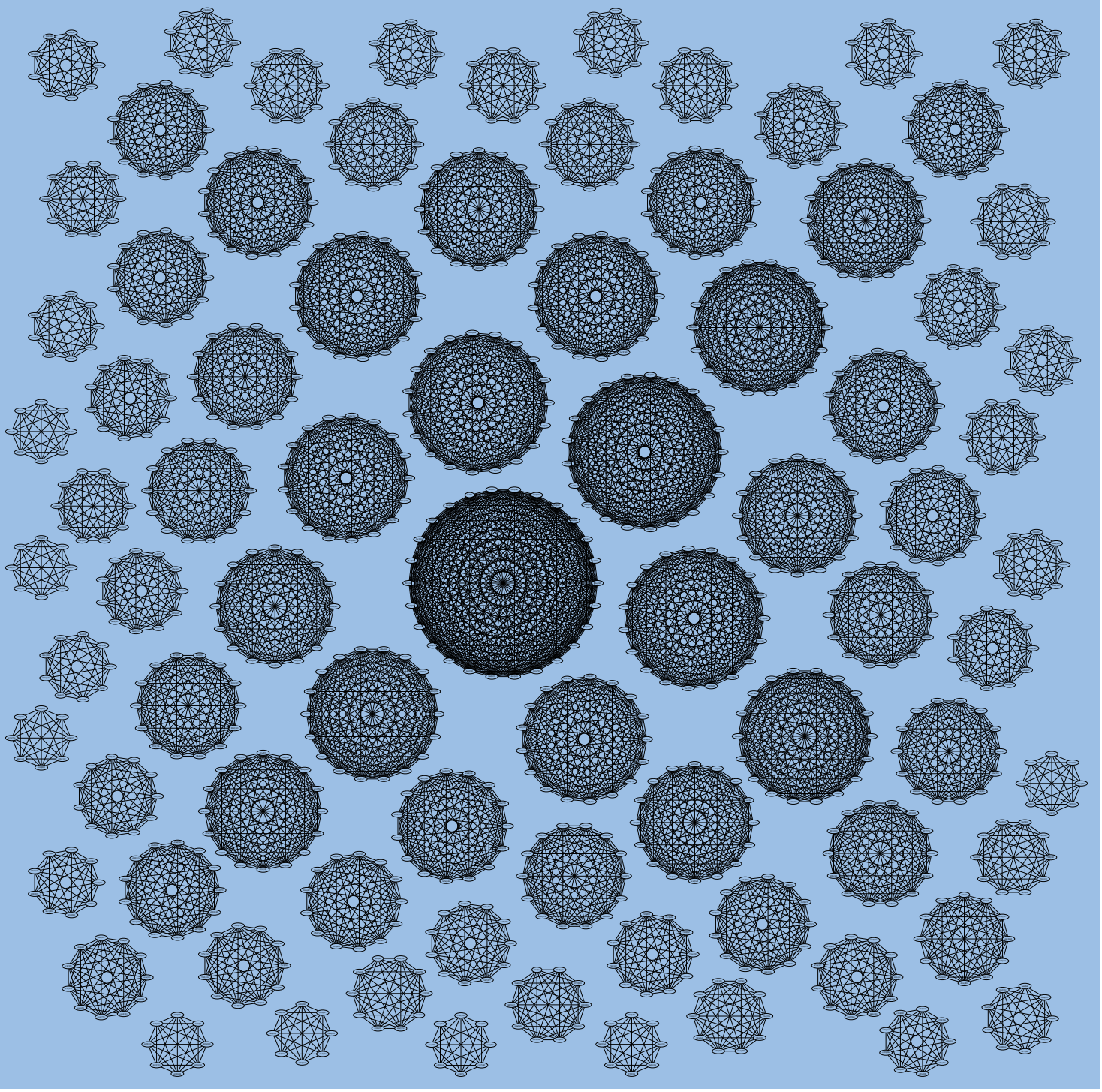
University of Stavanger

RACIN NYGAARD

FACULTY OF SCIENCE AND TECHNOLOGY

Improving Data Availability in Decentralized Storage Systems

PhD Thesis UiS no. 693 - April 2023



Improving Data Availability in Decentralized Storage Systems

by

Racin Nygaard

Thesis submitted in partial fulfilment of
the requirements for the degree of
PHILOSOPHIAE DOCTOR (PhD)

Faculty of Science and Technology
Department of Electrical Engineering and Computer Science
April 2023

University of Stavanger
NO-4036 Stavanger
NORWAY
www.uis.no

ISBN: 978-82-8439-158-8
ISSN: 1890-1387
PhD: Thesis UiS No. 693

This thesis is affectionately dedicated to
Filip and Regine

Preface

This dissertation is submitted in partial fulfillment of the requirements for the degree of Philosophiae Doctor (PhD) at the University of Stavanger. The research was conducted at the University of Stavanger from August 2018 to March 2023.

The dissertation consists of a collection of 3 research papers and 1 poster paper. Two research papers have been published, and one is to be submitted. The poster paper was presented during the conference but was not included in the proceedings. We give detailed information about the papers in the list of papers section. The dissertation is an individual contribution that aims at providing a holistic view of the research.

The papers are included in the dissertation after realignment and transformations to adhere to the requisite format. The content of the papers has been kept intact.

Apart from the papers, this dissertation also contains the necessary background information to read the papers. For improved readability, the dissertation re-uses much content from the included papers.

Racin Nygaard

Abstract

Preserving knowledge for future generations has been a primary concern for humanity since the dawn of civilization. State-of-the-art methods have included stone carvings, papyrus scrolls, and paper books. With each advance in technology, it has become easier to record knowledge. In the current digital age, humanity may preserve enormous amounts of knowledge on hard drives with the click of a button.

The aggregation of several hard drives into a computer forms the basis for a storage system. Traditionally, large storage systems have comprised many distinct computers operated by a single administrative entity.

With the rise in popularity of blockchain and cryptocurrencies, a new type of storage system has emerged. This new type of storage system is fully decentralized and comprises a network of untrusted peers cooperating to act as a single storage system. During upload, files are split into chunks and distributed across a network of peers. These storage systems encode files using Merkle trees, a hierarchical data structure that provides integrity verification and lookup services.

While decentralized storage systems are popular and have a user base in the millions, many technical aspects are still in their infancy. As such, they have yet to prove themselves viable alternatives to traditional centralized storage systems.

In this thesis, we contribute to the technical aspects of decentralized storage systems by proposing novel techniques and protocols. We make significant contributions with the design of three practical protocols that each improve data availability in different ways.

Our first contribution is Snarl and entangled Merkle trees. Entangled Merkle trees are resilient data structures that decrease the impact hierarchical dependencies have on data availability. Whenever a chunk loss is detected, Snarl uses the entangled Merkle trees to find parity chunks to repair the lost chunk. Our

results show that by encoding data as an entangled Merkle tree and using Snarl's repair algorithm, the storage utilization in current systems could be improved by over five times, with improved data availability.

Second, we propose SNIPS, a protocol that efficiently synchronizes the data stored on peers to ensure that all peers have the same data. We designed a Proof of Storage-like construction using a Minimal Perfect Hash Function. Each peer uses the PoS-like construction to create a storage proof for those chunks it wants to synchronize. Peers exchange storage proofs and use them to efficiently determine which chunks they are missing. The evaluation shows that by using SNIPS, the amount of synchronization data can be reduced by three orders of magnitude in current systems.

Lastly, in our third contribution, we propose SUP, a protocol that uses cryptographic proofs to check if a chunk is already stored in the network before doing wasteful uploads. We show that SUP may reduce the amount of data transferred by up to 94 % in current systems.

The protocols may be deployed independently or in combination to create a decentralized storage system that is more robust to major outages. Each of the protocols has been implemented and evaluated on a large cluster of 1,000 peers.

Acknowledgements

I want to thank my family and friends for their endless love and support: my children Filip and Regine, my wife Alina, my parents Gunn and Racin, and my brother Gaute and Johannes, Sofia, Gulnara, and Cristina.

I thank my supervisor, Professor Hein Meling, for his excellent guidance and advice over many years.

Thanks to the Relab research group members for many interesting and fruitful discussions.

Lastly, I thank all my colleagues and coworkers for their support, particularly Tore, Kjell-Ivar, Pål, and Vidar.

Contributions

In this chapter, I will present the contributions that were made during my PhD studies. First, I will present the list of research papers included in this thesis and their publication status. Second, I will present the source code and software artifacts that were developed during the PhD period. Next, I will present awards that were won during hackathons and other open public competitions. Then, I will list the presentations that were given during conferences and workshops. Following that, I will list my teaching activities during the PhD period. Lastly, I will list other works I was involved in both as a co-author, reviewer, and supervisor.

List of Included Papers

This section lists the research papers that are the basis for this article-based thesis. The content of each paper is included in Chapter 8.

1) Snarl: Entangled Merkle Trees for Improved File Availability and Storage Utilization

Racin Nygaard, Vero Estrada-Galiñanes, Hein Meling
Proceedings of the 22nd International Middleware Conference. Middleware '21. Québec city, Canada: Association for Computing Machinery, 2021, pp. 236–247.
URL: `https://doi.org/10.1145/3464298.3493397`

Keywords: *storage utilization, entanglement codes, erasure codes, cryptographic decentralized storage system, file availability*

2) SNIPS: Succinct Proof of Storage for Efficient Data Synchronization in Decentralized Storage Systems

Racin Nygaard, Hein Meling
To be submitted

Keywords: *data synchronization, proof of storage, decentralized storage system, maintenance, redundancy, data upkeep*

3) Cost-effective Data Upkeep in Decentralized Storage Systems

Racin Nygaard, Hein Meling, John Ingve Olsen
Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing. Tallinn, Estonia, 2023. URL: `https://doi.org/10.1145/3555776.3577728`

Keywords: *data upkeep, proof-of-storage, decentralized storage system, re-upload, peer-to-peer*

4) Lessons Learned from a Bare-metal Evaluation of Erasure Coding Algorithms in P2P Networks

Racin Nygaard
in: *arXiv* abs/2208.12360 (2022). URL: `https://arxiv.org/abs/2208.12360`

Keywords: *experimental evaluation, tooling, distributed storage, redundancy, peer-to-peer*

Source Code and Software Artifacts

The source code for the protocol and other software artifacts developed during the PhD period are available at `https://github.com/racin/phd`. The following is a non-exhaustive list of software artifacts that were developed during the PhD period.

- Source code for Snarl from **Paper 1**.
- Source code for SNIPS from **Paper 2**.

- Source code for SUP from **Paper 3**.
- Evaluation framework used for the evaluation. Inspired by **Paper 4**.
- GUI visualization tool for Snarl.
- Configuration files and scripts used for the cluster.

Awards

Winner "Ethereum Madrid Hackathon"

My team won 1st place in the "Ethereum Madrid Hackathon" for the submission "Entanglements in Swarm" [19] in Madrid in 2019.

Winner "Pentesting Ethereum Contracts"

My team won 1st place at the capture the flag event "Pentesting Ethereum Contracts" [54] hosted by Truffle during DevCon 5 in Osaka in 2019.

Winner "Best DApp using Swarm decentralized storage"

My team won 1st place in the category "Best DApp using Swarm decentralized storage" [79, 6] at WAM Hackathon in 2022.

Winner "Best search mechanism for Wikipedia offline snapshots"

My team won 1st place in the category "Best search mechanism for Wikipedia offline snapshots" [79, 7] at WAM Hackathon in 2022.

Grant "A spam-free, secure and decentralized email service on top of Swarm"

My team received a grant to develop Waggle [9] in 2020. Waggle is a spam-free, secure and decentralized email service on top of Swarm.

Presentations

ACM Middleware

I presented **Paper 1** during the ACM Middleware 2021 conference, titled "Snarl: Entangled Merkle Trees for Improved File Availability and Storage Utilization". The talk is available at `https://www.youtube.com/watch?v=fqnl1HHAKoM`.

ACM SAC

I presented **Paper 3** during the ACM SAC 2023 conference (DADS track), titled "Cost-effective Data Upkeep in Decentralized Storage Systems".

ACM Sigmetrics

I presented **Paper 4** during the poster session ACM SIGMETRICS 2021 conference, titled "Lessons Learned from Deploying a Decentralized Storage System".

University of Oslo

I had a guest talk at the University of Oslo in 2020, titled "Recent development of storage solutions in Ethereum".

DevCon 5

I co-authored a talk during DevCon 5 in Osaka in 2019, titled "When Merkle met Entanglements".

Credence Workshop

I presented a poster at the Credence Workshop in 2019, titled "Long-Term Storage In Decentralized Environments".

Swarm Orange Summit

I had a talk at the Swarm Orange Summit in 2019, titled "Motivating increased participation with incentives".

BBChain Workshop

I had a talk during the BBChain Workshop in 2019, titled “Persistent storage on Swarm.”. At the same workshop in 2021, I had a talk titled “Snarl: Entangled Merkle Trees for Improved File Availability and Storage Utilization”.

Teaching

DAT520 Distributed Systems

I was the course teacher for the course DAT520 Distributed Systems in 2021 and 2022. I also was the coordinator for laboratory exercises from 2019 to 2022.

DAT320 Operating Systems and Systems Programming

I was a teaching assistant for the course DAT320 Operating Systems and Systems Programming in 2020.

List of Other Works

The following contributions are not included in this thesis:

1. Vero Estrada-Galinanes, Racin Nygaard, Viktor Tron, Rodrigo Saramago, Leander Jehl, and Hein Meling. “[Invited talk] Building a disaster-resilient storage layer for next generation networks: The role of redundancy”. In: *IEICE Technical Report; IEICE Tech. Rep.* 119.221 (2019), pp. 53–58

The following theses have been inspired by my work:

2. Eivind Stavnes and Daniel Urdal. “Extending the Snarl File Repair Component for Distributed Storage Systems”. MA thesis. University of Stavanger, 2021

3. Bohua Jia. “Proving Redundancy In Decentralized Storage Networks”. MA thesis. University of Stavanger, 2022

4. Daniel Dirdal, Vebjørn Sundal, and Bartlomiej Stasiak. “Graphical Interface for File Repair Algorithm Entangle Visualizer 3D”. B.S. thesis. University of Stavanger, 2022

Contents

Chapter 1

Introduction

According to recent studies, the amount of digital data continues to grow at an extraordinary rate. [32]. Some data may be ephemeral, such as chats or temporary files created by applications and may be deleted after a short period. Other types of data, such as photos, videos, and documents may be of significant importance, and should never be deleted.

Storing such important data on a single hard drive is risky, as the hard drive may fail and the data may be lost. Naively increasing the number of hard drives to create additional copies is expensive, both in terms of the up-front cost of the hard drive, but also in terms of the cost of electricity. In addition, managing a large number of hard drives, including replacing failed hard drives, requires significant time and effort.

For the reasons above, it has become common to use cloud services such as Google Drive [34], Dropbox [17] and Microsoft Onedrive [46] to store data. Typically, a user pays a monthly subscription fee to the cloud service provider, and in return, the cloud service provider ensures that the data is stored reliably and is available when needed. The cloud service provider may use replication to create multiple copies of the data, in addition to other techniques to ensure that the data is available when needed.

Even though cloud services are convenient, they are not without their drawbacks. A primary concern from the user's point of view is that they represent a single legal entity, and may thus be a single point of failure. In addition, they offer limited transparency into how securely the data is stored and managed.

As an alternative to cloud services, decentralized storage systems have been

proposed [37, 13, 83, 74, 4, 39, 70, 84, 77]. In these systems, there is no single entity that controls the storage system. The storage system is comprised of a network of peers that cooperate to act as a single storage system. As such, these systems have the potential to be more robust to major outages and attempts of censorship, as the data is stored on multiple peers.

However, the decentralized nature of the storage system also introduces new challenges. The primary concern is that the peers are untrusted and unreliable. Hence, many of the previous technologies that were used to ensure data availability in cloud services are no longer applicable. As an example, simple replication requires significantly more coordination due to a lack of reliable information on the number of copies of the data that exists on the network.

The first generation of decentralized storage systems was inspired by the wave of peer-to-peer file-sharing systems of the late 1990s and early 2000s. Well-known examples include Napster [10], Gnutella [58], Freenet [11] and BitTorrent [12]. These systems were effective in distributing popular content but lacked the ability to preserve data that was rarely accessed. To solve the issue of long-term persistence, decentralized storage systems such as OceanStore [37], Cooperative File System [13] and Tahoe-LAFS [83] were proposed. However, these systems failed to gain wide adoption, partially due to technical challenges, but also due to a lack of incentives for the peers to operate correctly.

Ever since the hugely popular Bitcoin [47] protocol was released in 2009, there has been a surge of interest in blockchain technology and finding ways to use it to solve real-world problems. With the idea of *Web3* [78], there was a wide attempt at applying a token-based economy to many areas. This is also true for decentralized storage systems, and a new generation consisting of systems such as Swarm [74], IPFS [4], Filecoin [39], Storj [70], Arweave [84] and Sia [77] have emerged. These systems use blockchain technology to reward peers with cryptocurrencies, to ensure that they are incentivized to operate correctly.

However, many of the technical challenges of the first generation of decentralized storage systems remain unsolved. In this thesis, we contribute to solving some of the technical challenges. Specifically, we propose three protocols that each improve data availability in different ways. While the three protocols are designed to be used independently, they can also be used together to achieve even better results. As part of our work, we have implemented the protocols in the Go programming language and evaluated them on a large cluster of 1,000 peers.

In Figure 1.1, we show a simplified and generalized upload process in a decentralized storage system. During the upload process, the file is split into chunks and each chunk is given a unique identifier (labeled $a-e$). Based on the identifier, the chunks are then distributed to different peers in the network.

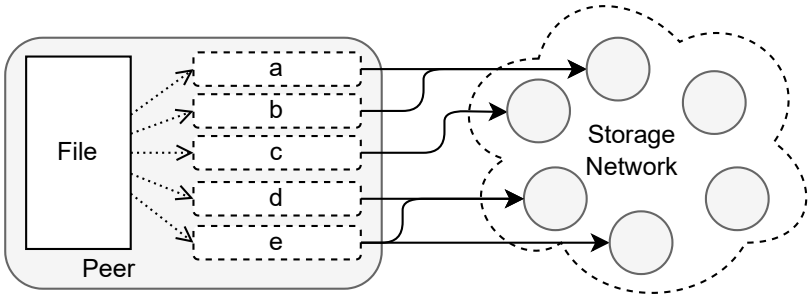


Figure 1.1: Upload process in decentralized storage systems.

The chunk distribution can be viewed as a mapping from chunk identifiers to the peers storing the file's chunks. Since all chunks are needed to reconstruct the file, the chunk mapping is essential for a functioning decentralized storage system. We refer to this mapping as the *lookup-metadata* of the file.

A key distinction between different decentralized storage systems is how they store the lookup-metadata. The first approach, used by Sia and Storj respectively, relies on the users themselves or a centralized authority to store a mapping for each chunk in the file. The second approach, used by Swarm, IPFS, and Filecoin is to encode the file into a Merkle-based structure [4, 45] so that the user only needs to store a single chunk identifier. The tradeoff for avoiding the centralization in the first approach is that the Merkle-based structure creates metadata chunks that are used for lookup and adds a hierarchical dependency between the metadata chunks and chunks of the file. Hence, the new metadata chunks become essential as without them, it is not possible to locate the chunks of the file in the network.

To prevent data loss, current systems typically add redundancy to the chunks by replicating them to multiple peers. However, it is well-known [59, 80, 43] that replication generates a high storage overhead compared to the redundancy provided. For better storage utilization, it is possible to use erasure codes instead of replication. An erasure code takes the chunks of the file as input and create addi-

tional parity chunks. However, to use erasure codes, additional *decode-metadata* is needed to know which parity chunks to use when recovering from data loss.

Similar to the distinction above, those systems that rely on the user or a centralized authority to store the lookup-metadata, use the same approach for storing the decode-metadata. However, for systems that use Merkle-based structures, we were unable to find any prior work that addresses the challenge of applying erasure codes to chunks that have a hierarchical dependency.

To the best of our knowledge, the first protocol in this thesis, Snarl, was the first work that considered the hierarchical dependencies between the lookup-metadata chunks and the chunks of the file when applying erasure codes. In addition, Snarl proposes a method to eliminate the need for a local or remote file for the decode-metadata. We designed a new Merkle-based structure called entangled Merkle trees that creates sufficient redundancy for both the lookup-metadata chunks and the chunks of the file. The entangling process achieves this by creating new parity chunks to intertwine both the lookup-metadata chunks and the chunks of the file. Our results show that Snarl increases storage utilization by 5× in Swarm with improved data availability. The file recovery is bandwidth-efficient and uses less than 2× chunks on average.

As depicted in Figure 1.1, chunks are sent to peers in the network. However, the receiving peer may already store the chunk, wasting significant bandwidth resources. To mitigate the wasted bandwidth, as the peers are untrusted, users cannot rely on simply asking the peer if it already has the chunk.

Our second protocol, Storage Upkeep Protocol (SUP) allows users to obtain cryptographic proof that its files (or chunks) are correctly stored in the network. To request a proof, a user will first generate a *nonce* (number used once). The nonce is then put into a challenge and sent to the network. Recipients of the challenge are expected to answer by generating a proof consisting of a cryptographic hash over the chunk data concatenated with a nonce. Including a nonce ensures that the peers had to be in possession of the given chunk at the time when creating the proof. The primary use case of SUP is to allow users to check the persistence of their data, before attempting to re-upload it. Our evaluation shows that SUP may reduce the amount of data transferred by up to 94 % and reduce the time needed to re-upload the data by up to 82 %.

The third protocol, Succinct Proof of Storage (SNIPS) targets the data synchronization between peers in the network. Efficient data synchronization is im-

portant for ensuring that sufficient redundancy is stored in the network, and responding to rapid changes in network topology. Yet, current methods for data synchronization are inefficient as they rely on long lists of chunk identifiers to determine which chunks are missing. In contrast, SNIPS use a novel approach based on minimal perfect hash functions to create succinct storage proofs that may be queried to determine missing chunks. Our results show a reduction in metadata transmitted to synchronize by up to three orders of magnitude.

While the jury is still out on the new generation of decentralized storage systems, they are promising. Even if the new generation of decentralized storage systems fails, the research in this thesis will be useful for future generations of storage systems. During our work, we produced large amounts of source code needed to implement, analyze, and evaluate the protocols. Parts of the source code have been made publicly available at `https://github.com/racin/phd`.

1.1 Common Definitions

Throughout this thesis, the terms *storage peer* and *peer* are used interchangeably, unless explicitly stated otherwise. We use the term *chunking* to refer to a process that splits a file into smaller *chunks*. Those peers that store a given chunk are called that chunk's *storer peers*.

We have also avoided using the term *node* when referring to a peer, except when made clear by the text when discussing other systems. In our text, we use the term node to refer to an element in a Merkle-based structure.

1.2 Thesis Outline

The rest of this thesis is arranged as follows. Chapter 2 presents a more holistic view of the background for the material covered in the papers. In Chapter 3 we build on the background material to present the research questions and main contributions that this thesis covers. In Chapter 4 we review our contributions and conjecture that by integrating the contributions we can further improve data availability in decentralized storage systems. We present a security analysis of the chunk proofs used in **Paper 2** and **Paper 3** in Chapter 5. Chapter 6 describes our experimental evaluation framework used to empirically evaluate our protocols.

Chapter 7 concludes the thesis, reflects on our contributions, and outlines ideas for future work. Finally, all 4 papers are included in Chapter 8.

1.3 Reading Guide

The chapters in the thesis are not independent of each other, and their chronological order does not reflect the optimal reading order. We recommend the following for the best reading experience:

1. Contributions
2. Introduction (Chapter 1)
3. Background (Chapter 2)
4. Research Questions (Chapter 3)
5. Included Papers (Chapter 8)
    (a) Snarl: Entangled Merkle Trees for Improved File Availability and Storage Utilization (**Paper 1**)
    (b) Lessons Learned from a Bare-metal Evaluation of Erasure Coding Algorithms in P2P Networks (**Paper 4**)
    (c) Cost-effective Data Upkeep in Decentralized Storage Systems (**Paper 3**)
    (d) SNIPS: Succinct Proof of Storage for Efficient Data Synchronization in Decentralized Storage Systems (**Paper 2**)
6. Improving Data Availability (Chapter 4)
7. Security Analysis of Chunk Proofs (Chapter 5)
8. Empirical Evaluations (Chapter 6)
9. Conclusion, Reflections and Future Prospects (Chapter 7)

Chapter 2

Background

In this chapter, we will present the necessary background material to read the rest of the thesis. We begin by discussing cryptographic hash functions and the Merkle tree. Next, we discuss two erasure codes, first the Reed-Solomon code, and then the Alpha Entanglement code. After this, we discuss distributed hash tables and proof of storage systems.

We then detail the Swarm storage network, which is a decentralized storage system that was used as a baseline for all our contributions. Finally, we discuss a range of decentralized storage systems and conjecture on how applicable our contributions are to these systems.

We note that much of the background material presented in this chapter also appears in the included papers. We present a more holistic view of the material here for the reader's convenience.

2.1 Overview

Over the years there have been numerous proposals for network-based storage systems. Notable systems that are well-known in the literature include NFS [49], OceanStore [37], Ceph [81], Cooperative File System [13] and Tahoe-LAFS [83]. The overarching goal of these systems is to provide a shared file system that is accessible by multiple clients.

A network-based storage system may be categorized as centralized, distributed or decentralized. In the following description, we have adapted the classical categorization found in [71] with more recent terminology found in [56] and [1]. A

centralized system will typically have a single point of failure and a single administrator that controls the system. A distributed system may have multiple points of failure and has a finite number of administrators that controls the system. A decentralized system has no single point of failure and no single administrator that controls the system. Each of these degrees of centralization has its own advantages and disadvantages, and the choice of protocols and algorithms will depend on the degree of centralization, amongst other parameters. As a rule of thumb, protocols and algorithms tend to become more complex as the degree of centralization decreases, with the most complex being decentralized systems. As such, algorithms and protocols from one system may not be directly applicable to another system.

A decentralized storage system is comprised of a network of storage peers that collaborate in storing data. The peers are untrusted and are typically connected to a subset of other peers in a Peer-to-Peer (p2p) network. This category of storage systems received considerable attention in recent years with the rise of Web3, cryptocurrencies and blockchain technology.

Using blockchains, the peers are incentivized to correctly store data as they receive crypto tokens for doing so. The potential of monetary benefits enables great competition between peers, as each peer wants to maximize their earnings. This competition may drive peers to rent out their storage space for lower costs, which in turn may drive down the cost of storage for the end user.

However, without a properly fine-tuned incentivization system, the fierce competition may lead to peers attempting to cheat the system. Cheating peers may compromise both the security and resiliency of the system and may ultimately lead to irrecoverable data loss. One such example is peers claiming to store data when they do not, in an attempt to earn additional crypto tokens.

Hence, to provide long-term data availability we need reliable and efficient protocols that are agnostic to monetary benefits. In the following sections, we will go into more detail on the protocols of decentralized storage systems. Our focus will be on the Ethereum Swarm storage network, however many of the concepts will be similar or relatable to protocols in other storage networks, such as IPFS.

2.2 Cryptographic Hash Functions

A cryptographic hash function is a common building block that maps an arbitrary length input to a fixed length output. The mapping is one-way, meaning that it is computationally infeasible to find the input given the output. Additionally, the mapping is deterministic, such that the same input will result in the same output.

Formally, we assume that a cryptographic hash function H satisfies the following security properties [64, 48].

1. Preimage resistance
2. Second preimage resistance (Puzzle friendliness)
3. Collision resistance

The preimage resistance property states that for any given hash value h, it is computationally infeasible to find a value y such that $\mathrm{H}(y) = h$.

Puzzle friendliness states that for every possible n-bit output value y, if k is chosen from a distribution with high min-entropy, then it is infeasible to find x such that $\mathrm{H}(k \mid\mid x) = y$, in time significantly less than 2^n. Second preimage resistance is stronger form of puzzle-friendliness, which says for any given value k, it is infeasible to find $k' \neq k$ such that $\mathrm{H}(k') = \mathrm{H}(k)$.

Collision resistance states that it is infeasible to find any pair of two distinct values x and y, such that $x \neq y$ and $\mathrm{H}(x) = \mathrm{H}(y)$.

2.3 Merkle Tree

A Merkle tree [45] is a hierarchical data structure whose original application was a digital signature system. Figure 2.1 illustrates a Merkle tree. The Merkle tree in the figure is binary, meaning that each node has at most two children. Merkle trees may also be k-ary, where each node has at most k children.

Merkle trees are constructed in a bottom-up fashion. The first step is to split the input data into smaller pieces, e.g. chunks of a file. Each chunk is then hashed to produce a leaf node, i.e. H_A is the hash of chunk A in Figure 2.1. The next step is to concatenate the value of k nodes and then hash the concatenated value to produce a parent node, e.g. $H_{AB} = H(H_A \parallel H_B)$. This step is repeated recursively for each level, until a single root remains, e.g. $H_{ABCDEFGH} = H(H_{ABCD} \parallel H_{EFGH})$.

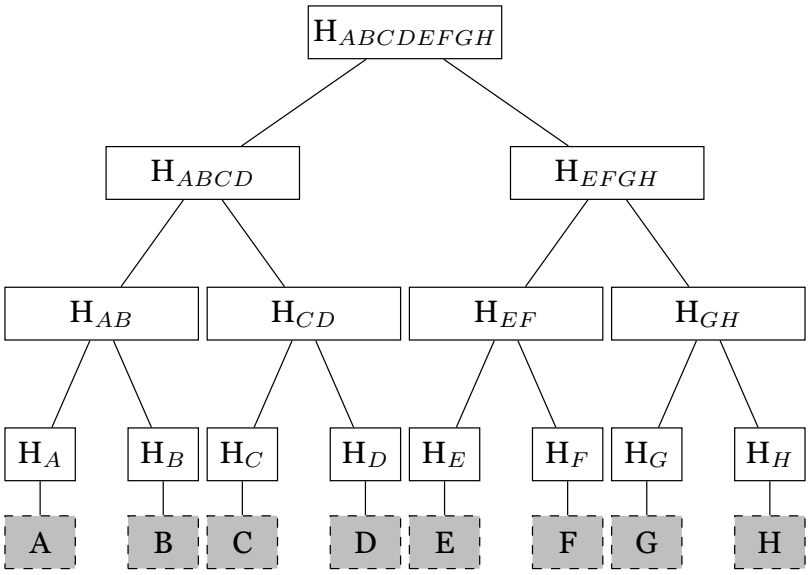


Figure 2.1: Binary Merkle tree with 8 leaves. The input values (A-H) are in gray.

The design choice of the number of children per node is application specific and is typically enforced by the protocol. For the same number of leaves, a binary tree will have a height of $\log_2(n)$, while a k-ary tree will have a height of $\log_k(n)$. The height of the tree is important as it determines the number of internal nodes that need to be computed to reach the root node. While a high k may reduce the height of the tree, it causes the internal nodes to span a larger number of children, which leads to larger inclusion proofs.

An inclusion proof is an important use case of Merkle trees that is used to prove to a third party that a certain node is part of the tree. To generate the inclusion proof, the prover recursively in a bottom-up fashion gathers the siblings of each parent until the root is reached. For example, to prove that leaf node H_C is part of the tree in Figure 2.1, the prover needs to gather H_D, H_{AB} and H_{EFGH}, and then send these values to the verifier. Using these values, the verifier can use H_C and H_D to compute H_{CD}, then use H_{AB} and H_{CD} to compute H_{ABCD}, and finally use H_{EFGH} and H_{ABCD} to compute the root $H_{ABCDEFGH}$. If the newly

computed root matches the root in the tree, the verifier can be sure that H_C is part of the tree.

Decentralized storage systems such as Swarm, IPFS, and Filecoin split the files into chunks and encode them in a Merkle-based structure. We say that this structure comprises both the lookup-metadata as internal nodes and the chunks of the file as leaves. To illustrate the use of lookup-metadata, we can use the example depicted in Figure 2.1 to see how a single chunk would be retrieved. Note that the retrieval of all chunks would be similar, with the main difference that all paths would be explored. Starting from the root, we have $H_{ABCDEFGH} = H(H_{ABCD} \parallel H_{EFGH})$. Consider we issue a request for the chunk identifier $H_{ABCDEFGH}$. The storage network will then return an internal node with the value $ABCDEFGH = H_{ABCD} \parallel H_{EFGH}$. Thus, we can continue by requesting another chunk identifier H_{ABCD}. We will then receive another internal node with the value $ABCD = H_{AB} \parallel H_{CD}$. The process is repeated with $H_{AB} \rightarrow AB = H_A \parallel H_B \rightarrow H_A \rightarrow A$.

2.4 Erasure Codes

To mitigate single points of failure, a common approach is to add redundancy to the data. The simplest form of adding redundancy is through replicating the data into multiple copies. However, it has been shown that by using erasure coding, similar levels of redundancy can be achieved with significantly lower storage overhead [24].

The most well-known erasure code is the Reed-Solomon (RS) code [57]. RS codes are typically defined over a finite field of two elements, denoted $GF(2^u)$ for u-bit code symbols. To encode, the RS encoding algorithm takes k information symbols as input and produces $n = 2^u - 1$ encoded symbols as output, where $n > k$. The standard notation is RS(n, k). RS codes are *maximum distance separable* (MDS) [33], and as a result, we may recover the original k symbols from any k encoded symbols. In other words, the RS code can recover from up to $n - k$ failures. However, the RS codes are not optimal for recovering from small errors. To recover any $[1, k\rangle$ original symbols, we still need k encoded symbols.

RS codes are a type of linear block code [33] and do not require keeping previous states in memory to do the encoding. Other types of codes, which we will discuss in the next section, require previous states to do the encoding.

As a last observation, we note that the optimal values of the n and k parameters are application-specific and may depend on the input data. For example, if the input was a file, and the k parameter was less than the size of the file, then the naive solution would be to split the file and perform multiple encodings. However, as we show in **Paper 1**, this approach achieves less tolerance to failures than a single encoding with a larger k parameter.

2.5 Alpha Entanglement Codes

Alpha Entanglement Codes (AE) [22] was proposed as practical and flexible erasure codes that increase redundancy by interconnecting storage devices (or storage nodes) with redundant information in large storage systems. The paper also suggests using AE codes on a much granular scale—to interconnect the chunks of a file. In our work with Snarl in **Paper 1** we propose using AE codes to create entangled Merkle trees, a resilient data structure that protects both lookup-metadata and the chunks of a file. Hence, in the following description of AE codes, we adapt the notation found in the original paper to the context of entangled Merkle trees.

We begin by giving an overview of AE codes and describing the simplest configuration. Then we discuss the general AE code and its parameters. Finally, we discuss our extensions to AE code to make them suitable for entangled Merkle trees and Snarl.

2.5.1 Overview

AE codes are designed to tolerate a large number of failures with low computation and bandwidth requirements. Redundancy is achieved by using the XOR (exclusive-or) operation to create an intertwined lattice. The lattice is a graph $G(V, E)$ where each vertex $v \in V$ represents a data chunk, and each edge $e \in E$ represents a parity chunk. The lattice is comprised of *strands* which is a chain of alternating data and parity chunks. A remarkable property of AE codes is that any chunk can be repaired by two other chunks, as opposed to requiring k chunks to repair a single chunk in RS codes (see Section 2.4). We call such a pair of chunks a *repair pair*.

In the following sections, we let d_i denote the i-th data chunk obtained from

chunking a file. We begin by illustrating a special case called simple entanglements [21] where the lattice forms a single chain in Figure 2.2. The data chunks are shown as circles and parity chunks as lines.

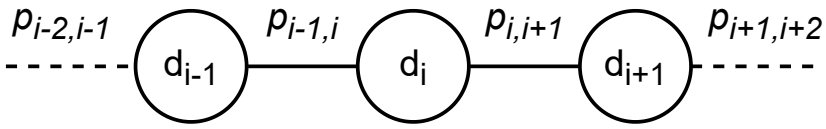


Figure 2.2: Simple Entanglement Codes ($\alpha = 1, s = 1, p = 0$).

Parities on a strand are calculated by XORing the previous data chunk with the previous parity chunk. At the very beginning of the strand, the parity $p_{i-1,i}$ does not exist and is artificially set to all zeros so that $p_{i,i+1}$ becomes a copy of d_i.

$$p_{i,i+1} = p_{i-1,i} \oplus d_i \tag{2.1}$$

Based on Equation (2.1) and the associative property of XOR, we can easily see that the repair pair of a data chunk consists of preceding and proceeding parity chunks.

$$d_i = p_{i-1,i} \oplus p_{i,i+1} \tag{2.2}$$

By the same reasoning, we can see how a parity chunk may be repaired by the proceeding data- and parity chunks.

$$p_{i,i+1} = d_{i+1} \oplus p_{i+1,i+2} \tag{2.3}$$

Hence, we see from Equation (2.1) and Equation (2.3) that a parity chunk has two possible repair pairs. Note that XOR is also commutative, so the ordering in the XOR operation does not matter.

2.5.2 Parameters of AE codes

We now discuss the parameters that make up the general AE code. The lattice is configured by three parameters: α, s and p, with $p \geq s$. The α parameter denotes the number of strands that are connected to each data chunk. Hence, α in essence controls how many parities will be created, which has a direct impact on the storage overhead. Each increase in α incurs the same storage overhead as having an

extra copy of the data itself.

In other words, $\alpha = 2$ incurs storage overhead similar to two extra copies of the data, and $\alpha = 3$ to three extra copies of the data. Higher values than $\alpha = 3$ have not been studied in the literature [22]. Increasing α also increases the number of repair pairs exponentially, which significantly increases fault tolerance, as there are more repair pairs to choose from.

The s and p parameters denote the number of horizontal and helical strands, respectively. In a two-dimensional representation, s may denote the number of rows and p the number of distinct columns. The columns are distinct in the sense a helical strand will revolve around the lattice structure after p columns. Thus, different values of s and p will decide which data chunks are connected to which strand. Hence, some configurations of s and p may provide different fault tolerance than others. However, increasing s or p does not impact the storage overhead, but may impact the time to encode.

By increasing α, the lattice becomes more intertwined, as illustrated by the $\alpha = 3,\ s = 5,\ p = 5$ configuration in Figure 2.3. With such a configuration, each data chunk will have $\alpha = 3$ strands that are connected to it and thus be associated with $\alpha = 3$ repair pairs. In other words, data chunk 7 may be repaired by either of the pairs $(p_{2,7} \oplus p_{7,12})$, $(p_{1,7} \oplus p_{7,13})$ or $(p_{3,7} \oplus p_{7,11})$.
As with simple entanglements, repairing either a data chunk or a parity chunk is a single XOR operation, as shown in Equation (2.1), Equation (2.2) and Equation (2.3). However, if we do not possess both chunks needed to fulfill the repair pair, a repair algorithm may attempt to expand in the lattice to first repair the chunks contained in the repair pair, and so on.

For the $\alpha = 3$ configuration, we label the three strand classes as *Horizontal* (H), *Right* (R) and *Left* (L). As $s = 5$ there are 5 horizontal strands, and similarly, as $p = 5$ there are 5 helical (right and left) strands. In the two-dimensional representation, such as Figure 2.3, the helical strands are illustrated as diagonal lines. To determine the labels of the parities connected to each data chunk, we may use the rules listed in Table 2.1. The top row of the lattice is denoted *top*, the bottom row *bottom*, and all others *center*. Formally, data chunk d_i is at the top iff $i \equiv 1 \pmod{s}$, bottom iff $i \equiv 0 \pmod{s}$, and center otherwise.

