentioned, from the perspective of the recoverer, there is no difference between a recoverer and a repairer. The communication interface between the requester

and the recoverer consists of recovery requests, recovery responses and the corresponding channels. The `RepairBlock` method is not used outside the implementation of the repairer itself. As such, the `Repairer` interface should be considered a guideline for alternative repair implementations.

4.1.5.3 The Recover and Respond Functions

These functions are provided to simplify use of the `Recoverer` interface. They functions act as wrappers around the communication channels between the requester and the recoverer. The requester may call `Recover` to correctly perform a recovery request. The function sets up a recovery request, sends the request to the recoverer, and awaits the response using a select statement, such that the context is respected. The recoverer may respond using the `Respond` function, which sets up a recovery response and sends it to the requester.

Usage of both functions is optional. The inputs to the `Recover` and `Respond` functions match the fields of the respective `RecoveryRequest` and `RecoveryResult` types. The functionality can easily be replicated by manually setting up request and response instances, and communicating using the channels. An example scenario where it is desirable to manually set up requests is for asynchronous communication. Both of these functions use channels synchronously; channels can be used asynchronously by replicating the behavior of the functions, which is simple.

4.1.5.4 Exposed Recovery Functions

The functions `RecoverDAG` and `RecoverWrite` recover the flattened Merkle DAG by communicating with the recoverer. Essentially, these functions operate as the requester in the recovery process described in Section 3.2.6. First they recover the root node of the data block Merkle DAG, and then use the `Walk` method to recover the remainder of the flattened Merkle DAG. The `RecoverDAG` function simply returns the flattened Merkle DAG. The `RecoverWrite` function additionally writes the contents of the recovered file to the writer provided by the `w` parameter, using the `ToRawData` method described in Section 4.1.1.2. If repair should be enabled, the recoverer must be initialized as such. The functions described in this section are unaware of the repair process.

4.1.5.5 Summary

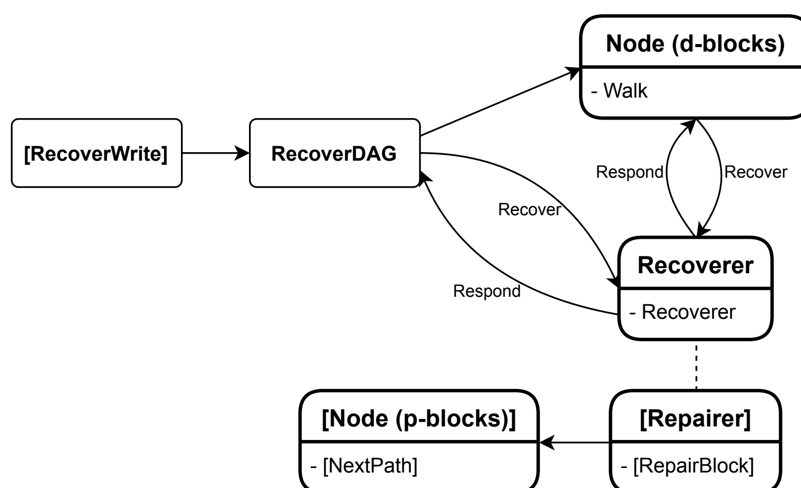

Figure 4.2: Overview of type relations in the repair process. Optional elements have names surrounded by brackets (`[]`). The `Recoverer` implementation optionally also implements the `Repairer` interface; this relationship is illustrated with a dotted line.

Figure 4.2 illustrates relationships between the abstraction layer types in the repair process. Either the `RecoverDAG` function, or the higher-level `RecoverWrite` function, is the entrypoint to the repair process. The function recovers the root data block from the recoverer by sending a request with the `Recover` function. The recoverer retrieves the data block, and responds with the `Respond` function.

Furthermore, the recovery function calls the `Walk` method on the root data block to recover the subtree, flattened in canonical order. The root data block, and any descendant data block, send recovery requests for child nodes to the recoverer, which retrieves the nodes and responds with the results. When all data blocks are recovered, the recovery function returns the flattened DAG or writes it to the provided writer.

Behind the scenes, the recoverer might implement the `Repairer` interface and a repair algorithm. In this case, when it fails to retrieve data blocks from the backend DSS, it uses the `RepairBlock` method to repair the data blocks. To repair data blocks it must retrieve parity blocks. It uses the `NextPath` method from the `Node` interface to determine the path it has to traverse in the parity file DAG to retrieve the required parity blocks, which are leaf nodes in the DAG.

4.2 Index Mappings

This section describes the index map data structure, and some of its use cases. First, it describes the reasons index mapping are useful. Then, the data structure itself is described. Finally, it describes two use cases which use the index mapping to simplify the implementation.

4.2.1 Motivation

A recurring problem is finding relations between DAG nodes. For example, both the *walk* and *next path* operations used during recovery and repair need to know the layout of the DAG they are working on. Being at a node with a certain canonical index, we need to know the indices of its child nodes to use these operations. There are many possible ways to determine these indices, and the implementation varies depending on the particular backend and the data structures it uses to store nodes. In backends with highly structured data layout, such as Swarm which uses Merkle trees with a fixed branching factor, it possible to determine indices of child nodes with an algorithm. Though in practice, it is difficult to come up with such an algorithm, and to be certain that it does not mistreat undiscovered edge cases. Other backends may have more varied DAG structures, which is the case for IPFS, making it even harder to predict DAG layouts.

4.2.2 The Index Mapping Data Structure

To solve these problems, a simple data structure called an *index mapping* is added to Snarl, intended for use in backend implementations. The index mapping is based on a mapping from parent indices to lists of child indices. The mapping can be generated for any backend by traversing a locally available DAG with the desired layout. During recovery, when the DAG of the requested file is not locally available, the backend implementation can generate a DAG with the same layout by running the backend's DAG construction algorithm on a buffer of the same size as the file that contains arbitrary data.

Figure 4.3 and Table 4.1 illustrate index mapping generation. If the backend generates a DAG with the layout illustrated in Figure 4.3, it can generate the corresponding index mapping easily. Starting from the root node, it traverses the DAG in the same order as the *walk* operation, storing mappings from parent index to child indices in a hash table. The resulting hash table values are displayed in Table 4.1.

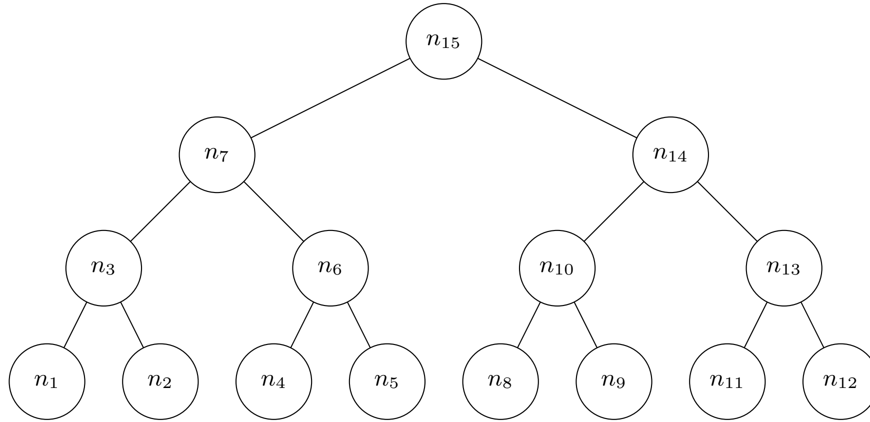

Figure 4.3: Binary tree with 15 nodes. Node subscripts contain the node's canonical index.

Table 4.1: Index mappings generated from the tree illustrated in Figure 4.3.

Parent node index (key)	**Child node indices (value)**
15	$[7, 14]$
14	$[10, 13]$
13	$[11, 12]$
10	$[8, 9]$
7	$[3, 6]$
6	$[4, 5]$
3	$[1, 2]$

When recovering a file, the index mapping can be generated and used for various purposes. For example, both the updated Swarm backend and the new IPFS backend generate and share an index mapping between all instance of nodes implementing the `Node` interface. This makes it easy to find indices of child nodes in the implementations of the `Walk` and `NextPath` methods.

Listing 4.6: Exported types and functions from `package indexmap`.

```
type IndexMap struct {
  Indices map[uint][]uint
  // other unexported fields
}

func FromMap(indices map[uint][]uint) *IndexMap
func (m IndexMap) String() string
func (m IndexMap) Get(index uint) ([]uint, error)
func (m IndexMap) RootIndex() uint
func (m IndexMap) LeafIndex(n uint) uint
func (m IndexMap) LeafMapping() []uint
func (m IndexMap) ParentMapping() map[uint]uint
func (m IndexMap) Height(index uint) uint
func (m IndexMap) NumNodes() int
func ShiftNodes(im *IndexMap, dag backend.FlatDAG, aeConf config.AEConfig)
            (backend.FlatDAG, error)
```

The exported API for `IndexMap` from the `indexmap` package is displayed in Listing 4.6. At its core, the index map is simply a mapping from a parent index (unsigned integer) to a list of child indices (slice of unsigned integers). A pointer to an `IndexMap` instance can be generated and stored by the deserializer, used during deserialization, and be shared among node instances. There are several exported methods which are convenience methods

for properties that can be discovered from the index mapping. For example, a node which knows its index i, and has an index mapping `imap`, can call `imap.Get(i)` to get the list of indices of its child nodes[2]. The operation of most of the convenience methods is self-explanatory, while some others are described in more detail below.

There is no constructor for the index map, since the layout depends on the particular backend. A backend implementation which wants to use the index map internally must generate an index map, and use the `FromMap` function to represent it as an `IndexMap` instance. As mentioned in Section 2.4.3.1, Snarl requires metadata such as the size of each block generated for the recovery process. The index map may be generated by the backend implementation simultaneously.

4.2.3 Accessing Parity Blocks

One use case for the index mapping is to access parity blocks. Recall that only leaf nodes in the parity file DAGs contain parity blocks. Thus, to access the n-th parity block in a DAG, one must find the n-th leaf node in the DAG. This process is the basis of the *next path* operation.

The index mapping can be used for this purpose. It provides two useful methods `LeafMapping` and `LeafIndex` to discover indices of leaf nodes. The `LeafMapping` method returns a sorted list l containing all leaf node indices. For example, the DAG illustrated in Figure 4.3 produces the list of leaf node indices $l = [1, 2, 4, 5, 8, 9, 11, 12]$. In this way, with $0 \leq i < |l|$, the element l_i contains the helical lattice index of leaf node number $i + 1$ (due to zero indexing). In the above example, the first leaf node has the index $l_0 = 1$, and the fourth leaf node has the index $l_3 = 5$. Similarly, method `LeafIndex(n)`, $1 \leq n \leq |l|$, returns the helical lattice index of the n-th leaf node. These convenience methods can be used to implement the `NextPath` of the `Node` interface. The Swarm and IPFS backends rely on the `LeafIndex` method.

4.2.4 Swapping Internal Nodes

Recall from Section 2.4.2.2 that Snarl uses an algorithm to swap the order of certain data blocks to improve recovery rates. In the original implementation, this algorithm was implemented in the `Lattice` type, and as a method `FlattenTreeWindow` for the Swarm backend. The algorithm had to be implemented for other backends as well, and it was desirable to abstract the process such that the backends could use the node swapping algorithm in a generic manner.