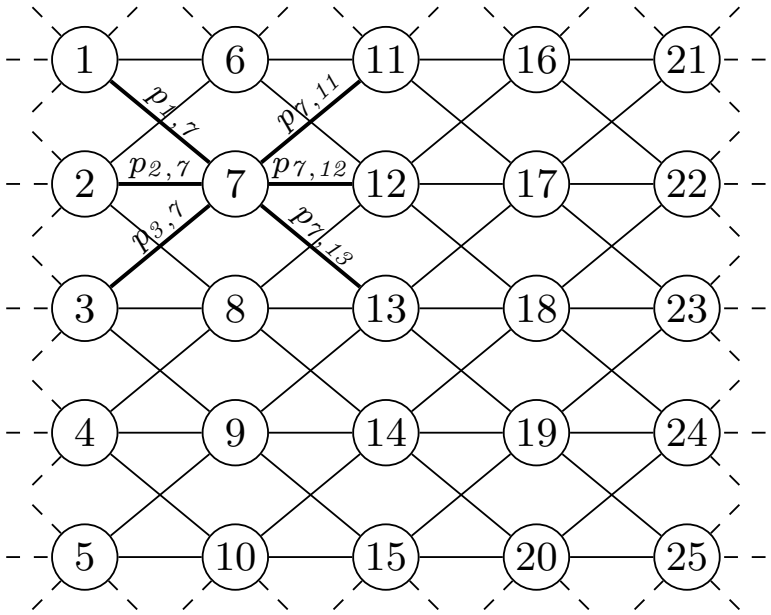


Figure 2.3: Alpha Entanglement with $\alpha = 3,\ s = 5,\ p = 5$ configuration.

2.5.3 Extending AE Codes for Entangled Merkle Trees

The work of **Paper 1** expands on the usage of AE codes by proposing an algorithm that allows complex lattices to be closed. Previous work [21] only considered closing the lattices with $\alpha = 1$. The closing is achieved by creating a new parity chunk in each strand that wraps around from the last data chunk in the strand to the first data chunk in the strand. We then recalculate the value of the first parity chunk using Equation (2.1), as it was initially a copy of the first data chunk.

A 1 MB file that is encoded into a Merkle tree in Swarm will comprise 256

Table 2.1: Repair pairs for data chunk d_i with $\alpha = 3$.

Data chunk d_i Placement	Horizontal strand Repair pair	Right strand Repair pair	Left strand Repair pair
Top	$p_{i-s,i} \oplus p_{i,i+s}$	$p_{i-s\cdot p+(s^2-1),i} \oplus p_{i,i+s+1}$	$p_{i-s+1,i} \oplus p_{i,i+s\cdot p-(s-1)^2}$
Center	$p_{i-s,i} \oplus p_{i,i+s}$	$p_{i-s-1,i} \oplus p_{i,i+s+1}$	$p_{i-s+1,i} \oplus p_{i,i+s-1}$
Bottom	$p_{i-s,i} \oplus p_{i,i+s}$	$p_{i-s-1,i} \oplus p_{i,i+s\cdot p-(s^2-1)}$	$p_{i-s\cdot p+(s-1)^2,i} \oplus p_{i,i+s-1}$

leaves, 2 internal nodes and the root. To protect the entire Merkle tree, we view all 259 elements as data chunks and then use AE codes to create entangled Merkle trees. In Figure 2.4 we show the closing of a lattice with 259 data chunks. As illustrated in the lower parts of the figure, closing the lattice changes the geometrical shape of each strand class (H, R, L) to a toroidal.

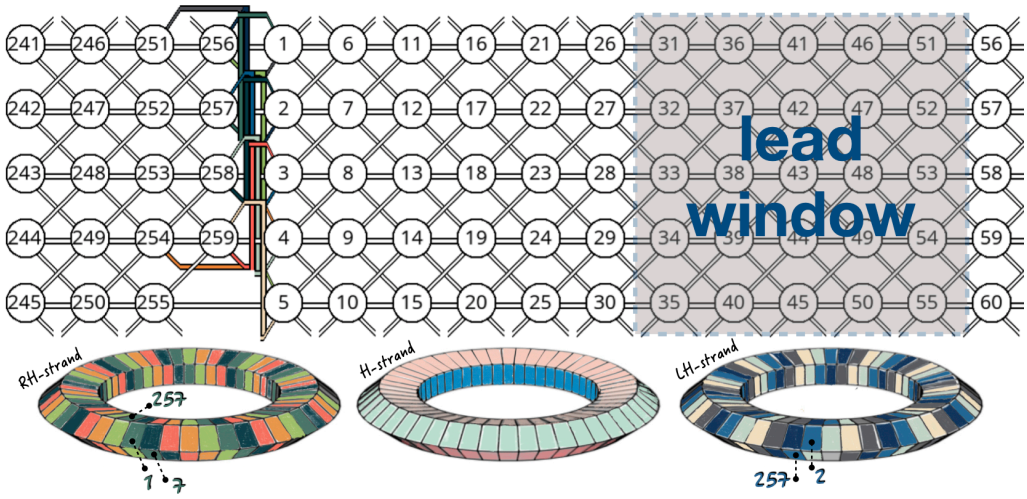


Figure 2.4: Toroidal lattice for a Merkle tree with 259 data chunks.

At first glance, the splice between the $[256, 259]$ column and the $[1, 5]$ column may seem arbitrary. However, the logic to close follows a simple rule. The rule states, that the splice from the last data chunk will be to the data chunk that initiated the strand. In the case of data chunk 259 in Figure 2.4, the right strand is initiated by data chunk 3. In other words, if we were to track the right strand from data chunk 3, we would end up at data chunk 259 ($3 \rightarrow 9 \rightarrow 15 \rightarrow 16 \rightarrow 22 \rightarrow \ldots \rightarrow 259$). Similarly, the left strand is initiated by data chunk 5. The revolving of the helical strands is also illustrated on the lower parts of the figure. It can be observed by following a color on either of the toroidal structures, e.g. green on the right strand, to see that it will revolve around after $p = 5$ chunks. In the special case where the number of data chunks is a multiple of $s \cdot p$, the splicing would follow the symmetry of the lattice, e.g., if the lattice contained 25 data chunks, for data chunk 25. Hence, for data chunk 25 the splice on the right strand would be to data chunk 1, and the splice on the left strand would be to data chunk 4.

2.5.4 Encoding the Entangled Merkle Tree

When a file is encoded as a Merkle tree, the tree comprises the following.

- the chunks of the file as leaves
- lookup-metadata in the form of internal nodes
- the root of the tree as the entry point for the file

The purpose of the entangled Merkle tree is to add fault tolerance to all of these components. Hence, the entangled Merkle tree should avoid interdependencies between the different components. For example, we cannot rely on an internal node for lookup being repaired by its children, as without the internal node, the children cannot be retrieved.

To avoid such interdependencies, the encoding should place a parent node and its children at least one *lead window* away from each other. A lead window for $\alpha = 3$ describes the number of data chunks that can be encoded before the helical strands (right or left) revolves around the lattice structure. As illustrated in Figure 2.4, consider the right strand that follows data chunk 31 on the path $31 \rightarrow 37 \rightarrow 43 \rightarrow 49 \rightarrow 55 \rightarrow 56 \rightarrow \ldots$. When it reaches 56, we can see that the pattern starts to repeat, and thus we can see that a lead window consists of 25 data chunks. Formally, the size of a lead window is defined as the product $s \cdot p$.

To manage these interdependencies, when constructing the entangled Merkle trees we attempt to place dependant chunks at least one lead window away from each other. We call this process *swapping* and have illustrated it in Figure 2.5.

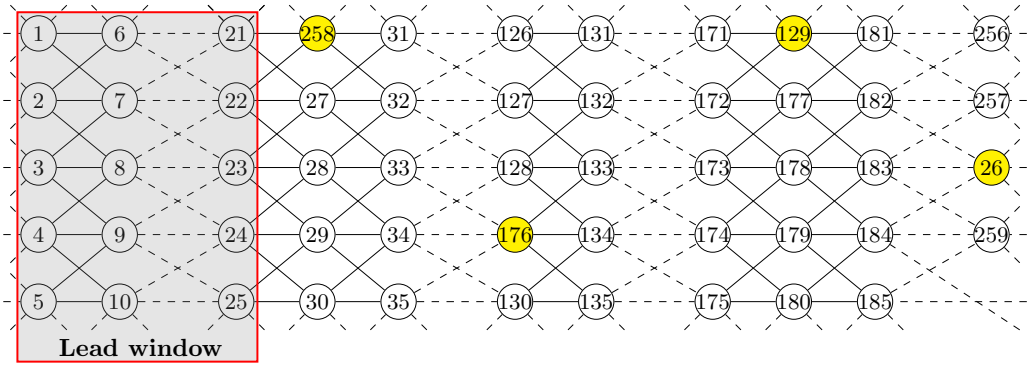


Figure 2.5: Managing interdependencies through swapping position in the lattice.

In the figure there we can see that data chunks 26 and 258 have been swapped. Similarly, data chunks 129 and 176 have been swapped. The reason for this is that 129 is the internal chunk that is the parent to chunks 1 to 128, and 258 is the parent to chunks 130 to 257. Hence, by having this new placement, we achieve improved fault tolerance as we avoid attempting to repair internal chunks using its children. For further details, we refer to **Paper 1** and its presentation from Middleware.

2.6 Distributed Hash Tables

An overlay network creates a logical network on top of the physical network to serve as the basis for most p2p systems [27]. One type of overlay network is structured p2p overlays, commonly referred to as Distributed Hash Table (DHT).

A DHT is a distributed system that provides a lookup service for a key-value store. Due to the logical routing, the DHT allows a peer to reach any other peer, while only requiring that each peer maintain connections to a small subset of the total peers in the network. Each peer participating in the DHT is given a unique peer address that is used to identify the peer. When a new key-value pair is inserted into the DHT, the peer address is used to determine which peers should store the key-value pair. Similarly, to later retrieve the value, a request containing the key is routed to the peers which are responsible for storing the value. This way, we only need to query a fraction of the total peers to retrieve the value. Notable DHTs that offer logarithmic lookup are Chord [66], Pastry [60], Tapestry [86] and Kademlia [44]. Both Swarm and IPFS use Kademlia as their DHT. In addition, a recent survey on decentralized storage systems shows that the most common DHT used in practice is Kademlia [14]. Therefore, the rest of this section will focus on Kademlia.

Kademlia assigns a 160-bit address to each peer when joining the system. Keys share the same 160-bit identifier space as peer addresses. To determine the distance between addresses or keys, the exclusive-or (XOR) operation is used. Peers maintain a routing table by storing information about other peers in *buckets*. However, each peer only maintains active connections to a handful of other peers in each bucket. The routing table for the peer P_{02} is shown in Figure 2.6.

As shown in the figure, peers have more granular information about peers that are closer to them. This allows for a logarithmic lookup time, as with each con-

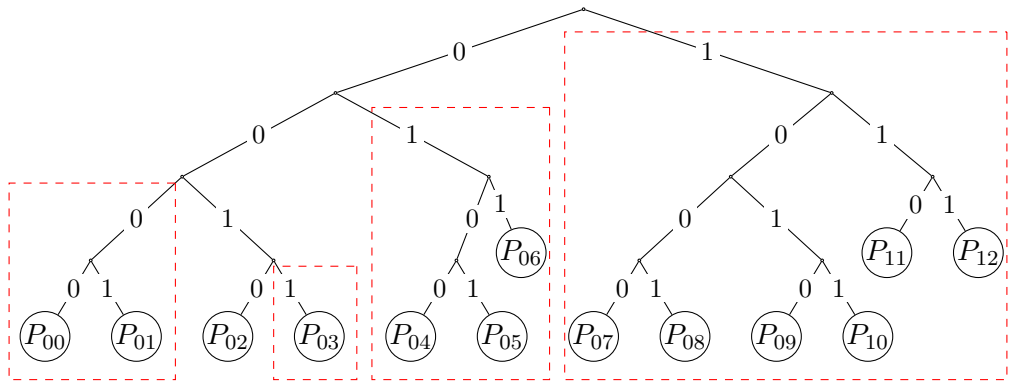


Figure 2.6: Kademlia tree with 13 peers. The red dashed rectangles indicate the buckets of peer P_{02}.

sequent step, the distance between the peer and the key is halved. To illustrate, consider a peer X, which wants to look up a value y in the Kademlia DHT. Peer X starts by checking the bucket where it knows y should be located. From the peers in that bucket, it selects the closest peer to y and queries it for the value. This peer, which we call Z, does not have y but instead returns a set of peers Z' that are even closer to y than itself. Peer X then selects one or more peers from Z' and queries them for y. This process continues until the closest peer in the network, Y, is discovered. Finally, the request for y is sent to Y, which returns the value to peer X.

2.7 Proof of Storage

Proof of Storage algorithms provides a way to outsource storage to a third party while being able to verify that the data is being correctly stored, without having to transfer the data itself. A Proof of Storage (PoS) system typically has three distinct actors; the *challenger*, which issues PoS queries; the *prover*, which responds to the queries by creating a proof; and the *verifier*, which verifies the proof.

The earliest works were published in 2007 [35, 3, 61]. Since that time, there has been considerable work to construct schemes with additional features and improved properties [85].

A common strategy in PoS algorithms is that the data owner samples the original data to create *proving-metadata*, before uploading it to storage peers. The proving-metadata must be kept secret from the storage peer, as it contains in-

formation on how to verify a proof. Hence, by having the proving data, the data owner can delete the original data and still be able to verify the data is stored correctly.

As an illustrative example, consider an imaginary PoS algorithm that randomly samples k bit values and their position p from the original data and stores them into a map with k entries $p \rightarrow \{0, 1\}$. To then create a challenge, the challenger selects n random positions from the map and sends them to the prover. The prover then needs to respond with the corresponding bit values. To verify the proof, the verifier checks that the bit values are correct.

Note that such a trivial scheme would have limited usage in practice, as with each new challenge, the prover learns more and more about the map (proving-metadata). When the prover has learned all the entries in the map, it can simply delete the uploaded data to save storage space while still being able to answer all subsequent challenges.

In **Paper 3** we propose a PoS scheme that verifies the entire data, not only random samples. Additionally, in **Paper 2** we propose a PoS-like construction that allows membership queries when multiple data items are included in the same storage proof.

2.8 Ethereum Swarm

Ethereum Swarm [76] (Swarm for short) is a global decentralized storage network. Swarm's monitoring website [26] shows that the network has over active 2,900 peers. Peers are incentivized to store and serve data by receiving BZZ tokens for doing so. The BZZ token is the native token of the Swarm network, and it resides on the Ethereum blockchain.

The following sections will detail the technical aspects of data storage, network topology and data synchronization in Swarm. We will use the *file* terminology to refer to any collection of data that a client wants to persist in the network. Collections of data include a regular single file, parts of a file and any other streams of bits.

2.8.1 Data Storage

Swarm splits files into 4 KB chunks. A unique *chunk identifier* is derived for each chunk by passing the chunk's content through a cryptographic hash function. Swarm creates a 128-ary Merkle tree where each of the file's chunks is a leaf. The internal nodes and root of the Merkle tree are also stored in 4 KB chunks and contain a concatenation of the chunk identifiers of their children. Thus, the integrity of a file is easily verified using the root hash value, as any modifications are propagated to the root.

This root also serves as the entry point when retrieving a file from the network. When a client wants to retrieve a file, it first queries the network for the root chunk. The root chunk is then decoded to reveal the chunk identifiers of the internal nodes. The same process is repeated for the internal nodes until the leaf chunks are reached. When all leaves have been retrieved, the file is reconstructed from the chunks.

2.8.2 Network Topology

Swarm uses a variant of Kademlia, called *forwarding Kademlia* [74], to maintain the overlay network. To illustrate forwarding Kademlia, we use the same scenario as for regular Kademlia, described in Section 2.6. We consider a peer X who wants to look up a value y. In the same way, peer X starts by finding the closest peer Z in the bucket where y belongs. When querying Z for y, instead of Z returning a list of peers that are closer to y, it will forward the request to one of them. This forwarding continues until the closest peer Y is reached. The peer Y then returns the value y through the same path as the request — but in reverse.

Forwarding Kademlia has two main advantages over regular Kademlia. First, each peer that is a *forwarder* only knows from which peer they received the request, and to which peer they forwarded the request. Therefore, it may be difficult to know which peer originated the request. This may be desirable for storage systems that want some notion of sender anonymity. Second, the workload of requesting and decoding responses is spread to more peers in the network. The distribution of the workload is a form of load balancing. However, as a tradeoff, as the workload of requesting is spread to more peers, we have an increased risk of failure due to message loss, as peers along the delivery path may fail or otherwise refuse to forward the request.

Swarm organizes peers into neighborhoods, based on the similarity of their peer addresses. The significance of neighborhoods is that each peer is supposed to replicate their data with its neighbors (see Section 2.8.3). In Figure 2.7 we show two example neighborhoods obtained in our cluster of 1,000 peers. The entire network topology is shown on the front page (zoom in to read the peers' labels) and represented as a histogram in Figure 2.8. In our experimental setup, we observed 81 distinct neighborhoods of various sizes $n \in [8, 26]$. As chunks are replicated by all peers in a neighborhood, the varying sizes of the neighborhoods cause the chunks' replication factor to vary accordingly.

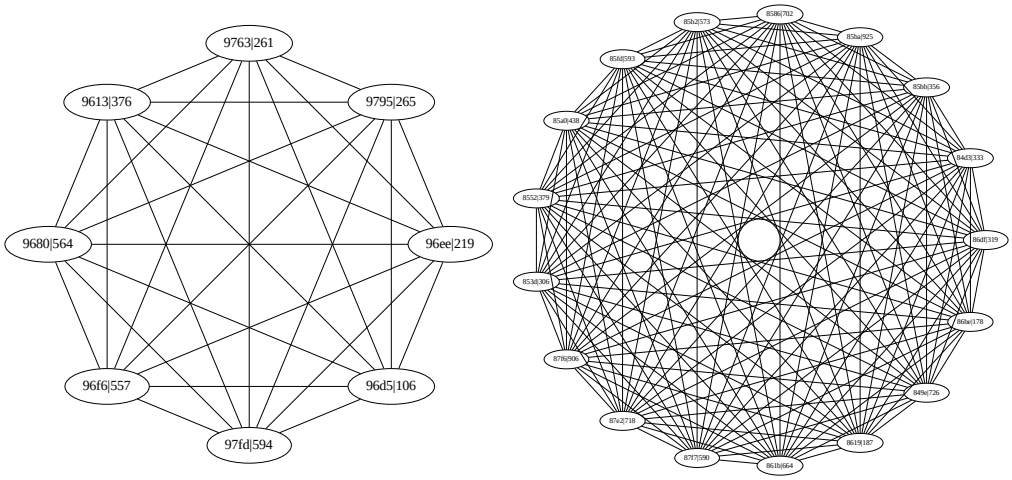


Figure 2.7: Swarm neighborhoods with 8 and 17 peers. Peers are labeled with their address prefix and their cluster instance.

2.8.3 Data Synchronization

In Swarm, chunk identifiers and peer addresses share the same address space. A neighborhood is a cluster of peers with similar addresses. During upload, each chunk is distributed to a neighborhood based on the chunk's content address. We call this the chunk's *closest neighborhood*, and the peers in the neighborhood are called the chunk's *storer peers*. These neighboring peers are responsible for storing the chunks whose content address is similar to their peer addresses. To download a file, a peer (or client) will similarly attempt to locate the file's chunks in the neighborhoods where the chunk is supposed to be stored.

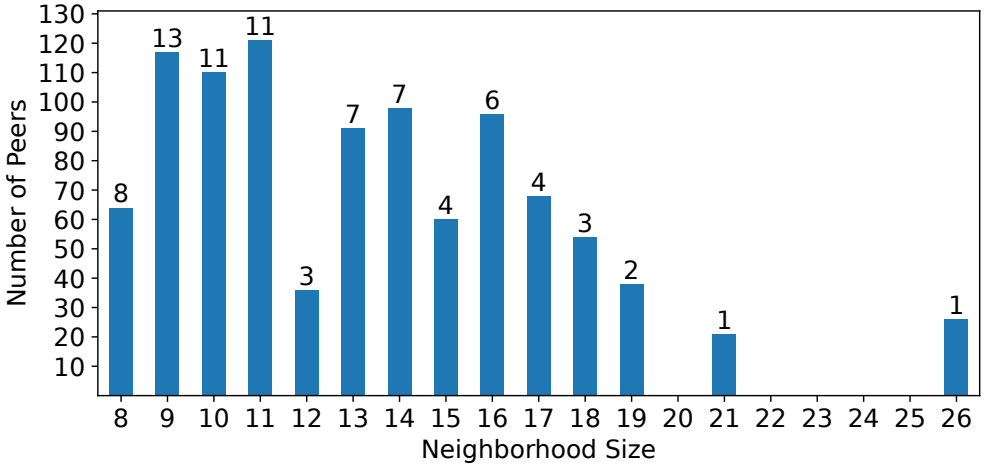


Figure 2.8: Distribution of neighborhood sizes in our cluster.

There are three protocols for data synchronization in Swarm; *push-sync*, *pull-sync* and *retrieval-cache*. The push-sync protocol involves transferring chunks from the uploader to storage peers, i.e., from the network's entry point to each chunk's closest neighborhood. Second, the pull-sync protocol is activated when a new peer enters or leaves the network. When pull-syncing, peers will query their connected peers for a list of chunk identifiers they are storing and then proceeds to request those within its address space. Lastly, the retrieval-cache protocol is activated when a peer forwards or otherwise receives a chunk.

In **Paper 3** we propose SUP which uses PoS queries to check the persistence of chunks in the network, before uploading them. Chunks that are not found, will be uploaded using the push-sync protocol. In **Paper 2** we discuss the fallacies of pull-sync and propose SNIPS as a replacement.

2.9 Other Decentralized Storage Systems

In this section, we briefly describe other current decentralized storage systems. Many aspects of these systems work similarly to Swarm, however, we will highlight some aspects in which they differ. In particular we will focus on IPFS [4], Filecoin [39], Storj [70], Arweave [84] and Sia [77]. A recent survey on decentralized storage systems [14] provides a more comprehensive overview of the field.

2.9.1 IPFS

The InterPlanetary File System (IPFS) is arguably the most well-known decentralized storage system. It has over 200,000 active peers [41], and over 2 million weekly users.

When a file is uploaded to IPFS, it is split into 256 KB chunks [75]. The chunks are then encoded into a Merkle DAG (Directed Acyclic Graph). The Merkle DAG is less rigid than the Merkle tree and allows for more flexible data structures. For example, the DAG does not need to be balanced, nodes can have values and nodes can have multiple parents.

IPFS peers are expected to store the entire file themselves and offer no built-in redundancy protocol. However, if an operator runs multiple IPFS peers, they can add redundancy to their data by connecting the IPFS peers using *IPFS Cluster* [38]. IPFS Cluster adds redundancy by replicating the data across the predefined peer instances. To store data at third parties, it is possible to use Filecoin [39]. Filecoin works as a layer on top of IPFS and adds replication and incentivizes storage.

To discover and transfer data, IPFS uses the *Bitswap* protocol [16]. To retrieve data that is stored in the network, the requesting peer uses Bitswap to send a *IWANT-HAVE* message to the peers it is connected to. This message contains a list of chunk identifiers. Then, each peer that is storing either of the chunks will respond with a *IHAVE* indicating which chunks it is storing. Finally, the requesting peer will send a new message *IWANT-BLOCK* to individual peers that are storing the requested chunks.

2.9.2 Filecoin

Filecoin is a decentralized storage network that uses IPFS as its underlying storage layer. Peers are rewarded with FIL tokens for correctly storing data. To verify that a peer is storing data correctly, Filecoin uses two separate PoS-based methods. First, to ensure that a peer is storing multiple copies of the data—on separate physical devices—Filecoin uses the *Proof of Replication* protocol [5]. Second, to ensure that a peer is storing the data continuously for a long period, Filecoin uses the *Proof of Spacetime* protocol [5].

A peer must meet demanding hardware requirements [40] before joining the network. In addition, a peer must put a deposit of FIL tokens into an account as

collateral. The collateral may be *slashed* (lost) if the peer fails to operate correctly.

2.9.3 Storj

Storj boasts over 21,000 active peers [67]. As with Filecoin, there is a minimum hardware requirement for joining the network [69]. However, Storj does not require a deposit of tokens as collateral. To join the network, a user must first register an account and then download the Storj client.

There are three different peer classes in the Storj network; *Uplink nodes*, *Storage nodes* and *Satellites*. Uplink nodes represent any service or application that uploads or downloads data to the Storj network. Storage nodes are the peers that store the data. Storage nodes are incentivized by receiving STORJ tokens for operating correctly. Lastly, a Satellite peer is a trusted centralized entity that manages the network. The tasks of the Satellite peer include [68]; 1) User account management, 2) Access management, 3) Payment, 4) Metadata storage, 5) Data repair and 6) Data auditing.

Storj uses the Reed-Solomon erasure coding scheme to add redundancy to the data. During the upload process, the Uplink node performs encryption, erasure coding and chunking. The Upload node then coordinates with the Satellite peer to distribute the chunks to the Storage nodes.

The auditing service of the Satellite node will periodically request *proofs of retrievability* from the Storage nodes. Failure to provide a proof promptly will result in reduced rewards.

2.9.4 Arweave

Arweave uses *blockweave*, a blockchain-like structure to store data in the network. As with regular blockchain structures, each block in blockweave has a link to the previous block. However, each block also has a link to another deterministically determined block, creating a structure that resembles a woven pattern—hence the name blockweave. This deterministically determined block is called the *recall block*, and it aids peers that want to keep track of the latest data in the network—but do not want to download the entire blockchain.

To ensure data permanence, Arweave stores uploaded data directly on-chain on blockweave. The redundancy is provided through the miners replicating the

blocks. The Arweave network has just over 100 peers [2]. The peers are rewarded with the native cryptographic token AR.

2.9.5 Sia

Sia was released in 2014 and is one of the first decentralized storage systems in the current generation. There are currently around 400 active peers [63].

Peers may be rewarded the cryptographic token *siacoin* in two ways [62]. The first way is by performing proof of work [48] to secure the Sia blockchain. The second way is to become a *host* that contributes storage to the network. To be eligible as a host, a peer must put a deposit of siacoins. In addition, the host will be subject to continuous auditing to ensure that they are storing the data correctly.

Sia splits files into 40 MB chunks, smaller files than 40 MB will be padded. The redundancy is provided through Reed-Solomon erasure coding, however, it offers no solution for the metadata, and lets the user decide how to store it. Should the metadata be lost, it may not be possible to recover the file. After creating the parity chunks, Sia will encrypt the file using Threefish [23]. The chunks are then uploaded to multiple peers, and their locations are also stored in the local metadata.

2.9.6 Conjectured Applicability of Our Contributions

We have presented a brief overview of the most popular decentralized storage systems. While these systems all aim at storing data, their approaches, implementation and programming languages differ. Hence, we cannot expect the contributions herein to work "out-of-the-box". Rather, the contributions will need to be adapted to the specific system. In Table 2.2 we summarize our conjectures about our contributions' applicability to these systems. We note that we are only considering the technical applicability of our contributions if implemented in each of these systems. In the following, we will discuss how our contributions relate to these systems and elaborate on our conjectures.

First, we have marked *Applicable* for all systems to adapt SUP from **Paper 3**. The reason for this is that all the systems have a similar structure in that they split a file into chunks and then distribute the chunks to the storage peers. By applying the techniques proposed in SUP, data transmission can be optimized as only the

Table 2.2: Conjectured applicability of our contributions.

Decentralized Storage System	Snarl **Paper 1**	SNIPS **Paper 2**	SUP **Paper 3**
Swarm	Applicable	Applicable	Applicable
IPFS	Applicable	Applicable	Applicable
Filecoin	Applicable	Applicable	Applicable
Storj	Artifacts applicable	Applicable with amendments	Applicable
Arweave	Applicable	Applicable with amendments	Applicable
Sia	Artifacts applicable	Applicable with amendments	Applicable

chunks that are missing need to be distributed. Hence, we conjecture that SUP is applicable to all systems.

Next, for Snarl from **Paper 1**, we observe that Swarm, IPFS, and Filecoin have similar structures for chunking and encoding the data into hierarchical structures. Previous work [65] has also implemented a version of Snarl in IPFS. Hence, we mark these systems as *Applicable* for Snarl.

Considering the data availability in Storj and Sia, they both use Reed-Solomon erasure coding to add redundancy to the data. However, their approaches have some weaknesses. In Sia, the repair metadata is stored by the user, and if it is lost, the data may be unrecoverable. In Storj, the repair metadata is stored by a trusted centralized entity, and if the entity is compromised, the data may be lost. In both of these systems, chunks are stored in a flat structure, unlike the hierarchical structure considered for Snarl. This makes the repair process significantly easier, as it does not need to consider dependencies between chunks. However, Snarl proposes a method to eliminate the need for a local or remote metadata file. Therefore, we mark these systems as *Artifacts applicable* for Snarl.

As for Arweave, the data is stored directly on the blockchain, and the redundancy is provided through miners replicating the blocks. However, replication has a high storage overhead compared to the provided redundancy [24]. Hence, we mark Arweave as *Applicable* for Snarl.

Finally with regards to SNIPS from **Paper 2**, we again argue that the structures of Swarm, IPFS, and Filecoin are sufficiently similar to suggest that the protocol is applicable for all of them.

We see that in Storj, Sia, and Arweave that the network topology is unstruc-

tured, and lookup is not decentralized. In Sia the user is responsible for keeping track of the location of its chunks. In Storj the Satellite peer is responsible for keeping track of where the chunks are stored. In Arweave, the lookup is probabilistic and involves searching the blockweave. SNIPS aims at efficiently synchronizing data among peers. In SNIPS we assumed a structured network where peers are clustered together in neighborhoods. We conjecture that even with an unstructured network topology, the design and methods proposed in SNIPS are relevant for these systems. Hence, we have marked the SNIPS protocol as *Applicable with amendments* for these systems.

Chapter 3

Research Questions

In this section, we build upon the Introduction (Chapter 1) and the Background (Chapter 2) to present the research questions (RQs) covered by this thesis. The overarching research topic in this thesis is to design protocols and techniques to improve data availability in decentralized storage systems. Having a reliable and efficient data storage system is a prerequisite for providing long-term persistence guarantees. Thus, the first research question in the thesis is as follows.

> **RQ 1:** *How can we improve data availability in decentralized storage systems?*

To answer this question, we need to understand the inner workings of decentralized storage systems. While studying the inner workings of decentralized storage systems, we discovered several design choices that result in suboptimal data availability. These design choices include the use of naive replication schemes which result in low storage utilization, ineffective data synchronization between peers, and the lack of storage guarantees for clients. We began our work by evaluating these design choices analytically. We developed analytical models, simulations and equations to understand how we could develop new protocols to improve data availability.

In addition to the analytical work, we also wanted to evaluate our protocols in a real-world setting. Evaluating decentralized protocols in a real-world setting is challenging due to the lack of a standardized testbed. To address evaluating our protocols, we post the second research question.

> **RQ 2:** *How can we evaluate protocols for decentralized storage systems in a real-world setting?*

Research questions **RQ 1** and **RQ 2** are the main research questions of this thesis. We have developed a set of subquestions to answer parts of **RQ 1**, which will be presented in the following subsections. Following the presentation of the research questions, we present a mapping between each of the research questions and our contributions enumerated in the List of Included Papers.

3.1 RQ 3: Applying Erasure Codes to Hierarchical Data Structures

Modern decentralized storage systems such as Swarm [74], InterPlanetary File System (IPFS) [4], and Filecoin [39] split files into chunks and encode the chunks into a Merkle-based structure. The Merkle-based structure comprises lookup-metadata in the form of internal nodes, the chunks of the file as leaves and the root of the tree. In the following, we will use a Merkle tree to explain the research question. However, the research question also applies to the Merkle DAG used in IPFS and Filecoin.

Figure 3.1 illustrates a file encoded as a Merkle tree. Note that the illustration uses a non-conventional circular representation of the Merkle tree to depict many leaf chunks. In the figure, the root node is labeled 25, and its children, 15 and 24 are internal nodes. The leaves are located on the outer periphery and are only accessible through the root and internal nodes.

When retrieving the file, the first request contains the chunk identifier of the root (25). The root is decoded to reveal the chunk identifiers of the internal nodes (15 and 24). These are then requested to reveal the chunk identifiers of either the next level of internal nodes or the leaves. The process terminates when all the leaf chunks ($[1-14]$ and $[16-23]$) are retrieved and the file can be reconstructed.

It is clear that all elements of the Merkle tree are required to retrieve the file. Thus, to add fault tolerance, we need to consider the entire structure. Currently, modern decentralized systems use naive replication schemes to add fault tolerance. However, it has been shown that replication is less storage efficient than erasure coding algorithms [59, 43, 80]. The inefficiency of replication schemes

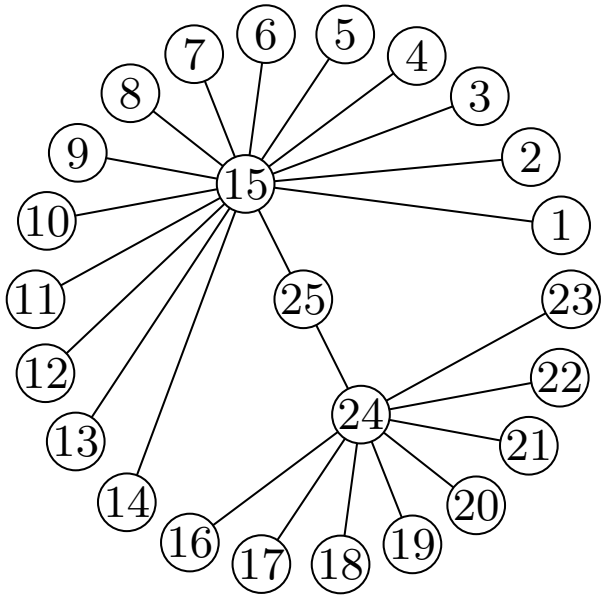


Figure 3.1: File encoded as a Merkle tree with post-order labeling on the nodes.

as compared to erasure codes can result in significantly higher storage consumption. In simple terms, a file that is replicated X times requires X times the storage space. In comparison, a file that is erasure coded may use orders of magnitude less storage space, while still providing the same level of fault tolerance. The tradeoff of erasure coding versus replication is additional computation to encode and decode the file and the need to have decode-metadata, which instructs how to recover from errors.

However, due to these hierarchical dependencies between leaves and internal nodes in the Merkle tree, it is non-trivial to apply erasure codes. Erasure coding algorithms are typically defined on flat structures, where the availability of chunks is independent of each other. To illustrate why applying erasure codes is non-trivial, we consider naively applying erasure coding to a mix of leaves and internal nodes. In the following, we will highlight two challenges with this approach.

First, we end up with a set of parity chunks that would need to be uploaded to the decentralized storage system. After uploading, we would need to record the chunk identifiers into the decode-metadata, so that we could use them when repairs are needed. Only those in possession of the decode-metadata are able to do

repairs. Hence, the decode-metadata would also need to be uploaded to the decentralized storage system. However, this represents a significantly reduced fault tolerance, as the decode-metadata cannot be part of the original erasure code.

Next, an erasure code is often described with two parameters (n, k), with n representing the number of encoded chunks, and k the number of original chunks. In this case, the maximum number of chunks that can be lost is $n - k$. However, if there is a dependency between the chunks, it means that the fault tolerance is significantly reduced, as the maximum number of chunks that can be lost is lower, potentially drastically lower.

To address the challenges above, we present the third research question.

> **RQ 3:** *How can we apply erasure codes when files are encoded in hierarchical data structures, such as Merkle trees?*

3.2 RQ 4: Synchronizing Storage Peers

Peers in decentralized storage systems are grouped together in smaller clusters based on their addresses. The peers inside a cluster aim to store the same data, and hence need to synchronize data with each other.

When a peer joins the storage system for the first time, it is assigned to a cluster. Upon joining, the new peer will attempt to retrieve data stored by other peers in the cluster. Similarly, when a peer temporarily leaves the storage system, and later joins again, it will attempt to synchronize with the other peers.

In current systems, synchronization relies on exchanging long lists of chunk identifiers to determine which chunks a peer is missing. To address the inefficiency of the synchronization process, we present the fourth research question.

> **RQ 4:** *How can we efficiently synchronize data between peers?*

3.3 RQ 5: Client Data Upkeep

Recent studies [53, 30, 75, 29, 15] have shown that the peer churn rate in decentralized storage systems is very high. A consequence of a high churn rate — especially without an efficient method to synchronize between peers, as mentioned in **RQ 4** — is that data is lost. Thus, a client that wants to ensure that their data is stored in the network must periodically re-upload their files to ensure that they

are stored in the network. Re-uploading entire files use an excessive amount of bandwidth, even though only a small fraction of the file's chunks might be lost. The above description leads us to the last research question.

> **RQ 5:** *How can clients efficiently ensure the persistence of their data?*

3.4 Mapping Research Questions to Papers

The relationship between the research questions and the research papers is illustrated in Figure 3.2. Research questions **RQ 3**, **RQ 4** and **RQ 5** are all derived from **RQ 1** and each is answered in a separate research paper. In addition, those papers partially answer **RQ 2**. However, **RQ 2** is also answered in a separate paper where it is the primary topic.

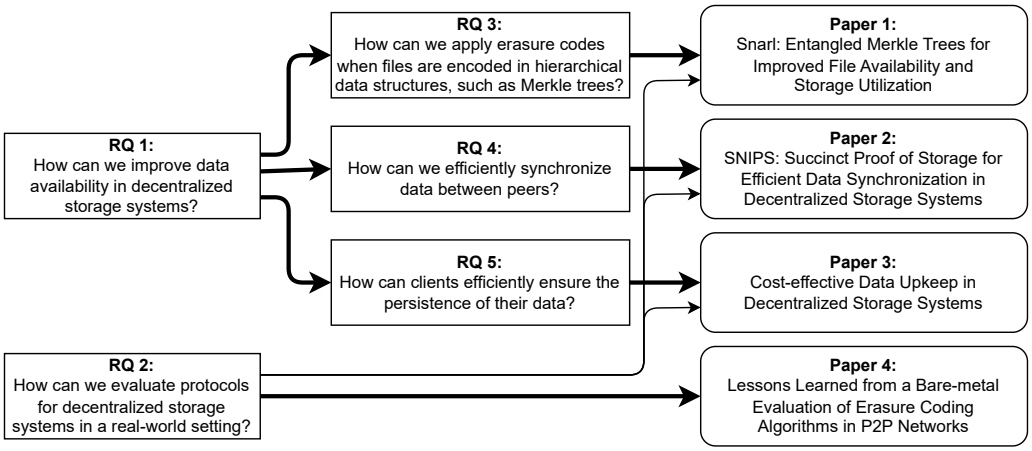


Figure 3.2: Mapping research questions to papers. The thicker line from research questions to papers indicates the main topic of the paper.

3.5 Main Contributions

The main contributions of the thesis and their relation to the research questions are outlined below. We have made the relevant source code for our contributions publicly available, and the experimental setup is described in the research papers. As such, all of our empirical studies should be reproducible.

3.5.1 Main Contribution 1

In **Paper 1** we answer **RQ 3** by proposing entangled Merkle trees and Snarl. An entangled Merkle tree is a resilient data structure that decreases the impact hierarchical dependencies have on data availability. Using the entangled Merkle tree, the root, internal nodes and leaves are encoded together in such a way that data can be recovered even if the root is missing from the original Merkle tree. To facilitate the recovery, the client interacts with the underlying decentralized storage system through Snarl. Snarl's repair algorithm will retrieve corresponding parity chunks from entangled Merkle trees to recover from missing or corrupt chunks. We found that by encoding data as an entangled Merkle tree and using Snarl's repair algorithm, the storage utilization could be improved by over 5 times.

3.5.2 Main Contribution 2

Research question **RQ 4** is answered by the proposal of SNIPS in **Paper 2**. Peers using the SNIPS protocol create a Minimal Perfect Hash Function (MPHF) [20, 42] that is exchanged with neighboring peers. Upon receiving the MPHF, the recipient peer can query it to determine missing chunks. The missing chunks are then requested from the sender to synchronize the peer. Our results show that by using SNIPS, the amount of synchronization data can be reduced by three orders of magnitude, compared to the state-of-the-art.

3.5.3 Main Contribution 3

We answer **RQ 5** in **Paper 3**, where we propose SUP. SUP is designed to reduce the amount of bandwidth required by clients that want to ensure that their data is stored in the decentralized storage system. In current systems, clients may only ensure data availability by re-uploading the entire file. Unnecessary uploading wastes significant amounts of bandwidth. The novelty of SUP is to use proof-of-storage queries to detect missing chunks and then only upload those that were missing. We show that SUP may reduce the amount of data transferred by up 94 % compared to the state-of-the-art.

3.5.4 Main Contribution 4

All contributions were evaluated using a large real-world cluster of 1,000 peers. The cluster consists of 30 bare-metal machines running Ubuntu. Real-world evaluation is difficult and time-consuming, as the systems and protocols are complex.

In **Paper 4** we detail our experience with the evaluations, with a particular focus on evaluating Snarl from **Paper 1**. Improvements to the evaluation framework are described in **Paper 3**. We present a holistic and coherent description of the evaluation framework in Chapter 6.

Chapter 4

Improving Data Availability

In this chapter, we connect our contributions and show how they improve data availability in decentralized storage systems. We recommend reading this chapter after first reading the included papers. A high-level illustration of how the contributions relate to each other is shown in Figure 4.1.

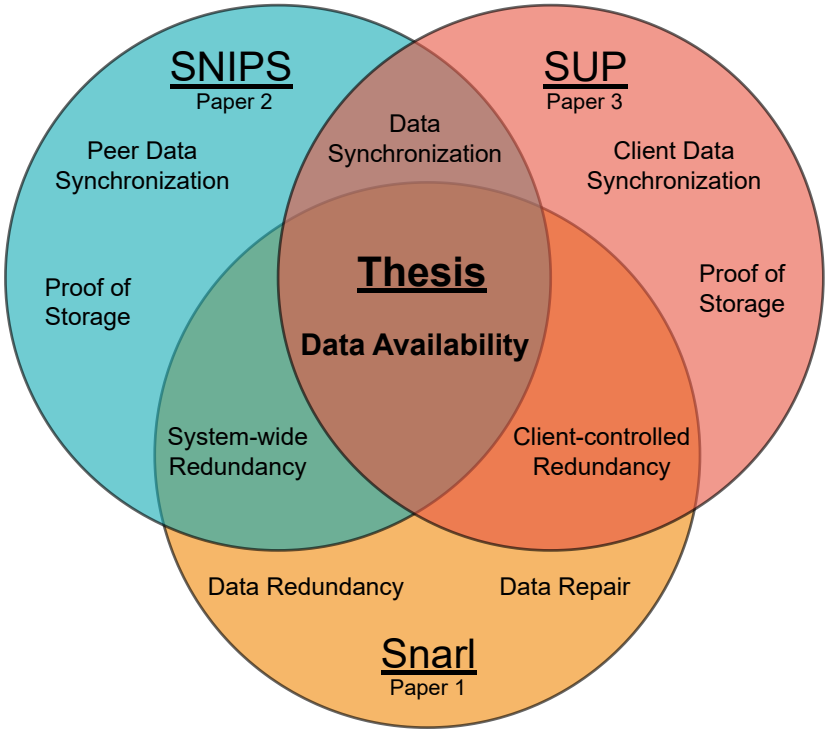


Figure 4.1: The relation between the Snarl, SNIPS, and SUP protocols.

The chapter is organized as follows. We begin with a brief summary of our contributions and relate them to Figure 4.1. Next, we discuss areas in which the contributions are complementary. Finally, we conjecture the potential benefits of integrating the contributions into the same decentralized storage system.

4.1 Brief Summary of Contributions

In **Paper 1** we present entangled Merkle trees and Snarl. Entangled Merkle trees provide data redundancy by creating repair pairs for all chunks in the Merkle tree. Whenever a chunk is missing, Snarl provides data repair by first retrieving a relevant repair pair from the network and then using the repair pair to reconstruct the missing chunk.

In **Paper 2** we present SNIPS, a novel approach to synchronizing data between peers efficiently. With SNIPS, a peer uses a Proof of Storage-like construction to create succinct storage proof for the chunks it wants to synchronize. Each peer then exchanges their storage proofs with other peers. Finally, the peers query the received storage proofs to determine which chunks they are missing and need to synchronize.

Finally, in **Paper 3** we present SUP, a protocol for efficient client data synchronization. SUP uses PoS-queries to determine which of the client's chunks are stored in the network. Then, to ensure persistence the client only needs to re-upload the chunks that are missing in the network.

We evaluated all three protocols under similar conditions on a large cluster of 1,000 peers. In Chapter 6 we present a holistic view of the evaluation setup and the evaluation framework. In the following section, we present some of the main results of our evaluation and discuss areas in which the contributions are complementary.

4.2 Highlighting the Main Evaluation Results

As our three contributed protocols, Snarl, SNIPS and SUP are independent and improve data availability in different ways, they have been evaluated using different metrics. In this section, we will go into further detail on the protocol and then highlight the main evaluation results for each protocol. The section serves as a

prelude to the detailed analysis of the integration of the protocols given in Section 4.3.

Using the entangled Merkle trees and the repair algorithm, Snarl can recover from corrupt or missing chunks. Our results show that compared to current systems, the storage utilization can be improved by over 5 times, with improved resilience against data loss. A file encoded with Snarl to have storage utilization comparable to 14 copies of a file achieves better recovery likelihood than a file replicated 72 times in Swarm. With a network of 1,000 peers, for a 1 MB file, the file encoded in Snarl has a 99 % recovery likelihood, with up to 50 % of the peers being offline or malicious.

While Snarl offers a significant improvement in terms of data availability, it relies on the internal nodes of the entangled Merkle trees being sufficiently replicated in the network to achieve optimal results. In addition, our results show that Snarl is more efficient when the total percentage-wise number of missing chunks is low, i.e. low chunk loss rate. We define *repair ratio* as the number of parity chunks retrieved, divided by the number of missing data chunks. The repair ratio is slightly higher than 2 for low chunk loss rates. For a 1 MB file, the average repair ratio reaches a peak of 2.25 when 25 % of the chunks are missing in the network. The increased repair ratio comes from the fact that entanglement codes need a complete pair to repair. Thus, if only one chunk in the pair is available, it might have been a wasteful download, as the repair algorithm must attempt to make progress using another pair. This suggests that Snarl should attempt to do repairs while the chunk loss rate is low. However, to detect the chunk loss rate, we need a mechanism that can efficiently determine which chunks are missing. Both of our other contributions, SNIPS and SUP, propose protocols that can be used to efficiently determine which chunks are missing. The SNIPS protocol may also be used to ensure sufficient replication of the internal nodes of the entangled Merkle trees.

SNIPS is a protocol that allows peers to efficiently synchronize data with each other. Efficient synchronization is vital for data availability because peer-to-peer networks are often subject to high churn. Our results show that SNIPS can reduce the amount of synchronization data by up to three orders of magnitude compared to current systems. In SNIPS, the peers within a neighborhood periodically exchange our proof of storage construction built on MPHFs. This construction allows the recipient to determine whether it is missing chunks from the sender, and,

if so—request the missing chunks. In addition, SNIPS can be set up so that peers exchange the MPHFs on the same frequency as blocks are committed to a public blockchain. For example, to exchange MPHFs once per day using the Ethereum blockchain, a data synchronization would have to be triggered for each 7200-th block. Thus, the efficient data synchronization of SNIPS complements Snarl, as efficient data synchronization will aid in maintaining sufficient chunk replication.

Our last contribution, SUP, uses PoS-queries to determine which of the client's chunks are stored in the network. The SUP protocol assumes that the client has a local copy of the chunks it wants to confirm the persistence of. For the evaluations, we considered a scenario where a client wants to re-upload its chunks to the network. We found that SUP can reduce the amount of data that needs to be uploaded by up to 94 % compared to current systems. In addition, the time spend re-uploading may be reduced by up to 82 %.

Finally, in **Paper 3**, we did a small study on the public Swarm network to determine the current data availability and rate of decay. Having such monitoring functionality is vital for the success of decentralized storage systems, as it allows the system to adapt to changing network conditions. For example, the system could dynamically adjust the redundancy levels or reduce the time between data synchronizations to maintain the desired data availability.

4.3 Integrating the Contributions

In this section, we conjecture the potential benefit in data availability of integrating the Snarl, SNIPS, and SUP protocols. In the following subsections, we will discuss the benefit of each integration. As a reminder, we illustrate the high-level relationship between the protocols in Figure 4.1. We note that the benefit of integrating the protocols is not necessarily additive, and new evaluations would be needed to determine the actual performance of the integrated protocols.

4.3.1 System-wide Redundancy

An integration between Snarl and SNIPS would complement each other in terms of data availability. First, we would gain fault tolerance against chunk loss from Snarl, and second, we would gain efficient data synchronization from SNIPS.

The benefit of the integration is made clear when considering the following

scenario. If we use SNIPS by itself, then if a chunk is lost by all peers in the network, there is no way to synchronize it back to existence. If we use Snarl by itself, then if the chunks of the entangled Merkle tree are not properly synchronized between the storage peers, the fault tolerance will be significantly reduced.

4.3.2 Client-controlled Redundancy

The integration of Snarl and SUP would allow clients to monitor and maintain the redundancy of their data. With SUP, a client can determine the persistence of their data in the network, and if necessary re-upload missing chunks.

A file encoded with Snarl adds fault tolerance by creating entangled Merkle trees. The new chunks of the entangled Merkle tree are distributed to the storage peers in the same way as the original chunks. Thus, clients can use SUP to determine the persistence and if necessary re-upload missing chunks from the entangled Merkle trees.

We note that Snarl already proposes *user-controlled redundancy*, which is the ability to control the redundancy of their data by choosing the encoding parameters.

4.3.3 Data Synchronization

The SNIPS protocol provides efficient data synchronization between peers, while the SUP protocol provides efficient data synchronization between clients and peers. We conjecture both types of synchronization are beneficial for achieving high data availability.

A key parameter with data synchronization is the interval at which it is performed. In **Paper 3**, we conducted an experiment in the public Swarm network in an attempt to find the upper bound for this interval. Our results show that chunks may become unavailable as soon as 6 days after the initial upload of a file.

These results suggest that a reasonable interval to run SNIPS or SUP could be daily. However, this is subject to change, depending on the network conditions.

4.3.4 Load-Balancing

Although not directly related to data availability, the integration of SNIPS and Snarl would enable improved load-balancing. By load-balancing, we mean that more storage peers are involved in answering a request to retrieve a file.

After SNIPS have been used to synchronize chunks between peers in a neighborhood, each peer will store the same chunks. Thus, these peers may collaborate in answering queries for chunks that belong to this neighborhood.

When Snarl encodes a file into an entangled Merkle tree, parity chunks will be created. These parity chunks will have their own unique identifier and they will be stored in the network in the same way as the original chunks. Thus, when retrieving a file, it is possible to shift the load of answering the query from the peers storing the original chunks to the peers storing the parity chunks.

Hence, we conjecture that the integration of SNIPS and Snarl would enable a more efficient load-balancing of the network.

4.3.5 Full Integration

In the previous sections, we discussed the possible benefits of integrating each pair of protocols. In this section, we discuss integrating all three protocols into the same system.

While we conjecture that the integration of SNIPS, Snarl, and SUP would be beneficial for data availability, it remains future work to design, implement, and evaluate such a system.

As noted in Section 7.2, we suggest that a tighter coupling to the decentralized storage system would be beneficial for the performance of SNIPS. We note that both SNIPS and SUP use *chunk proofs* for creating their respective storage proofs. We will analyze chunk proofs in Chapter 5.

Lastly, with the promising benefits of integrating Snarl and SNIPS, the need for re-uploading chunks may be reduced. However, SUP may still have a role to play in the system, as an independent mechanism to monitor or audit the data availability of the system.

Chapter 5

Security Analysis of Chunk Proofs

In this chapter, we will analyze the security of the *chunk proof* proposed in SUP and also used in SNIPS. Hence, we recommend reading this chapter after reading **Paper 3** and **Paper 2**. As a reminder, a chunk proof is defined as the cryptographic hash of a *nonce* (number used once) concatenated with the chunk's data. We repeat the chunk proof definition for chunk A in Equation (5.1).

$$\text{ChunkProof}_A: \quad cp_A = \text{H}(nonce \,||\, A) \tag{5.1}$$

The overarching goal is to show that a valid chunk proof can only be generated by a prover that has both the nonce and the chunk itself. To achieve this, we define two security games, inspired by multiple sources [3, 85, 8]. In the security games, we define two actors, first a powerful *probabilistic polynomial time* (PPT) adversary $\mathcal{A}$ (i.e., dishonest storage peer) and second a PPT verifier $\mathcal{V}$. We say that $\mathcal{A}$ wins the security game if it can use an invalid proof to pass the verification made by $\mathcal{V}$. If $\mathcal{A}$ is unable to do so, unless, with a negligible probability, we say that $\mathcal{V}$ wins the security game, and the protocol is secure.

In the first security game, $\texttt{Game}_{\texttt{soundness}}$, the adversary wins if it can pass verification without having both the chunk and the correct nonce. In the second security game, $\texttt{Game}_{\texttt{freshness}}$, the adversary wins if it can pass verification after saving storage space by precomputing chunk proofs, then deleting the chunk itself.

5.1 Definitions and Theory

We define some common notations used in security games. In addition, we give the theoretical foundation used in our security analysis. The proofs presented uses an asymptotic approach and are designed for sufficiently large input values. The security level is determined by the bit lengths of $\mathcal{P}_{seed}$, $\mathcal{P}_{prbg}$, and $\mathcal{P}_{hash}$. We make use of the following notation:

- Let $|X|$ denote the size in bits of a data object X.
- Let cp_A denote the chunk proof for chunk A.
- Let ID_A denote the chunk identifier for chunk A.
- Let $\mathcal{KV}$ denote a key-value store of chunk proofs.
- Let λ denote the security parameter.

We identify negligible quantities by using negl as a function that outputs a value less than the inverse of any polynomial. Hence, for every value $u > 0$, we define

$$\text{negl}(\lambda) < \frac{1}{\lambda^u}$$

We assume access to a cryptographic hash function H with the security properties outlined in Section 2.2 that takes an arbitrary length input and that outputs $\mathcal{P}_{hash}$ bits.

$$\text{H} : \{0,1\}^* \rightarrow \{0,1\}^{\mathcal{P}_{hash}}$$

We also assume access to a pseudo-random bit generator (PRBG) that takes a seed and outputs a uniformly random bit string of length $\mathcal{P}_{prbg}$.

$$\text{PRBG} : \{0,1\}^{\mathcal{P}_{seed}} \times \{0,1\}^{\mathcal{P}_{prbg}} \rightarrow \{0,1\}^{\mathcal{P}_{prbg}}$$

Lastly, the proofs make use of the binomial theorem [82]. The binomial theorem states the following algebraic expansion for an integer $k \geq 0$.

$$(1+x)^k = 1 + kx + \frac{k(k-1)}{2!}x^2 + ... + kx^{k-1} + x^k.$$

5.2 Soundness

The security game $\texttt{Game}_{\texttt{soundness}}$ between a *probabilistic polynomial time* (PPT) adversary $\mathcal{A}$ and a PPT verifier $\mathcal{V}$ is defined below. Adversary $\mathcal{A}$ and the verifier $\mathcal{V}$ both store the chunk A which has a unique chunk identifier ID_A. The challenger $\mathcal{C}$ runs the PRBG algorithm to generate a unique nonce. The challenger uses ID_A to request a chunk proof for chunk A. However, the nonce is only sent to $\mathcal{V}$. The adversary $\mathcal{A}$ makes polynomially many attempts at choosing a value M to replace the nonce and use it to generate a chunk proof using Equation (5.1).

$$cp_A \leftarrow \mathrm{H}(M \mid\mid A). \tag{5.1}$$

The verifier $\mathcal{V}$ runs the same algorithm as $\mathcal{A}$ to generate cp_A, but instead of choosing M, it uses the nonce given by challenger $\mathcal{C}$. Adversary $\mathcal{A}$ wins the $\texttt{Game}_{\texttt{soundness}}$ security game if any chunk proof generated by $\mathcal{A}$ is identical to the chunk proof generated by $\mathcal{V}$.

5.2.1 Proof

We want to prove that the $\Pr[\mathcal{A} \text{ wins } \texttt{Game}_{\texttt{soundness}}] \leq \mathrm{negl}(\lambda)$ holds. That is, chunk proofs satisfy the soundness definition, and there cannot exist a PPT adversary $\mathcal{A}$ that can generate valid chunk proofs without knowing the nonce with a higher than negligible probability.

The number of actions by the PPT $\mathcal{A}$ is upper bounded by a polynomial. For a given security parameter λ, we say that the upper bound for the number of chunk proofs generated by $\mathcal{A}$ is given by $O(\lambda^x)$.

There are $2^{\mathcal{P}_{hash}}$ possible distinct chunk proofs. The probability that the adversary $\mathcal{A}$ generates a valid proof and thus wins the security game is given by the following equation.

$$\Pr[\mathcal{A} \text{ wins } \texttt{Game}_{\texttt{soundness}}] = \frac{\lambda^x}{2^{\mathcal{P}_{hash}}}. \tag{5.2}$$

Let $\mathcal{P}_{hash} = \lambda$, meaning there are 2^λ possible distinct chunk proofs. We want to show that $\dfrac{\lambda^x}{2^\lambda}$ approaches 0 as $\lambda \to \infty$. We begin by defining

$$c = 2^{1/x} = 1 + b \quad (\text{for } b > 0)$$

Then we find the x-th root of $\frac{\lambda^x}{2^\lambda}$,

$$\sqrt[x]{\frac{\lambda^x}{2^\lambda}} = \frac{\lambda}{c^\lambda}$$

By the binomial theorem, for $\lambda \geq 2$, we have

$$c^\lambda = (1+b)^\lambda \geq 1 + \lambda b + \frac{\lambda(\lambda-1)}{2!}b^2. \tag{5.3}$$

We re-arrange Equation (5.3) and drop the $1 + \lambda b$ terms since they are negligible compared to the other terms. We thus obtain the inequality

$$\frac{\lambda}{c^\lambda} < \frac{2}{(\lambda-1)b^2},$$

Clearly, the right-hand size

$$\frac{2}{(\lambda-1)b^2}$$

approaches 0 as $\lambda \to \infty$. Hence,

$$0 \leq \frac{\lambda}{c^\lambda} < 1$$

Recall that $c = 2^{1/x}$ and thus

$$(\frac{\lambda}{c^\lambda})^x = \frac{\lambda^x}{2^\lambda} \leq \frac{\lambda}{c^\lambda}$$

Hence, we see that the following fraction approaches 0 as $\lambda \to \infty$.

$$\frac{\lambda^x}{2^{\mathcal{P}_{hash}}}$$

Finally, we see that the following inequality holds.

$$\Pr[\mathcal{A} \text{ wins } \mathtt{Game}_{\mathtt{soundness}}] \leq \text{negl}(\lambda)$$

■

5.3 Freshness

The security game $\texttt{Game}_{\texttt{freshness}}$ is played by a PPT adversary $\mathcal{A}$. We assume the existence of an oracle $\mathcal{O}$ that outputs a valid chunk proof for any given nonce and chunk identifier. Adversary $\mathcal{A}$ can access the oracle and makes polynomially many queries with identifier ID_A and a unique nonce for each query. The chunk proofs are kept in a key-value store, $\mathcal{KV}$, with the nonce as key and cp_A as value.

The challenger $\mathcal{C}$ runs the PRBG algorithm to generate a unique nonce. Adversary $\mathcal{A}$ wins the $\texttt{Game}_{\texttt{freshness}}$ security game if $\mathcal{KV}$ contains a valid chunk proof cp_A for the nonce generated by $\mathcal{C}$ with a higher than negligible probability.

5.3.1 Proof

We want to prove that the $\Pr[\mathcal{A} \text{ wins } \texttt{Game}_{\texttt{freshness}}] \leq \text{negl}(\lambda)$ holds. That is, there cannot exist a PPT adversary $\mathcal{A}$ that can provide a valid chunk proof without storing the chunk with more than a negligible probability. Adversary $\mathcal{A}$ makes polynomially many queries to the oracle $\mathcal{O}$ with identifier ID_A and a unique nonce for each query. Each response is stored in a key-value store, $\mathcal{KV}$. The number of actions by the PPT $\mathcal{A}$ is upper bounded by a polynomial. Hence, for a given security parameter λ, the upper bound for the number of chunk proofs $\mathcal{A}$ can store in $\mathcal{KV}$ is given by $O(\lambda^x)$.

The number of possible distinct nonces is $2^{\mathcal{P}_{prbg}}$. The probability that $\mathcal{KV}$ contains a valid chunk proof for a given nonce gives the probability for the adversary winning the security game

$$\Pr[\mathcal{A} \text{ wins } \texttt{Game}_{\texttt{freshness}}] = \frac{\lambda^x}{2^{\mathcal{P}_{prbg}}}.$$

Let the security parameter $\lambda = \mathcal{P}_{prbg}$. By the same arguments used for the soundness proof in Section 5.2, we can see that the following inequality holds.

$$\Pr[\mathcal{A} \text{ wins } \texttt{Game}_{\texttt{freshness}}] \leq \text{negl}(\lambda)$$

■

5.3.2 Practical Example

In this section, we will illustrate a practical example of the freshness security game as played by a rational storage peer $\mathcal{R}$. We want to show that $\mathcal{R}$ cannot reduce storage consumption by pre-computing chunk proofs for chunk A and storing them in a key-value store $\mathcal{KV}$. The values we have chosen for the security parameters are weaker than what likely would be used in practice. In particular, we have chosen $\mathcal{P}_{prbg}$ = 160 and $\mathcal{P}_{hash}$ = 160, which is the same as in SHA-1 [18], and $|A|$ = 256 KB, which is the same as in IPFS [72].

We argue that the size of a chunk proof is less than the chunk itself. If $|cp_A| \geq |A|$, a rational storage peer $\mathcal{R}$ will save storage space, computation, and bandwidth by proving chunk possession by simply transmitting the chunk. Thus, we can assume $|cp_A| < |A|$.

The security parameter $\mathcal{P}_{hash}$ gives the number of bits required to store each chunk proof, and $\mathcal{P}_{prbg}$ gives the number of bits required to store each nonce. Thus, storing a single pre-computed chunk proof will require at least $\mathcal{P}_{hash}+\mathcal{P}_{prbg}$ bits. Let y denote the number of chunk proofs in the $\mathcal{KV}$ key-value store. Thus, the size of the key-value store is $|\mathcal{KV}| = y(\mathcal{P}_{hash} + \mathcal{P}_{prbg})$ bits.

For a rational storage peer $\mathcal{R}$ to reduce storage consumption the following inequality must hold; $|cp_A| < |\mathcal{KV}| < |A|$. By putting in the numbers for the security parameters, we get the following inequality.

$$160 < 320y < 2097152$$

By solving for y, we see that the inequality holds if $\mathcal{KV}$ contain less than 6553 chunk proofs. The probability that one of these 6553 pre-generated chunk proofs could be used to answer the next challenge is negligible,

$$\Pr[\mathcal{R} \text{ wins } \texttt{Game}_{\texttt{freshness}}] = \frac{6553}{2^{160}} < 2^{-147}.$$

Chapter 6

Empirical Evaluations

In this chapter, we will present a holistic view of our evaluation setup and the evaluation framework we developed to evaluate the performance of our protocols.

Our protocols were not deployed on the public Swarm network. When operating on the public Swarm network, there is no way to control the number of peers in the network, the amount of data stored by them, their geographical location, reliability, and their behavior in general. Hence, for evaluations, we used a private network of 1,000 Swarm peers that closely resembles the public network. We believe this was necessary to ensure fair and reproducible results for each individual protocol.

As a step in the direction of reproducible research, we have made the relevant source code for the evaluation framework and necessary configuration files publicly available together with the other software artifacts of this thesis. We note that some of the content in this section overlaps with the included papers.

6.1 Evaluation Setup

The experiments were conducted on a cluster of 30 physical machines running Ubuntu 18.04.4 LTS. Each machine sports an Intel Xeon E-2136 3.30 GHz CPU, 32 GB RAM, a 1.5 TB SSD disk, and 10 Gbit/s NIC. We used the cluster to run a large network of 1,000 Swarm peers. Using Kubernetes [36] and Helm [28], we distributed the Swarm peers on 28 machines, using one to host a private Ethereum network and one for managing the experiment execution. To collect metrics from each Swarm peer, we used Prometheus [55], a monitoring system

and time series database. In addition, we used Grafana [25] to visualize the metrics. Parsing and aggregating the collected results from the 1,000 peers are primarily done via Python scripts. Figure 6.1 shows a screenshot from Grafana taken during the experiment execution. We can see that global Pullsync activity switches between activity and inactivity. This switching is due to the experiments with different parameters being executed. Using such visualization tools, we can easily monitor the evaluation execution to determine whether the experiments are running as expected.

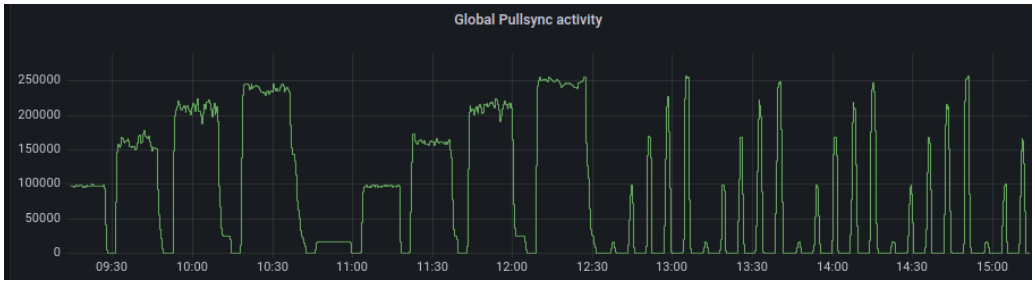


Figure 6.1: Screenshot from Grafana showing the accumulated Pullsync activity.

Helm uses a single configuration file as input to a dozen configuration files during the cluster's initialization. In the single configuration file, we specify parameters such as storage capacity, cluster placement, scaling, run-time parameters of Swarm, and much more. We base our Helm scripts on those provided by the Ethereum Swarm organization [73].

Peers running in Kubernetes are ephemeral by default, with limited options for persistent storage. The before-mentioned Helm scripts only supported persistent storage options for cloud providers, but not for private clusters. To provide persistent storage to the peers from the local SSD, we had to create a *Persistent Volume* (PV). Each peer has a dedicated PV, and each PV is linked to a physical location on the SSD. We used the modulus operator in the Helm configuration file to create the link so that peer y's data was allocated to a PV assigned to machine

$$m = 3 + y \mod 28.$$

This elegantly avoids using the first two machines $m \in \{1, 2\}$ for storing peer data, as they are used for other purposes. We used the same logic to place the Swarm peers to avoid using the first two machines.

6.2 Evaluation Framework

Our evaluation framework was developed and improved upon in several incremental steps throughout the PhD period. The evaluation framework has three main components:

1. Peer state manipulation
2. Experiment execution
3. Metrics collection

By manipulating the peers' storage, i.e., adding and removing chunks, we can evaluate how the protocols perform when chunks are lost in the network and how they recover from such loss. Our framework supports modifying the peers' online status, which allows us to evaluate how the protocols perform under varying degrees of churn. The framework also uses the state manipulation component to ensure identical system states across experiment runs, reducing the chance of bias in the results.

The framework is also used to interact with peers and execute the experiments. The execution of experiments may be initiated by the command line or through an API.

Lastly, the framework is used to collect metrics from peers to obtain the result of an experiment. We obtain the metrics by querying the peers right before the experiment is run and then again after the experiment has been executed. This allows us to determine the impact of the experiment as we obtain the delta difference from the experiment execution. The framework can aggregate metrics from all peers into a single output file, which subsequently may be used to generate graphs and tables.

Earlier versions of the evaluation framework were primarily based on Bash and Python. The Bash scripts were used to execute and manage the experiments and to control the Kubernetes pods directly. However, we found that to modify the state of the peers, e.g., to delete or add chunks to the local storage, the pods could not be running simultaneously. Thus, this approach was not ideal, since terminating pods is very time-consuming. To continue an experiment execution, we had to start the pods again and wait until the Swarm instances were discovered and connected so that the Swarm network was fully operational.

The current version of the framework is written in Go and offers improvements over the previous versions. One of the main improvements is that the framework can modify the state of peers without terminating them, thereby significantly reducing the time to execute experiments. To manipulate the state, we first integrated our framework with Swarm's API. Then, we extended Swarm's API with a new snapshot feature. The snapshot feature allows us to create snapshots of the local storage and later restore the state from the snapshots. Using such snapshots, we ensure an identical system state across experiment runs and thus increase the confidence in the results we obtain. To interact with the Kubernetes pods, the current version uses the Kubernetes port-forwarding API instead of relying on the Kubernetes CLI.

When running experiments, the framework is given a set of configuration parameters to describe the experiment, such as chunk loss, protocols, file addresses, and snapshots. The framework is then able to repeat the experiment multiple times, often running several days before completion. We conclude by listing features of the current evaluation framework:

- **Dataset.** Generate and upload files to the Swarm network based on a dataset specification. This functionality accepts a JSON configuration file with the desired amount of files, their size and optionally the address prefix of the chunks. After uploading, the framework returns the Swarm addresses of the files.

- **Distribution.** Collects data on the replication degree in the network. This feature enables us to have a fine-grained view of which peers are storing which chunks. It may be queried using a file's address, and will then return the list of peers storing the chunks that comprise that file.

- **Experiment execution.** Start running the experiment. Supported protocols are Data Stewardship, Pullsync, SUP, and SNIPS from **Paper 2** and **Paper 3**.

- **Listing chunk identifiers.** Retrieve a list of chunk identifiers referenced by a file's Swarm address. This list of chunk identifiers can be used to decide what chunks to remove from Swarm peers to experiment with chunk loss.

- **Metrics collection.** Collects performance and protocol-specific metrics

from the Swarm peers. The metrics are used to calculate the results from experiments.

- **Neighborhood information.** Queries each peer for their neighborhood information and creates a relation graph. This feature was used to create the image on the front page and the graphs in Figure 2.7.
- **Protocols.** Enable and disable protocols. This can be used to study the protocols separately.
- **Snapshots.** Create snapshots containing the addresses of all chunks stored by each Swarm peer. The snapshots can later be uploaded and applied to restore each peer to its previous state.

Chapter 7

Conclusion, Reflections and Future Prospects

The research presented in this dissertation contributes to the field of decentralized storage systems. In particular, it proposes novel protocols for improving data availability. The protocols have been shown to be practical and have been extensively evaluated. The following contains a list of the major highlights and takeaways of this dissertation.

- In **Paper 1** we propose Snarl. Snarl creates entangled Merkle trees to protect both data and metadata against data loss. In addition, Snarl offers decentralized repair, where peers can repair data without the need for a central authority.
- In **Paper 2** we propose SNIPS for peer data synchronization. SNIPS use MPHFs to send succinct proofs to neighboring peers. The neighbors then query the proof to find missing chunks.
- In **Paper 3** we propose SUP, a protocol for cost-effective uploads of data. Using SUP, a client sends Proof of Storage challenges, to determine if a chunk already exists in the network before uploading it.
- In **Paper 4** we present some aspects of our evaluation framework. In addition, the evaluation framework is discussed holistically in Chapter 6.
- We have discussed how the three protocols may be integrated to provide even higher data availability in Chapter 4.

- We have discussed that the protocols are applicable to a range of decentralized storage systems in Section 2.9.6.
- In Chapter 5 we presented security proofs for the chunk proofs used in SNIPS and SUP.

7.1 Reflections

The contributions presented in this thesis are the result of several years of research. As the research targeted cutting-edge technology within the field of decentralized storage systems, we encountered several challenges. In this chapter, we will reflect on how we overcame the challenges and discuss the limitations of our work.

We started using Ethereum Swarm to build and evaluate our protocols in 2019. During this time, the current version of Swarm was v0.4.1, and its code was still a package in the Ethereum code base. After a few months, the Swarm team decided to move the code to a separate repository. We continued our work with Swarm through its ”Alpha”, ”Beta”, ”testnet” and ”mainnet” releases, with the current Swarm release in our latest paper, **Paper 2**, being v1.9.0.

Doing research so closely on a cutting-edge software project and under such rapid development was challenging and time-consuming. We constantly had to adapt our protocol implementations to the changes in the Swarm code base. For example, the initial versions of Swarm used a different networking framework and did not have an incentive system.

In addition, the releases had their fair share of bugs and issues, which we had to work around. As documentation was scarce or rapidly outdated, we regularly had to resort to reading the source code and the latest commits to understand the changes.

Nevertheless, working on Swarm was interesting and a great learning experience. We thank the members of the Swarm team for helpful discussions and technical assistance throughout the years.

7.2 Future Prospects

The field of decentralized storage systems is still in its infancy and is under active research and development. Even though this involves constantly changing landscapes and conditions, we conjecture that the techniques and designs presented in the protocols of this dissertation may be applicable to both current and future generations of decentralized storage systems.

In the following, we discuss some future prospects of our work. We separate future prospects into two categories. The first category is protocol improvements, which are proposals for improving the presented protocols. In the second category, we speculate on how our work could be extended to areas outside decentralized storage systems.

7.2.1 Protocol Improvements

In this section, we discuss some future prospects for continuing the work and improving the presented protocols.

First off, we discussed the potential benefit of integrating our protocols in Chapter 4 for increased data availability. This integration may be used to extend an existing decentralized storage system or serve as a basis for a new one. It remains a future prospect to design and implement such a system.

Another related future prospect is to extend the protocols to other decentralized storage systems than Swarm. We have conjectured their applicability to other decentralized storage systems in Section 2.9.6.

Snarl was designed with the goal that no changes should be made to the Swarm code base. This design goal was necessary to make Snarl independent of Swarm and allowed it to be applied to and achieve similar results in IPFS. However, we believe that a tighter coupling to the storage system may lead to even better results. By having Snarl as an integral part of the storage system, it would be possible to periodically check the integrity of the data and its parities and repair them if necessary.

Snarl was designed to allow the end-user to choose the desired level of resilience. However, the evaluations in **Paper 1** were done using a single setting, $\alpha = 3$, $s = 5$, $p = 5$. Further research is needed to determine the optimal settings for different use cases.

The repair algorithm in Snarl has the potential for high degrees of parallelism. However, the current implementation has limited parallelism. In particular for parity chunk retrieval and simple repairs. A future prospect is to apply parallelism to complex repair operations. In addition, it remains future work to prove the correctness of the repair algorithm.

In SNIPS, we use MPHFs to send succinct proofs to neighboring peers. We did not implement the MPHF ourselves but used an existing Go-implementation [31]. However, there are several variants of MPHFs, each offering different performance characteristics. To further improve the performance of SNIPS, future work remains to consider the optimal tradeoff between characteristics, and if necessary implement our own MPHF.

When multiple peers are finding missing chunks, the current implementation of SNIPS does not take advantage of the same gossip mechanism as Pullsync. In the future, we may implement this to further improve the performance of SNIPS.

In the SUP protocol specification, the PoS challenges are batched. However, as noted in the paper, in the implementation, challenges are not batched. In the future, we may implement batching to further improve the performance of SUP.

7.2.2 Extending Our Work to Other Areas

In this section, we discuss how our work could be extended to areas outside decentralized storage systems.

First, we conjecture that SNIPS and the PoS-like construction may find applications outside our proposed use case. In **Paper 2** we focused on a single use case, namely data synchronization in decentralized storage systems. However, the need for data synchronization is not limited to decentralized storage systems. Hence, the techniques that SNIPS propose may be applicable for data synchronization in cloud services, distributed databases and other distributed systems. In addition, we conjecture that SNIPS may be used for other use cases, such as for monitoring the network's global redundancy level and for incentives.

Next, clients using the SUP protocol can verify the persistence of their data in the decentralized storage system. We conjecture that this may be extended to any storage system. A client may be interested in verifying the persistence of their data, irrespective if the storage system is trusted or not.

Bibliography

[1] Dimitri Antonenko. *Centralized vs Decentralized vs Distributed Networking Explained*. Accessed: 2023-03-14. Feb. 2022. URL: https://www.businesstechweekly.com/operational-efficiency/computer-networking/centralized-vs-decentralized-vs-distributed-networking-explained/.

[2] *Arweave Explorer | ViewBlock*. Accessed: 2023-03-15. URL: https://viewblock.io/arweave.

[3] Giuseppe Ateniese et al. "Provable Data Possession at Untrusted Stores". In: *Proceedings of the 14th ACM conference on Computer and communications security*. New York, NY, USA: Association for Computing Machinery, 2007, pp. 598–609.

[4] Juan Benet. "IPFS - Content Addressed, Versioned, P2P File System". In: *arXiv* abs/1407.3561 (2014).

[5] Juan Benet, David Dalrymple, and Nicola Greco. "Proof of Replication". In: (2017).

[6] *Best Dapp Using Swarm Decentralized Storage*. Accessed: 2023-03-14. 2022. URL: https://gitcoin.co/issue/fairdatasociety/wam/1/100027830.

[7] *Best Search Mechanism For Wikipedia Offline Snapshots*. Accessed: 2023-03-14. 2022. URL: https://gitcoin.co/issue/fairdatasociety/wam/19/100027845.

[8] Dan Boneh and Victor Shoup. "A Graduate Course in Applied Cryptography". In: *Draft 0.6* (2023). Accessed: 2023-03-17. URL: http://toc.cryptobook.us/.

[9] *Buzz is in the air — Swarm grants results are in!* Accessed: 2023-03-14. 2020. URL: https://medium.com/ethereum-swarm/buzz-is-in-the-air-swarm-grants-results-are-in-a030ab9178a9.

[10] Bengt Carlsson and Rune Gustavsson. "The Rise and Fall of Napster-an Evolutionary Approach". In: *Active Media Technology: 6th International Computer Science Conference, AMT 2001 Hong Kong, China, December 18–20, 2001 Proceedings 6*. Springer. 2001, pp. 347–354.

[11] Ian Clarke et al. "Freenet: A Distributed Anonymous Information Storage and Retrieval System". In: *Designing privacy enhancing technologies: international workshop on design issues in anonymity and unobservability Berkeley, CA, USA, July 25–26, 2000 Proceedings*. Springer. 2001, pp. 46–66.

[12] Bram Cohen. "Incentives Build Robustness in BitTorrent". In: *Workshop on Economics of Peer-to-Peer systems*. Vol. 6. Berkeley, CA, USA. 2003, pp. 68–72.

[13] Frank Dabek et al. "Wide-area cooperative storage with CFS". In: *ACM SIGOPS Operating Systems Review* 35.5 (2001), pp. 202–215.

[14] Erik Daniel and Florian Tschorsch. "IPFS and Friends: A Qualitative Comparison of Next Generation Peer-to-Peer Data Networks". In: *IEEE Communications Surveys & Tutorials* 24.1 (2022), pp. 31–52.

[15] Erik Daniel and Florian Tschorsch. "Passively Measuring IPFS Churn and Network Size". In: *arXiv preprint arXiv:2205.14927* (2022).

[16] David Dias, Jeromy Johnson, and Juan Benet. *Bitswap - Protocol Specification*. Accessed: 2023-03-14. URL: https://github.com/ipfs/specs/blob/main/BITSWAP.md.