For this purpose, the algorithm is implemented in the `ShiftNodes` function. The function accepts a flattened DAG and its corresponding index mapping, determines which nodes need to be swapped according to Algorithm 1, swaps those nodes, and returns the modified flattened DAG. It can be used by any backend implementation which creates an index mapping for the DAG constructed during the entanglement process. The `Shift` method of the `ShiftingConstructor` interface described in Section 4.1.2.1 can thus be implemented easily by calling the `ShiftNodes` function. This approach is used both in the updated Swarm backend implementation and in the IPFS backend implementation.

4.3 Changes to the Lattice Type

In Snarl, the `Lattice` type in the `entangler` package is the primary component used for recovery and repair. In the original implementation, many methods of this type depended on

[2]The `Get` function is just a convenience function around Go's map get operation, but it also checks for the existence of the parent index in the map. The same could be achieved by something like `if index, ok := imap.Indices[i]; ok {...}`, which sets the boolean `ok` to true if the key i exists in the map.

the Swarm backend or were otherwise programmed in a way which depended on implementation details of the Swarm backend. Most dependencies were removed by the introduction of the abstraction layer described in Section 3.2 and Section 4.1. Additionally, certain logical changes were necessary to work with parts of the abstraction layer.

The remainder of this section describes notable changes to the `Lattice` type and its related methods. The updated implementation adapts the original implementation by adding new, relatively isolated processes, while making minimal changes to methods of the `Lattice` type. The new processes use a CSP-approach to handle requests for data blocks and parity blocks. Several distinct processes communicate using channels, and primarily share memory by communication, reducing lock contention and shared state.

4.3.1 Implementing the Repairer Interface

In this section we look at changes made to the exposed API of the `Lattice` type to make it a valid recoverer within the new framework. In order to be a valid recoverer to the abstraction layer, either the `Recoverer` or the `Repairer` interface, described in Section 4.1.5, must be implemented. In the case of `Lattice`, it must be able to repair blocks, so the `Repairer` interface is implemented.

In the original implementation, the Swarm backend implementation directly called the `GetChunk` and `RepairChunk` methods of the `Lattice` type. These methods then retrieved or repaired the requested data block and update the internal state of the `Lattice`. A goal when adapting the `Lattice` to using the new repairer abstraction is to require the least amount of changes to internal logic, as it is rather complex. The adaptation involves two steps.

The first step is implementing the `RepairBlock` method, which only differs from the original `RepairChunk` method in that `RepairBlock` additionally accepts a `context.Context` parameter. As such, the implementation of `RepairBlock` simply wraps around the original `RepairChunk` method without changing it. In addition to executing the original function, it respects the `context.Context` parameter, which the requester might use for cancellation or timeouts.

The second step is to accept recovery requests and use `Lattice` methods to retrieve and repair blocks. For this, a recovery request handler process (recoverer) was added to the `Lattice`. The pseudocode for the recoverer is displayed in Algorithm 2. When initialized, the `Lattice` now sets up a channel for recovery requests, and starts in a separate goroutine a process which handles incoming requests. The algorithm repeatedly receives recovery requests from the channel c, as long as it remains open. Each request is handled in a separate goroutine. There are two scenarios for block retrieval: direct retrieval, and retrieval using the repair algorithm, described in the paragraphs below. An additional change is that raw blocks of data also need to be deserialized according to the abstraction layer, which occurs in the final step of the algorithm.

Direct retrieval This occurs (1) when $r_{Index} = 0$, i.e. when the caller explicitly indicates the repair algorithm should not be used, or (2) when $r_{IsParity}$ is true, i.e. for nodes in a parity file DAG. The block identified by r_{ID} is retrieved directly from the backend with the `GetBlock` function provided by the backend DSS implementation. Scenario (2) is not related to data block requests from the external requester, but rather to parity block requests from another process started by the `Lattice`, which is described in Section 4.3.2.2.

Retrieval using the repair algorithm This occurs for data blocks which might need to be repaired. We attempt to retrieve the block from the backend using the `GetChunk` function, which wraps around the backend's `GetBlock` function, additionally updating the `Lattice` state. If the block retrieval fails, we need to repair the block, which is performed using the `RepairBlock` function. While data blocks are always repaired with a pp-tuple,

Algorithm 2 Pseudocode for the recovery request handler of the `Lattice` type.

```
while open(c) do
    r ← receive(c)
    if r_IsParity ∨ r_Index = 0 then                                ▷ direct retrieval
        block, err ← GetBlock(r_ID)
    else                                         ▷ retrieval using the repair algorithm
        block, err ← GetChunk(r_ID, r_Index)
        if err ≠ nil then
            block, err ← RepairBlock(r_Index)
            missing ← missing blocks       ▷ missing data blocks for repair of b, if any
            if err ≠ nil ∧ |missing| > 0 then
                for all b ← pop(missing) do           ▷ iterate over each missing block
                    if DownloadFailed(b) ∧ RepairFailed(b) then
                        continue           ▷ skip blocks failing to download and repair
                    await update                 ▷ await status change on data block b
                    if HasData(dblock) then      ▷ check if b has data following update
                        block, err ← RepairBlock(r_Index)
                        if err = nil then
                            break
    if err ≠ nil then
        Respond(nil, err)
    else
        Respond(Deserialize(block, r_ID, r_Index, r_IsParity))
```

the pp-tuple might contain parity blocks that need to be repaired as well, which requires other data blocks to be present. As such, in order to repair a data block, we might need to wait for the presence of another data block. When this occurs, an error from `RepairBlock` contains indices of other data blocks that are necessary to repair the current data block. Unless the prerequisite data blocks are known to have failed both download and repair, we await updates on these data blocks, and reattempt the repair.

4.3.2 Updated Parity Block Retrieval Approach

This section describes the new approach to parity block retrieval. The new implementation moves most of the logic out of each backend and into the `Lattice` type. It also uses a CSP-approach to concurrency based on a single coordinator process that manages parity node state, starts other concurrent processes for slow work, and coordinates communication between the processes. The remainder of this section describes the motivation to implement a new approach, and interesting details regarding the new approach.

4.3.2.1 Motivation

There are certain changes to parity block retrieval and repair, to adapt to the abstraction layer API. The differences between the original approach and the new approach are described in Section 3.2.7.1. Originally, the `Lattice` method `GetParity` requested parity blocks by sending retrieval requests to the Swarm backend's getter, providing the identifier of the corresponding parity file DAG root node, and using the `context.Context` field of the getter function to specify the index of the requested parity block. The `Context.WithValue` method [22] was used to provide the parity block index as a value along with the context. The Swarm backend checked whether such a key-value pair was present, and determined the request was for a data block if the key-value pair was absent, or for a parity block otherwise. The original approach has a few drawbacks:

- The use of `context.Context` to carry a key-value pair containing the parity index is not idiomatic Go code. The documentation for the `context` package states the following: *"Use context Values only for request-scoped data that transits processes and APIs, not for passing optional parameters to functions"* [22]. Our interpretation of this is that the context should not be the primary way to provide the parity block index parameter, nor to determine if a request is for a data block or a parity block. The purpose of the context in this case is to enable the requester to cancel the work being executed by the recoverer.

- As mentioned in Section 3.2.7.1, the backend implementation must do most of the work with the original approach. When provided the index i, it must determine how to access the i-th parity block. It also has to come up with an approach to temporarily storing parity nodes that might be needed for subsequent requests. This logic must be duplicated for each backend implementation.

4.3.2.2 Updated Parity Block Getter

The updated approach is based on the *next path* operation described in Section 3.2.3.2. Recall that the algorithm works by traversing a branch of the parity file DAG until reaching a leaf node, which is the requested parity block. Nodes from each backend implement the `NextPath` method described in Section 4.1.1, which simply produces links and indices to the next node in the path to a requested leaf node. To use that algorithm, the repairer must implement another algorithm which repeatedly retrieves parity nodes, and calls `NextPath` to get the reference to the subsequent node, until it retrieves the desired parity block.

Challenges Implementing an algorithm for recovering parity blocks requires solving a few challenges:

- The original `Lattice` implementation had no knowledge of internal nodes in the parity file DAGs; it was only concerned with the leaf-nodes, which are the parity blocks. The updated solution needs a way to retrieve, store and address all parity nodes, not just the leaf nodes. Additionally, it must ensure that the same parity node is not downloaded twice from a backend DSS.

- The existing `Lattice` methods achieve synchronization by complicated use of locks, making it hard to safely introduce new logic. Ideally, the new solution should integrate with the existing methods with the least amount of changes, e.g. by creating an independent process.

- The new solution must maximize concurrency. We consider downloading data from the backend DSS to be the slowest part of Snarl. Thus, to minimize retrieval times, the solution should maximize the number of parity nodes it downloads concurrently.

Overview of updated solution A high-level overview of the updated repair process is given in Figure 4.4. In the figure, the communication between the *Requester*, `NodeGetter` implementation, *Recoverer*, and `Lattice` type are those described in Section 4.3.1. The algorithm for recovery of parity blocks is implemented in the *ParityRecoverer* process in the illustration, which we call r. The process is responsible for identifying, retrieving and storing all parity nodes necessary for the repair algorithm. The `Lattice` type sends requests for parity blocks to r, identifying the parity blocks by type of strand and n for the n-th leaf parity node.

If the parity block is already stored by r it immediately responds with the block. Otherwise, the parity block must be recovered by r, and the response to the parity block request is sent once r has the result. To recover the n-th parity block, r starts another process p_n,

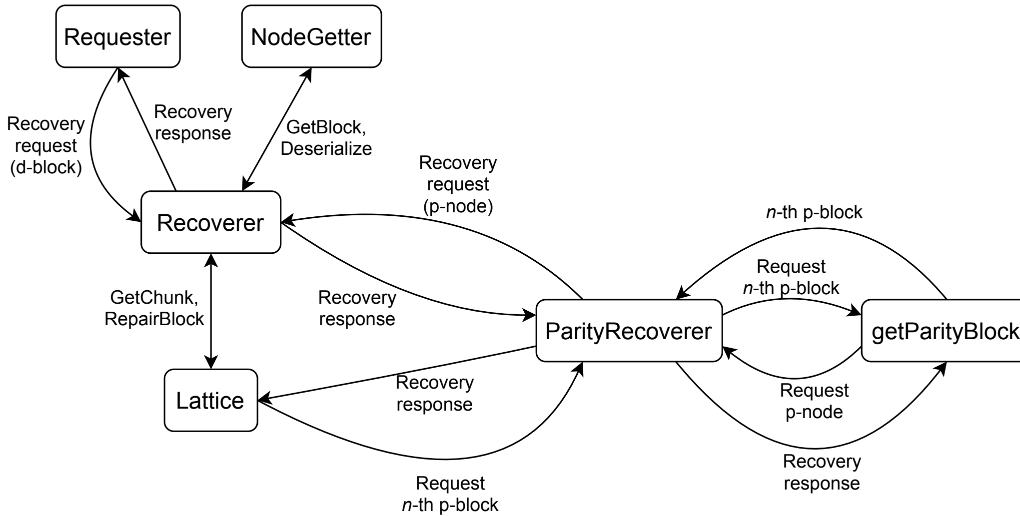

Figure 4.4: High-level overview of processes in updated repairer. Bidirectional edges denote function calls, while unidirectional edges denote channel messages.

which concurrently runs the `getParityBlock` function, to traverse a branch of the parity file DAG until it finds the requested parity block. Several such processes p_i may be running at any time, each responsible for finding a distinct parity block. The i-th parity block will only ever be retrieved by a single process p_i.