[17] Inc. Dropbox. *Dropbox*. Accessed: 2023-03-14. URL: https://www.dropbox.com/.

[18] Donald Eastlake 3rd and Paul Jones. "US Secure Hash Algorithm 1 (SHA1)". In: *Network Working Group RFC3174*. 2001.

[19] *Entanglements In Swarm*. Accessed: 2023-03-14. 2019. URL: https://devpost.com/software/entanglementsinswarm.

[20] Emmanuel Esposito, Thomas Mueller Graf, and Sebastiano Vigna. “RecSplit: Minimal Perfect Hashing via Recursive Splitting”. In: *2020 Proceedings of the Twenty-Second Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM. 2020, pp. 175–185.

[21] Vero Estrada-Galinanes, Jehan-François Pâris, and Pascal Felber. “Simple Data Entanglement Layouts With High Reliability”. In: *2016 IEEE 35th International Performance Computing and Communications Conference (IPCCC)*. IEEE. 2016, pp. 1–8.

[22] 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)*. IEEE. 2018, pp. 183–194.

[23] Niels Ferguson et al. “The Skein Hash Function Family”. In: *Submission to NIST (round 3)* 7.7.5 (2010), p. 3.

[24] Roy Friedman, Yoav Kantor, and Amir Kantor. “Replicated Erasure Codes for Storage and Repair-Traffic Efficiency”. In: *14-th IEEE International Conference on Peer-to-Peer Computing*. IEEE. 2014, pp. 1–10.

[25] Grafana Labs. *Grafana: The open observability platform*. Accessed: 2023-03-17. URL: `https://grafana.com/`.

[26] Janoš Guljaš. *Network Statistics - Swarm Scan*. Accessed: 2023-03-14. URL: `https://swarmscan.resenje.org`.

[27] Bernhard Heep. “R/Kademlia: Recursive and Topology-aware Overlay Routing”. In: *2010 Australasian Telecommunication Networks and Applications Conference*. IEEE. 2010, pp. 102–107.

[28] Helm. *The package manager for Kubernetes*. Accessed: 2023-03-14. 2023. URL: `https://helm.sh/`.

[29] Sebastian Henningsen et al. “Crawling the IPFS Network”. In: *2020 IFIP Networking Conference (Networking)*. IEEE. 2020, pp. 679–680.

[30] Sebastian Henningsen et al. “Mapping the Interplanetary Filesystem”. In: *2020 IFIP Networking Conference (Networking)*. IEEE. 2020, pp. 289–297.

[31] Sudhi Herle. *Fast Scalable Minimal Perfect Hash for Large Keysets*. Accessed: 2023-03-13. URL: https://github.com/opencoff/go-bbhash.

[32] Miranda Honnoll. *The World Is Moving Beyond Big Data, According to Ocient Survey of 500 Data and Technology Leaders*. Accessed: 2023-03-14. Aug. 2022. URL: https://www.businesswire.com/news/home/20220809005494/en/The-World-Is-Moving-Beyond-Big-Data-According-to-Ocient-Survey-of-500-Data-and-Technology-Leaders.

[33] William Cary Huffman and Vera Pless. *Fundamentals of Error-Correcting Codes*. Cambridge university press, 2010.

[34] Google Inc. *Google Drive*. Accessed: 2023-03-14. URL: https://www.google.com/drive/.

[35] Ari Juels and Burton S Kaliski Jr. "PORs: Proofs of Retrievability for Large Files". In: *Proceedings of the 14th ACM Conference on Computer and Communications Security*. New York, NY, USA: Association for Computing Machinery, 2007, pp. 584–597.

[36] Kubernetes. *Production-Grade Container Orchestration*. Accessed: 2023-03-14. 2022. URL: https://kubernetes.io/.

[37] John Kubiatowicz et al. "Oceanstore: An Architecture for Global-Scale Persistent Storage". In: *ACM SIGOPS Operating Systems Review* 34.5 (2000), pp. 190–201.

[38] Protocol Labs. *Automated data availability and redundancy on IPFS*. Accessed: 2023-03-11. URL: https://ipfscluster.io/.

[39] Protocol Labs. *Filecoin: A Decentralized Storage Network*. Technical report. Accessed: 2023-03-14. July 2017. URL: https://filecoin.io/filecoin.pdf.

[40] Protocol Labs. *Lotus: Hardware requirements*. Accessed: 2023-03-19. URL: https://lotus.filecoin.io/storage-providers/get-started/hardware-requirements/.

[41] Protocol Labs. *Protocol Labs is building the next generation of the internet*. Accessed: 2023-03-18. URL: https://protocol.ai/.

[42] Antoine Limasset et al. "Fast and scalable minimal perfect hashing for massive key sets". In: *arXiv preprint arXiv:1702.03154* (2017).

[43] WK Lin, Dah Ming Chiu, and YB Lee. “Erasure Code Replication Revisited”. In: *Proceedings. Fourth International Conference on Peer-to-Peer Computing, 2004. Proceedings.* IEEE. 2004, pp. 90–97.

[44] Petar Maymounkov and David Mazières. “Kademlia: A Peer-to-Peer Information System Based on the XOR Metric”. In: *International Workshop on Peer-to-Peer Systems*. Berlin, Heidelberg: Springer, 2002, pp. 53–65.

[45] Ralph C Merkle. “A Digital Signature Based on a Conventional Encryption Function”. In: *Conference on the theory and application of cryptographic techniques*. Springer. 1987, pp. 369–378.

[46] Microsoft. *Microsoft OneDrive*. Accessed: 2023-03-14. URL: `https://onedrive.live.com/`.

[47] Satoshi Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System*. 2008. URL: `https://bitcoin.org/bitcoin.pdf`.

[48] Arvind Narayanan et al. *Bitcoin and Cryptocurrency Technologies: A Comprehensive Introduction*. Princeton, NJ, USA: Princeton University Press, 2016.

[49] Bill Nowicki. “NFS: Network File System Protocol Specification”. In: *Network Working Group RFC1094*. 1989.

[50] Racin Nygaard. “Lessons Learned from a Bare-metal Evaluation of Erasure Coding Algorithms in P2P Networks”. In: *arXiv* abs/2208.12360 (2022).

[51] Racin Nygaard, Vero Estrada-Galiñanes, and Hein Meling. “Snarl: Entangled Merkle Trees for Improved File Availability and Storage Utilization”. In: *Proceedings of the 22nd International Middleware Conference*. Middleware ’21. Québec city, Canada, 2021, pp. 236–247.

[52] Racin Nygaard and Hein Meling. “SNIPS: Succinct Proof of Storage for Efficient Data Synchronization in Decentralized Storage Systems”. In: *arXiv* abs/2304.04891 (2023). URL: `https://arxiv.org/abs/2304.04891`.

[53] Racin Nygaard, Hein Meling, and John Ingve Olsen. “Cost-effective Data Upkeep in Decentralized Storage Systems”. In: *Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing*. Tallinn, Estonia, 2023.

[54] *Pentesting Ethereum Contracts With Ganache*. Accessed: 2023-03-14. 2019. URL: `https://twitter.com/trufflesuite/status/1181790431023157249`.

[55] Prometheus. *Prometheus - Monitoring system & time series database*. Accessed: 2023-03-17. URL: https://prometheus.io/.

[56] Saad Qadeer. *What are centralized, decentralized, and distributed systems?* Accessed: 2023-03-14. URL: https://www.educative.io/answers/what-are-centralized-decentralized-and-distributed-systems.

[57] Irving S Reed and Gustave Solomon. "Polynomial Codes Over Certain Finite Fields". In: *Journal of the society for industrial and applied mathematics* 8.2 (1960), pp. 300–304.

[58] Matei Ripeanu. "Peer-to-Peer Architecture Case Study: Gnutella Network". In: *Proceedings first international conference on peer-to-peer computing*. IEEE. 2001, pp. 99–100.

[59] Rodrigo Rodrigues and Barbara Liskov. "High Availability in DHTs: Erasure Coding vs. Replication". In: *International Workshop on Peer-to-Peer Systems*. Springer. 2005, pp. 226–239.

[60] Antony Rowstron and Peter Druschel. "Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems". In: *Middleware 2001: IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg, Germany, November 12–16, 2001 Proceedings 2*. Springer. 2001, pp. 329–350.

[61] Mehul A Shah et al. "Auditing to Keep Online Storage Services Honest". In: *Proceedings of the 11th USENIX Workshop on Hot Topics in Operating Systems*. USA: USENIX Association, 2007.

[62] *Sia Docs*. Accessed: 2023-03-11. URL: https://docs.sia.tech/.

[63] *SiaStats.info - Index*. Accessed: 2023-03-14. URL: https://siastats.info/.

[64] William Stallings. *Cryptography And Network Security: Principles and Practice, 7th Edition*. Upper Saddle River, NJ, USA: Pearson Education, 2017.

[65] Eivind Stavnes and Daniel Urdal. "Extending the Snarl File Repair Component for Distributed Storage Systems". MA thesis. University of Stavanger, 2021.

[66] Ion Stoica et al. “Chord: A Scalable Peer-to-Peer Lookup Protocol for Internet Applications”. In: *IEEE/ACM Transactions on networking* 11.1 (2003), pp. 17–32.

[67] Inc. Storj Labs. *Be the decentralized cloud.* Accessed: 2023-03-18. URL: https://www.storj.io/node.

[68] Inc. Storj Labs. *Satellite (Metadata Region)*. Accessed: 2023-03-12. URL: https://docs.storj.io/dcs/concepts/satellite.

[69] Inc. Storj Labs. *Storj Node Operator*. Accessed: 2023-03-12. URL: https://docs.storj.io/node/before-you-begin/prerequisites.

[70] Inc. Storj Labs. *Storj: A Decentralized Cloud Storage Network Framework*. Accessed: 2023-03-11. Oct. 2018. URL: https://www.storj.io/whitepaper.

[71] Andrew S Tanenbaum and Robbert Van Renesse. “Distributed Operating Systems”. In: *ACM Computing Surveys (CSUR)* 17.4 (1985), pp. 419–470.

[72] IPFS team. *Kubo CLI | IPFS Docs*. Accessed: 2023-03-13. URL: https://docs.ipfs.tech/reference/kubo/cli/#ipfs-add.

[73] Swarm team. *Ethersphere Helm Charts*. Accessed: 2023-03-17. URL: https://github.com/ethersphere/helm-charts.

[74] Swarm team. *Storage and Communication Infrastructure for a Self-Sovereign Digital Society*. Accessed: 2023-03-11. 2021. URL: https://www.ethswarm.org/swarm-whitepaper.pdf.

[75] Dennis Trautwein et al. “Design and Evaluation of IPFS: A Storage Layer for the Decentralized Web”. In: *Proceedings of the ACM SIGCOMM 2022 Conference*. 2022, pp. 739–752.

[76] Viktor Trón. *The Book of Swarm - v1.0 pre-release 7 November 17, 2020*. Accessed: 2023-03-11. 2020. URL: https://www.ethswarm.org/The-Book-of-Swarm.pdf.

[77] David Vorick and Luke Champine. “Sia: Simple Decentralized Storage”. In: (2014). Accessed: 2023-03-11. URL: https://sia.tech/sia.pdf.

[78] Shermin Voshmgir. *Token Economy: How the Web3 reinvents the Internet*. Vol. 2. BlockchainHub Berlin, 2020.

[79] *WAM Hackathon 2022 Winners - Fair Data Society*. Accessed: 2023-03-14. 2022. URL: https://github.com/fairDataSociety/WAM/blob/main/WAM%20Hackathon%202022%20Winners%20-%20Fair%20Data%20Society.md.

[80] Hakim Weatherspoon and John D Kubiatowicz. "Erasure Coding vs. Replication: A Quantitative Comparison". In: *International Workshop on Peer-to-Peer Systems*. Springer. 2002, pp. 328–337.

[81] Sage A Weil et al. "Ceph: A Scalable, High-Performance Distributed File System". In: *Proceedings of the 7th symposium on Operating systems design and implementation*. 2006, pp. 307–320.

[82] Wikipedia contributors. *Binomial theorem — Wikipedia, The Free Encyclopedia*. Accessed: 2023-03-12. 2022. URL: https://en.wikipedia.org/w/index.php?title=Binomial_theorem&oldid=1105633884.

[83] Zooko Wilcox-O'Hearn and Brian Warner. "Tahoe – The Least-Authority Filesystem". In: *Proceedings of the 4th ACM international workshop on Storage security and survivability*. 2008, pp. 21–26.

[84] Sam Williams et al. "Arweave: A Protocol for Economically Sustainable Information Permanence". In: *arweave. org, Tech. Rep* (2019).

[85] Anjia Yang et al. "Lightweight and Privacy-Preserving Delegatable Proofs of Storage with Data Dynamics in Cloud Storage". In: *IEEE Transactions on Cloud Computing* 9.1 (2021), pp. 212–225.

[86] Ben Y Zhao et al. "Tapestry: A Resilient Global-Scale Overlay for Service Deployment". In: *IEEE Journal on selected areas in communications* 22.1 (2004), pp. 41–53.

Chapter 8

Included Papers

This chapter contains the included research papers. Each paper is preceded by a title page that shows its publication status.

Paper 1

Snarl: Entangled Merkle Trees for Improved File Availability and Storage Utilization

Racin Nygaard, Vero Estrada-Galiñanes, Hein Meling

Published in ACM Middleware 2021 [51]

Abstract

In cryptographic decentralized storage systems, files are split into chunks and distributed across a network of peers. These storage systems encode files using Merkle trees, a hierarchical data structure that provides integrity verification and lookup services. A Merkle tree maps the chunks of a file to a single root whose hash value is the file's content-address.

A major concern is that even minor network churn can result in chunks becoming irretrievable due to the hierarchical dependencies in the Merkle tree. For example, chunks may be available but can not be found if all peers storing the root fail. Thus, to reduce the impact of churn, a decentralized replication process typically stores each chunk at multiple peers. However, we observe that this process reduces the network's storage utilization and is vulnerable to cascading failures as some chunks are replicated $10\times$ less than others.

We propose *Snarl*, a novel storage component that uses a variation of alpha entanglement codes to add user-controlled redundancy to address these problems. Our contributions are summarized as follows: 1) the design of an entangled Merkle tree, a resilient data structure that reduces the impact of hierarchical dependencies, and 2) the Snarl prototype to improve file availability and storage utilization in a real-world storage network. We evaluate Snarl using various failure scenarios on a large cluster running the Ethereum Swarm network. Our evaluation shows that Snarl increases storage utilization by $5\times$ in Swarm with improved file availability. File recovery is bandwidth-efficient and uses less than $2\times$ chunks on average in scenarios with up to 50 % of total chunk loss.

1 Introduction

Large-scale decentralized peer-to-peer (p2p) storage systems, such as *InterPlanetary File System* (IPFS) [2] and *Ethereum Swarm* [27], aim at providing novel services to satisfy current and future storage and communication needs. In such p2p systems, integrity verification plays a pivotal role [23], as peers are untrusted and the network may exhibit significant churn. Further, each peer typically only stores a subset of each file in fixed-sized chunks, and each chunk is given a content address for later retrieval. A content address is unique; it is the cryptographic hash of the chunk's data. To assist in this integrity verification and retrieval, a Merkle tree [18] data structure is typically used, where the tree's nodes are distributed across the peers of the p2p network based on their content address.

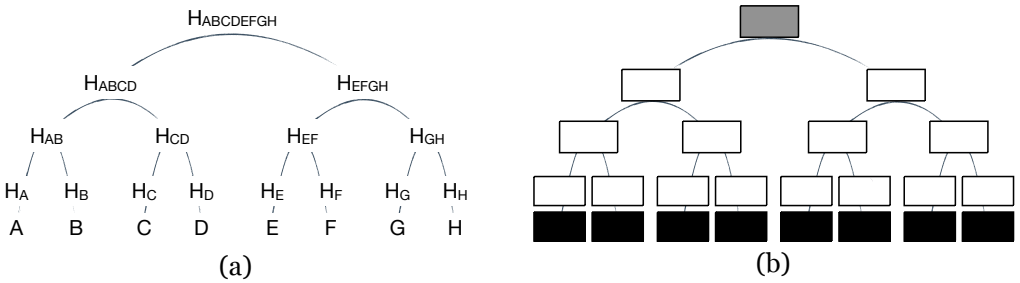


Figure 1: Merkle tree: (a) Binary Merkle tree built with 8 data chunks (A-H); (b) content addressing application.

A Merkle tree is a cryptographic data structure that maps multiple elements (nodes) to a single root node, as seen in Figure 1a. Internal nodes are obtained by hashing their children recursively using a bottom-up approach, so the root hash value covers the entire tree. The tree's integrity can be verified using the root hash value as any modification is propagated up. Merkle trees are widely used as they minimize audit costs and facilitate a wide range of applications, as mentioned in §2.

Figure 1b shows a file in a content-addressing system, such as the aforementioned Swarm and IPFS, where all content is placed at the lowest level of the tree, while the internal nodes and root are metadata required for lookup. Any given file will be represented by a unique Merkle tree, where the chunks of the file are the content at the lowest level. Internal nodes and the root are also stored as chunks in the p2p network, and contain a concatenation of the content addresses of their children. Thus, to retrieve the content, we must first request the root using its

content address and then recursively retrieve internal nodes until we reach the leaves. Consequently, the loss of the root or an internal node causes a cascading failure as it is impossible to request their children without the content address.

The data loss in real systems is further exacerbated because they use k-ary Merkle trees with a large branching factor, k. To illustrate the problem, Figure 2 depicts a radial visualization of the 128-ary Merkle tree for a 100 MB file stored in our 1000-peer Swarm cluster. The tree contains 24426 leaves, 191 internal nodes at level 1, two internal nodes at level 2, and the root at level 3. This figure also captures information about imbalances caused by the decentralized replication process in Swarm. Specifically, the replication factor (R) for each chunk is in the range of $R \in [9, 160]$. The root, labeled "1", has $R = 132$, with its two children labeled "2" and "3" having $R = 108$ and $R = 31$, respectively. This means that all data below internal node "3" is far more vulnerable than the data below "2".

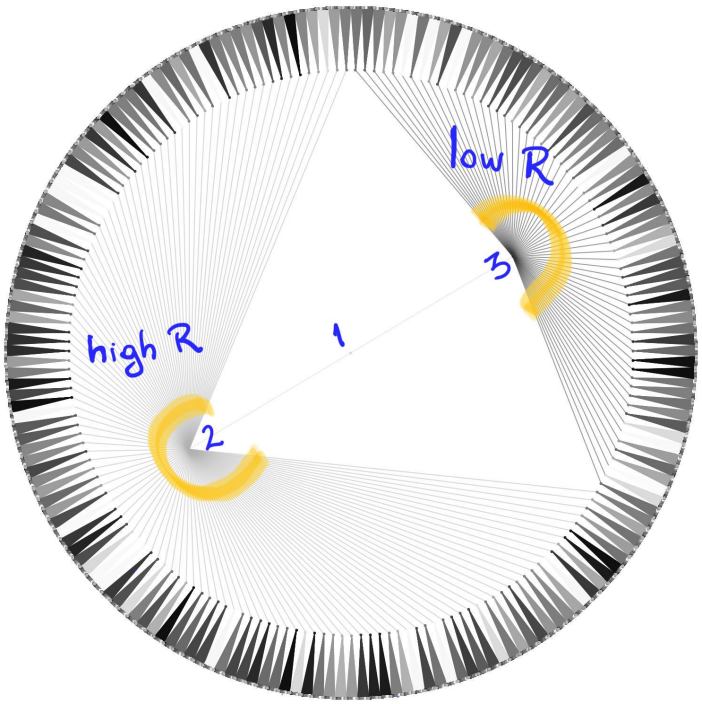


Figure 2: The 128-ary Merkle tree for a 100 MB file obtained in our cluster. The leaves are along the outer orbit and the root in the center. Edges connecting parents with children are colored according to their replication factor, with darker edges corresponding to weakly replicated parents.

1.1 Motivations for This Work

The Merkle trees used in decentralized storage systems introduce *hierarchical dependencies* between the content and the *metadata*, i.e., the content addresses used to locate the content. Hence, the root and internal nodes must also be available to ensure that the content is available. In other words, any replication added to the leaves is futile if the pointers to the leaves are not reachable. Previous experiences support our claim. Wilcox-O'Hearn admitted that about 90 % of all the content that has ever been stored in the Tahoe-LAFS [32] grids might have died despite using "erasure coding parameters with massive fault tolerance" [31]. He highlighted that administrators get a false impression by computing reliability under the assumption of independent failures because the numbers lead to many nines of reliability. In particular, for the standard 3-out-of-10 erasure code and $p = 0.90$ used in Tahoe-LAFS, we get 6 nines of reliability. He concluded that monitoring is more important than the redundancy parameters used in the system.

The assumption of independent failures is widely accepted despite that researchers have shown its pitfalls [22]. However, to the best of our knowledge, the risk of cascading failures in Merkle trees has not been studied. Previous research on cryptographic file systems focuses on the intersection between integrity and confidentiality. While some works combine Merkle tree and erasure coding or replication, the availability of the root and internal nodes has not been considered. Hence, we observe that the intersection between integrity and availability has not been sufficiently studied.

Coding algorithms are more storage efficient than replication [21, 14, 29], but they require *repair metadata* to decode. Decoding algorithms need some clue to start decoding, e.g., the index number of each coded block. A common solution is to use a metadata file or manifesto. For instance, Tahoe-LAFS treats the metadata as an additional erasure-coded file distributed on k servers [24]. Another approach is to store metadata in a local manifest. The caveat is that a local manifest is not publicly accessible and thus limits user collaboration. Hence, having to manage an additional metadata file is undesirable, especially in decentralized storage systems.

Some decentralized systems include incentive mechanisms to improve availability [6]. Currently, the Swarm Foundation is about to deploy an incentive

layer that will create liabilities for chunk availability in the Ethereum Swarm network [26]. Still, redundancy mechanisms are critical for content survivability. However, it is well-known that systems built for fault tolerance in networks with untrusted peers use hefty amounts of redundancy [20, 16, 31]. Thus, *we posit that a method imposing low redundancy yet achieving high fault tolerance will improve the network's storage utilization.*

1.2 Contributions

We propose a novel solution that combines Merkle trees with alpha entanglement (AE) codes [8] to address the gaps above, aiming at high fault tolerance, low bandwidth, and low storage requirements. To the best of our knowledge, this is the first work that addresses the intersection between integrity and availability when using Merkle trees.

As with other codes, AE codes also require repair metadata. However, we observe that by exploiting synergies between the Merkle tree structure and the encoding/decoding algorithm, we can eliminate the need for a local or remote metadata file. Therefore, our solution only requires the content address of the tree's root to locate content even if it is encoded and distributed in a p2p network.

We present Snarl, another way of saying entangle. Snarl does not require any modification to the underlying storage system and seamlessly integrates with a local peer in a p2p storage network. Users interact with Snarl when uploading or downloading content. The content is encoded in an entangled Merkle Tree (eMT) structure before being uploaded to the network. Further, Snarl offers user-controlled redundancy in the sense that the encoding configuration can be adjusted to match the behavior of the storage network, i.e., higher network churn requires more redundancy.

We have implemented Snarl and deployed it in our 1000-peer cluster running Ethereum Swarm. Our results demonstrate that data integrity and availability for decentralized storage systems can be obtained without significant storage or bandwidth overhead. Our evaluation shows that Snarl is capable of simultaneously improving the storage utilization and file availability in Swarm. Specifically, with the Snarl-14 configuration, **five times** the amount of data could be stored in the network with failure-resiliency comparable to Swarm. We show that file recovery is bandwidth-efficient in the presence of failures and uses less than $2.08\times$

the chunks on average in scenarios with up to 50 % of total chunk loss.

2 Preliminaries

Integrity and redundancy are interrelated. Systems introduce some form of redundancy to ensure data integrity [25]. Integrity checking requires comparing data with some derived piece of information to offer a certain level of assurance. For example, disk mirroring and checksums can detect integrity violations but cannot recover the original data. Assurance can be increased in a system that stores $N > 2$ copies using a majority vote, i.e., the majority of the copies are accepted as valid with some degree of confidence. Other widespread techniques are RAID parity and error detection and correction algorithms, which can recover data up to some level. For example, a (n, k) MDS erasure code encodes k data symbols into n coded symbols, and any k out of these n coded symbols can be used to decode the k data symbols. MDS stands for maximum distance separable codes and, in practice, it means that a (n, k) MDS code can recover data up to $n-k$ failures.

Hash functions generate a derived piece of information with negligible overhead. This property made them suitable to check correctness in memory stacks and queues [3]. Merkle trees [18] leverage the benefits of hash functions. Their original application was a digital signature system, where the growth of the signature is logarithmic with the number of messages signed. With its efficient verification of large amounts of data, it found its way into several other areas—particularly p2p-based systems, where it is necessary to verify the integrity of data received from untrusted parties. Often, cryptographic file systems, e.g., SiRiUS [10], TDB [15], Tahoe-LAFS [24], use Merkle trees to guarantee integrity. It is also used in blockchains like Bitcoin [19] and Ethereum [33], decentralized storage systems like IPFS [2] and Swarm [27], revision control systems like Git [4], and protocols like BitTorrent [5].

2.1 Swarm Overview

Swarm is a decentralized storage system that distributes stored data to a network of peers. The peer-to-peer network is based on Kademlia [17] for discovery and routing. By following the protocol, storing, and delivering content, peers are re-

warded BZZ tokens through a smart contract on the Ethereum blockchain. According to a monitoring website [1], the public network has more than 270 000 peers.

Connecting to the Swarm network requires running a local peer or using a public gateway. Peers runs the Bee client, which is under active development at the time of writing. The Bee client will incorporate incentive mechanisms to reduce network churn. In this work, we used the more stable Swarm client, v0.5.8. We run a cluster with 1000 peers isolated from the public Swarm network.

Swarm splits files into 4 KB chunks and places them as leaves in a 128-ary Merkle tree. The internal nodes and root of the Merkle tree are also 4 KB chunks and contain a concatenation of the content addresses of their children. Chunks in Swarm are given a unique content address derived from a cryptographic hash function over the chunk's data. All chunks are distributed to peers whose address has the same prefix as the chunk's content address, also known as an address space neighborhood.

Redundancy is of significant importance in Swarm. It is deeply embedded in its design to provide fault tolerance, censorship resistance, DDoS resistance, and zero downtime.

Currently, the redundancy in Swarm is provided by a decentralized replication process called *syncing*. Syncing is separated into three mechanisms. First, *push-sync* involves transferring chunks from the uploader to storage peers, i.e., from the network's entry point to each chunk's corresponding address space neighborhood. Once the chunk reaches the neighborhood, it is replicated to all the peers within it. Second, when a new peer enters the network, it initiates *pull-sync* with its neighbors. When pull-syncing, the new peer queries its connected peers for a list of chunks they are storing and then proceeds to request those within its address space. Lastly, each time a chunk is transferred in the network, all peers that act as relays between the sender and receiver may choose to replicate the chunk.

2.2 System Requirements

Our primary design goal for Snarl is to increase the resilience of data stored in a cryptographic decentralized storage system without adding storage overhead. Snarl should be optional and user-controlled, meaning that the user should be able to specify the resilience level and choose how much of the resilience will be

disclosed to the public. Snarl should enable file recovery despite the failure of a large part of the underlying storage system. Further, any peer should be able to repair efficiently, without overhead when there are no failures.

Snarl requires no modification to the underlying storage system. However, we specifically target storage systems that connect the stored data in a graph with a hierarchical dependency between nodes, typically a Merkle tree. We assume the presence of APIs to upload and download content.

3 Entangled Merkle Tree

Let $k\text{-}MT(d_1, \ldots, d_n)$ denote a *k-ary Merkle tree*, where data items $d_1, \ldots, d_n$ are at the leaves of the tree, and k is the branching factor limiting the number of children of a node. An internal node of the tree i_h at height h with children $c_1, \ldots, c_k$ is the hash of the concatenation of its children, i.e., $H(c_1|| \ldots ||c_k)$, for H a collision-resistant hash function. If the concatenated leaves form a file f, we can refer to its Merkle tree as $k\text{-}MT_f$.

Cryptographic decentralized storage systems often split the content into chunks and provide integrity via a Merkle tree. The Merkle tree maps chunks to a single root hash value used as a content identifier and entry point for retrieval. However, a serious concern is that the hierarchical dependencies between chunks can render several chunks irretrievable, even though they are stored at available peers. This can happen if a critical storage peer fails, resulting in a logic failure cascade. To reduce the impact of these dependencies within Merkle trees, we aim to transform the Merkle tree into a failure-resilient structure.

To fulfill our aims, we define a *k-ary entangled Merkle tree* ($k\text{-}eMT$) data structure, constructed from a $k\text{-}MT$ using these functions in sequence:

```
k-MT | mapper | swapper | entangler | k-eMT
```

Snarl implements these functions. The original $k\text{-}MT$ contains information (the user file or any arbitrary content) at the leaf level. However, the mapper, swapper, and entangler consider all the nodes of the $k\text{-}MT$ as information. The entangler generates α $k\text{-}eMT$s as output. These carry redundant information to recreate all nodes in the original $k\text{-}MT$.

We start by describing $k\text{-}eMT$ in §3.1 and §3.2, followed by the *entangler* in §3.3 and §3.4. The *mapper* is described in §3.5, and lastly, the *swapper* in §3.6.

3.1 The k-eMT: Challenges and Solutions

Let $k\text{-}eMT(p_1, \ldots, p_m)$ denote a *k-ary entangled Merkle tree*, where items $p_1, \ldots, p_m$ are at the leaves of the tree, and m is the total number of nodes in the original k-MT. An item p_i is a parity obtained by the entanglement algorithm and is a shorten for a parity $p_{i,j}$, more details are explained in §4.4. An internal node of the tree i_h is computed as in k-MT.

Our k-eMT design solves the problem of handling metadata for decoding mentioned in §1. We avoid the use of extra metadata by carefully crafting the entangled chunks in a forest formed by the original k-MT and a few k-eMTs created with the entanglement. By leveraging content-addressing and AE codes, the decoding process only needs to know the roots. In fact, some roots may be kept private and released only when needed via a smart contract or some other medium.

Moreover, the forest reduces the hierarchical dependencies of the original k-MT, as with the addition of k-eMTs there are alternative paths for traversal. In addition, as will be detailed in §4, the new structure can recover from missing and corrupt nodes.

We highlight that the design of k-eMT is non-trivial. It involves mapping the elements of a hierarchical structure into a helical lattice structure constructed by the AE encoding algorithm. As we explain later, the process needs to remove specific dependencies between chunks, as otherwise, the protection offered by AE codes would be reduced due to the appearance of additional irrecoverable failure patterns.

Our solution is designed for k-MT with $k \gg 2$, see §3.6 for more details. This is consistent with real-world systems, e.g., Swarm uses $k = 128$ and IPFS uses $k = 174$. Large k generates a shallow tree that accelerates the lookup in content address networks.

3.2 Canonical Naming

To aid in the reconstruction of the tree, we devise a canonical naming scheme. A similar scheme was proposed by Merkle [18] for a 2-MT with the root node labeled 1, the left child of node i labeled $2i$ and right child of node i labeled $2i + 1$. Our scheme supports k-ary branching, and also allows us to assign labels to nodes

during initial chunking, without knowing the size of the data beforehand. The scheme is based on a post-order traversal algorithm, with naming starting at the leaf level (see §3.5 for an example).

3.3 Overview of Alpha Entanglement Codes

AE codes [8] are designed to tolerate a large number of failures with low computation and bandwidth requirements. The encoding algorithm produces *entangled chunks* embodying parities to be disseminated across the system. Assuming the chunks are distributed to different storage peers, the decoding algorithm can use multiple "paths" in the lattice structure to decode the content. Each path requires the availability of a set of distinct peers. Paths are formed by an n-length combination of data and entangled chunks, with $n \geq 2$. The shortest path to repair a single failure has length two. Multiple paths improve the access and recovery likelihood for temporarily and permanently unavailable content, respectively. As such, paths provide for an efficient recovery mechanism for high churn scenarios.

The entanglement process creates a graph of interdependent chunks. In its simplest form, where $\alpha = 1$, the graph is a path, usually referred to as a chain or strand. It starts with $e_{\_,1} = 0$, a dummy chunk and continues with $e_{1,2} = d_1$, and then alternates unencoded data chunks and entangled chunks. The strand is constructed by encoding the entangled elements $e_{i,j}$ according to: $e_{n,\_} = e_{n-1} \oplus d_n$, with $n > 1$ where d_i are data chunks presented as input to the encoding algorithm, and $e_{i,j}$ is adjacent to d_i and d_j. Note that d_i are the ordered chunks obtained by chunking a file f, or the chunks of any arbitrary data stream.

The Cylindrical Helical Lattice

With $\alpha > 1$, the graph becomes a *lattice* composed of intertwined strands, where each data chunk d_i belongs to α strands.

Let $LAT_\alpha(d_1, \ldots, d_n)$ denote an $AE(\alpha, s, p)$-*lattice* for $\alpha > 1$, where data items $d_1, \ldots, d_n$ are the lattice vertices v. LAT_α is a regular graph of degree 2α with d_i's position in the lattice according to: *top* v for $i \equiv 1 \pmod{s}$, *central* v for $i \not\equiv 1 \wedge i \not\equiv 0 \pmod{s}$, or *bottom* v for $i \equiv 0 \pmod{s}$. LAT_α is composed of $s + (\alpha - 1) \cdot p$ intertwined strands. Each strand can be constructed independently using the equations in §3.3; therefore, lattice construction can be parallelized for efficiency.

For $\alpha \in [2, 3]$, the lattice can be thought of as a weaker version of graph embedding on a cylinder, i.e., we relax the embedding definition by omitting the non-intersection condition for edges [12]. In LAT_3, p strands are cylindrical double-helix, denoted *RH- and LH-strands* (right-handed and left-handed helical strands), and the remaining s strands are in parallel with the cylinder axis, hence, denoted *H-strands* (horizontal strands). In LAT_2, the strands are not a double-helix since there are only $s + p$ intertwined strands, choosing RH- or LH-strand does not yield any difference. Helical strands revolve around the imaginary central axis of a cylinder. RH-strands connect vertices in cycles, alternating a sequence of a top vertex, $s - 2$ central vertices, and a bottom vertex. LH-strands connect vertices in cycles, alternating a sequence of a bottom vertex, $s - 2$ central vertices, and a top vertex. H-strands connect vertices of the same type.

Let $LW(d_i, \ldots, d_{i+s \cdot p-1})$ denote a *lead window* for $\alpha = 3$, where data items $d_{i+j \cdot (s+1)}$ with $j \in [0, p-1]$ are connected to the same helical strand. The lead window describes the interval it takes a helical strand to revolve around the cylinder's axis. The concept is similar to the pitch of a helix, i.e., the height of one complete helix turn, measured parallel to the helix's axis. Given a flat representation of the lattice, we can say that if d_i is a top vertex, a lead window spans an area defined by s rows and p columns with its top-left vertex d_i.

3.4 A Variation for AE codes: A Toroidal Lattice

We propose a variation for AE codes based on closed entanglements [7]. Closed entanglements are entanglements with $\alpha = 1$ that generate a closed path with a slight modification to the algorithm to connect d_n with d_1 by recomputing $e_{1,2}$ and replacing the dummy chunk with $e_{n,1}$. The objective of closing the path is to better protect the elements at the extreme of the path. The original AE codes, however, used an "open" lattice. We modified the original design by closing each path that forms the lattice. As a result, the cylindrical helical lattice is transformed into a toroidal lattice. The subtleties of the closing are related to the number of nodes, N, in the tree. If N is multiple of $s \cdot p$ (the number of vertices in a lead window), the closing is straightforward. For other cases, the connections added to close the lattice prioritize connecting vertices from the same path. Figure 3 illustrates the toroidal lattice for a case where N is not multiple of $s \cdot p$.

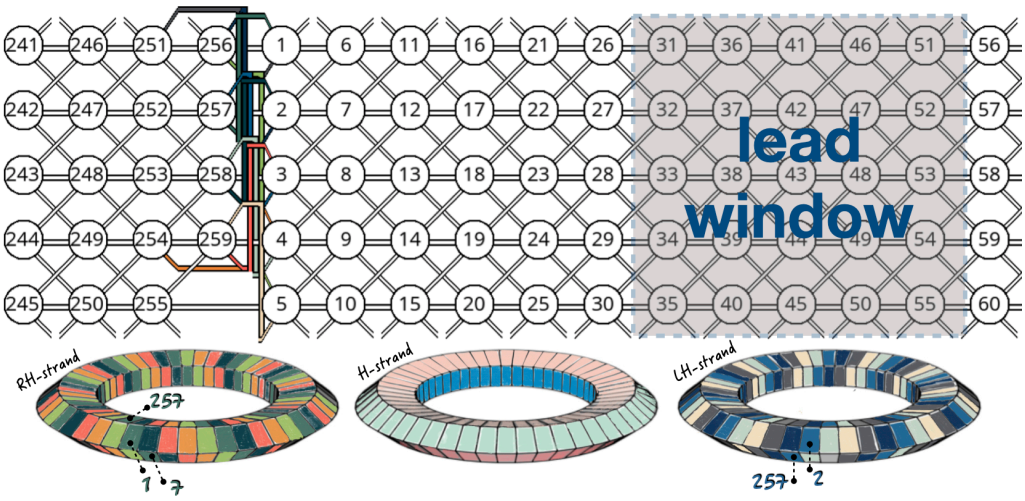


Figure 3: Toroidal lattice for a Merkle tree with 259 nodes.

3.5 Mapping the Tree Into a Lattice

Mapping is the first of two steps needed to prepare the input for the entanglement. The encoding algorithm (entangler) takes an input stream of "ordered" chunks (information), i.e. entangler is deterministic and produces an output that depends on the order of the input chunks. Internally, the entangler creates LAT_α, where its vertices $d_1, \ldots, d_m$ correspond to the input stream (m is the total number of k-MT nodes and not just the leaves). All nodes are treated as information in our redundancy scheme in order to create redundancy for the internal nodes and the root too. Each vertex represents a distinct chunk. The order of the input chunks is reflected in the way the lattice unfolds.

To map the tree into the lattice, the *mapper* uses a post-order traversal algorithm that reads the tree and generates the input stream. To illustrate, reading the 2-MT showed in Figure 1a, $MT_f(A, \ldots, G)$ produces the ordered input: $A \rightarrow B \rightarrow H_{AB} \rightarrow C \rightarrow D \rightarrow H_{CD} \rightarrow H_{ABCD} \rightarrow E \rightarrow F \rightarrow H_{EF} \rightarrow G \rightarrow H \rightarrow H_{GH} \rightarrow H_{EFGH} \rightarrow H_{ABCDEFGH}$. Using our canonical naming, the elements of the sequence are labeled $1, 2, 3, \ldots, 15$. Therefore, this example creates a LAT_α with 15 vertices. The number of strands in LAT_α is defined by the entangler (obtained by the coding parameters). Before, we need the swapper to finish preparing the input for the entangler.

3.6 Swapping

Swapping is the second and final step to prepare the input for entanglement. Nodes that have a parent-child relationship cannot be adjacent or in a close neighborhood in the LAT_α. The reason for this is to avoid that a parent is entangled with its children or with their neighbors. If that occurs, and the chunk represented by the parent node is missing, then it is not possible to retrieve the elements in the lattice that are used to recover the missing chunk. The swapping algorithm (swapper) moves the parents at least one LW away from their children. The swapper looks for a candidate vertex position where none of the vertices in its neighborhood are children of the swapped vertex.

4 Snarl

The high-level architecture comprises three components: Snarl, a proxy, and the underlying storage system. Figure 4 shows the architecture and the main packages of Snarl.

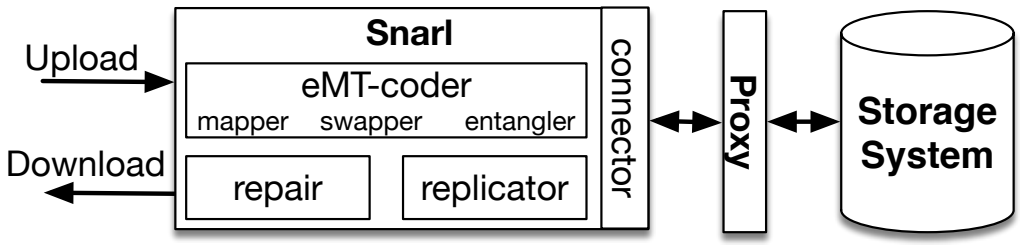


Figure 4: Snarl architecture.

Snarl is designed to be a general-purpose tool for providing user-controlled redundancy to any Merkle tree, and requires no modification to the underlying storage system. Snarl uses a modular approach and comprises four packages: (i) *eMT-coder* implements encoding and decoding algorithms for the eMT (§3). (ii) *repair* contains the algorithms needed to recover from failures. (iii) *replicator* combines eMTs and replication to further improve storage utilization. (iv) *connector* abstracts the low-level details of the storage system.

The repair algorithm locates parity chunks in a bandwidth-efficient manner and is only applied when failures are detected. When downloading content not encoded with an eMT, Snarl reverts to the retrieval mechanism of the underlying

storage system. Likewise, a non-Snarl user can download content encoded with an eMT, as the original Merkle tree is untouched and reachable.

Snarl comprises 4800 lines of Go code and 2200 lines for testing and benchmarking purposes. We have made the code available in our GitHub repository (`https://github.com/relab/snarl-mw21`).

4.1 User Interaction With Snarl

The user interacts with Snarl using the command-line. To encode content – without interacting with the storage system, we provide the *entangle* command. The entangle command takes as input the content that needs protection and the desired level of protection. It will generate α files, one for each eMT, and no other metadata.

To interact with the storage system, the commands *download* and *upload* are used. Upload calls entangle internally before uploading the original Merkle tree and α eMTs to the storage system. After each upload, the storage system replies with the content address of the root. The user must persist the content addresses of the roots for the original Merkle tree and the eMTs, as they are necessary to retrieve the content later. Each content address is small; usually, 32 bytes, depending on the underlying storage system.

To download content from the storage system, a user must supply the original Merkle tree root's content address. When downloading through Snarl, a user may also pass the content address of the eMTs root, allowing Snarl to repair the original content if chunks are missing. The repair process is seamless for the user, downloading parity chunks on-demand and reconstructing the content, despite widespread data loss.

4.2 Snarl Interaction With the Storage System

This section explains how Snarl interacts with the underlying storage system before detailing how we successfully deployed Snarl using Swarm as the storage system.

Snarl does not require any change to the underlying storage system. To achieve this, we assume a *proxy* that exposes APIs to upload and download content. In a decentralized storage system, the proxy will be a peer participating in the network. As we expect the various storage systems' APIs to be slightly different, we

provide a set of interfaces in our *connector* package. This allows a separation of concerns in the other packages, as all storage system-specific details are implemented in separate packages.

To satisfy the connector package's interfaces, specific details such as the Merkle tree branching factor, its balancing algorithm, and the maximum chunk size must be implemented. By designating this to its own package, the behavior can be mirrored either by calling the proxy, importing a library, or as a separate implementation of its specification.

Swarm as the Storage System.

Swarm uses a 128-*MT* construction for each file. A chunk's maximum payload size is 4096 bytes, with an extra 8 bytes describing the accumulated size of all its leaves if it is an internal node, otherwise its length. The tree is constructed from left to right ensuring that all non-leaves, except the righter-most, are of the same size and height. These 8 bytes also subverts the *length extension attack* [30] and can be used to determine the canonical index of the node without exploring the tree. At the same time, the 8 bytes create some difficulty, as our eMT is limited by the same 4096 bytes. Thus, we cannot encode the chunk size information in the eMT, as we would have to fit 4104 bytes inside 4096. Instead, we observe that since the tree's construction is deterministic, the chunk size can be derived from the size of the eMT and the chunk's position.

4.3 Local Repair Information

The edges of the toroidal lattice outlined in §3.4 are the parity chunks needed for repair, and knowledge of their content address is the only way to retrieve them.. The mapping of parities to content addresses is called *repair metadata* and is used by Snarl when requesting parities.

As the size of repair metadata grows linearly with the content's size, it could be impractical to store locally for large files. Further, it hinders *collaborated repairs*, as only those with access to the metadata can partake in repairs. Storing the metadata in the storage system itself is also a bad idea, as the original content's resilience would be reduced to the single failure of the metadata.

In Snarl, we instead implement the eMT from §3, which allows requesting correct parities knowing only the root chunk's content address. An eMT has the

same number of leaves as there are chunks in the original Merkle tree. The leaves are ordered such that the left parity for the chunk with canonical index n is leaf n in an eMT. To find the leaf index for the right parity for the chunk, we implemented an algorithm according to §3.3.

We now illustrate how to retrieve the desired leaves from the $k\text{-}eMT(d_1, \ldots, d_n)$. To retrieve a leaf d_i from the $k\text{-}eMT$, we first request its root to know the content addresses for their children. Each child is a leaf or an internal node that points to a subtree. Every subtree of a parent is of equal size, apart from the right-most. To calculate the subtree's size, we realize that their size must be a power y of the branching factor k, where y is the chunk's level, starting at 0 on the leaves. Thus, given a parent at level h, each parent's child will have size k^{h-1}, apart from the right-most. Therefore, to find a path to leaf x, we traverse through the g-th sibling defined by $g = \lceil \frac{x}{k^{h-1}} \rceil$. This continues recursively by redefining $x = x - (g-1) \cdot k^{h-1}$ for each step, until we reach the desired leaf. The pseudo-code is given in Algorithm 1.

Algorithm 1 Parity retrieval from an entangled Merkle tree

```
1: func GetParity(parentAddr, leafId)
2:     parent ← Download(parentAddr)
3:     childSize ← parent.branchFactor ∧ (parent.level − 1)
4:     if childSize = 1 then                          ▷ All children are leaves
5:         return Download(parent.child[leafId])
6:     next ← math.Ceil(leafId / childSize)           ▷ g
7:     leafId ← leafId − (next − 1) · childSize       ▷ x
8:     return GetParity(parent.children[next], leafId)
```

4.4 Repairing Failures

Snarl can detect and repair failures due to both corrupt and missing chunks. It is agnostic to the reason for a failure, whether it is due to data loss, network partition, malicious behavior, or churn. Integrity verification of chunks is done by comparing the cryptographic hash of the received data with the requested content address. Therefore, we treat corrupt chunks the same way as missing chunks.

A repair begins as soon as a failure is detected during a download and runs concurrently with the remaining download. Snarl uses the same algorithm to

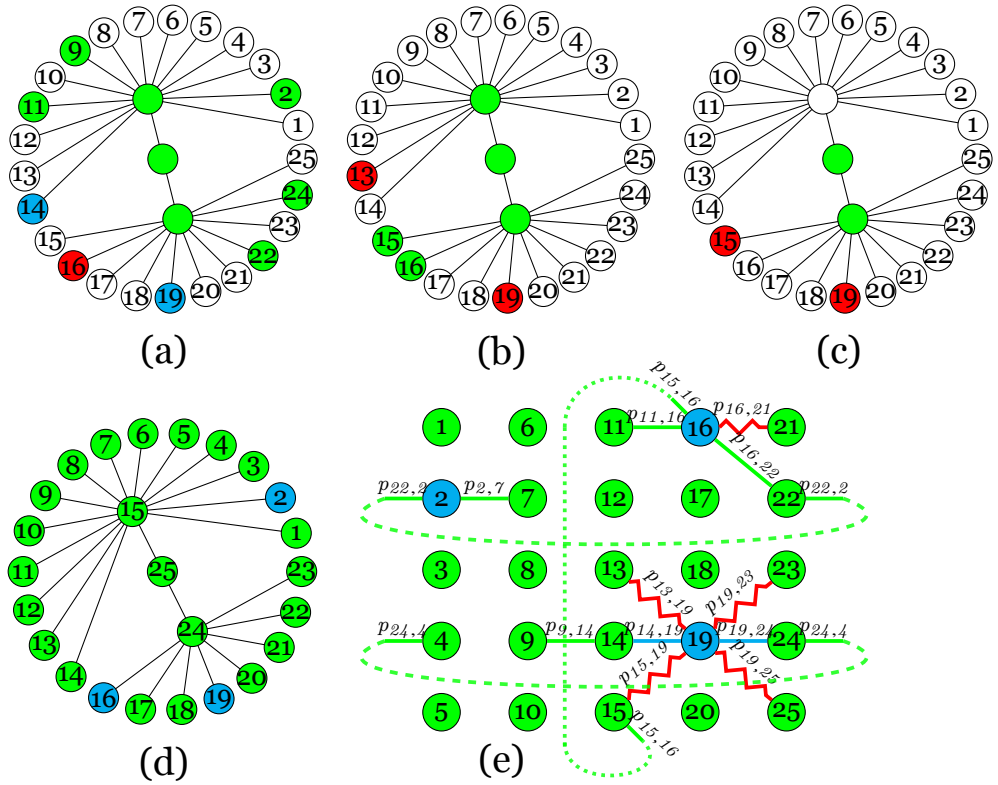


Figure 5: Repair algorithm: (a,b,c) horizontal (H-eMT), right (RH-eMT) and left (LH-eMT) entangled Merkle trees, (d) original Merkle tree, (e) lattice. Failed downloads are colored with red, successful downloads are colored in green. Repairs are colored with cyan. White is not requested. Users can collaborate to maintain the system by re-uploading repaired chunks.

repair any node of the Merkle tree. It is intelligently aware that the loss of an internal node prevents downloading any of its children and will thus give priority to repairing internal nodes first.

We illustrate the repair algorithm in Figure 5 with three example scenarios. For illustrative simplicity, we use a Merkle tree with three levels and a branching factor of 14.

Snarl starts by requesting the root of the original Merkle tree, labeled “25” in Figure 5d. Then it proceeds through the internal nodes, labeled “15” and “24”, to the leaves, which hold the content. If a chunk loss or corruption is detected, even at the root chunk, Snarl immediately requests parity chunks from the eMTs,

also shown in Figure 5 with three strand types ($\alpha = 3$), (a) H-eMT, (b) RH-eMT, and (c) LH-eMT. The lattice shown in Figure 5e is a virtual data structure used to coordinate repairs. Each vertex in the lattice represents a chunk from the original Merkle tree. The parities in the lattice are illustrated as edges labeled $p_{X,Y}$, where X is the vertex connected on the left, and Y is the vertex connected on the right. When downloading a parity, we request the n-th leaf chunk of an eMT of the correct strand type, where n is the left-connected vertex's index.

Using the toroidal structure of the lattice, Snarl can repair in many failure scenarios in which the underlying storage system cannot. After successful repair, the user can re-upload the repaired data and parity chunks to the storage system, thus aiding system-wide maintenance.

We consider three example failure scenarios. First, the most basic scenario, where a single data chunk is lost. As soon as Snarl cannot download data chunk "2", it requests parities $p_{2,7}$ and $p_{22,2}$ from H-eMT. Both are available, and data chunk "2" is repaired with two parity downloads.

In the second scenario, data chunk "16" is lost, and when attempting to download the parities $p_{11,16}$ and $p_{16,21}$ from H-eMT, Snarl discovers that $p_{16,21}$ is unavailable. Snarl then requests the two adjacent parities from the RH-eMT, i.e., parities $p_{15,16}$ and $p_{16,22}$. Both are available, and data "16" is repaired with three parity downloads.

Data chunk "19" is lost in the last scenario, including all its adjacent parities in the H-eMT, RH-eMT and LH-eMT. Snarl can recover from such a scenario by repairing one of the parity pairs. It requests parity $p_{9,14}$ and uses that together with data "14" to repair $p_{14,19}$. Similarly, parity $p_{24,4}$ and data "24" are used to repair $p_{19,24}$. Thus, with a complete parity pair, data "19" can be repaired with only two parity downloads.

During repair, Snarl will expand both vertically and horizontally in the lattice until either the chunk is repaired or an irrecoverable pattern is detected. The expansion is recursive and follows the order; H, RH, and LH, referring to the eMT from which parity chunks are requested first.

4.5 Replicating Entangled Merkle Trees

It has been shown that a combination of erasure codes and replication [9] generally achieves better storage utilization and lower repair overhead than the two

methods separately. With Snarl, we propose a similar combination of eMTs and replication.

We distinguish between replication of internal and leaf chunks. These must be weighed against each other, as a higher replication factor for internal chunks improves the likelihood that a leaf chunk can be located but does not contribute to the repair algorithm. On the other hand, a higher replication factor for leaf chunks does contribute to the repair algorithm. However, a loss of their parent results in the de facto loss of all leaves.

We use (1) as a measure to balance the replication of internal versus leaf chunks. Equation (1) captures the likelihood that a chunk replicated r times, with each replica placed on a distinct peer, would still be available after f peers failed in a network with n peers. We observe that adding replicas is an effective measure to increase recovery likelihood if the replication factor is low. However, at higher replication factors, the gains of adding another replica approach 0. In fact, with only 45 replicas, the recovery likelihood is more than 99.9 %, with up to 86 % peer failure.

$$P(x) = 1 - \prod_{i=0}^{n-f-1} \frac{n-r-i}{n-i} \tag{1}$$

Snarl can scale the replication factor, allowing a user to specify the total storage consumption of the encoded data to be proportional to uniformly replicating the original Merkle tree. Further, the weighting of the replication factor for internal and leaf chunks can also be adjusted. By default, we cap the number of internal chunk replicas at 45. To differentiate between Snarl configurations, we label them with the ratio of storage consumption compared to the original Merkle tree. For example, with Snarl-5, we use the same storage consumption as having 5 copies of the original Merkle tree.

5 Evaluation

We evaluate Snarl using the toroidal lattice variant of closed entanglements described in §3.4. The encoding parameters are $\alpha = 3$, $s = 5$, $p = 5$, as it has been previously studied in the literature [8]. It's worth noting that Snarl allows the user to adjust these parameters to fine-tune the performance and redundancy to

its requirements. We compare Snarl with full uniform replication, where every chunk has the same number of copies, and with Swarm's default redundancy.

In our first evaluation, we studied how files in Swarm are replicated in our cluster. We then evaluate the chunk distribution of files encoded with Snarl. Next, we study Snarl's file availability on our cluster of 1000 Swarm peers to show how Snarl can increase storage utilization by over 80 %. We show that the encoding speed of Snarl is linear, requiring only 3.6 seconds for a 1 GB file. The effectiveness of our repair algorithm is evaluated by examining bandwidth overhead incurred in different failure scenarios.

5.1 Experimental Setup

We ran our experiments on a cluster of 30 machines running Ubuntu 18.04.4 LTS. Each machine is equipped with Intel Xeon E-2136 3.30 GHz CPU, 32 GB RAM, 1.5 TB SSD disk, and 1 Gbit/s NIC. We used Helm [11] and Kubernetes [13] to distribute 1000 Swarm peers on 28 machines. We use the remaining two machines to host the Snarl client, the bootstrapping peer, and manage the experiment execution.

The bootstrapping peer allows the Swarm peers to discover each other and to achieve their desired connectivity. Swarm's *maxpeers* parameter is left unchanged, allowing a peer to connect with up to 50 peers in the network.

Each chunk's replication factor is highly variable in Swarm, as we discuss in §5.2. To compensate for this variability and make the comparisons fair, we need to adjust each chunk's replication factor to match the evaluated coding scheme. To facilitate these adjustments, we have developed a set of tools.

Our first tool, *listchunks*, lists the content addresses of all the chunks of each file in our storage network. The second tool, *deletelist*, determines which chunks must be deleted from which peer. To evaluate files of different sizes, we must run listchunks and deletelist for each file. To obtain an aggregated delete list, we use a third tool, *combinelist*. Finally, the aggregated list is passed to *deletechunks*, to delete the chunks on the peers in the storage network.
Swarm has three mechanisms for chunk distribution, also called *syncing*. Two of these mechanisms, *push-syncing*, and *pull-syncing*, can be disabled with the *nosync* command-line option when starting a peer. However, the syncing process for chunks delivered through the Kademlia [17] DHT cannot be disabled. This

posed a challenge for us, as the increased chunk replication would unduly skew the results for the different coding schemes.

Hence, to ensure an identical system state across experiment runs, we must counteract this built-in syncing process. To that end, we create a snapshot of the entire storage network and recover from a snapshot between each experiment run. Further, we ensure that each peer is well-connected before each experiment run. To reach sufficient connectivity, the peers must first discover each other. We monitor the discovery process by periodically polling Swarm's inter-process communication file, *bzzd.ipc*. As soon as the desired connectivity is reached, we start the next experiment run.

5.2 Replication in Swarm

Our study reveals that the replication factor (R) in the Swarm network is not uniform at the chunk level, meaning that some chunks are more replicated than others. However, the file size does not appear to impact R notably. For file sizes 1 MB, 10 MB, and 100 MB, we found that each chunk was in the range $R \in [9, 162]$, with an average R of 72. The relatively high average R of 72 means that for every gigabyte of original data stored in the network, the collective resources of the storage peers will consume at least 72 gigabytes.

We plot the relative chunk replication for a 100 MB file in Figure 6. The file consists of 25803 unique chunks, with a R that ranges from 9 for 26 chunks (0.1 %) to 162 for 3 chunks (0.01 %), with an average R of 72.36. The horizontal axis shows the replication factor, and the vertical axis shows the relative frequency of occurrence.

Figure 7 shows how the chunks are distributed on the storage peers by plotting the number of chunks each peer stores for the same 100 MB file. We have sorted the peers in ascending order of the number of chunks stored. Interestingly, the values appear to follow a power-law distribution. The number of chunks stored ranges from 105 on peer 1 to 12865 on peer 1000, with an average number of chunks stored 1867.

5.3 Chunk Distribution

We observe that a file encoded with Snarl occupies a broader range of the address space than with full replication.

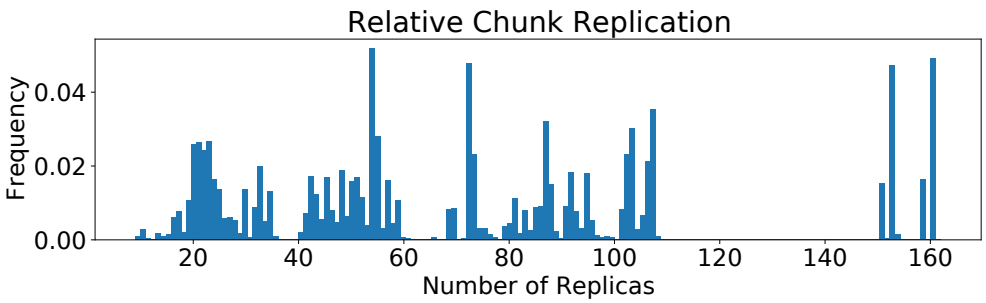


Figure 6: Relative chunk replication for a 100 MB file stored in a Swarm network consisting of 1000 peers.

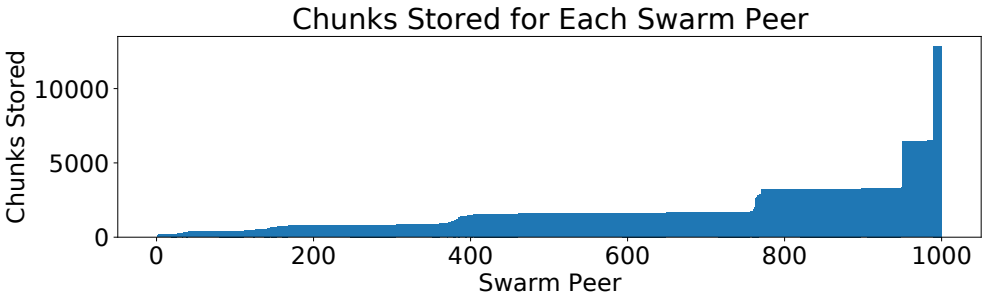


Figure 7: Storage consumption for a 100 MB file in a Swarm network consisting of 1000 peers. Peers are enumerated on the horizontal axis, sorted by the number of chunks stored.

By mapping the first two bytes of the content address for each chunk, we get an idea of how a file would be distributed in the network, as chunks are placed at peers that share the same prefix. We repeated the experiment 10000 times with randomly generated files and report the average result.

For a 1 MB file in Swarm, there are 259 unique chunks with Swarm's full replication, with 258 distinct prefixes, meaning that 2 chunks share the same two first bytes in their address. Snarl encoding generates 1048 unique chunks, with 1039 distinct prefixes.

A file encoded with Snarl is evenly spread and occupies 1.59 % of the address space, while a replicated file occupies only 0.39 %. Consuming more address space is advantageous to mitigate an attack attempting to make a file unavailable

by monopolizing a small section of the address space.

5.4 File Availability

Next, we empirically estimate Snarl's file availability from the recovery likelihood for different failure scenarios. We compare the results of Snarl to full uniform replication, where each chunk in the Merkle tree is replicated the same number of times. We also compare with Swarm, which has a chunk distribution similar to that shown in Figure 6. In the case of replication, it is clear that the file can be recovered as long as at least one replica of each chunk that composes the file is available. We conduct two types of evaluations for file availability. In the first experiment, we obtain the recovery likelihood by artificially marking some percentage of *chunks unavailable* and trying to recover the file. The second experiment is similar, except that we mark some percentage of *storage peers unavailable*. Our evaluations show that Snarl provides a higher recovery likelihood for a lower storage consumption than the case for full replication. We list the storage consumption used by each scheme in Table 1.

5.4.1 Chunk Loss

We compare files of sizes 1, 10, and 100 MB encoded with Snarl-5, Swarm with replication factor 5 (R-5), and replication factor 10 (R-10). We run 10000 iterations of each experiment, with new random input data for each. Snarl-5 used for this evaluation has the same storage consumption as R-5; thus, we use this as the baseline for our evaluations.

Because we are only interested in recovering the entire file, a single chunk loss will make the file unavailable. Hence, the recovery likelihood is expected to decrease as the file size increases. As the number of chunks that compose a file increases, given the same chunk loss percentage, the likelihood of an irrecoverable error also increases. From Figure 8, we can observe that the impact on recovery likelihood for larger files is less for Snarl than for full replication for all three file sizes.

From Figure 8, we can see that Snarl-5 has a significantly higher recovery likelihood than both R-5 and R-10, with up to 50 % chunk loss. Given 45 % chunk loss, Snarl-5 has a recovery likelihood of 99 % for a 1 MB file, while R-5, in comparison, has 1 % recovery likelihood, and R-10 92 %.

Table 1: Snarl performance when encoding files of different size: storage consumption given in MB and total chunks number, the number of internal nodes in the *k-MT* and the α *k-eMT* unique and total nodes including replicas, the maximum chunk loss/peer failure for 99 % recovery likelihood per redundancy scheme. Higher values are better for max. chunk loss/peer failure.

	Storage Consumption: MB::>total chunks		
Scheme	1 MB	10 MB	100 MB
R-5	5.06::>1295	50.41::>12905	503.96::>129015
R-10	10.12::>2590	100.82::>25810	1007.93::>258030
Snarl-5	5.06::>1295	50.41::>12905	503.96::>129015
Snarl-14	13.15::>3626	131.07::>36134	1310.31::>361242
Swarm-72	72.60::>18585	723.69::>185264	7292.91::>1866986

	Internal Nodes: unique::>total		
Scheme	1 MB	10 MB	100 MB
R-5	3::>15	21::>105	203::>1015
R-10	3::>30	21::>210	203::>2030
Snarl-5	15::>165	87::>2784	818::>33518
Snarl-14	15::>330	87::>3915	818::>36810
Swarm-72	3::>104	21::>1548	203::>14258

	Max. Chunk Loss			Max. Peer Failure		
Scheme	1 MB	10 MB	100 MB	1 MB	10 MB	100 MB
R-5	14 %	8 %	5 %	14 %	14 %	14 %
R-10	37 %	29 %	22 %	40 %	40 %	40 %
Snarl-5	45 %	38 %	34 %	50 %	47 %	43 %

Table 1 summarizes the maximum chunk loss percentage that the redundancy scheme can tolerate and still provide 99 % recovery likelihood.

5.4.2 Peer Failure

We now evaluate Snarl deployed with 1000 peers, where each peer stores some of the chunks that compose the file. For this evaluation, we use Snarl-5 and Snarl-14. We choose Snarl-14 to evaluate the improved storage utilization Snarl offers, as Snarl-14 has a storage consumption roughly 81 % lower than regular files in Swarm. For each experiment we run 10000 iterations.

We can see from Figure 9 that Snarl-5 outperforms both R-5 and R-10, even considering that R-10 requires twice the storage. Interestingly, Snarl-14 outperforms the redundancy provided by Swarm while using less than 20 % of the storage. In other words, if every file in Swarm were encoded with Snarl, storage uti-

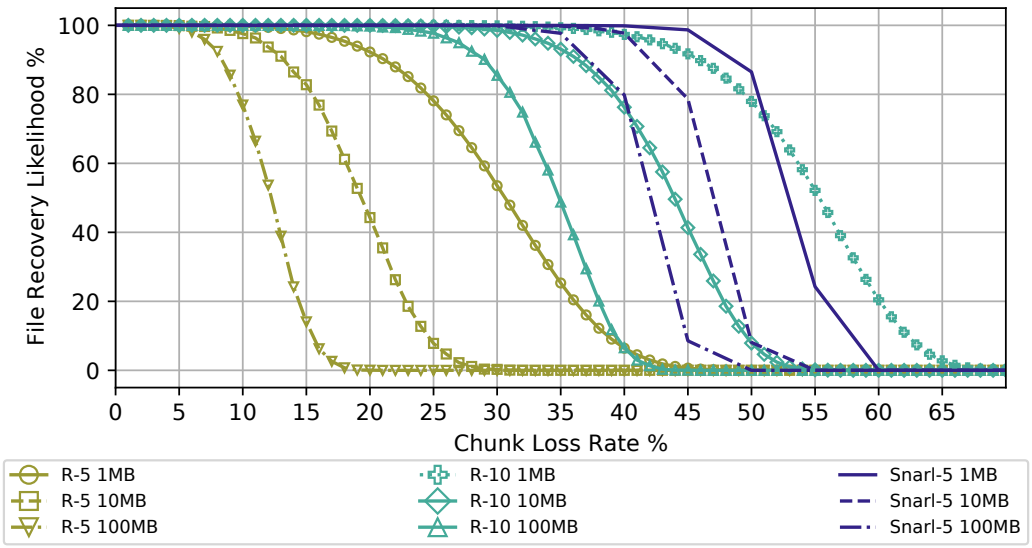


Figure 8: Recovery likelihood for various chunk loss rates.

lization would increase so that five times the amount of data could be stored in the network, with improved resiliency against failures. The recovery likelihood of replicated files for R-5, R-10, and Swarm, seems to be largely unaffected by the file size. This is because we need all chunks that compose the Merkle tree to recover the file, combined with the fact that chunks in Swarm are not uniformly distributed over the entire network. In other words, chunks are placed at peers with similar addresses, and therefore we are evaluating the likelihood that all peers in at least one address region are failing. Because chunks for various file sizes in Swarm are distributed to the same address region, the recovery likelihood is independent of file size, at least for files 1 MB and up.

Table 1 summarizes the maximum peer failure percentage that the redundancy scheme can tolerate and still provide 99 % recovery likelihood.

5.5 Encoding Speed

We evaluate the encoding performance of Snarl by measuring the time for encoding files from 1 MB to 1 GB. We ran this experiment on a single machine in our cluster.

Since encoding is based on lightweight XOR operations, it can be implemented

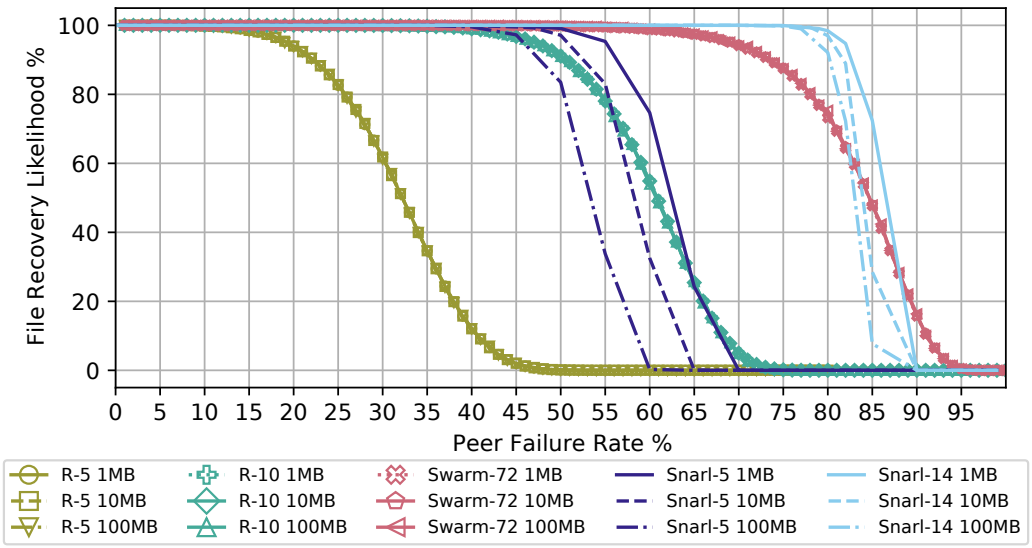


Figure 9: Recovery likelihood for various peer failure rates.

efficiently. We iterate over the file, and for every chunk, we XOR with the previously added chunks to create the parity chunks. We keep the last parity chunks in memory to perform the XOR operation with the newly added chunk to create the new parity. The cumulative time it takes to execute the XOR operations is the most significant factor for larger file sizes. As expected, the encoding time is linear with the file size; a file of 1 MB takes 3.8 milliseconds, a 10 MB file takes 37 milliseconds, a 100 MB file takes 360 milliseconds, and a file of 1 GB takes 3.6 seconds.

Decoding is also based on XOR, where two parity chunks are used as input to the XOR operation to reconstruct a data chunk. Thus, decoding a file requires using the same number of XOR operations as chunks in the file, resulting in linear time complexity. The actual time required for decoding, however, will largely depend on the network latency.

5.6 Network Overhead

We have previously outlined the repair algorithm in §4.4. As long as there are no failures, there is no need for any parity chunks. Only when failures occur will Snarl start requesting necessary parity chunks from the network. Each data

chunk is connected with three parity pairs in the lattice, as shown in Figure 5e. At best, a single failure or missing data chunk can be repaired with only a single pair. Both chunks in the pair are requested in parallel, and if either of them is unavailable, the next pair will be requested. However, in the case none of the three pairs can be downloaded, Snarl will expand outwards in the lattice, requesting additional parities to repair the parity pairs. This process is recursive and terminates only when the data chunk is repaired or an irrecoverable pattern is detected.

Therefore it is crucial that Snarl is bandwidth-aware and only requests the minimal number of parities necessary for successful retrieval. In addition, Snarl should be able to traverse the eMT to find the correct parities needed for each type of failure.

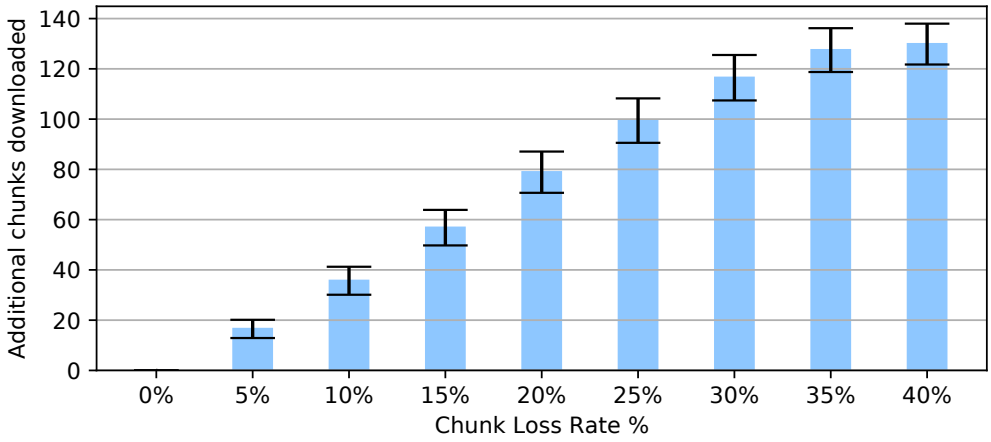


Figure 10: Download overhead for a 1 MB file in Snarl with increasing levels of chunk loss.

We measure the bandwidth efficiency of Snarl by counting how many additional chunks are downloaded compared to Swarm with no failures. Figure 10 shows how many additional chunks were downloaded for various amounts of chunk loss in a 1 MB file. Each experiment was executed with 10000 iterations. The error bars show the standard deviation. We can see that Snarl does not retrieve any parities when there are no failures, and with failures present, the number of chunks downloaded grows linearly with the failure rate.

To evaluate the repair efficiency of Snarl, we define the *repair ratio* as the number of parities retrieved, divided by the number of lost data chunks. Figure 11

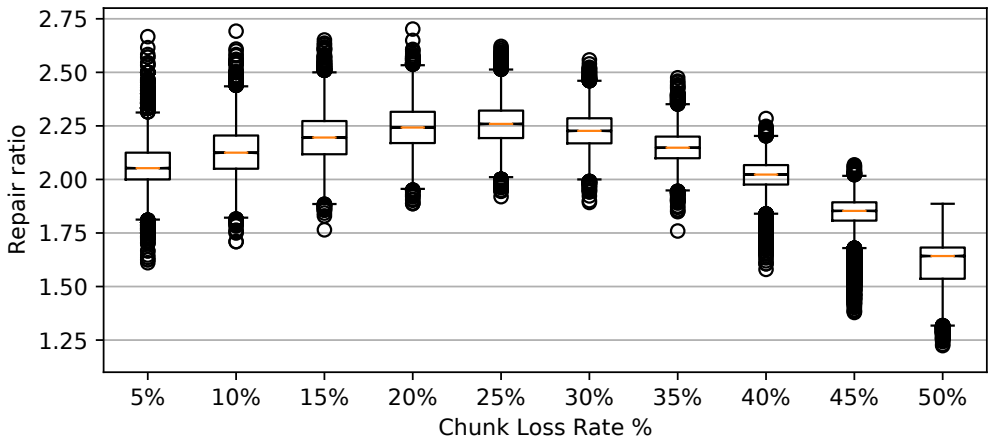


Figure 11: Repair overhead for a 1 MB file in Snarl with increasing levels of chunk loss.

plots our findings, showing that the repair ratio is slightly higher than 2 until there is a high number of chunk losses, after which it starts to decrease. It decreases because each parity is connected to two data chunks in the lattice, and thus if both of these data chunks are lost, Snarl will only retrieve this parity from the network for the first repair operation and then from the local storage for the second one.

6 Design Alternatives

We now discuss two interesting alternatives based on erasure coding to add resiliency to the Merkle tree.

6.1 Swarm Tree

Swarm tree is a design alternative outlined in [28, §5.1], which to our knowledge, has not yet been implemented. In Swarm tree, erasure coding is added at the system level to protect the tree from data loss. The tree's non-leaf nodes are encoded using a (128,112) MDS erasure code, i.e., each node can have 112 child nodes, with 16 entries for parities.

Since Swarm tree is a system-level redundancy scheme, it does not allow users to control the replication parameters of the coding scheme. Further, since each

internal node in the Swarm tree is part of a stripe with 112 data chunks and 16 parity chunks, users *must* retrieve parities, even if there are no failures. By encoding all redundancy information in the same Merkle tree as the original file, users *must* disclose resiliency levels publicly. As with any (n, k) MDS code, we need any k-of-n chunks to recover the stripe. Thus, to repair a single chunk, we first need to retrieve k chunks to recover the stripe and then reconstruct the chunk.

An unknown replication factor protects the root chunk, and thus we ignore this apparent weakness in our evaluations. Swarm tree's description states that there should be 16 parity blocks in each stripe, independent of data elements.

We can calculate the probability of an irrecoverable error for chunk losses by realizing that this scheme is equivalent to a composition of a sequence of MDS-stripes, i.e., a 1 MB file would be split into 4 stripes; $(128, 112)$-$(128, 112)$-$(48, 32)$-$(19, 3)$, where the first number inside each bracket is the stripe length n and the second number is the data elements k. The first three stripes contain the 256 leaf nodes, while the last stripe contains the internal nodes. Thus, an irrecoverable error in any of these stripes effectively renders the entire composition, or file, unavailable.

To evaluate the irrecoverable likelihood, we have developed equation (2), where c is the total number of chunk losses, S_p is the number of parity blocks in a stripe, S is the set of stripes and d is the set of chunk losses per stripe. The set D captures all possible combinations of chunk losses per stripe, e.g., given $|S| = 3$, we can have x chunk losses in stripe 1, y losses in stripe 2, and z losses in stripe 3, where $x + y + z = c$. Let L be a subset of D, as described by (2), containing those scenarios that lead to critical failure. The product $\prod_{n=1}^{|S|} \binom{S_n}{d_n}$ gives the number of combinations of the given failure scenario using multivariate geometric distribution, i.e., all possible ways to select x chunks from stripe 1, y from stripe 2, and z from stripe 3.

$$P(c) = \frac{\sum^{|L|} \prod_{n=1}^{|S|} \binom{S_n}{d_n}}{|D|}, \; where \; |D| = \binom{\sum S}{c}, c \leq \sum S, \tag{2}$$
$$L \leftarrow \{\forall d \in D : |d| = |S| \wedge \sum d = c \wedge \exists d_n \in d : d_n > S_p\}$$

Further, if we assume that the probability of chunk loss (b) is independent, all

stripes are of identical length (n), and with the same parity parameter (n-k), we can simplify (2) to (3). For a stripe to be available, we need at least k out of n elements, thus the sum $\sum_{k=0}^{n-k}$ the probability that we have up to k losses for a given chunk loss b. The product $\prod^{|S|}$ gives the recovery likelihood for the entire chain.

$$P(X) = 1 - \prod^{|S|} \sum_{k=0}^{n-k} \binom{n}{k} b^k \times (1-b)^{n-k} \tag{3}$$

Equation (3) shows the pitfall of this approach—*recovery likelihood decreases drastically with file size*. This is because Swarm's branching factor is constant at 128, and thus the only way to accommodate a larger file is to add branches, resulting in more stripes and higher $|S|$.

File availability given chunk loss is shown for various file sizes in Figure 12. In the legend, [sample] denotes results from random sampling; similar for [eq (2)] and [eq (3)]. Interestingly, results from (3) with stripes 20 and 200 fit nicely with results obtained from random sampling for file sizes 10 MB and 100 MB, respectively. From this, we can deduce that with a constant parity parameter, empty branches contribute little to the number of irrecoverable errors.