While traversing the branch of the DAG, p_n needs all parity nodes in the branch in order to call `NextPath` to determine which node is next. Since parity nodes are only stored in r, p_n sends requests for each parity node it requires to r. If the parity node is already in r's storage, it immediately responds with it. Otherwise r concurrently requests it to be recovered by the recoverer process from Section 4.3.1, and responds to p_n when it receives a response from the recoverer.

Preventing duplicate recovery processes When r communicates with several concurrent processes, there may be several requests for the same parity node. r must prevent starting duplicate recovery processes for the same node, or for nodes it has already failed to retrieve. It handles this with recovery statuses and subscriber lists. By combining recovery statuses and subscriber lists, r is ensured to not start a duplicate recovery process, and not to request the same node twice.

Whenever r receives a request or response related to a node, it sets the *recovery status* for the node. Requests for nodes r has not seen requests for previously cause the recovery status of the nodes to be set to pending. When r gets a retrieval result, it updates the recovery status of the node. On successful node retrieval, the pending recovery status is unset. Upon retrieval failure, r marks the node as failed. In this way, if r receives another request for the same node before the result is known, it does not start a duplicate process to recover the same node. If r receives a request for a failed node, it immediately responds that the node is irretrievable.

The concept of subscriber lists is also used to prevent duplicate work. Whenever r receives a request for a node without a known result, it adds the request as a subscriber. Later, when r knows the result of the node retrieval, it sends the result to all relevant subscribers at the same time, and removes them from the subscriber list. This indirect approach of responding to requests lets r coordinate other work simultaneously, without blocking while waiting for retrieval.

Handling requests and recovery responses As illustrated in Figure 4.4, the *ParityRecoverer* process r must handle several requests and responses. r is responsible for

coordinating recovery requests, storing parity nodes, and announcing results to any relevant subscribers. There are two types of requests sent to r: (1) external requests are sent from the `Lattice` type to request a parity block; (2) internal requests, requesting a parity node in the branch to a leaf parity node, are sent from a process p_n running the `getParityBlock` function. Additionally, two types of recovery responses sent to r: (1) external responses from the recoverer in response to requests for retrieval of parity nodes; (2) internal responses from a process p_n, when it finds the n-th parity block, or if an error occurs in the algorithm. To handle all of these cases, r uses a select-case[3] on three channels:

1. *External requests:* The channel for external requests receives parity block requests from the `Lattice` type. During initialization of r, this channel is provided. When r receives requests on this channel, it immediately responds with the parity block if it already stores it, or an error if the same request has failed previously. Otherwise, it sets the recovery status of the parity block to pending, registers the request as a subscriber, and concurrently starts a process p_n to recover the parity block.

2. *Internal requests:* The channel for internal requests receives requests for internal and leaf parity nodes from processes running the `getParityNode` function. Requests for nodes with known results are immediately responded to. Otherwise, r sets the recovery status of the node to pending, registers the request as a subscriber, and concurrently starts a process to request the node from the external recoverer.

3. *Recovery responses:* Responses to recovery requests from both the external recoverer and processes running `getParityBlock` arrive on the same channel. r determines which the type of sender by looking at a flag in the recovery response. The process updates the recovery status of the node, either marking it as failed if the recovery response contains an error, or unsetting the pending recovery status if it is successful. Furthermore it finds subscribers waiting for the result, removes them from the subscriber lists, and sends the result to each relevant subscriber.

By using a single coordinator process r, concurrency is simplified. No locks are necessary, since all parity nodes are only stored in r, and r only handles a single request or response at any given time. r coordinates other concurrent processes, but the processes only share memory by communicating, such that there cannot be any data races. It is simple to implement the other processes that communicate with r, since they only need to communicate with r over a request-response channel pair. r should not be a significant bottleneck, since r coordinates slow work such as node retrieval to other concurrent processes, meaning it can respond to requests quickly.

The getParityNode function Pseudocode for the `getParityNode` function is displayed in Algorithm 2. The function takes three parameters: a parity block request req, a channel to send parity node requests $c_{request}$, and a channel to send its result to c_{result}. As mentioned, the algorithm retrieves and calls `NextPath` on every parity node in the branch leading to the requested parity block. First the root node for the type of strand is requested. Subsequently, nodes referenced by the result of calls to `NextPath` are requested. All nodes node requests are sent to the parity recoverer process r through the $c_{request}$ channel, which either already stores, or retrieves the nodes. Upon the first error or leaf parity node, the function terminates, sending the result to r through the c_{result} channel. The function is simple, leaving parity node retrieval and result announcements to the parent process r.

[3] The `select` statement executes the "statement list" of the first selected `case` statement, similarly to a switch-case, but for communication operations [32]. For example, we may attempt to receive from two channels and send to another channel; the first case that succeeds will execute the related statement list.

Algorithm 3 Pseudocode for the `getParityBlock` function used by the parity block getter.

```
function getParityBlock(req, c_request, c_result)
    strand ← req_strand                                  ▷ type of strand of requested p-block
    node, err ← request(nil, 0, strand)                  ▷ request root parity node
    if err ≠ nil then
        respond(nil, err)
        return
    prev ← node_ID
    loop
        link, index, err ← node.NextPath(req_n)          ▷ next path to n-th leaf node
        if err ≠ nil then
            respond(nil, err)
            return
        node, err ← request(link, index, strand)
        if err ≠ nil then
            respond(nil, err)
            return
        if node_IsLeaf then                              ▷ parity block found
            respond(node, nil)
            return
        if link = prev then                              ▷ cycle in NextPath implementation
            respond(nil, error)
            return
        prev = link
```

4.3.3 Updated Constructor Function

Listing 4.7: Header for the `Lattice` constructor.

```
func NewLattice(ctx context.Context, conf config.Config,
                getter backend.NodeGetter, dataRootID []byte,
                parityRootIDs [][]byte, meta []backend.BlockMetadata)
                (*Lattice, error)
```

The constructor function for `Lattice` is updated. The original constructor function did not suffice for a backend-independent implementation based on the abstraction layer. Instead, it depended on specifics of the Swarm backend implementation. The header for the updated constructor function is provided in Listing 4.7. Some notable parameters:

- `conf` provides the configuration the `Lattice` uses, This includes the AE parameters α, s and p. Additionally it includes the backend configuration, where the backend type and block size are of interest, due to some edge cases where `Lattice` handles the Swarm backend differently when decoding blocks.

- `getter` provides a backend implementation of the `NodeGetter` interface from the abstraction layer, as described in Section 4.1.4. This enables Snarl to retrieve and deserialize blocks regardless of the backend being used. The *NodeGetter* process in Figure 4.4 represents the instance provided here.

- `dataRootID` and `parityRootIDs` provide the identifiers of the root data block and root nodes of the parity file DAGs, used for data block retrieval and parity block retrieval. End users must be able to provide these identifiers to perform recovery and repair.

- `meta` provides the block metadata. It contains the DAG layout and size of each

block, necessary for initialization of the `Lattice` data structure. Previously, this was not provided as a parameter to the constructor, but performed in the `Lattice` initialization based on the file size. Providing the metadata as an argument lets backend implementers produce the metadata correspondingly.

Based on the provided parameters, the `Lattice` is initialized. As was the case in the original implementation, the constructor sets up placeholder instances of Snarl's `Block` type for each data block and parity block. As block retrieval and repair occurs, Snarl updates the state of these blocks.

Furthermore, two updated constructor starts two processes in separate goroutines. The first is the recoverer described in Section 4.3.1, to handle external recovery requests for data blocks, and internal recovery requests for parity nodes. The second is the parity block getter described in Section 4.3.2, to recover parity blocks for the repair algorithm.

4.4 IPFS Backend

In this section we look at implementation details of the IPFS backend. This backend implements the abstraction layer described in Section 4.1 using code built on top of exported APIs and types of IPFS. Only the parts considered most important or interesting are described. This includes the `Node` and `NodeGetter` interface implementations, generation of flattened Merkle DAGs, and an IPFS proxy type used to communicate with the DSS.

4.4.1 Node Implementation

In this section we look at the implementation of the `Node` interface from Section 4.1.1 for the IPFS backend.

4.4.1.1 The Node Type

Listing 4.8: `Node` interface implementation for IPFS.

```
type Node struct {
  ipld.Node
  mDag ipld.DAGService
  cid []byte
  index uint
  indices *indexmap.IndexMap
  isLeaf bool
}
```

The `Node` type from the `snarl/ipfs` package is displayed in Listing 4.8. It is a thin wrapper around a type used by IPFS internally, the `ipld.Node` interface. Additionally it contains a few extra fields used for the implementation of the Snarl interface.

- The `Node` and `mDag` fields are IPFS data structures. As mentioned, the `ipld.Node` interface, which the `Node` field is an instance of, is used internally by IPFS. Among other things it provides the methods `RawData` for the raw data stored by a Merkle DAG node, `Cid` for the content identifier of the node, `Links` for links to child nodes of internal nodes and `Stat` for metadata such as block size and cumulative size [14, 7]. These methods are used for the implementation of the Snarl `Node` interface. The `mDag` field implements the `ipld.DAGService` interface. Among other things, it provides a method `Get` retrieves IPFS nodes identified by CID [15]. This is used internally by the `FlatDAGConstructor` implementation.
- The `cid` field stores the CID of the node, which is used by the `ID` implementation.

- The `index` field contains the canonical index of the node. It is used by both `Walk` and `NextPath` implementations for determining a node's index as well as the indices of its children.
- The `indices` field contains an index mapping as described in Section 4.2. It is used to determine indices of child nodes and to find paths to leaf nodes.
- The `isLeaf` field is true for leaf nodes, and false for internal nodes. A call to the `IsLeaf` method returns this value. This is necessary for the `Lattice` to terminate the *next path* operation (see Section 4.3.2). Additionally, when a complete flattened DAG has been recovered, the recovery algorithm uses only data from leaf nodes to reconstruct the original file.

4.4.1.2 Walk Implementation

As part of the `Node` interface, there is an IPFS implementation of the `Walk` method specified in Section 3.2.3.1. The implementation consists of two parts. The first is the exposed `Walk` method, which can be seen as the entry point to the algorithm. The second part is an inner, unexported `walk` method, which takes an additional parameter for cancellation. The unexported `walk` method recursively retrieves child nodes as specified by the *walk* operation.

Listing 4.9: `Walk` implementation and header for unexported `walk` method for the IPFS backend `Node` type.

```
func (n *Node) Walk(ctx context.Context,
                     recoverer chan<- backend.RecoveryRequest)
                    (flatDAG backend.FlatDAG, err error) {
  if recoverer == nil {
    return nil, backend.ErrNilRecoverer
  }
  cancelCtx, cancel := context.WithCancel(ctx)
  defer cancel()
  return n.walk(cancelCtx, cancel, recoverer)
}

func (n *Node) walk(ctx context.Context, cancel context.CancelFunc,
                     recoverer chan<- backend.RecoveryRequest)
                    (flatDAG backend.FlatDAG, finalErr error)
```

Listing 4.9 displays the implementation of the exported `Walk` method and the header of the unexported `walk` method. The following paragraphs describe the reasoning for the split implementation and the concurrent algorithm implemented by the unexported `walk` method.