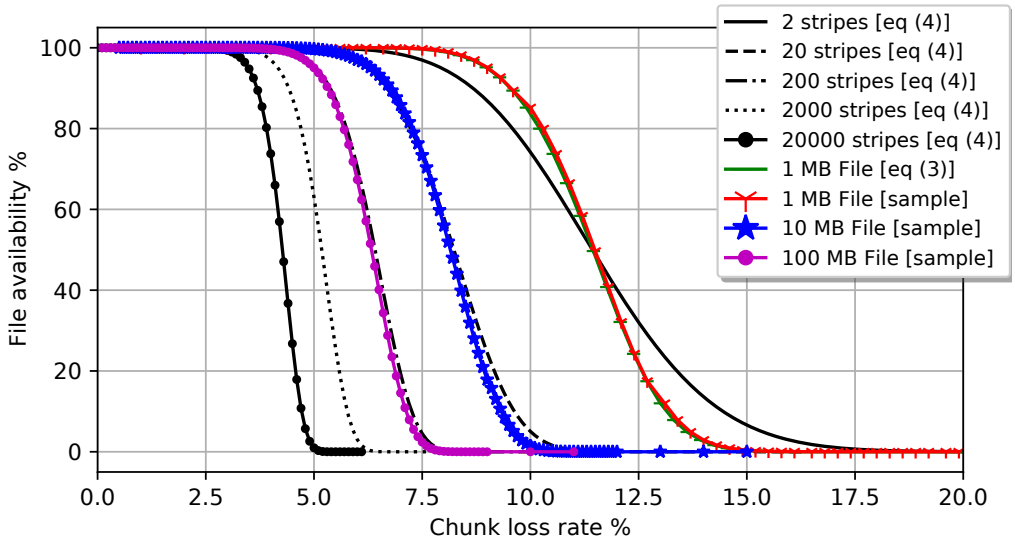


Figure 12: Availability of Swarm tree for various file sizes.

6.2 Coded Merkle Tree

Coded Merkle tree [34] is similar to Swarm tree in that they extend each level of the Merkle tree with additional erasure-coded blocks. There are two main differences between them. First, in Swarm tree, the input data is segmented into stripes of length 112 and extended with 16 parity blocks to 128 in total. In coded Merkle tree, all input data is placed in the same stripe and then extended with the appropriate number of parity blocks, depending on the coding rate ($r = k/n$). Secondly, Swarm tree always extends a stripe with 16 parity blocks, even if the stripe is not complete. In coded Merkle tree, the length of the extension depends solely on the coding rate, leading to several different coding settings.

7 Conclusion

This paper introduced Snarl, a user-controlled storage component that lets users control how to store data redundantly. Snarl can also improve storage utilization in cryptographic decentralized storage systems. These systems typically split the content into chunks and provide integrity verification and lookup services via a Merkle tree. The root and internal nodes are used to locate data in content-addressed storage and must thus be stored redundantly.

We have designed an entangled Merkle Tree, a resilient data structure that protects all nodes in the Merkle tree.

Based on our evaluation on Ethereum Swarm, we conclude that if every file was encoded with Snarl, five times as much data could be stored in the network, with a comparable failure-resiliency. We believe that these findings may prompt developers to incorporate Snarl into their systems.

Acknowledgments

We thank Leander Jehl for providing valuable feedback on earlier versions of this paper, Rodrigo Q. Saramago for technical assistance, and the members of the Swarm Foundation for helpful discussions. This work is partially funded by the BBChain and Credence projects under grants 274451 and 288126 from the Research Council of Norway.

References

[1] *Bee Nodes Live*. https://beenodes.live/. Accessed: 2021-06-01.

[2] Juan Benet. "IPFS-Content Addressed, Versioned, P2P File System". In: *arXiv preprint arXiv:1407.3561* (2014).

[3] Manuel Blum et al. "Checking the Correctness of Memories". In: *Algorithmica* 12.2 (1994), pp. 225–244.

[4] Scott Chacon and Ben Straub. "Pro Git". In: 2nd ed. 10.2. Apress, 2014. Chap. Git internals - Git objects.

[5] Bram Cohen. *Incentives Build Robustness in BitTorrent*. https://www.bittorrent.org/bittorrentecon.pdf. Accessed: 2021-10-26. 2003.

[6] Erik Daniel and Florian Tschorsch. "IPFS and Friends: A Qualitative Comparison of Next Generation Peer-to-Peer Data Networks". 2021.

[7] Vero Estrada-Galinanes, Jehan-François Pâris, and Pascal Felber. "Simple Data Entanglement Layouts With High Reliability". In: *2016 IEEE 35th International Performance Computing and Communications Conference (IPCCC)*. IEEE. 2016, pp. 1–8.

[8] 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)*. IEEE. 2018, pp. 183–194.

[9] Roy Friedman, Yoav Kantor, and Amir Kantor. "Replicated Erasure Codes for Storage and Repair-Traffic Efficiency". In: *14-th IEEE International Conference on Peer-to-Peer Computing*. IEEE. 2014, pp. 1–10.

[10] Eu-Jin Goh et al. "SiRiUS: Securing Remote Untrusted Storage". In: *NDSS*. Vol. 3. Citeseer. 2003, pp. 131–145.

[11] Helm. *The package manager for Kubernetes*. https://helm.sh/.

[12] Naoki Katoh and Shin-ichi Tanigawa. "Enumerating Constrained Non-crossing Geometric Spanning Trees". In: *Computing and Combinatorics*. Ed. by Guohui Lin. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 243–253.

[13] Kubernetes. *Production-Grade Container Orchestration*. https://kubernetes.io/.

[14] WK Lin, Dah Ming Chiu, and YB Lee. “Erasure Code Replication Revisited”. In: *Proceedings. Fourth International Conference on Peer-to-Peer Computing, 2004. Proceedings.* IEEE. 2004, pp. 90–97.

[15] Umesh Maheshwari, Radek Vingralek, and William Shapiro. “How to Build a Trusted Database System on Untrusted Storage”. In: *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation-Volume 4*. 2000.

[16] Petros Maniatis et al. “The LOCKSS peer-to-peer digital preservation system”. In: *ACM Transactions on Computer Systems (TOCS)* 23.1 (2005), pp. 2–50.

[17] Petar Maymounkov and David Mazieres. “Kademlia: A Peer-to-Peer Information System Based on the XOR Metric”. In: *International Workshop on Peer-to-Peer Systems*. Springer. 2002, pp. 53–65.

[18] Ralph C Merkle. “A Digital Signature Based on a Conventional Encryption Function”. In: *Conference on the theory and application of cryptographic techniques*. Springer. 1987, pp. 369–378.

[19] Satoshi Nakamoto et al. “Bitcoin: A Peer-to-Peer Electronic Cash System”. In: (2008).

[20] Sean C Rhea et al. “Pond: The OceanStore Prototype.” In: *FAST*. Vol. 3. 2003, pp. 1–14.

[21] Rodrigo Rodrigues and Barbara Liskov. “High Availability in DHTs: Erasure Coding vs. Replication”. In: *International Workshop on Peer-to-Peer Systems*. Springer. 2005, pp. 226–239.

[22] Bianca Schroeder and Garth A Gibson. “Understanding Disk Failure Rates: What Does an MTTF of 1,000,000 Hours Mean to You?” In: *ACM Transactions on Storage (TOS)* 3.3 (2007), 8–es.

[23] Thomas SJ Schwarz and Ethan L Miller. “Store, Forget, and Check: Using Algebraic Signatures to Check Remotely Administered Storage”. In: *26th IEEE International Conference on Distributed Computing Systems (ICDCS’06)*. IEEE. 2006, pp. 12–12.

[24] Mennan Selimi and Felix Freitag. “Tahoe-LAFS Distributed Storage Service in Community Network Clouds”. In: *2014 IEEE Fourth International Conference on Big Data and Cloud Computing*. IEEE. 2014, pp. 17–24.

[25] Gopalan Sivathanu, Charles P Wright, and Erez Zadok. “Ensuring Data Integrity in Storage: Techniques and Applications”. In: *Proceedings of the 2005 ACM workshop on Storage security and survivability*. 2005, pp. 26–36.

[26] V Trón et al. *Swap, swear and swindle: incentive system for swarm, May 2016*.

[27] Viktor Trón et al. *Swarm Documentation - Release 0.5*. `https://swarm-guide.readthedocs.io`. Accessed: 2021-05-25.

[28] Viktor Trón. *The Book of Swarm - v1.0 pre-release 7 November 17, 2020*. `https://www.ethswarm.org/The-Book-of-Swarm.pdf`. Accessed: 2021-10-26.

[29] Hakim Weatherspoon and John D Kubiatowicz. “Erasure Coding vs. Replication: A Quantitative Comparison”. In: *International Workshop on Peer-to-Peer Systems*. Springer. 2002, pp. 328–337.

[30] Wikipedia contributors. *Length extension attack — Wikipedia, The Free Encyclopedia*. `https://en.wikipedia.org/w/index.php?title=Length_extension_attack&oldid=934998209`. [Online; accessed 26-October-2021]. 2020.

[31] Zooko Wilcox-O’Hearn. *[tahoe-dev] erasure coding makes files more fragile, not less*. `https://tahoe-lafs.org/pipermail/tahoe-dev/2012-March/007185.html`. Accessed: 26.10.2021.

[32] Zooko Wilcox-O’Hearn and Brian Warner. “Tahoe – The Least-Authority Filesystem”. In: *Proceedings of the 4th ACM international workshop on Storage security and survivability*. 2008, pp. 21–26.

[33] Gavin Wood et al. “Ethereum: A secure decentralised generalised transaction ledger”. In: *Ethereum project yellow paper* 151.2014 (2014), pp. 1–32.

[34] Mingchao Yu et al. “Coded Merkle Tree: Solving Data Availability Attacks in Blockchains”. In: *International Conference on Financial Cryptography and Data Security*. Springer. 2020, pp. 114–134.

Paper 2

SNIPS: Succinct Proof of Storage for Efficient Data Synchronization in Decentralized Storage Systems

Racin Nygaard, Hein Meling

To be submitted [52]

Abstract

Data synchronization in decentralized storage systems is essential to guarantee sufficient redundancy to prevent data loss. We present *SNIPS*, the first succinct proof of storage algorithm for synchronizing storage peers. A peer constructs a proof for its stored chunks and sends it to verifier peers. A verifier queries the proof to identify and subsequently requests missing chunks. The proof is succinct, supports membership queries, and requires only a few bits per chunk.

We evaluated our SNIPS algorithm on a cluster of 1000 peers running Ethereum Swarm. Our results show that SNIPS reduces the amount of synchronization data by three orders of magnitude compared to the state-of-the-art. Additionally, creating and verifying a proof is linear with the number of chunks and typically requires only tens of µs per chunk. These qualities are vital for our use case, as we envision running SNIPS frequently to maintain sufficient redundancy consistently.

1 Introduction

Decentralized storage systems such as Ethereum Swarm [52], Filecoin [32], and InterPlanetary File System (IPFS) [6] are built on top of large peer-to-peer networks. The network's peers collaborate in pooling their storage capacity to provide a single storage system. The peers in Filecoin and Swarm are rewarded to incentivize their contribution of storage and bandwidth to the network. Peers do not trust each other; instead, they rely on cryptographic primitives to verify the integrity of the data they are tasked to store. The integrity verification is performed on fixed-size chunks obtained by splitting files. Chunks are immutable and are identified by their content address, a cryptographic hash of the chunk's content. Figure 1 illustrates the process of uploading a file to a decentralized storage system.

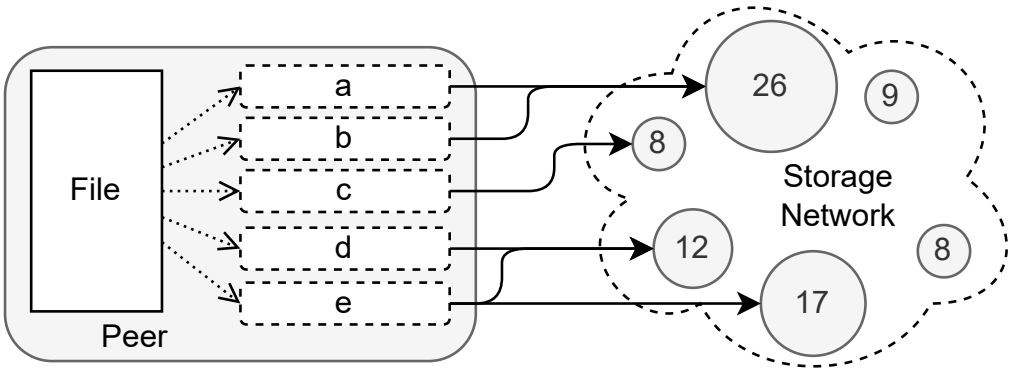


Figure 1: A file is split into chunks and distributed to neighborhoods of different sizes (number in circles).

During upload, each chunk is distributed to a neighborhood based on the chunk's content address. A neighborhood is a cluster of peers with similar addresses. These neighboring peers are responsible for storing the chunks whose content address is similar to the peers' addresses. To download a file, peers will similarly attempt to locate the neighborhoods storing the file's chunks. As Figure 1 shows, the neighborhoods can be of varying cardinality. Thus, the size of a chunk's neighborhood determines the chunk's replication factor. And since reconstructing a file requires access to all of its chunks, having sufficient redun-

dancy becomes paramount.

Recent studies [43, 28, 50, 27, 13] of the network dynamics of Ethereum Swarm and IPFS show that these systems may have tens of thousands of peers connected simultaneously. However, the studies also show high churn rates, where only a small minority of peers will remain online continuously for 24 hours. Consequently, peer neighborhoods are constantly changing, potentially compromising the availability of stored files. One study shows that data loss is inevitable, as it might happen as quickly as within a few days [43]. Hence, storage systems must maintain sufficient redundancy to reduce the likelihood of data loss. And maintaining redundancy requires *efficient data synchronization* methods.

State-of-the-art protocols for data synchronization in current systems are highly inefficient. In Swarm, the protocol is called Pullsync, and it relies on peers exchanging long lists of the chunk identifiers they are storing so that the other peer can request the missing chunks. In an attempt to optimize Pullsync, each peer keeps track of the last synchronization state of its neighbors. However, as we show in Section 2.3.3, this optimization is flawed, as inconsistencies can occur. Filecoin [32] and IPFS uses a protocol called Bitswap [15] for data synchronization. Like Pullsync, Bitswap also exchanges long lists of chunk identifiers for content discovery and synchronization. Bitswap is under active research [14, 34, 35], and various optimizations have been proposed, e.g., GraphSync [33] and CAR mirror [55]. However, these proposals still rely on exchanging long lists of chunk identifiers.

Previous work [1, 9, 17] on data synchronization in peer-to-peer networks proposed protocols using Bloom filters [7] for approximate reconciliation. As the name suggests, approximate reconciliation does not guarantee that all chunks are synchronized. A subset of a file's chunks could be enough to reconstruct the file if it was erasure-coded [45]. However, adding erasure coding to storage systems based on Merkle trees is non-trivial, because of hierarchical dependencies between the content and the metadata [42]. Hence, we focus on the case where all chunks must be synchronized.

A space-efficient way for peers to ensure they are fully synchronized is to use Proof of Storage (PoS) algorithms [2]. PoS algorithms allow peers to convince each other what data they are storing without transferring the data itself. These algorithms typically have three distinct actors; the *challenger*, which issues PoS queries; the *prover*, which responds to the queries by creating a proof;

and the *verifier*, which verifies the proof. However, verification typically either completely fails or passes; there is no partial verification. Even though a peer might have a subset of chunks, the verification will fail as the peer does not have all the chunks. Moreover, to synchronize data, the verifier would not only need to partially verify proofs but also determine which subset of chunks were missing. A naive PoS-based partial verification approach that creates a proof for each chunk would require communication overhead comparable to exchanging the chunk identifiers themselves.

1.1 Contributions

This paper presents SNIPS, a novel protocol for data synchronization in decentralized storage systems. SNIPS uses a novel PoS-like construction to generate succinct storage proofs. The proofs are small, typically only a few bits per chunk, and can be generated and verified efficiently. Our PoS construction is non-interactive, allowing verifiers to make use of the storage proof without a preceding *challenge* phase typically used in PoS algorithms. If both prover and verifier have an identical set of chunks, then both will immediately be convinced of each others state. Moreover, our construction also allows the verifier to perform membership queries on the proof to determine which chunks are missing, even when the prover and verifier have disjoint sets of chunks. The verifier may observe false positives for some local chunks that were not part of the prover's proof. However, our SNIPS protocol can reconcile these false positives.

Peers using the SNIPS protocol generate storage proofs and exchange them with their neighbors. Upon receiving a storage proof, the peer can query the proof to identify any chunks that it is missing and request the missing chunks from the sender. False positives are eliminated iteratively with increasingly more accurate proofs. In summary, a SNIPS proof provides two features; (1) it convinces the recipient that the sender has the data, and (2) the recipient can query the proof to determine which chunks it is missing.

Our implementation of SNIPS in Ethereum Swarm shows that it is a practical protocol that can be used in decentralized storage systems. The evaluations show significant bandwidth savings with a reduction in synchronization data transmitted by up to three orders of magnitude compared to current systems. Moreover, creating and verifying proofs typically requires only tens of microseconds per

chunk. While our evaluation does not compare with Bitswap and its derivatives, SNIPS's approach complements these protocols, and we expect Bitswap would also benefit from our approach.

Our contributions are summarized as follows:

- A new PoS-like construction for creating storage proofs amenable to partial verification and identification of missing chunks.
- The design of SNIPS, a data synchronization protocol for decentralized storage systems, and its implementation in Ethereum Swarm.
- A rigorous simulation of bandwidth usage and synchronization overhead and a performance evaluation of SNIPS on a real-world cluster.

2 Preliminaries

We assume a content-addressed decentralized storage system [6, 52] built on top of a peer-to-peer network, where each *storage peer* is connected to a subset of the peers in the network. That is, the peers are clustered into small neighborhoods, which collectively share the responsibility of persisting chunks from a specific section of the address space.

We assume no trust in the storage peers and the peer-to-peer network may experience significant churn. We rely on the storage system's underlying network to route messages between peers.

Each storage peer has access to cryptographic key pairs for signing and verifying digital signatures and we assume that cryptographic primitives cannot easily be circumvented. Let $\langle m \rangle_p$ denote a message m signed by peer p.

A peer's public key is further used to generate a unique address for use in the overlay network. This address serves as the baseline for the connectivity graph for the overlay network, such that peers are more likely to be connected to other peers with similar addresses.

In our algorithm, we assume that all peers have access to a shared source of randomness. One approach to obtain shared randomness is to use the block hash of a future block in a public blockchain [46]. Alternative methods include multi-party computation [10], Shamir's secret sharing [48], verifiable delay function [8], or verifiable random functions [40].

We define the following metrics for evaluating SNIPS.

Definition 1 (Similarity) *The similarity between two sets, A and B, is the overlap coefficient [53];* $|A \cap B|/\min(|A|, |B|)$.

Definition 2 (Proof Accuracy) *Let M_i be the set of missing chunks identified when querying a proof. Let M_p be the set of chunks a peer is missing from a proof. The proof accuracy is then* $|M_i|/|M_p|$.

2.1 Proof of Storage Model

In our PoS model, there are only provers and verifiers, also referred to as storage peers. Our PoS-like construction allows peers to (i) efficiently synchronize chunks and (ii) verify the integrity of their stored chunks with their neighbors. We assume that peers have full copies of the chunks they store.

Our construction does not provide storage guarantees to clients, as we do not rely on a challenge phase. We assume clients upload their data to the storage system. Moreover, chunks in decentralized storage systems are stored on different peers independently of their location within a file. Thus, our PoS scheme does not bind chunks to a specific location within a file, as it is not required for our use case. Other protocols can provide these guarantees to clients [43, 2].

2.2 Threat Model

We consider the threats that can be waged against SNIPS by insider attackers (peers). We do not consider outsider attackers without credentials, as the storage peers will reject messages from such attackers. Colluding peers is also not considered; other mechanisms are needed to mitigate collusion attacks (see Section 7.2). Denial of service attacks is out of scope for this paper, as the underlying storage system should mitigate such attacks. We focus on forms of attack that aims to compromise the integrity of the storage peers' stored data:

Replay attacks. We consider peers attempting to replay protocol messages to trigger illicit behavior.

Upload attack. After receiving a request for a missing chunk, an attacker may attempt to upload a different chunk.

Pollution attack. Malicious peers may attempt to pollute the storage system by creating and distributing invalid chunks.

Non-repudiation attack. A SNIPS proof can be viewed as a commitment to synchronize the containing chunks. We consider a non-repudiation attack, where the attacker attempts to modify the containing chunks after distributing the proof.

Moreover, an attacker may attempt to construct a storage proof that results in a false consistency (see Section 3.4). Section 4.5 discusses these threats and their mitigations.

2.3 Swarm

Swarm [52] is a global decentralized storage and communication system that distributes stored data to a network of peers. Swarm's p2p overlay network is based on Kademlia [39], and has more than 2000 active peers [25].

Each peer in Swarm deploys a smart contract, called checkbook, to an EVM-compatible blockchain, e.g., Ethereum or Gnosis. The contract is used to reward peers with BZZ tokens when they contribute resources to the network. Specifically, peers pay to download chunks and are rewarded for delivering chunks and forwarding messages. After deploying the contract, each peer generates a unique peer address used to identify it in the network. The address is generated by hashing the concatenation of the peer's Ethereum public key, the network identifier, and the hash of the block immediately following the one that deployed the peer's checkbook contract. The peer is then placed in the neighborhood that shares the longest prefix with the peer's address.

2.3.1 Data Storage in Swarm

Swarm splits files into 4 KB chunks. A unique *chunk identifier* is derived for each chunk from a cryptographic hash of its content, also referred to as its content address. Swarm creates a 128-ary Merkle tree where each of the file's chunks is a leaf. The internal nodes and root of the Merkle tree are also stored in 4 KB chunks and contain a concatenation of the chunk identifiers of their children.

Chunk identifiers and peer addresses share the same address space. When a chunk is uploaded to the network it is sent to the closest peer, based on the chunk's identifier [49]. The chunk is then replicated by each peer in the closest peer's neighborhood. Figure 2 shows the distribution of neighborhood sizes obtained in our cluster of 1000 peers. We observed 81 distinct neighborhoods of various sizes (n) with $n \in [8, 26]$. As chunks are replicated by all peers in a neighborhood,

the varying sizes of the neighborhoods cause the chunks' replication factor to vary accordingly.

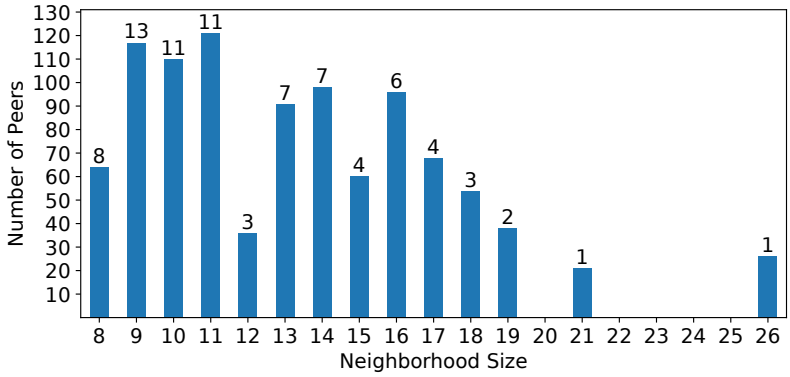


Figure 2: Distribution of neighborhood sizes in our cluster.

2.3.2 Swarm's Pullsync Protocol

The goal of Swarm's Pullsync protocol [52, 49] is to synchronize chunks between neighboring peers. Pullsync aims to provide eventual consistency among peers within a neighborhood despite churn. However, Pullsync does not scrub or verify the integrity of the stored data. Figure 3 gives an overview of the protocol; the following is a simplified description of the steps.

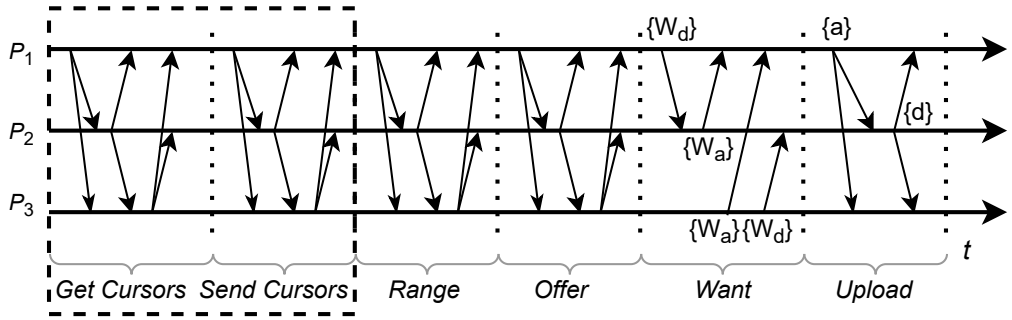


Figure 3: Overview of Swarm's Pullsync protocol.

All peers within a neighborhood run Pullsync with each other. Each peer monitors the network for topology changes, such as peers joining or leaving, and trig-

gers the Pullsync protocol on such events. Consider two peers P_1 and P_2 responding to a topology change.

Peer P_1 sends a request to its neighboring peers to get the number of chunks (*cursor*) stored in its neighborhood. Based on the received cursors, P_1 determines if it is missing any chunks, and which peer has the chunks that it may need.

Upon receiving a request for a range of chunks, peer P_2 creates an *offer* request containing an array of chunk identifiers in the range. When P_1 receives the offer request, it populates a bit vector with 1s for the chunk identifiers it is missing, and 0s otherwise. The bit vector is sent back to P_2, which will retrieve the missing chunks and deliver them to P_1, concluding the synchronization process.

2.3.3 Inconsistencies in Pullsync

Pullsync's inconsistencies can be explained by examining the first three phases in Figure 3. The *Get Cursors* and *Send Cursors* are only sent the first time two peers synchronize. After this, each of the peers will keep the other's cursor in memory, and use those for the *Range* phase.

The issue with this arises when a peer deletes a chunk. Deletion of chunks may be caused by malicious behavior or errors in the Swarm Bee client. However, deletions may also be completely benign, such as due to garbage collection or mechanisms in the incentive layer [22, 52]. In addition, the Swarm Bee client exposes an API for the user to delete chunks.

When a peer deletes a chunk, unless it is the last chunk added to the peer, the internal cursors will not be updated. Therefore, no matter how many times the peer runs the Pullsync protocol, it will not synchronize the deleted chunk. This causes inconsistencies in which chunks are stored by the neighborhood, and may ultimately result in chunks being lost.

2.4 Minimal Perfect Hash Function

Our enhanced PoS construction described in Section 3 uses a Minimal Perfect Hash Function (MPHF) [18, 37] to generate storage proofs. An MPHF is a bijective map from a set of N elements (keys) to the integers $[1, N]$ (index values). Each key of the set is mapped to exactly one value, and each value is paired with exactly one key. Figure 4 illustrates the mapping.

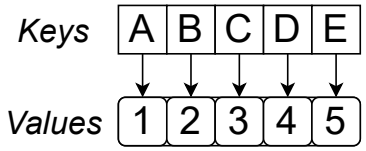


Figure 4: Minimal Perfect Hash Function bijective mapping.

The MPHF allows constructing a space-efficient, collision-free mapping of huge datasets with up to 10^{12} keys. While the theoretical limit is as low as 1.44 bits per key [4], practical implementations typically require a few bits per key. The MPHF is intended for static datasets and has no operations for adding, updating, or deleting elements. That is, the set of keys must be static and known in advance. The expected lookup time is $O(1)$, and a mapping can be constructed in $O(N)$ time.

Querying the map for a key that was part of the original set returns the corresponding value in the range $v \in [1, N]$. However, if the key was not part of the original set, it may return any value in the range $v \in [0, N]$. When the value 0 is returned, the key was not part of the original set; however, any other value $v \in [1, N]$ may be a false-positive. We define a Find function for MPHF:

$$\mathrm{Find}(elm) : \begin{cases} 0 & elm \text{ definitely not in the set} \\ v \in [1, N] & elm \text{ might be in the set} \end{cases}$$

3 Enhanced Proof of Storage Construction

We are now ready to introduce our enhanced non-interactive PoS construction. The construction generates proofs with the following properties: (1) it convinces the recipient that the sender has the data, and (2) the recipient can query the proof to learn which elements are missing. The construction consists of two algorithms executed by a *prover* and a *verifier* peer.

3.1 Prover Algorithm

The prover uses the CreateProof function in Algorithm 1 to generate a *storage proof* for the set of chunks, $\mathcal{CP}$, that the prover stores. The storage proof is con-

structed in two steps.

3.1.1 Create a proof of possession for each stored chunk

The prover computes a proof of possession for each chunk $a \in \mathcal{CP}$ using Equation (1), as shown in Lines 7-10 of Algorithm 1. We refer to this proof as the *chunk proof* [43].

$$\text{ChunkProof}_a : \quad cp_a = \text{H}(nonce \,||\, a) \tag{1}$$

The chunk proof is the cryptographic hash of a *nonce* (number used once) concatenated with the chunk's data. As long as the nonce is unpredictable, the prover cannot claim to store a chunk that it does not, since it must have both the chunk and nonce when generating the proof. We, therefore, derive the nonce from a shared source of randomness, as discussed in Section 2. To that end, the nonce captures the recency of the chunk proof. We discuss the frequency of proof generation in Section 4.4. As an optimization, the prover may generate a storage proof for a limited range $[start, end]$. In our evaluation, however, we use the entire range of chunks.

3.1.2 Compress possession proofs into a single storage proof

The prover compresses the chunk proofs to a succinct storage proof. We accomplish this by constructing an MPHF (Line 11), such that each chunk proof *cp* is mapped to a unique index $idx \in [1, N]$, where N is the number of chunks.

Finally, on Lines 12-14, we build a reverse mapping from the MPHF indices to the chunk identifiers. Find always returns a valid mapping here since the MPHF was constructed from the same chunk proofs. The prover can use this reverse mapping to identify the missing chunks requested by a verifier.

$$\begin{aligned} \textit{index-id} &: idx \mapsto \textit{chunk.id} \qquad \text{Reverse map} \\ \textit{mphf} &: cp \mapsto idx \end{aligned}$$

Algorithm 1 Prover: Storage Proof Generation

```
 1: Local persistent state at prover:
 2: CP                                              ▷ Set of chunks stored by prover
 3: index-id                                        ▷ Reverse map: MPHF index to chunk ID
 4: func CreateProof(nonce, start, end)
 5:   chunkProofs ← { }
 6:   cp-id ← ⟨ ⟩                                   ▷ Temporary map: chunk proof to chunk ID
 7:   for each chunk ∈ CP : chunk.id ∈ [ start, end ] do
 8:     cp ← H(nonce ‖ chunk)                       ▷ Chunk proof
 9:     chunkProofs ← chunkProofs ∪ cp
10:     cp-id[cp] ← chunk.id
11:   mphf ← new MPHF(chunkProofs)
12:   for each cp ∈ chunkProofs do                  ▷ Fill reverse map
13:     idx ← mphf.Find(cp)                         ▷ Index of cp in proof
14:     index-id[nonce][idx] ← cp-id[cp]            ▷ Save cp's chunk ID
15:   return [ mphf, nonce, start, end ]            ▷ Storage Proof
```

3.2 Verifier Algorithm

Next, we explain how the verifier can use a storage proof from the prover to identify chunks that the verifier may be missing using the FindMissingChunks function shown in Algorithm 2.

Given a storage *proof*, the verifier computes chunk proofs for each local $chunk \in CV$ over the proof's range. These chunk proofs are then used to query the prover-generated MPHF using the Find function (Line 8). Thus, if the chunk proof is in the MPHF proof, Find returns an $idx \in [1, N]$. Otherwise, $idx = 0$ is returned. We use *idx* as an index into the *mfc* array. On Line 10, the verifier decrements the *idx*-th entry in *mfc* to indicate that the local $chunk \in proof$. Once the verifier has processed all the chunk proofs, we use *mfc* to identify the missing chunks and collisions as follows:

$$mfc[idx] \begin{cases} = 1 & \text{verifier } \mathbf{m}\text{issing chunk for } idx \\ = 0 & \text{verifier } \mathbf{f}\text{ound chunk locally} \\ < 0 & \text{verifier has } \mathbf{c}\text{ollision(s) for } idx \end{cases}$$

That is, entries in *mfc* whose value is still equal to 1 must be missing (Line 15), and we add them to the *missing* set. Entries whose value equals 0 are not missing and require no further processing. For entries with a negative value (Line 17),

however, the verifier found two or more chunk proofs that mapped to the same index value, i.e., a collision. We discuss this issue in Section 3.3.

It is essential to recognize that the verifier cannot specify the missing chunks in the form of chunk identifiers or chunk proofs. Instead, the *missing* set contains index values that the prover can use to look up the missing chunks' identifiers in the *index-id* reverse mapping. Recall that the prover built the *index-id* on Lines 12-14 of Algorithm 1.

Algorithm 2 Verifier: Find Missing Chunks

```
 1: Local persistent state at verifier:
 2: CV                                                  ▷ Set of chunks stored by verifier
 3: func FindMissingChunks(proof)
 4:   size ← | proof.mphf |                             ▷ Number of chunks in proof
 5:   mfc ← [1, ..., 1]^size                           ▷ Initially, all chunks are missing
 6:   for each chunk ∈ CV : chunk.id ∈ [ proof.start, proof.end ] do
 7:     cp ← H(proof.nonce || chunk)
 8:     idx ← proof.mphf.Find(cp)                       ▷ Index of cp in proof
 9:     if idx ≠ 0 then                                 ▷ Might be in the set
10:       mfc[idx] ← mfc[idx] − 1                      ▷ Found chunk or collision
11:   missing ← { }
12:   collision ← false
13:   for i ← [1, size] do
14:     if mfc[i] = 1 then
15:       missing ← missing ∪ i                        ▷ Missing chunk
16:     else if mfc[i] < 0 then
17:       collision ← true                             ▷ Found collision
18:   return [ missing, collision ]                    ▷ Missing chunks
```

3.3 Collisions

This section explains why collisions can occur when verifying a proof. Recall that MPHF's Find function may return a false positive, which can happen when a chunk not part of the original set responds with "*might be in the set*." Formally:

Definition 3 (False Positive) *Let $\mathcal{CP}$ be the set of chunks used to construct the storage proof and let $cp \notin \mathcal{CP}$. Then we have a false positive if* $\mathrm{Find}(cp) > 0$.

Given that MPHF admits false positives, it is also possible to have a collision. A collision happens when multiple chunks map to the same index value in the

proof. Having a collision means that there is at least one false positive. Formally:

Definition 4 (Collision) *Let $\mathcal{CP}$ be the set of chunks used to construct the storage proof, and let $cp_a \in \mathcal{CP}$ and $cp_b \notin \mathcal{CP}$ such that $cp_a \neq cp_b$. Then we have a collision if:*

$$\text{Find}(cp_a) = \text{Find}(cp_b)$$

The prover and verifier independently build their chunk proofs from $\mathcal{CP}$ and $\mathcal{CV}$, respectively. Since $\mathcal{CV}$ may differ from $\mathcal{CP}$, and the verifier use chunk proofs from $\mathcal{CV}$ to query the prover's storage *proof*, the verifier may observe a collision if two or more chunk proofs map to the same index value (Line 8 in Algorithm 2).

We simulated a single peer verifying proofs of different sizes to determine the probability of false positives. As we can see from Figure 5, the probability of observing false positives increases with the number of chunks in the proof. Given a proof of 10^8 chunks, the probability of observing a false positive is 99.78 %. Although false positives and collisions can occur, their impact is limited as SNIPS uses a strategy of repeated proof generation until all missing chunks are discovered. We describe the strategy in Section 4 and evaluate it in Section 6.4. In fact, we conjecture that even if the proof had a 100 % probability of false positives, SNIPS would be able to discover all missing chunks. We reason by observing that even if false positives exist, it is still possible to discover missing chunks — unless these false positives result in false consistency, which we discuss next.

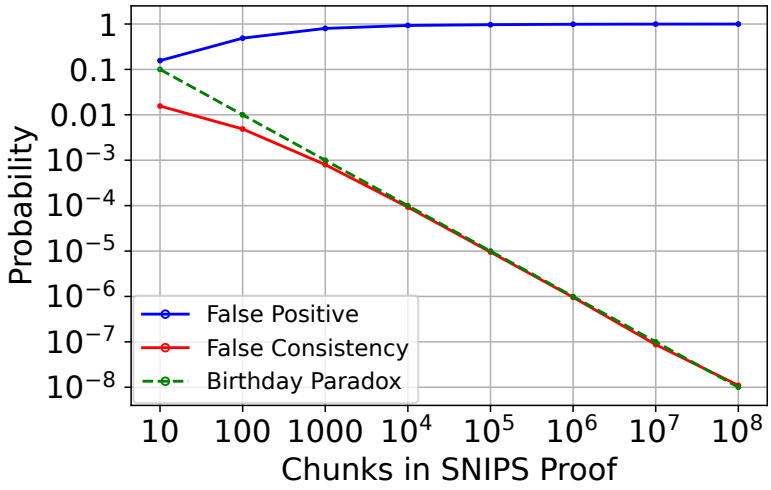


Figure 5: Probability of false positive and false consistency as a function of the number of chunks in the proof. The dotted green line shows an estimation of false consistency.

3.4 False Consistency

As we explained in Section 3.3, it is possible that the verifier peer can observe a collision if two or more chunk proofs map to the same index value. While such collisions are unlikely, they are easy to detect and recover from. In this section, we discuss *false consistency*—a related problem that may only occur when similarity is below 1, i.e., both peers have chunks that the other does not.

Definition 5 (False Consistency) *When a peer believes it has no missing chunks after querying a proof. However, at least one chunk used to query the proof gave a false positive, and the value of the false positive exactly matches the last remaining expected value.*

To explain false consistency, consider a proof exchange between two peers, P_1 and P_2. Peer P_1 stores chunks $\{a, b, c\}$ and peer P_2 stores chunks $\{a, b, d\}$, where chunk $c \neq d$. When peers P_1 and P_2 evaluate each other's *proof* to identify missing chunks, it is possible that they both have a false positive, resulting in two

distinct mappings, as illustrated here:

$$P_1 : [a \mapsto 3, b \mapsto 1, d \mapsto 2]$$
$$P_2 : [a \mapsto 1, b \mapsto 2, c \mapsto 3]$$

In this case, there are no collision mappings nor any mapping to 0. That is, P_1 believes it has all chunks in the storage *proof* sent by peer P_2 and vice versa. Thus, both peers falsely conclude that they have the same set of chunks.

Figure 5 shows the results of simulating a single peer verifying proofs of different sizes to determine the probability of false consistency. The probability of false consistency decreases linearly with an increasing number of chunks in the proof, and becomes negligible for a large number of chunks. As an example, for a proof with 1000 chunks, corresponding to a peer storing only 4 MB, the probability of observing a false consistency is less than 0.09 %.

Fortunately, as the number of chunks increases, observing a false consistency is unlikely in practice. By using a different nonce for each proof, there is a very high probability that the false consistency will be resolved in the next proof exchange.

We can model the probability of a false consistency using the *birthday paradox* [23], shown as a green line in Figure 5. The birthday paradox is a classic technique used to find the probability of a collision of two or more randomly chosen elements in a set. In our case, we let N be the cardinality of the set of all possible index values in the proof, and let k represent the number of peers exchanging proofs. The equation is given in Equation (2) as follows:

$$P(c) = 1 - \frac{N!}{(N-k)!}\frac{1}{N^k} \tag{2}$$

As there are only 2 peers exchanging proofs, we can simplify to get the expression $P(c)|_{k=2} = 1/N$.

4 The SNIPS Protocol

This section presents SNIPS, a data synchronization protocol for decentralized storage systems. SNIPS allows storage peers to perform periodic checks for missing chunks with negligible bandwidth overhead and without a preceding chal-

lenge phase.

SNIPS provides a mechanism for peers to exchange storage proofs, and to iteratively improve synchronization accuracy. Used together with the PoS construction in Section 3, we can use fewer bits per chunk in the storage proof. Thus, SNIPS helps to strike a balance between the size of the storage proof versus the iterations needed to synchronize the peers.

We first give an overview of SNIPS' three phases, as shown in Figure 6. While Figure 6 illustrates the different phases as if they are synchronized, SNIPS operates entirely asynchronously. Each peer can progress at their own pace.

In the *Prove* phase, each peer constructs a storage proof for its chunks. Peer P_1 constructs the proof $S_{a,b,c}$, representing that it stores the chunks $\{a, b, c\}$. Similarly, P_2 constructs $S_{b,c,d}$, and P_3 constructs $S_{b,c}$. The peers P_1, P_2, and P_3 send their storage proofs to the other two.

Upon receiving the storage proof $S_{b,c,d}$ from P_2, P_1 detects that it is missing chunk $\{d\}$ and requests it from P_2 with the R_d message in the *Select* phase. Similarly, P_2 detects that it is missing chunk $\{a\}$ and requests it from P_1. In the same way, P_3 detects that it is missing chunks $\{a, d\}$ and requests them from P_1 and P_2, respectively. Finally, the requested chunks are uploaded in the *Upload* phase, and the storage redundancy is recovered. Next, we explain each phase in more detail.

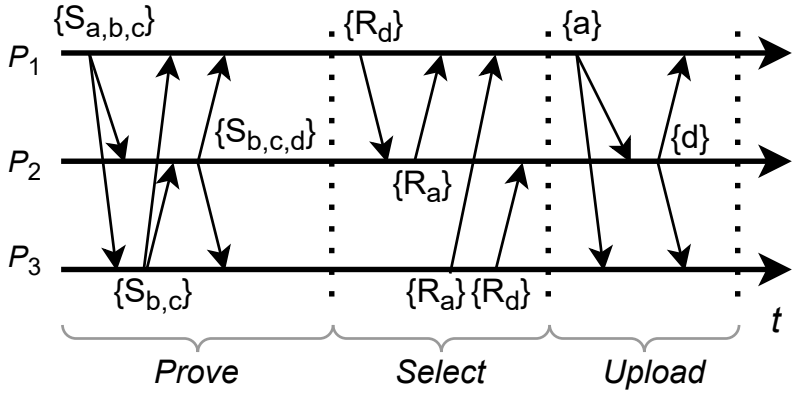


Figure 6: Space-time diagram of SNIPS's phases.

4.1 Prove Phase: Storage Proof Generation

In the Prove phase, shown in Algorithm 3, each peer constructs a storage proof for the set of chunks that the peer stores using the CreateProof function in Algorithm 1. The proof is then digitally signed to preserve its integrity and to link it to the prover's public key.

The Prove phase is triggered via the NewProof message in two ways: periodically from the local peer (see Section 4.4) or from another peer to quickly recover from a collision (see Line 8 in Algorithm 4).

Algorithm 3 Prove Phase: Storage Proof Generation

```
1: upon receive ⟨ NewProof | nonce, start, end ⟩ do
2:     proof ← CreateProof(nonce, start, end)
3:     send ⟨ Prove | proof ⟩_p                    ▷ Signed by peer p
```

4.2 Select Phase: Request Missing Chunks

In the Select phase, shown in Algorithm 4, a storage peer expects to receive Prove messages from other peers periodically. Upon receiving a storage *proof*, the peer calls the FindMissingChunks function to identify any potentially missing chunks. If the peer detects missing chunks, it can request them by sending a Select message to the peers that created the corresponding storage proofs.

As we mentioned in Section 3.2, evaluating the storage proof can result in collisions. The *collision* flag indicates this. Thus, to reconcile collisions, we trigger a new Prove phase (Line 8). However, we wait until the peer's requested chunks have been received, as indicated by the UploadDone message. Depending on the peer neighborhood's storage size to synchronize and the similarity between prover and verifier, it may require multiple iterations to reconcile all collisions.

Since the verifier only knows the prover's missing chunk indexes, we may request the same chunk from multiple peers. Hence, we sequentially process the proofs from different peers to avoid duplicate chunk uploads. However, we request chunks from the same peer concurrently. Upon receiving chunks, we evaluate them against all proofs.

Algorithm 4 Select Phase: Request Missing Chunks

```
1: upon receive ⟨ Prove | proof ⟩_p do                    ▷ Verified proof from peer p
2:     [ missing, collision ] ← FindMissingChunks(proof)
3:     if missing = ∅ then
4:         return                                          ▷ Nothing is missing
5:     reply ⟨ Select | proof.nonce, missing ⟩             ▷ Request chunks
6:     if collision then
7:         upon receive ⟨ UploadDone ⟩ do
8:             reply ⟨ NewProof | proof.nonce, proof.start, proof.end ⟩
```

4.3 Upload Phase: Send Missing Chunks

The Upload phase presented in Algorithm 5 is simple. In this phase, the prover responds to Select messages by uploading the missing chunks to the verifier.

The prover peer that created the storage proof stores the *index-id* reverse mapping for the nonce associated with the storage proof. Hence, to facilitate the upload, the prover uses the *index-id* mapping. We find the chunk identifier in the reverse mapping for each index value in the request's *missing* set. Then we use that chunk identifier to find the chunk in the local storage, $\mathcal{CP}$, and send that chunk to the requesting verifier peer. Finally, the UploadDone message signals to the verifier peer that the upload is done.

Algorithm 5 Upload Phase: Send Missing Chunks

```
1: Local persistent state at prover:
2: CP                                                    ▷ Set of chunks stored by prover
3: index-id                                              ▷ Reverse map: MPHF value to chunk ID
4: upon receive ⟨ Select | nonce, missing ⟩ do
5:     for each idx ∈ missing do
6:         chunkid ← index-id[nonce][idx]
7:         chunk ← {c : c ∈ CP ∧ c.id = chunkid}
8:         reply ⟨ Upload | chunk ⟩
9:     reply ⟨ UploadDone ⟩
```

4.4 Proof Generation Frequency

We briefly explore how to limit the execution frequency of SNIPS. For example, in a decentralized storage system, we envision SNIPS running approximately once per day.

To accomplish this, we can configure protocol execution to follow the block generation frequency of a public blockchain. For example, we can trigger the Prove phase by extracting a new nonce from the blockchain every k-th block. The nonce can be the hash of the blockchain's current block header. Since blocks are produced at regular intervals, we can limit protocol execution to the desired frequency. For example, Ethereum has an average block time of around 12 seconds [20, 19]. Thus, to generate a storage proof once a day, we can trigger execution every 7200-th block ($24\text{h} \cdot 60\text{m} \cdot 60\text{s}/12\text{s}$).

Having the peers issue proofs on the same interval for the same nonce allows the peers to amortize the cost of proof generation. We leave it for future work to explore how the peers can coordinate the proof generation to save costs.

4.5 Security Analysis and Mitigation Strategies

This section provides a cursory security analysis of SNIPS and some mitigations. We note that the underlying storage system already mitigates several attack vectors. For example, we do not consider omission and freeloading in this work, since the underlying storage system should mitigate these attacks. More work is needed to understand if SNIPS exposes additional attack vectors for such attacks and how to mitigate them. We analyze the storage peer integrity attacks outlined in Section 2.2 in the following.

Replay attack. An attacker can wage a replay attack by sending the same Prove or Select message multiple times. A peer may process these replayed messages without affecting the storage peer's integrity. However, such attacks impact the protocol's performance due to repeated work and added network traffic. We note that processing NewProof and Prove messages are more costly than processing Select messages (see Figure 10a). Thus, we may also rely on the storage system's mechanisms for rate-limiting peers.

An alternative mitigation strategy is discarding replayed messages based on the nonce and peer address. With this strategy, peers cannot retransmit messages for the same nonce and must wait for the next synchronization round instead.

Upload attack. A misbehaving peer may upload different chunks than those requested in the Select phase. To mitigate this, the receiving peer first checks that the uploaded chunks belong to its neighborhood. Next, the peer checks that the uploaded chunks map to the expected index values in the storage proof previously

received from the uploading peer.

To illustrate, consider the example in Figure 6, where P_2 sent a request for R_a and then received chunk a from P_1. Thus, P_2 can check that a maps to R_a in the original storage proof $S_{a,b,c}$ sent by P_1. If the mapping is incorrect, then P_2 rejects the chunk. If P_1 sent one of the other chunks in $S_{a,b,c}$ that P_2 already has, then P_2 will also reject the chunk.

Pollution attack. A misbehaving peer may attempt to pollute the storage system by creating invalid chunks, e.g., chunks not uploaded by a client. The attacking peer may include the invalid chunks in the storage proof and hope that other peers will request them. SNIPS does not prevent this attack; we leave it to the underlying storage system to mitigate it. A trivial mitigation strategy requires peers to embed the client's signature with each upload.

Non-repudiation attack. When a peer distributes a storage proof, it commits to the chunks in the proof. However, an attacker could attempt to deceive other peers into accepting a non-verified chunk by exploiting a collision in the proof sent during the Prove phase.

Consider a scenario with two peers, P_1 and P_2, where P_1 is the attacker. Let P_1 have the chunks $\{a, b\}$, and P_2 have $\{a\}$, but not b. At the start of the protocol, P_1 creates a storage proof $S_{a,b}$ and sends it to P_2. Next, P_2 sends a request R_b for chunk b from P_1. Assume now that P_1 can find a trojan chunk $b' \neq b$, such that $\mathrm{Find}(cp_{b'}) = \mathrm{Find}(cp_b)$. Finally, P_1 can upload b', and P_2 would accept b' as a valid chunk instead of b, even though b' is not contained in $S_{a,b}$.

We argue that this attack is difficult to perform in practice for typically sized storage systems. First, the attacker must find a candidate chunk b' that collides with b. Second, b' must have the same chunk identifier prefix as b; otherwise, b' would belong to a different neighborhood, and P_2 would not accept it. This involves hashing over each candidate b' until the resulting chunk identifier has a shared prefix. Then, the attacker must compute the chunk proof of b', as described in Section 3.1, and then determine if it collides with b. If the chunk proof does not collide, the attacker moves to the next candidate b'.

Given a prefix length of l bits and a proof of n chunks, the probability of finding a chunk b' with both a shared chunk identifier prefix and that collides with b is $1/(2^l n)$. Hence, given a 16-bit prefix and a proof of 1 million chunks, the probability of finding a trojan chunk is $1.53 \cdot 10^{-11}$.

To further mitigate the non-repudiation attack, e.g., for smaller-sized stor-

age systems, we can include a "proof checksum" in the storage proof. The proof checksum is a hash of the chunk proofs in the storage proof, i.e., $\mathrm{H}(cp_1 \parallel cp_2 \parallel \dots)$.

We mention briefly that an attacker may also attempt to exploit the possibility of a false consistency. However, this attack would be at least as costly as the non-repudiation attack.

5 Implementation

We implemented SNIPS as a package in Swarm Bee v1.9.0. The implementation comprises about 800 lines of Go code plus about 1900 lines for benchmarking, testing, and metrics collection. Our implementation is based on the protocol described in Section 4. For the MPHF, we used the implementation from [29] with the recommended expansion factor $\gamma = 2$.

In this section, we describe the implementation details of the protocol. We start by describing the protocol messages and the data structures used by the prover and verifier. Next, we describe how we implemented the prover and verifier, including some optimizations. Lastly, we describe some aspects of our evaluation framework.

5.1 Protocol Messages

We defined SNIPS' four messages using protocol buffers [24]. First, the Proof message contains an 8-byte nonce, two 32-byte start and end fields, a 4-byte entry for the proof's length, and a variable-length entry for the MPHF proof itself. Additionally, the proof is signed, adding another 97 bytes for a Swarm-specific signature. The Swarm-specific signature allows peers to verify that the proof's creator is within the same neighborhood.

The Select message only contains an 8-byte nonce and a bit vector. The bit vector allows us to efficiently select multiple chunks in the same message.

The NewProof message only contains an 8-byte nonce. This is a deviation from the protocol described in Section 4, where the NewProof message also contains the start and end of the proof. This is because, in our implementation, the prover keeps track of the start and end, and it is thus unnecessary to include them in the message. Lastly, the UploadDone message is only used as a signal and does not contain any data.

5.2 Prover

At some point, each peer in the network will act as a prover. Therefore, as soon as a peer starts, it will begin listening for new blocks to be added to the blockchain.

The other trigger to initiate proof generation is when a peer receives a New-Proof message. In this case, the peer will check if a proof for the same nonce already exists in the *id-proof* cache. If so, the peer will obtain the proof from the *id-proof* cache instead of generating a new one. Hence, we can amortize the cost of generating the chunk proofs by caching the chunk proofs. This is particularly useful if peers issue proofs on the same interval for the same nonce, e.g., every 7200-th block.

5.3 Verifier

The verifier uses the same *id-proof* cache as the prover. In addition, the verifier uses a bit vector to represent the missing chunks in the Select message instead of a list of indexes. We observe the benefit of using a bit vector increase as the number of missing chunks increases. The added overhead of the bit vector makes it comparable to sending the index for only a single missing chunk.

5.4 Blockchain as a Shared Randomness Source

To obtain a shared randomness source, we implemented a small probe that listens for new blocks from the Ethereum blockchain. Once a new block is received, the probe will determine if it is the next one in the interval by comparing its block number with the next expected block number. If the block number is greater or equal to the next expected block number, the probe will query the blockchain for the block hash of the next expected block number. Thus, even if we were slow to query the blockchain for new blocks, once we catch up, we will use the same block hash as the other peers for the nonce.

5.5 Evaluation Framework

We give a brief overview of the evaluation framework we implemented to evaluate the performance of SNIPS and Pullsync. To collect metrics, we use the Prometheus [44] monitoring system. We registered interesting metrics for SNIPS and Pullsync, such as the number of messages sent, the number of bytes in messages, the time to

process messages, and more. In addition, as the protocols do not wait for chunks to be synchronized before terminating, we used Prometheus to monitor the peers' activity to determine when synchronization was completed.

In both SNIPS and Pullsync, communication is contained within neighborhoods. To ensure that we only collected metrics from peers in the neighborhoods and did not have any outside interference, we added an API endpoint to the Swarm Bee client that allowed us to query the neighborhood of a peer.

6 Evaluation

This section presents our evaluation of SNIPS. Using a real-world Swarm deployment on our cluster, we measured SNIPS's performance in terms of the amount of transmitted data, bandwidth savings, synchronization time, computation time, and per-chunk bandwidth requirements. We compared our measurements to Swarm's Pullsync protocol. We evaluate full synchronization runs in scenarios with *chunk loss* (CL) and *adding new chunks* (CA).

Our results show that SNIPS uses up to three orders of magnitude less synchronization data than Pullsync. SNIPS's computational overhead shows that it is a practical protocol requiring only tens of microseconds per chunk to create and verify proofs. We also simulated the performance of SNIPS in the most challenging synchronization scenarios, where the *similarity* between peers is 0. The simulations show that SNIPS performs well under challenging conditions and can always efficiently synchronize peers. We varied the number of chunks so that the neighborhoods' total storage ranges from 1 MB to 1000 MB and up to 10 GB for some experiments.

6.1 Experimental Setup

The experiments were conducted on a cluster of 30 physical machines running Ubuntu 18.04.4 LTS. Each machine has 32 GB RAM, an Intel Xeon E-2136 3.30 GHz CPU, a 1.5 TB SSD disk, and 10 Gbit/s NIC. We used the cluster to run a large network of 1000 Swarm peers. Using Kubernetes [31] and Helm [26], we distributed the Swarm peers on 28 machines, using one to host a private Ethereum network and one for managing the experiment execution.

The distribution of our Swarm network is shown in Figure 2. There are 81 neighborhoods whose size varies between 8 and 26 peers. Since SNIPS's synchronization operations are confined to a single neighborhood, we evaluated the protocol with neighborhoods of $8, 17$, and 26 peers. Hence, a rough Fermi estimate for the system-wide bandwidth savings would be $81\times$ the individual savings of one neighborhood.

6.2 Consistent Pullsync

Comparing the performance of SNIPS and Pullsync is challenging due to Pullsync's inconsistencies. In addition, SNIPS has fewer phases than Pullsync, and the initiator of SNIPS wants to upload chunks, while the initiator of Pullsync wants to download chunks. To compensate, our duration measurements are taken from when the protocol is initiated until all peers are fully synchronized. However, this is not without caveats, as Pullsync gossips new chunks to its neighbors, reducing the chunk distribution time while potentially wasting bandwidth due to duplicate chunk uploads. While SNIPS could trivially implement a similar optimization, our priority is to reduce the amount of transmitted data.

For data transmission measurements, we were able to isolate data synchronization traffic from chunk uploads. However, to evaluate the chunk loss scenario, we modified Pullsync so that it does not cause inconsistencies. When new chunks are added, we compare SNIPS with vanilla Pullsync.

6.3 Real-world Comparison of SNIPS and Pullsync

We first compare SNIPS with Pullsync by measuring the *data transmitted* and *time to synchronize* a neighborhood under the same conditions in our cluster. We repeated the experiments 10 times for each configuration, and fixed the *similarity* between peers to 1, meaning that $|A \cap B| = \min(|A|, |B|)$.

In the CL scenario, peers initially store between 1 MB and 1000 MB of chunk data. We then varied the amount of chunk loss from 0 % to 100 %. Figure 7 (a, b, c) shows the *metadata transmitted* to synchronize a neighborhood of sizes $8, 17$, and 26, respectively. SNIPS (solid lines) transmits 2-3 orders of magnitude less metadata than Pullsync (dashed lines). The results are consistent for all neighborhood storage sizes.

Figure 8 (a, b, c) shows the *time to synchronize* a neighborhood of sizes $8, 17$, and 26, respectively. The synchronization time includes downloading the synchronization metadata and uploading the chunks. SNIPS is always faster than Pullsync for small storage sizes (1, 10 MB) and the 8-peer neighborhood. For the 17- and 26-peer neighborhoods and larger storage sizes (100, 1000 MB), Pullsync is slightly faster when there is more than 30 % and 50 % chunk loss. Pullsync is faster in these cases despite transmitting more metadata because of Pullsync's gossip optimization, explained in Section 6.2. Our experiments were performed on cluster nodes connected via unsaturated 10 Gbit/s links. We conjecture that the gossip optimization will be less effective in an Internet environment where peers are connected over lower-bandwidth links.

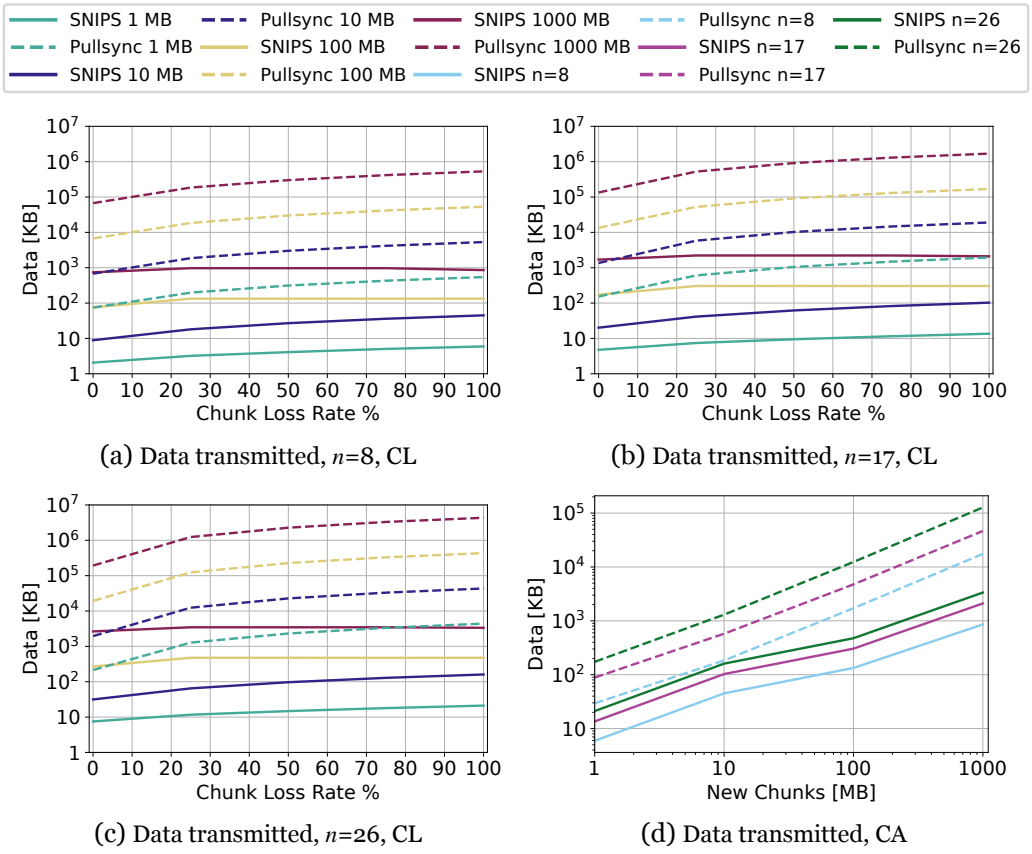
In the CA scenario, we add 1 MB to 1000 MB of new chunk data to a peer and observe the synchronization behavior. The *metadata transmitted* to synchronize a neighborhood of sizes $8, 17$, and 26 are shown in Figure 7d. As expected, the amount of metadata is linear with the new chunk data uploaded for Pullsync. However, SNIPS has a sublinear relationship, which becomes more pronounced for larger uploads. Overall, SNIPS transmits 1-1.5 orders of magnitude less data than Pullsync.

Figure 8d shows the *time to synchronize* a neighborhood of sizes $8, 17$, and 26. SNIPS is always faster than Pullsync for 10 MB of new chunks. However, due to the gossip optimization, Pullsync has a slight advantage when there are 100 MB or more new chunks.

We summarize the same results in Table 1 in the form of average *bandwidth savings* and *speedup*.

Table 1: Average bandwidth savings (KB) and speedup (ms) with SNIPS vs Pull-sync. RSE is the relative standard error.

		CL: Bandwidth Savings			CA: Bandwidth Savings		
Peers	Size	Avg (KB)	Avg (%)	RSE (%)	Avg (KB)	Avg (%)	RSE (%)
8	1 MB	308	98.7	0.4	23	79.8	0.6
	10 MB	2981	99.1	0.1	139	75.5	0.4
	100 MB	29849	99.6	0.1	1582	92.2	0.1
	1000 MB	298957	99.7	0.0	16630	95.1	0.3
17	1 MB	1038	99.1	0.4	75	84.7	0.2
	10 MB	10166	99.4	0.1	474	82.2	0.1
	100 MB	91024	99.7	0.0	4449	93.6	0.1
	1000 MB	909988	99.8	0.0	44463	95.5	0.0
26	1 MB	2293	99.4	0.3	152	87.8	0.6
	10 MB	22619	99.6	0.1	1142	87.7	0.2
	100 MB	226274	99.8	0.1	11907	96.2	0.1
	1000 MB	2265603	99.9	0.0	123972	97.4	0.1
		CL: Speedup			CA: Speedup		
Peers	Size	Avg (ms)	Avg (%)	RSE (%)	Avg (ms)	Avg (%)	RSE (%)
8	1 MB	25400	47.0	1.2	25069	46.6	1.4
	10 MB	31821	52.0	1.5	31859	50.6	2.2
	100 MB	89170	64.6	1.9	99949	59.1	2.1
	1000 MB	741126	73.0	1.5	651543	55.8	1.5
17	1 MB	24056	44.9	1.2	23150	43.4	1.0
	10 MB	20513	38.4	2.3	14385	27.2	3.7
	100 MB	-3833	-3.5	1.4	-56847	-37.5	0.6
	1000 MB	-156825	-20.7	1.5	-614137	-52.3	1.2
26	1 MB	955	3.1	1.3	1354	4.4	1.6
	10 MB	4451	11.1	2.5	-307	-0.7	3.2
	100 MB	96969	41.7	1.2	17451	6.6	1.3
	1000 MB	-29329	-2.8	1.6	-788973	-57.3	1.8



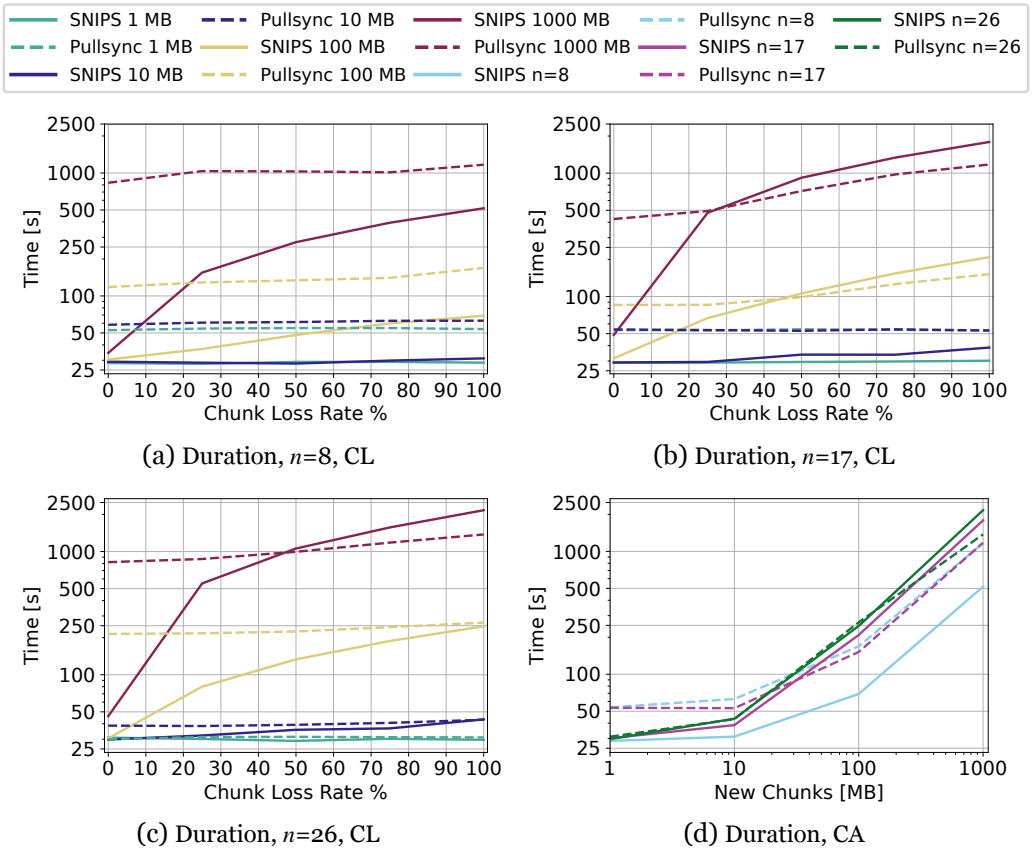
(a) Data transmitted, n=8, CL

(b) Data transmitted, n=17, CL

(c) Data transmitted, n=26, CL

(d) Data transmitted, CA

Figure 7: Comparison of data transmitted between SNIPS and Pullsync when synchronizing peer neighborhoods. We vary the neighborhood storage sizes between 1, 10, 100, and 1000 MB. The peer neighborhood sizes $n = \{8, 17, 26\}$. Plots (a, b, c) show the data transmitted as we vary the chunk-loss rate (CL). Plot (d) shows the data transmitted as we vary the amount of new chunk data added (CA).



(a) Duration, n=8, CL

(b) Duration, n=17, CL

(c) Duration, n=26, CL

(d) Duration, CA

Figure 8: Comparison of time spent between SNIPS and Pullsync when synchronizing peer neighborhoods. We vary the neighborhood storage sizes between 1, 10, 100, and 1000 MB. The peer neighborhood sizes $n = \{8, 17, 26\}$. Plots (a, b, c) show the duration as we vary the chunk-loss rate (CL). Plot (d) shows the duration as we vary the amount of new chunk data added (CA). The duration includes the time to transmit synchronization metadata and uploading chunks.

6.4 SNIPS With Varying Degree of Similarity

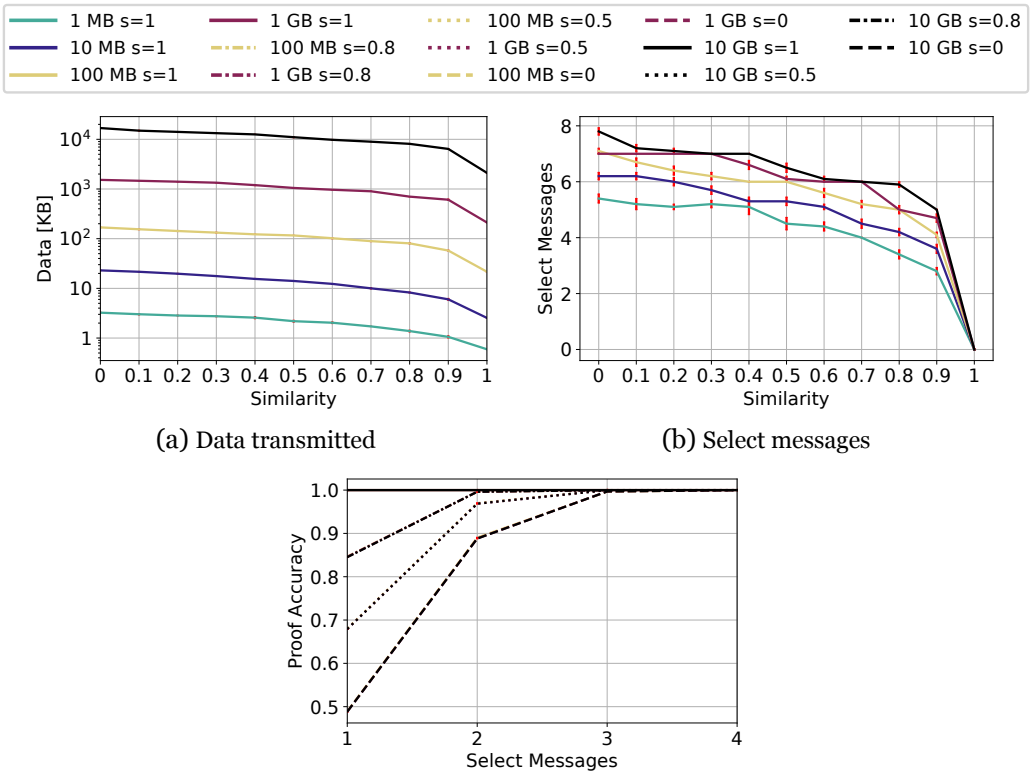
Recall the metrics *similarity*, $|A \cap B|/\min(|A|, |B|)$ and *proof accuracy*, $|M_i|/|M_p|$, defined in Section 2. We now simulate SNIPS's performance when similarity < 1, meaning both peers have chunks that the other does not. As Section 3.3 explains, this complicates the synchronization as querying with chunk proofs not part of the original proof may cause collisions.

We initialize the peers with the same amount of chunks for all simulations. Depending on the similarity setting, the total amount of chunks a peer is storing after concluding synchronization will range from the initial amount when similarity is 1 (chunks are fully overlapping) to $2\times$ the initial amount when similarity is 0 (chunks are fully disjoint).

We first simulate the transmitted synchronization metadata. As expected, the transmitted metadata per peer when similarity is 1 in Figure 9a closely matches the transmitted metadata when chunk loss is 0 % in Figure 7. For all storage sizes, the transmitted metadata increases by slightly less than one order of magnitude when similarity decreases from fully overlapping (1) to fully disjoint (0). The growth in transmitted metadata comes from the need for additional sequential Prove and Select phases until the peers have fully synchronized. For each round of Select, the peers become more synchronized, and their subsequent proofs become larger and more accurate.

Figure 9b shows the number of Select messages needed for two peers to synchronize fully. As the peers initially have the same amount of chunks, when similarity is 1, peers are already synchronized, and no Select messages are needed. The number of Select messages increases as the similarity decreases.

Finally, we evaluated the accuracy of the proofs, as shown in Figure 9c. Interestingly, the amount of chunks does not impact the proof accuracy. When similarity is 1, the proof accuracy is always 1, since all missing chunks are identified with a single Select message. Moreover, the proof accuracy improves with the number of Select messages. This is because each proof contains more chunks than the previous, allowing the peer to identify more missing chunks. The highest number of Select messages needed to fully synchronize was 4.



(a) Data transmitted

(b) Select messages

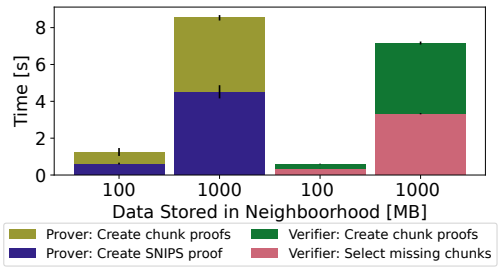
(c) Proof accuracy

Figure 9: SNIPS's performance with varying degrees of similarity $s = [0, 1]$. The labels define each peer's initial storage before the synchronization process begins. (a) Synchronization data for two peers. (b) Number of Select messages needed for two peers to fully synchronize. (c) Proof accuracy as a function of iterative Select messages. Vertical lines show the standard error.

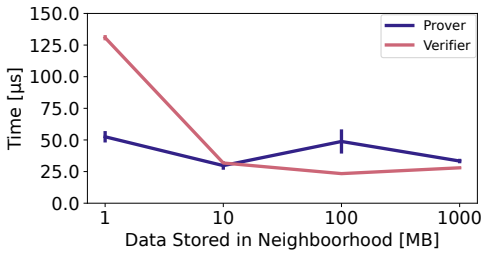
6.5 Computation Overhead

We measured the time spent creating and selecting missing chunks (verifying) from a SNIPS proof by extracting the time from the logs obtained from running the experiment in Section 6.3 on our cluster. We isolate the time spent creating chunk proofs as this is common for both operations. Our results are shown in Figure 10a and Table 2.

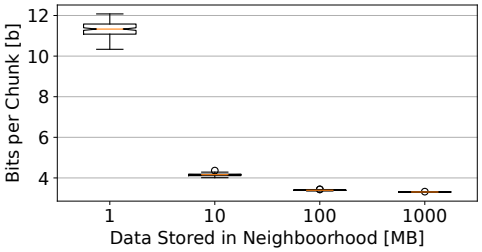
The results show that the verifier spends slightly less time creating chunk proofs than the prover. In addition, the time spent selecting missing chunks is slightly less than the time spent creating the proof.



(a) Time to create and select



(b) Time per chunk to create and select



(c) Per chunk overheads

Figure 10: Space and time overheads. (a) Time spent creating and selecting missing chunks from a SNIPS proof. (b) Time spent per chunk creating and selecting missing chunks from a SNIPS proof. (c) Per chunk overhead for a SNIPS proof for different neighborhood storage sizes (bits per chunk). Vertical lines represent error bars.

In comparison, we found that the average time to answer one peer with a long list of chunk identifiers in Pullsync's Offer phase amounts to 73.2 seconds for 100 MB of chunks and 728.6 seconds for 1000 MB of chunks. The Offer phase is executed partially in parallel and thus further study is needed to determine the exact time spent in sequential operations.

Next, we measured how the time spent correlates with the size of the proof. Figure 10b shows that only tens of microseconds are necessary for either operation. As the size of the proof increases, the time spent per chunk seems to stabilize. The time per chunk is slightly higher for 1 MB proofs due to the constant overhead for signing the proof. This overhead is amortized over many chunks for larger proofs, thus lowering the per-chunk time.

Table 2: Time spent creating and selecting missing chunks from a SNIPS proof.

Operation	Prover		Verifier	
	100 MB	1000 MB	100 MB	1000 MB
Chunk proofs	622 ms	4517 ms	359 ms	3327 ms
Create	628 ms	4017 ms		
Select			240 ms	3838 ms

6.6 Storage Overhead

We extracted the size of the SNIPS proofs from the experiments presented in Section 6.3. As the size of the proof is proportional to the number of chunks, we measured the storage overhead in terms of bits per chunk. Our results are shown in Figure 10c. The overhead approaches 3.3 bits per chunk for a proof containing 1000 MB of chunks. As in Section 6.5, the overhead for 1 MB of chunks is dominated by constant overheads. In this case, the higher number of bits per chunk for 1 MB of chunks is due to the digital signature, the nonce, and the $[start, end]$ range. However, for larger storage sizes, this overhead is amortized.

7 Related Work

We contrast our work with two main categories of related work: PoS constructions and data synchronization protocols.

7.1 Bloom Filter Variants and Set Reconciliation

A Bloom filter [7] is a probabilistic data structure for dynamic sets that supports a similar membership query as our PoS-like construction. The Bloom filter uses k hash functions to map each element to k bits in a bit array. When querying a Bloom filter, the results will be k array positions. If either of the k array positions is 0, then the element is *definitely not in the set*. Otherwise, we say that the element *might be in the set*, as the membership query suffers from false positives. The false positive rate of a Bloom filter can be improved by increasing the number of bits used per element.

A crucial difference between the Bloom filter and our PoS-like construction is that our membership query always returns an index value index value $[0, N]$ that

can be used to identify missing elements. As pointed out in previous works [16], it is not obvious how to identify missing elements from the Bloom filters array positions, other than testing membership on all possible elements in the universe.

An invertible Bloom filter (IBF) [16] is an extension to the standard Bloom filter that allows a query to extract the elements in the set—if the set is small enough. The IBF embeds an XOR field in each cell which is calculated over all the keys in the cell. Two IBFs can attempt to use the XOR fields to identify the elements in their symmetric difference.

Set reconciliation using IBF was suggested in [17]. Their results show that the method can only identify the missing elements with high probability when the symmetric difference is less than 30 %. In comparison, we have shown that SNIPS can always identify the missing elements, even when the symmetric difference is 100 % (the maximum).

A cuckoo filter [21] is an alternative to a Bloom filter that supports deletion and approximate membership tests. Elements are inserted into the cuckoo filter by hashing the element with two different hash functions and then mapping the hash value to at least one of the two resulting candidate buckets.

Recent work [38] has proposed *MCFSyn* for multi-party set reconciliation using marked cuckoo filters (MCF). Each peer generates an MCF vector that is sent to a centralized participant. The centralized participant then creates an overall MCF vector and distributes it to the peers. The peers can then identify the missing elements by comparing their MCF vectors with the overall MCF vector. The protocol requires that the set of peers are fixed for the duration of the protocol. Both false positives (identifying a missing element as being in the set) and false negatives (identifying an element as missing from the set) are possible in the protocol, resulting in excessive communication and inaccurate reconciliation. In contrast, SNIPS is a decentralized protocol that does not require a fixed set of peers and does not suffer false negatives.

7.2 Proof of Storage Algorithms and Accumulators

PoS algorithms [2, 30, 47] allow a peer to prove that it possesses a chunk of data without revealing the chunk itself. Three actors are involved in a PoS protocol: a challenger, a prover, and a verifier. As summarized in [54], the features, security, and performance of PoS algorithms vary greatly.

A related feature is *public verifiability*, first introduced in [2]. With this feature, anyone can take on the verifier role and verify the proof. Algorithms with this feature require that the original data owner generates and persists some metadata to be used by other peers to create storage challenges and to verify the proofs. The metadata adds extra overhead, and it needs to be generated for each data chunk that should be publicly verifiable. Should this metadata be lost, the original data owner must regenerate it. Our PoS-like construction allows anyone to possess the original data set to verify the proof without needing additional metadata.

We mention briefly that some PoS schemes [12, 11] can detect colluding peers by encoding each replica differently or by relying on timing assumptions. However, these reliances are unsuitable for decentralized storage systems.

An accumulator is a cryptographic construction representing a set of elements and allows the issuance of membership proofs without revealing the elements themselves [5]. A universal accumulator [36] allows a prover to generate both membership and non-membership proofs. Other variants include accumulators based on Bilinear maps [41] and Merkle trees [3]. Unlike our PoS-like construction, accumulators can give definite answers to membership queries. However, they would require a different witness for each chunk, which must be constantly updated, potentially overwhelming the system.

7.3 rsync

A popular tool for data synchronization is *rsync*. The rsync algorithm [51] is designed to reduce the amount of data needed to transfer files between two peers. Rsync achieves this by only transferring blocks not already at the destination and using a rolling checksum to detect changes within files. However, rsync is unsuitable for decentralized storage systems as it is limited to synchronizing one peer at a time.