Exported Walk method The implementation of the exported `Walk` method, displayed in Listing 4.9, is a minimal wrapper around the internal `walk` method. It is the entry point to the algorithm, which is called externally, e.g. by the constructor and recoverer. The wrapper wraps the input context variable `ctx` with the `context.WithCancel` function, which provides an associated `context.Context` instance and a cancellation function [21]. The `cancelCtx` context and `cancel` function are given as inputs to each concurrent call to the internal `walk` method. If an error occurs in any concurrent call to the `walk` method, the cancellation function can be called to signal to the other goroutines, through the `cancelCtx` context, that an error has occurred and that they should abort their work as well. Additionally, a deferred call to `cancel` is executed by the `Walk` method after the call to `walk` terminates, which releases resources used by the context, as suggested by the Go documentation.

Internal walk method The internal `walk` method contains the logic for implementing the *walk* operation. It does the following:

- Set up a channel `done` to signal completion. Use Go's `select` statement to execute the first of two available branches:

  1. Receive a value on the aforementioned `done` channel. In this case the operations performed elsewhere in this call to `walk` is terminated and signals completion. We return the final results produced elsewhere in the method.
  2. Receive a value from the channel returned by `ctx.Done()`. In this case, the execution of the *walk* operation is canceled by any concurrent call to `walk`, possibly this call. It indicates that some node is irretrievable, or that some error occurred, such that the flattened DAG cannot be recovered in its entirety, so we abort any further computation and retrieval.

- Concurrently, we recover child nodes and produce the flattened DAG in canonical order. Any error in the following steps aborts the execution of all goroutines descended from the overlying call to `Walk`.

  – The internal function `getChildren` concurrently sends recovery requests for each child node of a node. Using a `select` statement, the function either returns the results of the recovery requests if it receives a value on its internal `done` channel, or aborts execution if `cancel` has been called by another goroutine, signalled by receiving a value on the channel returned by `ctx.Done()`. If the recoverer responds to any recovery request with an error, this function aborts further execution by sending to its `done` channel, returning the related error.

  – If the child nodes were retrieved without error, we perform a recursive, concurrent call to `walk` on each child node. Each child node recovers and returns its subtree flattened in the canonical order. The result of each call to `walk`, called a subresult, is sent to a channel `subResults`.

  – Another goroutine handles subresults received on the `subResults` channel. It catenates the subresults in the correct order, finally returning the flattened DAG in the canonical order, and signalling the completion of the algorithm.

The implementation described above performs most work concurrently. Using a CSP approach which performs work concurrently using goroutines, and communicating by channels, enables highly concurrent code. Concurrent calls to `walk` are made on each child node, which in turn does the same to its child nodes as soon as possible. As a result, the recoverer is often handling multiple recovery requests simultaneously, which decreases the total recovery time. Additionally, the shared context and cancellation function enables us to abort and prevent unnecessary work when the algorithm is known to fail. Since the context is also attached to recovery requests, the recoverer also aborts its work when the `cancel` function is called elsewhere.

4.4.1.3 NextPath Implementation

The `NextPath` method is also implemented as part of the `Node` interface. It implements the *next path* operation described in Section 3.2.3.2, which is used to find links to leaf nodes, used by the repair algorithm to recover parity blocks.

Listing 4.10: `NextPath` implementation for the IPFS backend `Node` type.

```
func (n *Node) NextPath(index uint) ([]byte, uint, error) {
  index = n.indices.LeafIndex(index)
  if index == n.index {
    return n.ID(), index, nil
  }
  if n.index < index {
    return nil, 0, errNoPathToLeaf
  }
  childIndices, err := n.indices.Get(n.index)
  if err != nil {
    return nil, 0, err
  }
  for i, childIndex := range childIndices {
    if childIndex >= index {
      links := n.node.Links()
      if len(links) > i {
        return links[i].Cid.Bytes(), childIndex, nil
      }
      break
    }
  }
  return nil, 0, errNoPathToLeaf
}
```

The implementation of `NextPath` is provided in Listing 4.10. First, the index of the requested leaf node is translated using the index map, such that if the input $index = i$, the translated index j is the index of the i-th leaf node in the Merkle DAG. In canonical ordering, child indices are always lesser than their parent's index, which is checked by the second if-clause. Furthermore we get indices of each child node of the present node n as a sorted list, and iterate over them. The first child node whose index is greater than or equal to the index of the leaf node we are looking for is either a parent to the leaf node (if index is greater), or the leaf node itself (if index is equal). We return the link to this child node, which is the next child node in the path to the requested leaf node.

In the Merkle DAG there are many nodes whose index exceeds the index of the requested leaf node. Therefore, the algorithm may not perform correctly if `NextPath` is called on a node that is not in the path to the leaf node. However, as long as requester calls `NextPath` on the root node first, and then on each linked node, the algorithm will perform correctly.

The most important property of this implementation is its simplicity. There are few steps involved in discovering the path to the leaf node, and it is straightforward to understand how the algorithm works and in which cases it might fail. As discussed in Section 3.2.7.1, the majority of the logic resides in the recoverer.

4.4.2 Node Getter

The `NodeGetter` type in the `snarl/ipfs` package implements the `NodeGetter` interface[4]. It is used to retrieve raw, serialized blocks of data from IPFS, and to deserialize that data into `Node` instances. The following paragraphs describe the `NodeGetter` type and its two constructor functions.

[4]In this section, we let "the `NodeGetter` interface" denote the interface belonging to the abstraction layer, as specified in Section 4.1.4, and "the `NodeGetter` type" denote the type in the `snarl/ipfs` package which implements the aforementioned `NodeGetter` interface.

Listing 4.11: The IPFS backend `NodeGetter` type, related constructors, and the `Deserialize` method header.

```
type NodeGetter struct {
  mDag                        ipld.DAGService
  dataIndices, parityIndices *indexmap.IndexMap
  backend.BlockGetter
}

func NewNodeGetter(mDag ipld.DAGService, fileSize, pFileSize uint,
                   getter backend.BlockGetter, conf config.BackendConfig)
                  (*NodeGetter, error)

func NewConstructingNodeGetter(mDag ipld.DAGService,
                                getter backend.BlockGetter)
                               *NodeGetter

func (g *NodeGetter) Deserialize(rawNode []byte, id []byte,
                                  index uint, isParity bool)
                                 (backend.Node, error)
```

The NodeGetter type The type, the headers for related constructor functions, and the header for the `Deserialize` method are displayed in Listing 4.11. The fields in the struct have the following purpose:

- The `mDag` field is provided through the constructor functions and used to set the `mDag` field of `Node` instances during deserialization.
- The `dataIndices` and `parityIndices` fields contain index maps for the data block Merkle DAG and parity file Merkle DAGs. During deserialization, the deserialized `Node` instance is initialized with one of the index maps, depending of the deserialized node is a data node or parity node. This is necessary due to the different layouts of Merkle DAGs containing data blocks and parity blocks. Recall that nodes use the index map in the `Walk` and `NextPath` methods.
- The field `backend.BlockGetter` uses Go's type composition to embed any type implementing the `BlockGetter` interface, causing the `NodeGetter` type to implement this interface as well. The advantage of embedding the `BlockGetter` interface here is that any implementation of it can be used, e.g. we could use an implementation that downloads blocks from IPFS proper, or one that retrieves blocks from local storage, or even one that is programmed to fail in certain situations for testing purposes.

Constructor functions Furthermore, we have two constructors for the `NodeGetter` type. The `NewNodeGetter` function sets up a node getter to be used in the recovery process. It generates arbitrary bytes of data for the data file based on the `fileSize` parameter, and the same for the parity files based on the `pFileSize` parameter. Based on the layout configuration in the `conf` parameter and the aforementioned bytes of data, it then generates index maps for the data file and the parity files by generating two IPFS Merkle DAGs. Knowing the DAG layouts is necessary for node deserialization. The `pFileSize` field may be set to zero, in which case no index map is generated for the parity file Merkle DAGs, such that parity nodes cannot be deserialized. This may be desirable if recovery is to be used with the repair algorithm disabled.

The other constructor function, `NewConstructingNodeGetter`, is used by the flattened DAG constructor described in Section 4.4.3. It either uses the provided `ipld.DAGService`, or generates an in-memory version if the argument is nil. The index mappings are empty for this version, since they are not necessary for the flattened DAG constructor.

4.4.3 Flattened Merkle DAG Constructor

The `FlatDAGConstructor` in the `snarl/ipfs` package implements the `FlatDAGConstructor` interface. It is used to produce a flattened Merkle DAG in the canonical order from the contents of an input reader. The following paragraphs describe the `FlatDAGConstructor` type implementing the interface of the same name, and the implementation of the `Construct` method.

Listing 4.12: Type, constructor and method headers for the `FlatDAGConstructor` implementation the IPFS backend.

```
type FlatDAGConstructor struct {
  BS   blockstore.Blockstore
  Conf config.BackendConfig
}

func NewConstructor(bs blockstore.Blockstore, conf config.BackendConfig)
                    FlatDAGConstructor

func (c FlatDAGConstructor) Construct(ctx context.Context, r io.ReadSeeker)
                                      (flatDAG backend.FlatDAG, err error)
```

The FlatDAGConstructor type The `FlatDAGConstructor` type, its constructor, and the header for the `Construct` method are displayed in Listing 4.12. The type contains a blockstore which it uses to store any IPFS blocks from the Merkle DAG generated from the input reader. The blockstore may optionally be provided in the constructor for the type, e.g. to store the blocks permanently, or for testing purposes. If the blockstore is not provided, an in-memory blockstore is used instead. The `Conf` field is used to determine the Merkle DAG layout.

The Construct method The implementation of the `Construct` method is based on a combination of IPFS exported APIs and the Snarl abstraction layer. It generates a Merkle DAG from the input reader using exported APIs from IPFS. The Merkle DAG uses the UnixFS format with raw leaf nodes, as described in Section 2.2.3.5. From the generated Merkle DAG it generates an index map and sets up a `MerkleDAGBlockGetter` g, which implements the `BlockGetter` interface, and can retrieve IPFS blocks from the Merkle DAG by CID. Using g as the block getter, it sets up a `NodeGetter` ng which can additionally deserialize blocks into `Node` instances. Furthermore it runs a simple recoverer r in a separate goroutine, which accepts recovery requests and calls `GetBlock` and `Deserialize` on ng to produce nodes. Finally, `Walk` is called on the root node, providing the recovery request channel to r. It produces the flattened Merkle DAG in canonical order.

4.4.4 The Proxy Type

The `Proxy` type is a proxy for communicating with the IPFS backend. The type is used for the portions that require communication with other IPFS peers, namely uploading blocks or files, and retrieving nodes. During initialization, the proxy sets up a connection to the IPFS daemon using an instance of the `Shell` type from the `go-ipfs-api/shell` package [13]. This type establishes a connection to a locally running IPFS daemon which it communicates with through the IPFS HTTP API [11].

Using the shell, the proxy communicates with the IPFS daemon to upload and download blocks. The `FileUploader` and `BlockUploader` interfaces are implemented as the `uploader` type, and the `BlockGetter` interface is implemented as the `BlockGetter` type. Each method uses the `Shell` instance provided by the `Proxy` type to upload or download blocks from IPFS, thus indirectly using the IPFS HTTP API. The `Proxy` type embeds both types such that it implements the interfaces as well.

Furthermore, the `Proxy` can produce an instance of the `NodeGetter` interface. This instance uses the `Proxy` to retrieve serialized data from IPFS, and uses the `Deserializer` method implementation described in Section 4.4.2 for node deserialization. This can be used by the `Lattice` type during file recovery and repair.

4.5 Updates to the Swarm Backend

In this section we look at changes made to the Swarm backend for Snarl, which resides in the `snarl/swarmconnector` package. We make certain changes to adapt the Swarm backend to the abstraction layer. Large parts of the abstraction layer are inspired by the operation of the original Swarm backend, aiming to fix perceived shortcomings and make it applicable to other backends as well. As such, most of the Swarm backend only required minor adaptations, and only a few cases demanded entirely new logic. In the remainder of this section we go through some notable changes made to the Swarm backend for Snarl as part of this thesis.

4.5.1 Node Implementation

In this section we look at the implementation of the `Node` interface from Section 4.1.1 for the Swarm backend.

4.5.1.1 The TreeChunk Type

Listing 4.13: `TreeChunk` type from the Swarm backend.

```
type TreeChunk struct {
  Depth       int
  BranchCount int64
  Length      int
  SubtreeSize uint64
  Data        []byte
  Key         []byte
  Index       int
  Children    []*TreeChunk
  Parent      *TreeChunk
  // new fields
  opt     *BuildTreeOptions
  indices *indexmap.IndexMap
}
```

The original Swarm backend implementation used the type `TreeChunk` to represent Merkle tree nodes. The updated definition for the type is displayed in Listing 4.13. Observe that the updated version is slightly extended, adding the options field `opt`, which controls how the `Walk` implementation behaves, and the index map field `indices`, which is used by the `NextPath` implementation.

4.5.1.2 Walk Implementation

The `Walk` implementation for the `TreeChunk` type is adapted from a similar method which existed in the original Swarm backend. Originally, the Swarm backend used a function `BuildCompleteTree`. This function recovered the root node, and then traversed the tree using an internal method `walkTreeChunk`, which recursively and concurrently recovered child nodes until the entire Merkle tree was recovered. These functions accepted as parameters, among other things, a getter to retrieve blocks of data from Swarm, and a repairer to repair irretrievable blocks.

Listing 4.14: `Walk` implementation and header for unexported `walk` method for the Swarm backend `TreeChunk` type.

```
func (n *TreeChunk) Walk(ctx context.Context,
                         recoverer chan<- backend.RecoveryRequest)
                        (flatDAG backend.FlatDAG, err error) {
  if recoverer == nil {
    return nil, backend.ErrNilRecoverer
  }
  cancelCtx, cancel := context.WithCancel(ctx)
  defer cancel()
  return n.walk(cancelCtx, cancel, 0, recoverer)
}

func (n *TreeChunk) walk(ctx context.Context, cancel context.CancelFunc,
                         parentOffset int,
                         recoverer chan<- backend.RecoveryRequest)
                        (flatDAG backend.FlatDAG, err error)
```

The updated implementation is split into two parts. The part which retrieves the root node and starts traversal of child nodes is moved to the implementation of the `FlatDAGConstructor` interface. This is described in further detail in Section 4.5.2.

The `Walk` method is adapted from `walkTreeChunk`. Similarly to the `Walk` implementation for the IPFS backend described in Section 4.4.1.2, the Swarm backend splits `Walk` into an exported method and an internal method accepting more arguments. The method implementation and the header for the internal `walk` method are displayed in Listing 4.14. The `cancel` argument to `walk` is used to abort execution if any error occurs in any concurrent call to `walk` descended from the call to `Walk`.

Following the abstraction layer API, an external recoverer is used, provided by the `recoverer` argument. This means that the backend implementation no longer needs to explicitly retrieve and repair blocks. It simply sends recovery requests and handles the responses it receives.

Furthermore, the code is adapted to use a CSP-style concurrency model. Each child node is recursively walked in a separate goroutine, each sending its result to a channel handled in the primary goroutine. `select` statements are used to select the first occurrence of a subresult from a child node or a canceled context. Subjectively, the CSP approach makes the concurrent code easier to understand.

4.5.1.3 NextPath Implementation

The `NextPath` method is implemented almost exactly like the IPFS implementation described in Section 4.4.1.3. The only difference is that the Swarm backend uses a different approach to obtain links to child nodes. Internal nodes in Swarm store as their data a sequence of CIDs linking to their child nodes, which are catenated into a single sequence of bytes. To obtain the CIDs we check the size of the node's data, then extract the CIDs from the data by extracting $|CID|$ bytes until reaching the end of the data sequence.

4.5.2 Entanglement and Recovery

The implementations of the interfaces used for entanglement and recovery are similar to the IPFS implementations described in Section 4.4.2 and Section 4.4.3.

Block getter The `BlockGetter` interface, used to recover blocks from the backend DSS, is implemented as a wrapper around existing getter methods. The original Swarm backend implements the `storage.Getter` interface, which includes the `Get` method [27]. It takes the same arguments and returns the same results as the `GetBlock` method from Snarl's `BlockGetter` interface. We simply wrap the function call to implement the new interface.

Node getter The `NodeGetter` type in the `snarl/swarmconnector` package implements the `NodeGetter` interface. As with the IPFS implementation described in Section 4.4.2, it embeds a type which can retrieve data from the backend DSS. Requests to get serialized blocks from the backend are handled through this embedded type. Other than that, the node getter contains index mappings for nodes of the data Merkle tree and parity Merkle trees, used for deserialization. The type is used both for flattened Merkle tree construction, described in the following paragraph, and for recovery.

Flattened Merkle tree constructor The `FlatDAGConstructor` type in the `snarl/swarmconnector` package implements the `FlatDAGConstructor` interface. The `Construct` method is implemented similarly to the IPFS implementation described in Section 4.4.3. It uses Swarm exported APIs to generate a Merkle tree from the contents of the input reader and generates an index mapping from the layout. Then it sets up a recovery request handler in a separate goroutine, recovers the root node, and finally calls the `Walk` method on the root node to generate the flattened Merkle tree.

Chapter 5

Evaluation

This chapter evaluates the performance of Snarl using the IPFS backend introduced in this thesis. It presents the results of several experiments with the IPFS backend in different configurations. In particular, the experiments vary in failure type, failure rate, and replication rate. The experiments are primarily focused on the file availability when using Snarl, compared to simply replicating the file. Additionally, network overhead is measured.

5.1 Experimentation Setup

This section describes the test setups used when running the experiments. The different simulation strategies are presented, as well as the procedure used to simulate different types of failure. Finally, the metrics gathered while running the simulations are presented.

5.1.1 Simulation strategies

This section describes the different simulation strategies used to conduct the experiments. The experiments are simulated in different ways depending on the scenario being tested. Node failures are simulated by running Snarl locally, while peer failures are tested using Docker containers.

Local testing When testing for file availability, we only need to check whether or not Snarl is able to recover a file given a certain shortage of nodes. This can be simulated simply by running Snarl locally, and artificially preventing it from accessing specific nodes. Compared to running multiple physical machines, or virtual machines in a simulated network, this saves a lot of time. Because of time constraints, this allows us to run significantly more tests than time would have allowed otherwise. The tests are described in greater detail in Section 5.2.1.

Docker containers In order to complement the local testing with some more realistic simulations, peer failure experiments are conducted in emulated networks using Docker containers. Both strategies simulate randomly distributed nodes, with similar node replication strategies. Given peer failure rates equal to the node failure rates used in local testing, both scenarios should give similar results. This is because they both amount to losing a random selection of nodes.

The peer failure tests were chosen for the Docker simulations, as they are simpler to simulate. Given the amount of time it takes to generate large amounts of results this way, only a small set of tests are performed. The Docker 20.10 series is used for the experiments. Two host machines are used, in the following configurations:

- The first machine runs Gentoo Linux with the Linux 5.10.37-gentoo kernel. It has an AMD Ryzen 9 3900X CPU with 12 cores and 24 threads, and 32 GB of memory.

- The second machine runs Arch Linux with the Linux 5.12.9-arch1-1 kernel. It has an Intel Core i7-8565U CPU, and 32 GB of memory.

5.1.2 Simulation procedure

This section describes the general procedure behind the simulations. It includes how nodes are generated and distributed, and how loss is simulated. The actual simulations use certain shortcuts to reduce runtimes, described in Section 5.2.

1. *Node generation:* A file with random content is generated, given a specific file size for the experiment. The file is entangled to produce three parity files. During the entanglement process, a flattened Merkle DAG containing all data blocks is generated. It also generates the flattened Merkle DAG for each parity file. Having access to all these nodes, the CID and size of each node is recorded.

2. *Distribution:* The test file and the three parity files are uploaded to the local IPFS daemon. Uploading the files will split them into IPFS nodes, the same nodes that were generated in the first step. Which nodes to duplicate, and how many replicas to create, is determined according to the replication strategy used. The nodes are then distributed to other IPFS peers by instructing certain peers to pin certain nodes. Since the nodes are available from the local IPFS daemon, the other IPFS peers can pin any node without issue. After distributing the nodes, the contents are deleted locally by unpinning the IPFS nodes and initiating the IPFS garbage collector.

3. *Simulating node loss:* Once the nodes are distributed, node loss is arranged according to the specification of the experiment. It is simulated by selecting a number of nodes that should be made unavailable, and instructing the relevant peers to delete those nodes. Deleting nodes involves unpinning the relevant CID, and running the garbage collector.

4. *Simulating peer failure:* Peer failure is simulated by choosing a random selection of servers, and telling them to stop their IPFS daemons. This ensures they are unavailable through the IPFS network.

5.1.3 Runtime Metrics

This section presents the metrics collected during the runtime of the experiments. To evaluate the performance of Snarl in various scenarios, the system must collect runtime metrics. To do this without modifying the Snarl codebase, we extend the implementation of the `BlockGetter` interface used by the IPFS backend's `Proxy` type described in Section 4.4.4. The block getter used in the experiments wrap around the original implementation, additionally storing a few metrics whenever a node is requested:

- *Node CID:* The CID of the requested node. This way, it is possible to determine exactly which nodes were requested while attempting to recover the file.

- *Retrieval result:* The number of bytes retrieved and the error message, if any. This way, it is possible to determine how many retrieval requests failed, and distinguish between retrieval times of successful downloads and failed downloads (which are canceled after a predefined timeout). It is also possible to determine the download overhead, i.e. the number of bytes that had to be downloaded to restore a file of n bytes.

The CID is used to determine the type of each node (data or parity), and their replication rate. By combining this data with the metrics collected by the block getter, we can determine how many downloaded blocks were data blocks or parity blocks, and the overhead of internal parity nodes.

At the end of each experiment, all collected metrics are written to a single JSON file. A script written in R processes the data after the experiments are finished.

5.2 File Availability

This section evaluates Snarl's performance with regards to file availability. To measure this, we inject failures into the "network" as described in Section 5.1.2, before attempting file recovery. If the file is recovered despite the failures it counts as a success, otherwise it counts as a failure. This is repeated for several trials. The file availability rate defined as

$$\frac{\#Successes}{\#Trials}, \tag{5.1}$$

where the number of successes and trials are recorded during a failure scenario with a distinct loss percentage. The file availability is measured for several node loss and peer failure scenarios with several trials for each.

5.2.1 Experiment: Node Loss

The node loss experiment measures the file availability of various configurations for different percentages of node loss. The experiments are conducted by running Snarl locally with a block getter implemented for the experiment, which only responds to requests for CIDs without registered node loss. The file size in each experiment is 100 MiB, and the results of 100 trials are considered for each node failure scenario. There are four configurations for node distribution:

1. *snarl5:* This configuration distributes an alpha-entangled file. The file is entangled by Snarl, and data and parity nodes are replicated until the total number of bytes reaches five times the size of the original data.
2. *snarl10:* Same as *snarl5*, but data and parity nodes are replicated up to ten times the storage.
3. *repl5:* This configuration produces the IPFS Merkle DAG representation of the input file, and replicates each node (data block) five times.
4. *repl10:* Same as *repl5*, but the nodes are replicated ten times.

Replication For the replication for *snarl5* and *snarl10*, the following algorithm is used. We keep a "pool" of CIDs, C, which is a sequence of CIDs where there may be duplicates. Initially, CIDs of all data and parity nodes are added. We alternately add CIDs of internal nodes and leaf nodes until the pool size reaches the desired replication rate. For each round, to add CIDs to the pool, we add all CIDs of the current type (internal or leaf node) to a list, shuffle the list, and iterate over it, adding CIDs one by one until the pool size is exceeded or until the list is empty. CIDs of internal nodes are added to the list twice before shuffling, i.e. they are duplicated twice as many times as CIDs of leaf nodes. This reduces the impact of cascading failures.

For *repl5* and *repl10*, the pool is simply filled with the CIDs of each data node, replicated five or ten times respectively.

Node loss After replication, we simulate the node loss. When the pool is filled, the order of all CIDs within it is randomized, to simulate randomized distribution of nodes. Let $0 < f < 1$ be the node loss ratio. We remove the first $f \cdot |C|$ entries from the pool, simulating the node loss. Finally, we remove duplicate entries, as they are unnecessary. The block getter only responds with nodes to requests with CID $c \in C$.

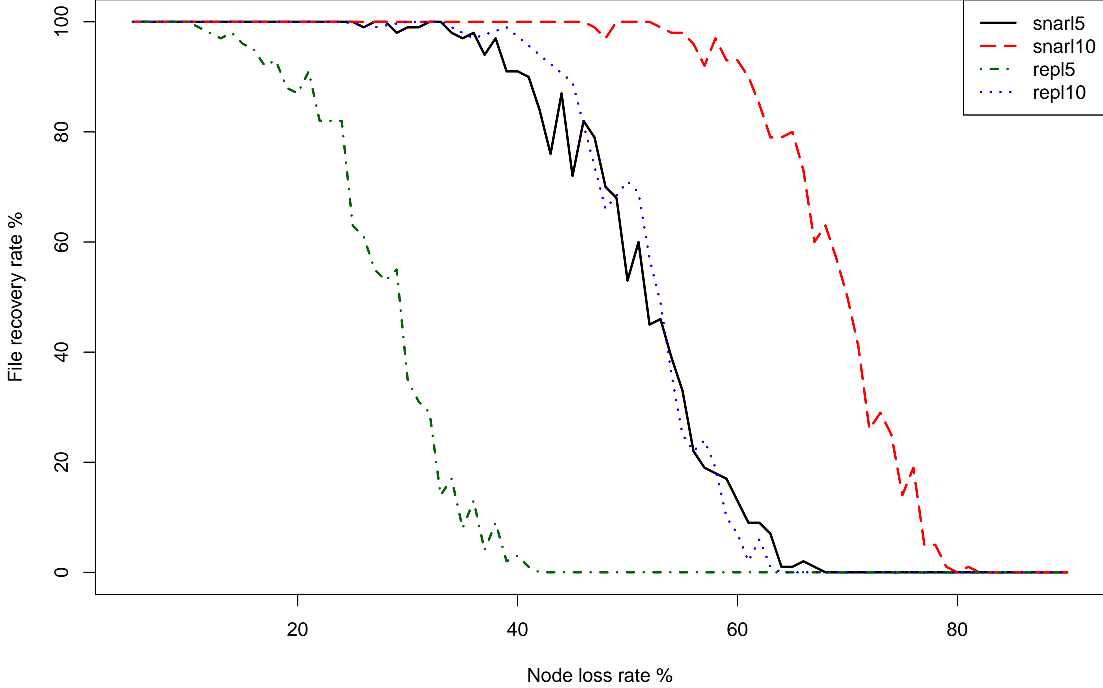

Figure 5.1: File recovery rate of 100 MiB files for node loss experiments, averaged over 100 trials.

Results and observations The results of the simulations are displayed in Figure 5.1. We can see that the recovery performance of *snarl5* is roughly equivalent to that of *repl10*, with only half the storage overhead. For *repl5*, the first failure was registered in the 11 % node loss scenario. In comparison, for *snarl5* the first failure was registered at 26 % node loss. For *snarl10*, the first failure was registered at 47 % node loss, while it is registered at 27 % node loss for *repl10*. The file availability of *snarl10* drops below 90 % between 60 and 70 % node loss rate, and has a non-zero file availability above 80 % node loss rate.

5.2.2 Experiment: Peer Failure

The peer failure experiment measures the file availability of Snarl for different percentages of peer failure, using the Docker setup described in Section 5.1.1. The node distribution and peer management is conducted using the PML. Peer failure is simulated by instructing a number of randomly selected peers to shut down their IPFS daemon, making the peer inaccessible to node requests. The experiments are conducted for the *snarl5* and *snarl10* replication scenarios, with 20 IPFS peers. The file size in each experiment is 50 MiB, and the results of 30 trials are considered for each peer failure scenario.

Replication Replication works similarly to Section 5.2.1, with minor differences. For the peer failure scenario, parity node CIDs are replicated $r \cdot 2^h$ times, with r being the replication rate, and h being the height of the node in the Merkle DAG, with the height of leaf nodes being zero. Additionally, data nodes are replicated evenly, with no extra duplication for internal nodes. The additional replication of higher parity nodes will further reduce the chance of cascading errors, as higher nodes will potentially make more nodes unavailable.

Node distribution After replicating the nodes, they are randomly distributed among each participating peer, ensuring every peer holds approximately the same amount of nodes. To achieve this, the list of replicated nodes is shuffled, and then treated as a queue. The nodes in the queue are always distributed to the peer with the lowest number of current nodes, using the first one found in the case of ties. The PML controller makes sure not to distribute the same node to a particular peer more than once by keeping an exclude-list. If a peer is found to already hold the CID currently being distributed, it will be added to the list and not be considered when re-choosing the peer.

Peer failure Peer failure is simulated by shutting down the IPFS daemons of random peers. For these experiments, peer failure is simulated in intervals of 10 %. For example, to simulate 60 % peer failure, the controller commands 60 % of peers to shut down their IPFS daemon. The tests are run using 20 peers, meaning $20 \cdot 0.6 = 12$ peers are instructed to shut down IPFS. In order to not waste time distributing nodes to peers before failing them, the peers are actually determined *before* node distribution. Whenever a node would normally be distributed to one of these peers, its distribution is skipped.

To find the most distinct data points with a minimal number of tests run, we chose to simulate loss percentages surrounding the slopes observed in Figure 5.1. For *snarl5*, the tests were run for 30 to 70 % loss, and for *snarl10* 40 to 80 % loss. However, the 80 % loss simulations encountered issues preventing usable results, possibly related to the deadlocking issue described in Section 6.4.

Table 5.1: Results from 30 trials of the peer failure experiments for 50 MiB files.

Peer failure %	30	40	50	60	70
Success (snarl5)	30	27	17	6	0
Fail (snarl5)	0	3	13	24	30
Rate (snarl5)	1.0	0.9	0.57	0.2	0.0
Success (snarl10)	N/A	30	30	28	17
Fail (snarl10)	N/A	0	0	2	13
Rate (snarl10)	N/A	1.0	1.0	0.93	0.57

Results and observations The results of the simulations are displayed in Table 5.1. We can see that the recovery performance is roughly the same as for the node loss scenario presented in Section 5.2.1. With 500 % replication (*snarl5*), the recovery rate doesn't fall below 90 % for up to 40 % peer failure, but decreases rapidly after this point. Using 1000 % replication (*snarl10*), the rate stays above 90 % for up to 60 % peer failure.

5.3 Network Overhead

This section evaluates network overheads of Snarl when using the repair algorithm to recover files from IPFS. When retrieving a file from the backend, Snarl invokes the repair algorithm whenever it fails to download a data block. At this point, it needs to use a pp-tuple (pair of parity blocks) to repair the data block. As such, it downloads parity blocks (leaf nodes) and internal parity nodes (needed to traverse the Merkle DAG) from IPFS. Furthermore, if any of these downloads fail, Snarl tries to recover them by recursively downloading and repairing other related, entangled blocks. The goal is to download the least number of parity blocks while being able to recover the contents of the requested file.

Definition and measurement approach To measure the network overhead, we measure the number of bytes downloaded by Snarl to recover the original file. The network overhead

is defined as

$$Network\,Overhead = \frac{Bytes\,Downloaded}{Size\,of\,File}, \tag{5.2}$$

such that we measure the percentage of excess bytes we needed to download to recover a file of n bytes. The size of the original file, e.g. 100 MiB, is considered, such that even if only data blocks are downloaded, there is a negligible overhead calculated by Equation 5.2. We measure the network overhead in the node loss experiments described in Section 5.2.1. The block getter stores the size of each node it successfully returns, where failed node requests are registered as zero bytes returned. In post-processing of the measurements, we calculate the network overhead for each successful trial in every failure scenario, and store the averages, variations, etc.

In addition to the network overhead, the type of each successfully downloaded block is recorded. Using this, the percentage of successfully downloaded blocks that are parity blocks is measured as

$$pnode\ \% = \frac{Parity\,Nodes\,Downloaded}{Nodes\,Downloaded}, \tag{5.3}$$

disregarding nodes of each type that fail to download.

Table 5.2: Network overhead measurements for node loss experiments in the *snarl5* configuration.

Node loss %	5	10	15	20	25	30	35	40	45	50
Mean overhead	1.039	1.085	1.137	1.19	1.249	1.307	1.36	1.411	1.436	1.442
σ	0.01	0.014	0.02	0.022	0.022	0.032	0.031	0.044	0.054	0.07
pnode %	8.5	16.5	24	30.7	37.4	43.4	48.6	54	58.9	63

Results and observations The network overhead measurements for the *snarl5* configuration in the node loss experiments are displayed in Table 5.2. We see that the network overhead increases linearly with the node loss percentage, and then flattens out around 45 % node loss. We also see that the percentage of downloaded blocks that are parity blocks increases along with the node loss percentage.

Chapter 6

Discussion and Insights

This chapter discusses the results of the evaluation presented in Chapter 5. It discusses the outcomes of the experiments and insights which the results provide. It also discusses some limitations of the experiments, and some problems that occurred while running some experiments.

6.1 File Availability Experiments

Node Loss Experiment Results In Section 5.2.1, we saw that with the same storage overhead, Snarl performs much better than simple replication. This is due to the fact that Snarl entangles data with AE codes, such that missing blocks of data can be repaired. Data entangled by Snarl, and replicated to 500 % of the size of the original file grants roughly the same file recovery rate as when the original data is replicated 1000 % without entanglement. In other words, by using AE codes to encode the data, we can either drastically reduce storage overheads while maintaining high reliability, or drastically increase reliability without increasing the storage overhead.

The observations for node loss are similar to the results presented by Nygaard et al in their paper introducing Snarl [19]. In that paper, the authors evaluate Snarl's performance using the Swarm backend. The chunk loss rate metric presented by the authors corresponds to the node loss metric evaluated in this thesis. With regards to file recovery rate, the IPFS backend performs similarly to the the results presented for the Swarm backend by Nygaard et al, as expected. The file recovery rate drops below 100 % in the interval between 35 % and 40 % node loss rate, which matches the results for the Swarm backend. However, it seems that the file recovery rate for the IPFS backend with a 100 MiB input file, in the *snarl5* scenario, goes towards zero slower than the Swarm backend does.

We believe this is due to the difference in block size and branching factor between the two backends. The Swarm backend uses a block size of 4 KiB, while the IPFS backend by default uses a block size of 256 KiB. Additionally, the Swarm backend uses a branching factor of 128, while the IPFS backend uses a branching factor of 174. As a result, there are fewer nodes in total in the IPFS backend, and additionally, the Merkle DAGs have fewer internal nodes, which means that cascading failures are less likely to occur.

Peer Failure Experiment Results Section 5.2.2 presented the results for the peer failure experiments. The number of trials per scenario was only 30, so the results are not very accurate. Nevertheless they seem to match with the results of the node loss experiments presented in Section 5.2.1. This is expected, since nodes are distributed evenly across the network, and as such, failing a percentage of peers should be equivalent to losing the same percentage of nodes. This result contrasts the result presented for the Swarm backend by Nygaard et al, where the file availability was significantly better in the peer failure scenario compared to the chunk loss (presented as node loss in this thesis) scenario [19]. We believe

this is because the nodes are not replicated evenly in the Swarm network, as displayed in Figure 6 of the referenced work.

Number of trials Ideally, more trials should be run for each experiment, in order to get more accurate results. Unfortunately, this was not possible due to time constraints. We do however believe that the results are relatively accurate, since they seem to match the results presented by Nygaard et al in the Snarl introductory paper [19].

Parameter Changes All the results presented are evaluated with the default configuration for the Snarl IPFS backend. In the default configuration, the block size is 256 KiB and the branching factor is 174. While the Snarl IPFS backend does not currently support alternative branching factors, it does support other block sizes. It would be interesting to determine how a different block size would affect file availability. Attempts were made to conduct the peer failure experiments with other block size configurations, however problems occurred when trying to pin the nodes at other peers. Unfortunately, due to time constraints, we could not track down the causes for these issues, and as such no results are presented here. We believe the problems were due to incorrect configuration of the IPFS daemon, and should be possible to fix. We have tested that the Snarl IPFS backend can entangle and recover content when using an alternative block size configuration, but only in local tests.

6.2 Distribution Strategies

It should be mentioned that the distribution strategy used in the node loss and peer failure experiments in Section 5.2 are not representative of the distribution strategy commonly used in IPFS. As mentioned in Section 2.2.3.3, data is distributed in IPFS through peers pinning the data. Each peer that has pinned a piece of data can provide it to other peers upon request. While we do not know for sure, we assume most pins in IPFS recursively pin the root node of files, thereby indirectly pinning its subtree. Meanwhile the node distribution employed in the experiments in this thesis distributes nodes evenly across the network. In the peer failure experiments, each peer directly pins a subset of nodes. The node failure experiments operate on the assumption that every node is distributed evenly across the entire network. In practice, it would be unlikely for large number of peers to pin individual nodes from the DAG of a file, such that the nodes in the DAG are spread evenly.

There are a few scenarios where we imagine this distribution could occur. Firstly, with proper incentive mechanisms, peers could be coordinated to pin nodes in a way that distributes data as much as possible. Furthermore, this could occur in private networks which control each IPFS peer, or in networked applications which use IPFS as their storage layer.

There are other possible distribution strategies that could be tested as well. The reason for considering other distribution strategies is that the utility of AE codes for file availability depends on how nodes are distributed in the network, and it would be interesting to see how well Snarl performs in those situations. Experiments for two other approaches were planned, but could not be completed due to time constraints. The following paragraphs describe the two alternative approaches that were considered.

File distribution We refer to the first alternative distribution strategy as file distribution. In this distribution, participating peers pin all nodes in a file, which we assume is how most users of IPFS behave. For experiments with this distribution, experiments could evaluate availability with different numbers of replicas, and by distributing parity files to separate peers. However, it is difficult to conduct meaningful experiments for this distribution. For example, if the experiment only injects peer failures, and no node losses, then the file can

be recovered as long as at least one remaining peer stores either the entire data block DAG or any of the parity block DAGs.

Subtree distribution The second alternative distribution considered is referred to as subtree distribution. In this distribution, participating peers pin subtrees of a file. Experiments with this distribution could evaluate availability by distributing subtrees to IPFS peers rather than individual nodes. Additionally, the height at which subtrees are chosen for distribution could vary. For example, a scenario distributing subtrees at height one (level above the leaf nodes) should perform differently from another scenario distributing subtrees at the height just below the root node, since the data distribution is higher in the former case. We expect subtree distribution to be more probable of seeing usage in the real-world than distributing nodes evenly, since it would be easier to incentivize it and to orchestrate the subtree distribution scheme. An example use case would be large data sets where peers only need portions of the data, and only pin subtrees containing portions of the desired data.

6.3 Network Overhead Results

This section discusses the network overhead results presented in Section 5.3. The results show that the number of bytes needed to recover a file through download and repair is reasonable. The overhead seems to increase linearly with the node loss percentage. As such, for small node losses, which we expect are more likely to occur in practice, the overhead is almost negligible.

Looking at the percentage of downloaded nodes being parity nodes (the *pnode %* metric) gives additional insights. Firstly, when looking at the different node loss percentage scenarios, the percentage of downloaded parity blocks is consistently higher than the network overhead. In other words, despite having to download several parity blocks to recover the original file, the impact on network overhead is minor. Furthermore, considering there are much more parity nodes in total compared to data nodes, this metric tells us that in the successful trials, we do not need to download the majority of parity blocks in order to repair the file. These observations imply that Snarl's recursive repair algorithm prevents us from having to download several nodes.

In the work introducing Snarl written by Nygaard et al [19], the Network Overhead section presents similar findings. There, the download overhead of the Swarm backend is measured as the number of additional chunks (referred to as nodes in this thesis) downloaded during repair compared to in Swarm with no failures. The experiment shows the results for a 1 MiB file with varying levels of chunk loss (referred to as node loss in this thesis). In their findings the overhead also increases linearly, before evening out at around 40 % node loss, confirming our findings.

6.4 Deadlocks

In a few experiments, Snarl got stuck in a deadlock. This occurred rarely, and only for experiments with very high node loss percentages or peer failure percentages, where the expected file availability rate was close to zero. We are not sure why the deadlocks occur, but they only happen in scenarios where it would be impossible to repair the file. If that suspicion is true, this means that the updated Snarl repair algorithm fails to give up in rare instances.

In the node loss experiments from Section 5.2.1, a timeout was added to work around this issue. These experiments time out trials after an amount of time that safely exceeds the expected recovery time including repairs, and disregards the result of the trial when it occurs. It is possible that this might have led to slightly inflated file availability rates

for the highest node loss percentages in the presented results, however we believe that the results are fairly accurate over all.

Chapter 7

Future Work

In this chapter we discuss several approaches that would be interesting to investigate for future work on Snarl. Many of the ideas proposed here were intended to be implemented during this thesis, but could not due to lack of time.

7.1 Swarm using the Bee Client

As mentioned in Section 2.2.2.2, the new Bee client is the currently supported Swarm client. Snarl uses the original Swarm client for its Swarm backend implementation. The Swarm backend should be migrated to the Bee client to be enable Snarl to operate on the public Swarm network. This could either be achieved by implementing an additional Snarl backend using the Bee client, inspired by the existing Swarm backend, or by reworking the existing Swarm backend to use the Bee client instead of the original Swarm client. We believe the former option is best, as it will enable the use of both backends.

7.2 Retrieval Optimization

In addition to using the repair algorithm for repairing unavailable data blocks, it should be possible to use it to reduce retrieval times. Due to the nature of DSSes, some nodes may take very long to retrieve, even if they are available, e.g. nodes that are not widely replicated, or stored by a congested or physically distant peer. Instead of downloading a data block, it is possible to use its related parity blocks to repair it, producing the same content. This method can be used to obtain data blocks that are technically available, but take a long time to retrieve.

To enable this, Snarl could determine during its runtime that downloading a certain block is slower than expected, and try to obtain the block in an alternative way. For instance, it should be possible to represent the different ways of obtaining a data block (that is, direct retrieval or repair through one of the related pp-tuples) as a graph. The graph may extend past the data block and related pp-tuples by incorporating other blocks that can be used to obtain the prerequisite blocks through recursive repair. A pathfinding algorithm could be used to determine the shortest path (involving the least amount of downloaded blocks) to recover the data. The algorithm could also incorporate polling to try other paths if none of the existing paths are succeeding at an acceptable rate.

Such an approach should also try to minimize the bandwidth consumption of the backend DSS. A naïve approach could simultaneously request all the redundant data related to a data block while requesting the data block itself, and use whichever source was available first to recover the data block. This puts unnecessary strain on the backend DSS as most of the downloaded content is not ultimately necessary to recover all data blocks. Therefore, a repair-to-retrieve approach should only be used when Snarl suspects that directly retrieving a data block is slow, or if some prerequisite blocks are already available locally.

7.3 Optimizing Memory Consumption

Currently, Snarl algorithms need a lot of memory. E.g. while Snarl is recovering a file, all data blocks and parity blocks that it retrieves are kept in memory until the algorithm completes. When entangling or recovering large files keeping everything in memory could be problematic. A future work may consider ways of reducing the memory consumption of Snarl algorithms, e.g. by writing intermediate content to temporary files at known offsets, and only storing the data in memory during the operations that need it.

7.4 Updated Repairer Implementation using CSP

It would be interesting to modify or replace the Snarl repairer implementation with a version using CSP for all concurrent operations. The new data block and parity block recovery mechanisms described in Section 4.3 use a CSP-based approach for concurrency, by setting up several concurrent processes that do various distinct tasks, and share memory by communicating. Compared to using locks, this approach is less susceptible to common problems such as data races or deadlocks. The repairer is also split into several domains that involve simple operations, making each domain simpler to reason about. A future work could either extend the newly introduced processes, or implement new ones, to convert the repairer to a pure CSP-based approach. A new implementation could also be more formally defined and evaluated using process calculus.

7.5 Updates to Parity File Layout

In Section 2.4, we saw that Snarl stores parity blocks in α distinct files, which are uploaded to the backend DSS along with the original file. Each parity file contains all parity blocks from one type of strand. This approach makes it easy to generate parity files during entanglement, and to address parity blocks during recovery. However, there are a few drawbacks to this approach:

- *Storage overhead:* Each parity file is constructed from the catenated content of all parity blocks. However, the backend DSS usually generates a Merkle DAG or similar structure when storing the content, for addressability and for confirming that the content is correct. Depending on the branching factor and the number of parity blocks, a lot of internal nodes might have to be generated for this Merkle DAG other than the parity blocks themselves, resulting in storage overhead.
- *End user requirements:* As described in Section 3.1.3, the end user of Snarl must be able to provide the CID of the root node of each parity file DAG to use the Snarl repair algorithm. By storing parity blocks in α files, the end user must remember α root node CIDs for each parity file, in addition to other essential information. Ideally, the end user should have to remember fewer details to be able to use Snarl as intended.
- *Cascading failures:* In the DAGs produced from parity files, only the leaf nodes contain the parity blocks. In other words, only the leaf nodes of the parity file DAGs are protected by the Snarl repair algorithm, since only parity blocks can be repaired. On the other hand, internal nodes in the parity file DAGs cannot be repaired by Snarl. If they are missing we have a cascading failure, where we cannot get references to child nodes of missing internal nodes, rendering parity blocks in their subtree unavailable as well. By storing α parity files as α DAGs, there are a large number of internal nodes, leading to an increased risk for cascading failures.

Because of the above reasons, we believe that storing all parity blocks in a single parity file could be beneficial for Snarl's performance. By storing all parity blocks in a single file,

the total number of internal nodes is decreased, which should mean that cascading failures are less likely to occur. Since parity blocks remain leaf nodes in the parity file DAG, the existing Snarl algorithms are almost unaffected by this change. Knowing the number of parity blocks n of each type of strand, we could simply provide an offset $n \cdot i | i \geq 0$ to access parity blocks from the i-th type of strand. The end user would only have to provide a single parity file root CID instead of α CIDs.

There is however a trade-off with this approach, relating to cascading failures. If a cascading failure occurs, a larger number of underlying parity blocks might be affected. For this reason, we believe it could be possible to provide additional protection for internal parity nodes, such that we use the same total storage overhead as in the original solution, but reduce the probability of cascading failures.

Chapter 8

Conclusion

This thesis updated the Snarl file repair component. The original Snarl implementation could only use the Swarm DSS as a backend. We introduced an abstraction layer and adapted Snarl code such that other backends could be used as well. An additional benefit is that it is now simpler to add additional backends. Using the introduced abstraction layer, we added support for IPFS as a backend for Snarl. The original Swarm backend is also adapted to the new logic, and subjectively speaking the logic of this backend is now simpler.

Furthermore, there are significant additions to the core Snarl repair algorithm. In the new solution, using an approach based on CSP, several processes with separate concerns interoperate to make the recovery and repair algorithms work. This leads to a flatter structure, and should serve as an inspiration to future modifications to Snarl.

Finally, we evaluated the performance of Snarl using the implemented IPFS backend. The first experiment considered node loss, i.e. how file availability is affected when arbitrary nodes are removed from the network. The second experiment considered peer failure, i.e. how file availability is affected when arbitrary peers are removed from the network. Snarl performed well in both experiments, and was found to perform roughly as well as simple replication that used twice the storage overhead. The network overhead during repair was also evaluated, and demonstrated that network overheads are reasonable despite high node loss percentages.

List of Figures

2.1 Merkle tree constructed from four data blocks. Illustration by David Göthberg. Source https://upload.wikimedia.org/wikipedia/commons/9/95/Hash_Tree.svg. . . . 7
2.2 Example Merkle DAG. The blue (dashed) nodes represent the original uploaded file, and the yellow (solid) nodes represent an updated version uploaded later on. . . . 10
2.3 Helical lattice for the $AE(3, 4, 4)$ configuration. . . . 13
2.4 Illustration of pp-tuples. . . . 15
2.5 Illustration of dp-tuples. . . . 15
2.6 High-level overview of the architecture of Snarl. . . . 17
2.7 Illustration of split file during entanglement. . . . 18
2.8 Illustration of Merkle tree from input file during entanglement. . . . 18
2.9 Illustration of flattened Merkle tree during entanglement. . . . 18
2.10 Illustration of parity blocks generated during entanglement. . . . 20
2.11 Illustration of Merkle tree produced from parity blocks during entanglement. 20

3.1 High-level overview of the architecture of Snarl, including the introduced abstraction layer. . . . 28
3.2 Illustration of calls to *next path* requesting a leaf node. . . . 30
3.3 Flow chart of the requester in the recovery process. . . . 32
3.4 Flow chart of the recoverer in the recovery process. . . . 33
3.5 Flow chart of the recovery process of the repairer. . . . 35
3.6 Overview of operation of controller, Snarl and PML during experiments. . . 36

4.1 Overview of type relations in the entanglement process. Optional elements have names surrounded by brackets (`[]`), and are pointed to by dotted lines. 41
4.2 Overview of type relations in the repair process. Optional elements have names surrounded by brackets (`[]`). The `Recoverer` implementation optionally also implements the `Repairer` interface; this relationship is illustrated with a dotted line. . . . 46
4.3 Binary tree with 15 nodes. Node subscripts contain the node's canonical index. 48
4.4 High-level overview of processes in updated repairer. . . . 53

5.1 File recovery rate of 100 MiB files for node loss experiments, averaged over 100 trials. . . . 68

List of Tables

2.1 Entanglement rules to determine which parity blocks are used for *input* to triple entanglement [4]. 14
2.2 Entanglement rules to determine which parity blocks are *output* by triple entanglement [4]. 14

3.1 Results of calls to *next path* on nodes from the Merkle DAG illustrated in Figure 3.2. The call on the leaf node n_9 is not performed in practice, but the result is included for illustration. 31

4.1 Index mappings generated from the tree illustrated in Figure 4.3. 48

5.1 Results from 30 trials of the peer failure experiments for 50 MiB files. . . . 69
5.2 Network overhead measurements for node loss experiments in the *snarl5* configuration. 70

Listings

4.1 API for node and flattened DAG representations. 38
4.2 API for entanglement. 40
4.3 API for uploading blocks and files. 42
4.4 API for retrieving and deserializing nodes. 43
4.5 API for recovery and repair. 44
4.6 Exported types and functions from `package indexmap`. 48
4.7 Header for the `Lattice` constructor. 55
4.8 `Node` interface implementation for IPFS. 56
4.9 `Walk` implementation and header for unexported `walk` method for the IPFS backend `Node` type. 57
4.10 `NextPath` implementation for the IPFS backend `Node` type. 59
4.11 The IPFS backend `NodeGetter` type, related constructors, and the `Deserialize` method header. 60
4.12 Type, constructor and method headers for the `FlatDAGConstructor` implementation the IPFS backend. 61
4.13 `TreeChunk` type from the Swarm backend. 62
4.14 `Walk` implementation and header for unexported `walk` method for the Swarm backend `TreeChunk` type. 63

Bibliography

[1] Juan Benet. "IPFS - Content Addressed, Versioned, P2P File System (DRAFT 3)". In: (2014).

[2] The IPFS Community. *IPFS Docs*. 2020. URL: https://docs.ipfs.io.

[3] Alexandros G. Dimakis et al. "Network Coding for Distributed Storage Systems". In: 2007.

[4] Vero Estrada-Galiñanes. "Practical erasure codes for storage systems: The study of entanglement codes, an approach that propagates redundancy to increase reliability and performance". PhD thesis. May 2017. DOI: 10.13140/RG.2.2.31917.74725.

[5] Vero Estrada-Galiñanes et al. "Alpha Entanglement Codes: Practical Erasure Codes to Archive Data in Unreliable Environments". In: *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)* (June 2018). DOI: 10.1109/dsn.2018.00030. URL: http://dx.doi.org/10.1109/DSN.2018.00030.

[6] Charles Antony Richard Hoare. "Communicating sequential processes". In: *Communications of the ACM* 21.8 (1978), pp. 666–677.

[7] *IPFS Block format (Go file) from package ipfs/go-block-format*. Sept. 2018. URL: https://github.com/ipfs/go-block-format/blob/2b1318c1fcabb7cec4c74724edc1b3ba220d2693/blocks.go (visited on 05/30/2021).

[8] *IPFS Cluster Homepage*. 2020. URL: https://cluster.ipfs.io/ (visited on 06/04/2021).

[9] *IPFS Docs: File Systems and IPFS*. May 2021. URL: https://docs.ipfs.io/concepts/file-systems/ (visited on 06/07/2021).

[10] *IPFS Docs: Go-IPFS*. Apr. 2021. URL: https://docs.ipfs.io/reference/go/api/ (visited on 06/06/2021).

[11] *IPFS Docs: HTTP API reference*. Sept. 2020. URL: https://docs.ipfs.io/reference/http/api/ (visited on 06/06/2021).

[12] *IPFS Docs: Pin files using IPFS*. Apr. 2021. URL: https://docs.ipfs.io/how-to/pin-files/ (visited on 06/06/2021).

[13] *IPFS go-ipfs-api GitHub repository*. Apr. 2021. URL: https://github.com/ipfs/go-ipfs-api/ (visited on 06/06/2021).

[14] *IPLD format (Go file) from package ipfs/go-ipld-format*. Sept. 2018. URL: https://github.com/ipfs/go-ipld-format/blob/47f3c58ad85d03df2ed9e1385b3000b5d8f48e98/format.go (visited on 05/30/2021).

[15] *IPLD Merkle DAG definition (Go file) from package ipfs/go-ipld-format*. Apr. 2020. URL: https://github.com/ipfs/go-ipld-format/blob/47f3c58ad85d03df2ed9e1385b3000b5d8f48e98/merkledag.go (visited on 05/30/2021).

[16] Petar Maymounkov and David Mazières. "Kademlia: A Peer-to-peer Information System Based on the XOR Metric". In: 2002.

[17] Ralph C. Merkle. "Secrecy, authentication and public key system". PhD thesis. Dept. of Electrical Engineering, Stanford University, June 1979.

[18] Arvind Narayanan et al. *Bitcoin and Cryptocurrency Technologies*. Princeton University Press, 2016.

[19] Racin Nygaard, Vero Estrada-Galiñanes, and Hein Meling. "Snarl: Entangled Merkle Trees for Improved Availability and Storage Utilization". In: *Submitted for publication* (2021).

[20] *Original Swarm client GitHub repository*. 2021. URL: https://github.com/ethersphere/swarm/ (visited on 06/08/2021).

[21] *Package context*. 2021. URL: https://golang.org/pkg/context/ (visited on 05/30/2021).

[22] *Package context: func WithValue*. 2021. URL: https://golang.org/pkg/context/#WithValue (visited on 05/29/2021).

[23] *Package exec*. 2021. URL: https://golang.org/pkg/os/exec/ (visited on 06/06/2021).

[24] *Package io: type Writer*. 2021. URL: https://golang.org/pkg/io/#Writer (visited on 05/22/2021).

[25] William Stallings. *Cryptography and Network Security*. Pearson Education Limited, 2017.

[26] *Swarm Bee client GitHub repository*. 2021. (Visited on 06/08/2021).

[27] *Swarm storage type definitions (Go file) from package swarm/storage*. Feb. 2020. URL: https://github.com/ethersphere/swarm/blob/cf4cefbd7adc4aeeea8182ffc0a6b0d91a8abcb7/storage/types.go (visited on 06/01/2021).

[28] Sasu Tarkoma. *Overlay Networks: Toward Information Networking*. CRC Press, 2010.

[29] Ethereum Swarm team. *The sun is setting for the old Swarm network*. Dec. 2020. URL: https://medium.com/ethereum-swarm/the-sun-is-setting-for-the-old-swarm-network-46cdc8048f8b (visited on 06/08/2021).

[30] *The Go Programming Language Specification: Channel types*. Feb. 2021. URL: https://golang.org/ref/spec#Channel_types (visited on 05/22/2021).

[31] *The Go Programming Language Specification: Interface types*. Feb. 2021. URL: https://golang.org/ref/spec#Interface_types (visited on 05/22/2021).

[32] *The Go Programming Language Specification: Select statements*. Feb. 2021. URL: https://golang.org/ref/spec#Select_statements (visited on 05/30/2021).

[33] Viktor Trón. *The Book of Swarm - v1.0 pre-release*. https://www.ethswarm.org/The-Book-of-Swarm.pdf. 2020.