8 Conclusion

This work presents SNIPS, a novel protocol for data synchronization in decentralized storage systems. Having efficient data synchronization that can run frequently and respond to network churn is paramount for keeping sufficient re-

dundancy levels in the storage system. SNIPS features a PoS-like construction for creating storage proofs that support membership queries. Peers exchange storage proofs with neighboring peers and query the proofs to identify missing chunks and subsequently request them. The proofs typically require only a few bits per chunk.

Our contribution includes rigorous experiments on a real-world cluster to show that SNIPS is a practical protocol. In addition, we have demonstrated the correctness of SNIPS by simulating the performance under worst-case scenarios.

We compare SNIPS to Pullsync, the state-of-the-art protocol for data synchronization in Ethereum Swarm. Our results show that Pullsync is vulnerable to inconsistencies, does not provide storage guarantees, and uses up to three orders of magnitude more synchronization data than SNIPS.

The potential impact of SNIPS on other decentralized storage systems seems promising but is yet to be fully explored. We believe there are other uses for the PoS-like construction, but we leave them for future work.

References

[1] Sachin Agarwal and Ari Trachtenberg. "Approximating the number of differences between remote sets". In: *2006 IEEE Information Theory Workshop-ITW'06 Punta del Este*. IEEE. 2006, pp. 217–221.

[2] Giuseppe Ateniese et al. "Provable data possession at untrusted stores". In: *Proceedings of the 14th ACM conference on Computer and communications security*. 2007, pp. 598–609.

[3] Foteini Baldimtsi et al. "Accumulators with applications to anonymity-preserving revocation". In: *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE. 2017, pp. 301–315.

[4] Djamal Belazzougui, Fabiano C Botelho, and Martin Dietzfelbinger. "Hash, displace, and compress". In: *European Symposium on Algorithms*. Springer. 2009, pp. 682–693.

[5] Josh Benaloh and Michael de Mare. "One-way accumulators: A decentralized alternative to digital signatures". In: *Workshop on the Theory and Application of of Cryptographic Techniques*. Springer. 1993, pp. 274–285.

[6] Juan Benet. “IPFS-Content Addressed, Versioned, P2P File System”. In: *arXiv preprint arXiv:1407.3561* (2014).

[7] Burton H Bloom. “Space/time trade-offs in hash coding with allowable errors”. In: *Communications of the ACM* 13.7 (1970), pp. 422–426.

[8] Dan Boneh et al. *Verifiable Delay Functions*. Cryptology ePrint Archive, Paper 2018/601. `https://eprint.iacr.org/2018/601`. 2018. url: `https://eprint.iacr.org/2018/601`.

[9] John W Byers et al. “Informed content delivery across adaptive overlay networks”. In: *IEEE/ACM transactions on networking* 12.5 (2004), pp. 767–780.

[10] Ran Canetti et al. “Adaptively secure multi-party computation”. In: *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*. 1996, pp. 639–648.

[11] Reza Curtmola et al. “MR-PDP: Multiple-Replica Provable Data Possession”. In: *2008 The 28th International Conference on Distributed Computing Systems*. New York, NY, USA: IEEE, 2008, pp. 411–420.

[12] Ivan Damgård, Chaya Ganesh, and Claudio Orlandi. “Proofs of Replicated Storage Without Timing Assumptions”. In: *Advances in Cryptology – CRYPTO 2019*. Cham: Springer, 2019, pp. 355–380.

[13] Erik Daniel and Florian Tschorsch. “Passively Measuring IPFS Churn and Network Size”. In: *arXiv preprint arXiv:2205.14927* (2022).

[14] Alfonso De la Rocha, David Dias, and Yiannis Psaras. *Accelerating Content Routing with Bitswap: A multi-path file transfer protocol in IPFS and Filecoin*. 2021.

[15] David Dias, Jeromy Johnson, and Juan Benet. *Bitswap - Protocol Specification*. Accessed: 2023-02-20. url: `https://github.com/ipfs/specs/blob/main/BITSWAP.md`.

[16] David Eppstein and Michael T Goodrich. “Straggler identification in round-trip data streams via Newton’s identities and invertible Bloom filters”. In: *IEEE Transactions on Knowledge and Data Engineering* 23.2 (2010), pp. 297–306.

[17] David Eppstein et al. “What’s the difference? Efficient set reconciliation without prior context”. In: *ACM SIGCOMM Computer Communication Review* 41.4 (2011), pp. 218–229.

[18] Emmanuel Esposito, Thomas Mueller Graf, and Sebastiano Vigna. “RecSplit: Minimal perfect hashing via recursive splitting”. In: *2020 Proceedings of the Twenty-Second Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM. 2020, pp. 175–185.

[19] *Ethereum 2.0 Phase 0 – The Beacon Chain*. Accessed: 2022-09-19. url: https://github.com/ethereum/consensus-specs/blob/v0.11.1/specs/phase0/beacon-chain.md%5C#time-parameters.

[20] *Ethereum Average Block Time Chart*. Accessed: 2022-09-19. url: https://etherscan.io/chart/blocktime.

[21] Bin Fan et al. “Cuckoo filter: Practically better than bloom”. In: *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*. 2014, pp. 75–88.

[22] Swarm Foundation. *The Mechanics of Swarm network’s Storage Incentives*. Accessed: 2022-12-07. url: https://medium.com/ethereum-swarm/the-mechanics-of-swarm-networks-storage-incentives-3bf68bf64ceb.

[23] Isidore Jacob Good. “Probability and the Weighing of Evidence”. In: (1950).

[24] Google. *Protocol Buffers*. Accessed: 2022-05-24. 2022. url: https://developers.google.com/protocol-buffers.

[25] Janoš Guljaš. *Network Statistics - Swarm Scan*. Accessed: 2022-12-06. url: https://swarmscan.resenje.org.

[26] Helm. *The package manager for Kubernetes*. url: https://helm.sh/.

[27] Sebastian Henningsen et al. “Crawling the IPFS network”. In: *2020 IFIP Networking Conference (Networking)*. IEEE. 2020, pp. 679–680.

[28] Sebastian Henningsen et al. “Mapping the interplanetary filesystem”. In: *2020 IFIP Networking Conference (Networking)*. IEEE. 2020, pp. 289–297.

[29] Sudhi Herle. *Fast Scalable Minimal Perfect Hash for Large Keysets*. Accessed: 2022-12-07. url: https://github.com/opencoff/go-bbhash.

[30] Ari Juels and Burton S Kaliski Jr. “PORs: Proofs of retrievability for large files”. In: *Proceedings of the 14th ACM conference on Computer and communications security*. 2007, pp. 584–597.

[31] Kubernetes. *Production-Grade Container Orchestration*. url: `https://kubernetes.io/`.

[32] Protocol Labs. *Filecoin: A Decentralized Storage Network*. Technical report. Accessed: 2023-02-20. July 2017. url: `https://filecoin.io/filecoin.pdf`.

[33] Protocol Labs. *Implementation of GraphSync Wire Protocol*. Accessed: 2023-02-20. url: `https://github.com/ipfs/go-graphsync`.

[34] Protocol Labs. *Project: Beyond Bitswap*. Accessed: 2023-02-20. url: `https://github.com/protocol/beyond-bitswap`.

[35] Zeeshan Lakhani. *Notes through the multiverse: IPFS ↔ CAR Pool ↔ CAR Mirror*. Accessed: 2023-02-20. url: `https://talk.fission.codes/t/notes-through-the-multiverse-ipfs-car-pool-car-mirror/3647`.

[36] Jiangtao Li, Ninghui Li, and Rui Xue. “Universal accumulators with efficient nonmembership proofs”. In: *International Conference on Applied Cryptography and Network Security*. Springer. 2007, pp. 253–269.

[37] Antoine Limasset et al. “Fast and scalable minimal perfect hashing for massive key sets”. In: *arXiv preprint arXiv:1702.03154* (2017).

[38] Lailong Luo et al. “MCFsyn: A multi-party set reconciliation protocol with the marked cuckoo filter”. In: *IEEE Transactions on Parallel and Distributed Systems* 32.11 (2021), pp. 2705–2718.

[39] Petar Maymounkov and David Mazieres. “Kademlia: A Peer-to-Peer Information System Based on the XOR Metric”. In: *International Workshop on Peer-to-Peer Systems*. Springer. 2002, pp. 53–65.

[40] S. Micali, M. Rabin, and S. Vadhan. “Verifiable random functions”. In: *40th Annual Symposium on Foundations of Computer Science (Cat. No.99CB37039)*. 1999, pp. 120–130.

[41] Lan Nguyen. “Accumulators from bilinear pairings and applications”. In: *Cryptographers’ track at the RSA conference*. Springer. 2005, pp. 275–292.

[42] Racin Nygaard, Vero Estrada-Galiñanes, and Hein Meling. “Snarl: Entangled Merkle Trees for Improved File Availability and Storage Utilization”. In: *Proceedings of the 22nd International Middleware Conference*. Middleware ’21. Québec city, Canada: Association for Computing Machinery, 2021, pp. 236–247. url: https://doi.org/10.1145/3464298.3493397.

[43] Racin Nygaard, Hein Meling, and John Ingve Olsen. “Cost-effective Data Upkeep in Decentralized Storage Systems”. In: *Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing*. Tallinn, Estonia, 2023. url: https://doi.org/10.1145/3555776.3577728.

[44] Prometheus. *Prometheus - Monitoring system & time series database*. url: https://prometheus.io/.

[45] Michael O Rabin. “Efficient dispersal of information for security, load balancing, and fault tolerance”. In: *Journal of the ACM (JACM)* 36.2 (1989), pp. 335–348.

[46] Arseniy Reutov. *Predicting Random Numbers in Ethereum Smart Contracts*. Accessed: 2022-12-05. url: https://blog.positive.com/predicting-random-numbers-in-ethereum-smart-contracts-e5358c6b8620.

[47] Hovav Shacham and Brent Waters. “Compact proofs of retrievability”. In: *International conference on the theory and application of cryptology and information security*. Springer. 2008, pp. 90–107.

[48] Ewa Syta et al. “Scalable bias-resistant distributed randomness”. In: *2017 IEEE Symposium on Security and Privacy (SP)*. Ieee. 2017, pp. 444–460.

[49] Swarm team. *Storage and Communication Infrastructure for a Self-Sovereign Digital Society*. Accessed: 2022-09-10. url: https://www.ethswarm.org/swarm-whitepaper.pdf.

[50] Dennis Trautwein et al. “Design and evaluation of IPFS: a storage layer for the decentralized web”. In: *Proceedings of the ACM SIGCOMM 2022 Conference*. 2022, pp. 739–752.

[51] Andrew Tridgell. “Efficient algorithms for sorting and synchronization”. PhD thesis. The Australian National University, 1999.

[52] Viktor Trón. *The Book of Swarm - v1.0 pre-release 7 November 17, 2020*. Accessed: 2022-09-10. url: https://www.ethswarm.org/The-Book-of-Swarm.pdf.

[53] MK Vijaymeena and K Kavitha. "A survey on similarity measures in text mining". In: *Machine Learning and Applications: An International Journal* 3.2 (2016), pp. 19–28.

[54] Anjia Yang et al. "Lightweight and privacy-preserving delegatable proofs of storage with data dynamics in cloud storage". In: *IEEE Transactions on Cloud Computing* 9.1 (2018), pp. 212–225.

[55] Brooklyn Zelenka. *CAR Mirror v0.1.0*. Accessed: 2023-02-20. url: https://github.com/fission-codes/spec/blob/main/car-pool/car-mirror/SPEC.md.

Paper 3

Cost-effective Data Upkeep in Decentralized Storage Systems

Racin Nygaard, Hein Meling, John Ingve Olsen

Published in ACM SAC 2023, DADS track [53]

Abstract

Decentralized storage systems split files into chunks and distribute the chunks across a network of peers. Each peer may only store a few chunks per file. To later reconstruct a file, all its chunks must be downloaded. Chunks can disappear from the network at any time as peers are untrusted and may misbehave, fail or leave the network. Current systems lack a secure and cost-effective mechanism for discovering missing chunks. Hence, a client must periodically re-upload all of the file's chunks to keep it available, even if only a few are missing from the network. Needlessly re-uploading chunks waste significant amounts of the network's bandwidth, takes additional time to complete, and forces the client to pay for unwarranted resources.

To address the above problem, we propose SUP, a novel protocol that utilizes proof-of-storage queries to detect missing chunks. We have evaluated SUP on a large cluster of 1000 peers running a recent version of Ethereum Swarm. Our contributions include the design and implementation of SUP and a study of Swarm's redundancy characteristics. Our evaluation shows that SUP significantly improves bandwidth utilization and time spent on data upkeep compared to the existing solution. In common scenarios, SUP can save as much as 94 % bandwidth and reduce the time spent re-uploading by up to 82 %. While dependent on the storage network's bandwidth pricing policy, using SUP may also reduce the overall monetary costs of data upkeep.

1 Introduction

In large-scale decentralized peer-to-peer storage systems such as *Ethereum Swarm* [43, 39] and *InterPlanetary File System* (IPFS) [7], peers collaborate by storing and serving each other data. These systems aim to pool the storage and computational resources of the peers together to create affordable and reliable storage for everyone. Anyone can upload files to the network, which are split into smaller chunks, as shown in Figure 1. Each chunk is replicated in a small subset of the total peers in the network.

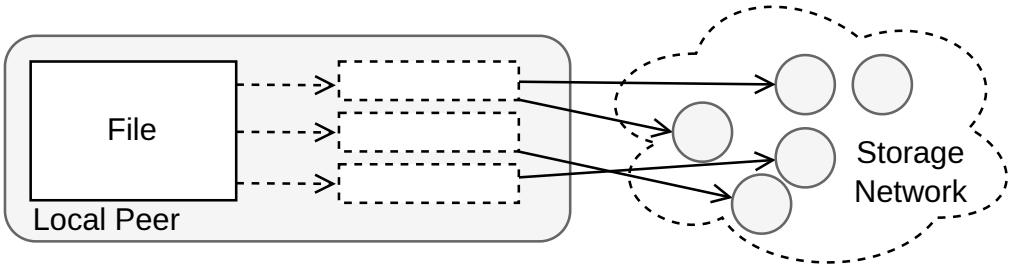


Figure 1: The local peer split files into chunks. The chunks are uploaded to different peers in the storage network.

The main challenge with this idea follows from the fact that the peers are untrusted, and the networks may exhibit significant churn. The peers may intentionally or by accident delete or corrupt the data they are tasked to store. Furthermore, a peer may itself become unreachable by the other peers due to issues with network connectivity, or it may have failed in other ways. To mitigate the unreliability of the peers, the incentives of the network motivate correct behavior with cryptographic tokens.

Even though previous work [29] has proposed solutions for data resilience, both IPFS [37] and Swarm still rely on a high degree of replication to keep data persistent. Without maintenance, as long as there are misbehaving peers and churn, the replication degree of chunks will decrease over time. The minimal replication of a file's chunks is also of particular interest, as the unavailability of a single chunk may cause the de-facto unavailability of the entire file. We have studied chunk availability in the public Swarm network and found that chunks can become unavailable as soon as 6 days after upload.

To maintain a high degree of replication, these systems rely on the client to continuously re-upload their data [20]. In Swarm, this process is called *Data Stewardship*. However, Data Stewardship always re-uploads all chunks of a file. For large files, this consumes unnecessarily large amounts of bandwidth. IPFS does not have a dedicated process for re-uploads [35, 42] and relies solely on its standard upload functionality.

In this work, we propose Storage Upkeep Protocol (SUP), a lightweight proof-of-storage system that uses storage proofs to save bandwidth when re-uploading data. A proof-of-storage system has three distinct actors, the challenger, the prover and the verifier. It is an essential requirement that the prover cannot precompute the proof before seeing the challenge. As such, each challenge is coupled with a unique nonce and thus a verified proof ensures that the prover had the data when the proof was created. SUP is targeted for usage in decentralized storage systems where the data is de-duplicated and peers are untrusted.

We evaluate SUP on a large cluster consisting of 1000 Swarm Bee peers. The results show that SUP can provide up to a 94 % reduction in the bandwidth consumed in the network when re-uploading. It also provides faster re-uploads; in common scenarios, the time spent re-uploading is reduced by up to 82 %.

Our contributions are summarized as follows:

- The design and implementation of SUP, a cost-efficient data upkeep protocol built for decentralized storage systems.
- A performance evaluation of SUP and Data Stewardship on a real-world cluster.
- Monitoring data availability over four weeks in the public Ethereum Swarm network.

2 Swarm Overview

In this section, we present an overview of the Swarm network and its data upkeep protocol, Data Stewardship.

Swarm [43] is a global decentralized storage and communication system that distributes stored data to a network of peers. According to a monitoring website [17], the public network has more than 2000 active peers in the past month,

and over 863 000 total peers. The peers connect to the Swarm network using the Swarm Bee client. Swarm's p2p overlay network used for routing and discovery is based on the Kademlia distributed hash table (DHT) [26].

Each peer in Swarm is required to deploy a smart contract, called checkbook, to an EVM-compatible blockchain, e.g., Ethereum or Gnosis. The smart contract is used to reward peers with BZZ tokens when they provide resources to the network. In the current version of Swarm, providing bandwidth is incentivized. Specifically, peers pay to download chunks and are rewarded for delivering chunks and forwarding messages. After deploying the smart contract, each peer generates a unique peer address used to identify it in the network. The address is generated by hashing the concatenation of the peer's Ethereum public key, the network identifier, and the hash of the block immediately following the one that deployed the peer's checkbook smart contract.

2.1 Data Storage in Swarm

Swarm splits files into 4 KB chunks. A unique *chunk identifier* is derived for each chunk by passing the chunk's content through a cryptographic hash function. Swarm creates a 128-ary Merkle tree where each of the file's chunks is a leaf. The internal nodes and root of the Merkle tree are also stored in 4 KB chunks and contain a concatenation of the chunk identifiers of their children.

Chunk identifiers and peer addresses share the same address space. When a chunk is uploaded to the network it is sent to the *closest peer*, whose address has the greatest proximity to the chunk's identifier. The *proximity* of two addresses is the number of equal prefix bits in both addresses [39].

Each chunk is replicated by its *storer peers*; these are the closest peer and that peer's *nearest neighborhood*. A neighborhood is a set of peers that have the same proximity to an address. A peer's nearest neighborhood is the neighborhood with 8 or more members that have the closest proximity to the peer's address.

Any peer may choose to store any given chunk. However, a chunk may be irretrievable by other peers unless it is stored by its storer peers. This follows from how messages are routed in Swarm, which we discuss next.

2.2 Message Routing in Swarm

The message routing protocol in Swarm, called *forwarding Kademlia*, differs from the original Kademlia description [26]. In the original description, the peer X, which wants to look up a value y in the DHT, starts by asking the closest peer it knows to y. This peer, which we call Z, then returns a set of peers Z' that are closer to y than itself. This process continues until the closest peer in the network, Y, is discovered. Finally, the request for y is sent to Y.

In forwarding Kademlia, instead of Z replying to X with a list of candidates that are closer to y, it will forward the request to one of them. That peer then forwards the request to the closest peer it knows. Eventually, the request reaches a peer Y that knows of no closer peer to y than itself. Then, Y returns y along the same path as the request, if it has y.

Kademlia allows peers to find chunks in logarithmically many steps; if the chunk is stored at the correct peers, that is, its storer peers as defined above. A chunk that is not stored by its storer peers is not guaranteed to be found through Kademlia routing. Peers may, however, choose to cache chunks to improve retrievability.

2.3 Data Stewardship

"It is in the nature of Swarm that data eventually disappears" [20]. Swarm clients use a mechanism called Data Stewardship [20] to mitigate this fact by periodically re-uploading their data to the storage network. This, of course, requires the client to have the data that it wishes to re-upload.

The client initiates Data Stewardship by specifying a chunk identifier. Data Stewardship then uploads the chunk to its storer peers using the *push-sync* protocol. If the chunk is the root of a Merkle tree, it uploads the entire tree.

2.4 Data Availability in Swarm

We conducted a four-week experiment on the public Swarm network to see how quickly chunks would disappear. We uploaded a 5 MB file to the *Swarm gateway* [40] and checked the availability of its chunks once a day. The chunk availability over time is shown in Figure 2. Green bars indicate that the whole file is retrievable, and red bars indicate that some chunks were unavailable. On the 6th

day, we observed that some of the chunks were unavailable, but they returned on the 7th day. After the 16th day, the file remained unavailable due to missing chunks.

We count chunks in one of 16 buckets (`0-f`), based on the first four bits of the chunk identifier, each time they are unavailable. Then, we normalized the buckets to account for the fact that there is some variation in how many of the file's chunks have the same 4-bit prefix. Finally, we plot the relative size of each bucket in Figure 3 as the *accumulated* (blue) bars. These bars show the frequency that chunks belonging to each bucket were found to be unavailable. We also count how many unique chunks fall into each bucket. That is, if the same chunk is unavailable multiple times, we count it only once. We normalize and plot the relative size of these counts also in Figure 3 as the *unique* (orange) bars.

In Figure 3, we see that chunks in buckets `c` and `e` were most often found to be unavailable. However, buckets `5` and `4` have the most unique chunks. Almost none of the unavailable chunks were found in bucket `8`. This seems to suggest that certain parts of the network are less reliable than others.

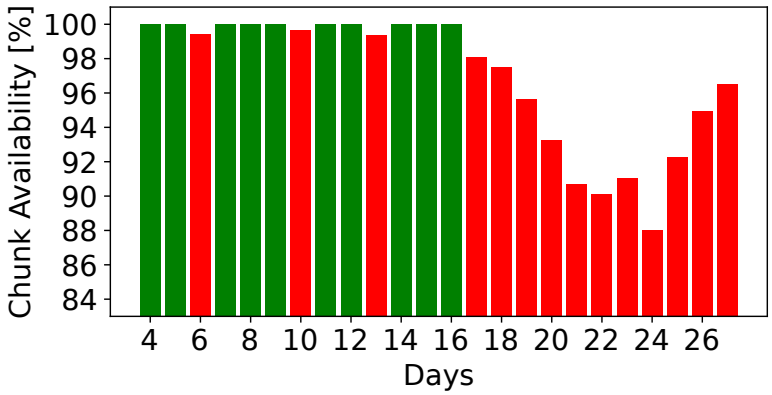


Figure 2: Chunk availability over time.

3 System Model and Requirements

We assume the presence of a p2p network, where each peer is connected to a subset of the total peers in the network. An overlay network is used to route messages between unconnected peers. Each peer generates a private key that is used to dig-

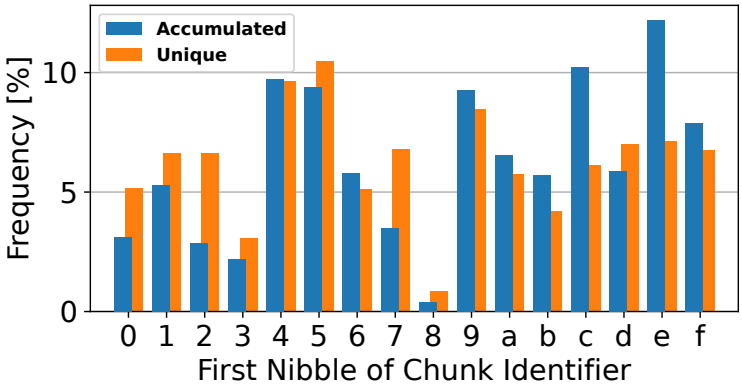


Figure 3: Faulty chunk identifier frequency.

itally sign messages, and a corresponding public key so that other peers can verify the signature. The key distribution and management are out of the scope of this paper.

The public key is further used to generate a unique address for use in the overlay network. The address serves as the baseline for the connectivity graph for the overlay network, such that peers are more likely to be connected to other peers with similar addresses.

There are three distinct actors in the protocol, the *Challenger* which issues proof-of-storage challenges, the *Prover* which receives the challenges and produces the proofs, and the *Verifier* which evaluates the correctness of the proof.

Requirements. 1) We design the protocol to prevent provers from computing valid proofs before receiving challenges. 2) A valid proof must ensure with an overwhelming probability that the prover was in possession of the data when the proof was created. 3) Proofs are bound to a specific prover, and cannot be reused by others. 4) Proving and verification must be efficient. 5) Proofs should be of a small constant size.

3.1 Threat Model

Challengers and provers in SUP may attempt to attack the system. Verifiers only receive proofs and thus cannot attack the system. A malicious challenger may trigger a Sybil attack by manufacturing a large number of challenges and sending these to the provers. The provers would then have to spend computational- and

I/O-resources to compute the proofs, which the attacker would discard. Individual provers may attempt to construct proofs for data they do not store. Similarly, a collection of provers may collude to answer proof-of-storage challenges for the same reason. Provers may also generate false proofs that verifiers would spend resources to reject.

We address malicious challengers in Section 4.3 and malicious provers in Section 4.2. We discuss colluding provers in Sections 4.2 and 4.4. Finally, we address false proofs in Section 4.5.

4 The Storage Upkeep Protocol

SUP is a novel protocol designed to save bandwidth and reduce time spent re-uploading data to decentralized storage networks comprised of untrusted peers. Re-uploading is defined as uploading previously uploaded data to the network and is often necessary to persist data, e.g., due to network churn or unreliable peers. The reduction in bandwidth and time spent comes from SUP's issuing of *storage challenges* and awaiting *storage proofs* before selectively uploading only those chunks that were missing or had invalid proof.

Our design is flexible, and modularized and does not require changes in the system's existing protocols. Instead, we add a new protocol to significantly reduce the amount of data transmitted during the re-uploading process.

SUP relies on the underlying P2P network to route messages between peers. The space-time diagram in Figure 4 illustrates SUP's protocol execution. In the figure, one entity acts as both the challenger and verifier, and three storage peers are labeled P_1, P_2, and P_3. The challenger, P_1 and P_2, all store a set of chunks labeled $\{a, b, c\}$, while P_3 only stores $\{a, b\}$. The protocol starts with the challenger sending a proof-of-storage challenge to the storage peers. Upon receiving the challenge, each storage peer computes a proof for those chunks it is storing and sends that proof to the verifier. The verifier processes each proof, and for P_3's proof, the verifier will detect that P_3 is missing chunk $\{c\}$. The verifier will then enter the re-upload phase and send the missing chunk to P_3.

Each peer has access to cryptographic keys for digitally signing and verifying messages. In the following algorithm description, we let $\Sigma(m)$ denote the digital signature of a message m and *pub* a peer's public key. The following subsections will

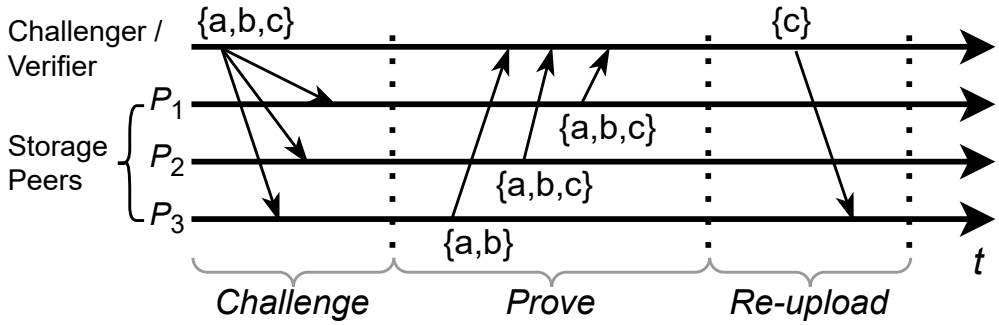


Figure 4: The message flow between the challenger, verifier, and the storage peers.

detail the most significant actors in the protocol.

4.1 Re-upload

Algorithm 1 gives a high-level pseudo-code for SUP. The client interacts with its local peer and calls `Reupload` with a list of chunk identifiers it wishes to persist in the storage network. If no list is given, the default is to use the identifiers of all chunks stored by the local peer. Given the list of chunk identifiers, the challenger creates a proof-of-storage challenge by calling `CreateChallenge` (Section 4.3). The challenge is then sent via the p2p network to the storage peers responsible for storing the chunks.

Each storage peer awaits challenge messages (Section 4.4) and generates a proof-of-storage proof for those chunks covered by the challenge that the peer is storing. Upon receiving a proof, the verifier will attempt to validate the proof using `VerifyProof` (Section 4.5). For each valid proof, the verifier removes chunk identifiers covered by the proof from the list of chunks to be re-uploaded.

Once the verifier has processed all proofs, the *uploadList* variable will only contain chunk identifiers that are missing from the network and must be re-uploaded. The verifier will iterate over the *uploadList* and upload the chunks to the storage peers.

We leave counting the proven chunk identifiers to get information about the redundancy in the network for future work.

Algorithm 1 Reuploader

```
1: Local persistent state at challenger/verifier:
2: C                                              ▷ Set of stored chunks
3: S                                              ▷ Set of storage peers
4: uploadList                                     ▷ Identifiers for chunks to upload
5: remaining                                      ▷ Outstanding storage proofs to verify
6: func Reupload(chunkIDs)
7:   if chunkIDs = ∅ then
8:     chunkIDs ← {c.id ∀ c ∈ C}                  ▷ Identifiers of all stored chunks
9:   chal ← CreateChallenge(chunkIDs)             ▷ Algorithm 2
10:  uploadList ← chunkIDs
11:  remaining ← |S|
12:  send ⟨ chal ⟩ to S                           ▷ Send chal to storage peers
13: upon receive ⟨ proof ⟩ do                     ▷ proof verified signature
14:   if VerifyProof(proof) then                  ▷ Algorithm 4
15:     for all chunkID ∈ proof do
16:       uploadList ← uploadList \ chunkID
17:   remaining ← remaining − 1
18: upon event remaining = 0 do                   ▷ All proofs processed or timeout occurred
19:   for all chunkID ∈ uploadList do
20:     chunk ← {c : c ∈ C ∧ c.id = chunkID}
21:     send ⟨ chunk ⟩ to S                       ▷ Upload missing chunk
```

4.2 Proof Construction

The primary purpose of our storage proof is to assure the challenger that the peer in question is currently storing the data queried in the challenge. To ensure that the peer is storing the data when generating the storage proof, we need to remove any way to pre-compute proofs before receiving the challenge. Resistance against pre-computing proofs is achieved by having the challenger embed a unique *nonce* (number used once) into each challenge query. The nonce is unknown to provers before they process the challenge.

The proof construction assumes the existence of a cryptographic hash function that is *preimage-resistant* [34] and *puzzle-friendly* [27]. The preimage resistance property states that for any given hash value h, it is computationally infeasible to find y such that $\mathrm{H}(y) = h$. Puzzle friendliness states that for every possible n-bit output value y, if k is chosen from a distribution with high min-entropy, then it is infeasible to find x such that $\mathrm{H}(k \,||\, x) = y$, in time significantly

less than 2^n. We define chunk a's chunk proof in (1), as the cryptographic hash of the concatenation of a nonce and a's data.

$$\text{CP}_a = \text{H}(\textit{nonce} \mid\mid a) \tag{1}$$

Hashing fixes the size of chunk proofs to a small constant. Note that the concatenation order *nonce* $\mid\mid a$ in (1) is essential, since otherwise the chunk proof may be vulnerable to the *length extension attack* [14]. This weakness is present in well-known iterative hash functions based on the Merkle-Damgård construction, such as MD5, SHA-1, and SHA-2. In the length extension attack, an attacker with knowledge of $\text{H}(m_1)$ and the length of m_1 can calculate $\text{H}(m_1 \mid\mid m_2)$ for an arbitrary message m_2 without knowing m_1. If we flipped the concatenation order in the chunk proof to be $\text{H}(a \mid\mid \textit{nonce})$, a dishonest prover could store only $\text{H}(a)$ and the length of a and delete chunk a. Upon receiving a new challenge, a prover could use the length extension attack to calculate $\text{H}(a \mid\mid \textit{nonce})$ without knowing a. We recommend computing the chunk proof in (1) using a strong cryptographic hash function without known weaknesses, such as the length extension attack.

The chunk proofs are aggregated to form a full storage proof for the challenge. We chose to aggregate the chunk proofs using XOR. Using XOR reduces memory consumption, as we only need to keep a small fixed amount of data in memory. The prover computes the hash of the concatenation of the aggregated chunk proofs and the prover's public key. Concatenating with the public key prevents other peers from copying the aggregated chunk proof to use it to claim they are storing the chunks. For a peer with the public key *pub*, the base proof for a set of chunks $\{a, b, c\}$ is given in (2).

$$\text{BP}_{a,b,c} = \text{H}((\text{CP}_a \oplus \text{CP}_b \oplus \text{CP}_c) \mid\mid \textit{pub}) \tag{2}$$

Since XOR is both associative and commutative, the order in which chunk proofs are aggregated does not impact the final result. We want that our scheme allows a prover to answer subsets of a challenge. For example, a challenger might issue a challenge for the chunks $\{a, b, c, d, e\}$, but the recipient peer can only prove $\{a, b, c\}$. To encode that only the chunks $\{a, b, c\}$ are included in the proof, we introduce a bitmap L. The i-th bit in L represents whether or not the i-th chunk is included in the proof. For example, $L = 11100_2$ means that chunks $\{a, b, c\}$ are

included in the proof.

In addition, we need to embed the nonce to allow the verifier to know which challenge is being proven, as the challenger might have issued multiple challenges. Lastly, we require that the prover digitally signs the base proof, providing message integrity and proof that it is authorized to issue proofs for the included public key. This allows the verifier to determine that the proof comes from the intended peer. Given a challenge requesting proofs for $\{a, b, c, d, e\}$, a peer storing chunks $\{a, b, c\}$ should respond with a proof according to (3), where bitmap $L = 11100_2$.

$$\text{Proof}_{a,b,c} = \{\text{BP}_{a,b,c},\ \textit{nonce},\ L,\ \Sigma(\text{BP}_{a,b,c}),\ \textit{pub}\} \tag{3}$$

4.3 Challenger

When a client wants to re-upload data using SUP, we send a challenge to the other storage peers to determine which chunks are missing from the network and must be uploaded. The challenge contains a list of chunk identifiers that the challenge concerns.

To prevent provers from pre-computing proofs, the challenger embeds a nonce in the challenge. The nonce is obtained by generating a random value and then cryptographically committing to the value by hashing it. The purpose of cryptographically committing is to be able to dispute any claim over who was the originator of the challenge, as only the challenger will be able to reveal the preimage of the committed nonce.

Finally, the list of chunk identifiers is packed together with the committed nonce and then digitally signed to create the challenge. By digitally signing the challenge, we can prevent unauthorized or excessive issuance of challenges. In addition, the signature allows provers to verify the integrity of the challenge. Algorithm 2 lists the pseudo-code.

4.4 Prover

Algorithm 3 lists the pseudo-code for proving chunk possession after receiving a challenge. The prover initially verifies the integrity of the challenge, and its digital signature to prevent misuse. For each chunk identifier contained in the challenge, the prover checks if it stores the chunk, and if so, it creates a chunk proof for it,

Algorithm 2 Create Challenge

```
1: Local persistent state at challenger:
2: commits                                          ▷ Map of nonce pre-images
3: challenges                          ▷ Map of chunk identifiers in sent challenges
4: pub                                              ▷ Challenger's public key
5: func CreateChallenge(chunkIDs)
6:   k ← RandomValue()
7:   nonce ← H(k)
8:   commits[nonce] ← k                             ▷ Commit to the nonce
9:   challenges[nonce] ← chunkIDs
10:  chal ← [ nonce, chunkIDs ]
11:  return [ chal, Σ(chal), pub ]
```

as described in (1), and then aggregates it to the base proof described in (2). The base proof is initialized to a null vector with the same length as the hash function.

Including both the committed nonce and the chunk data in the chunk proof means that the prover must know both simultaneously, making pre-computation infeasible. To aggregate the chunk proofs, we use XOR with the base proof. This reduces the algorithm's memory consumption, as the chunk's data can be garbage collected from memory after creating each chunk proof. The bitmap L indicates which chunks are included in the proof. As the chunks are processed in the same order as the challenge, we add 2^i to the bitmap to mark the i-th chunk as included.

After processing all the chunks, we cryptographically hash the concatenation of the base proof with the prover's public key to create the final base proof. Including the prover's public key prevents other peers from eavesdropping on the communication to re-use the proof for themselves. We then create the final proof using the final base proof, the received nonce and L. The final proof is returned to the challenger together with its digital signature.

4.5 Verifier

The pseudo-code for the verifier is listed in Algorithms 4 and 5. Any peer who stores the same chunks the storage proof covers can verify the proof. To verify a storage proof, the verifier calculates a new proof for the same chunks and then compares it to the received proof. When the proof covers fewer chunks than the challenge, we use the bitmap L to know which chunks are covered by the proof. The prover claims that the challenge's i-th chunk is covered by the proof only if L's i-th bit is set. The verifier returns *true* only if the received proof matches the

Algorithm 3 Prove Data Possession

Local persistent state at prover:
$\mathcal{C}$ ▷ Set of stored chunks
pub ▷ Prover's public key
upon receive $\langle$ *chal* $\rangle$ **do** ▷ *chal* verified signature
  $baseProof \leftarrow [0]^{\mathrm{H}.length}$
  $L \leftarrow 0$ ▷ Bitmap of chunks included in proof
  for $i \leftarrow 0$ to $|chal.chunkIDs|$ **do**
    $chunk \leftarrow \{c : c \in \mathcal{C} \wedge c.id = chal.chunkIDs[i]\}$
    if $chunk \neq \emptyset$ **then**
      $baseProof \leftarrow baseProof \oplus \mathrm{H}(chal.nonce \parallel chunk)$
      $L \leftarrow L \mid (1 \ll i)$ ▷ Add 2^i to L
  $baseProof \leftarrow \mathrm{H}(baseProof \parallel pub)$
  $proof \leftarrow [\, chal.nonce,\ L,\ baseProof \,]$
  reply $\langle\ proof,\ \Sigma(proof),\ pub\ \rangle$ ▷ Send *proof* to challenger

newly calculated proof.

Typically, a challenge is sent to multiple peers, and for verification of multiple proofs, the verifier can leverage caching to amortize the cost of I/O operations to fetch chunks. That is, the verifier caches each chunk proof so that it can verify multiple proofs from different storage peers without accessing its local storage for each of them. When iterating through the proof, each chunk proof is retrieved from the cache or computed on the fly and stored in the cache for future lookups.

Algorithm 4 Verify Proof

Local persistent state at verifier:
$\mathcal{C}$ ▷ Set of stored chunks
challenges ▷ Map of chunk identifiers in sent challenges
func VerifyProof(*proof*)
  $myProof \leftarrow [0]^{\mathrm{H}.length}$
  $chunkIDs \leftarrow challenges[proof.nonce]$ ▷ *chunkIDs* that may be in *proof*
  for $i \leftarrow 0$ to $|proof.L|$ **do** ▷ $|proof.L|$ is $\lfloor \log_2 proof.L \rfloor + 1$
    if $proof.L \,\&\, 2^i \neq 0$ **then** ▷ Chunk is included in proof
      $chunkProof \leftarrow$ GetChunkProof$(proof.nonce, chunkIDs[i])$
      if $chunkProof = nil$ **then**
        return *false* ▷ Error: Must store all proven chunks to verify
      $myProof \leftarrow myProof \oplus chunkProof$
  $myProof \leftarrow \mathrm{H}(myProof \parallel proof.pub)$
  return $myProof = proof.baseProof$

Algorithm 5 Get Chunk Proof

```
1: Local persistent state at verifier:
2: C                                              ▷ Set of stored chunks
3: chunkProofs                                    ▷ Map of all processed chunk proofs
4: func GetChunkProof(nonce, chunkID)
5:   if chunkProofs[nonce ‖ chunkID] = nil then
6:     chunk ← {c : c ∈ C ∧ c.id = chunkID}
7:     if chunk = ∅ then
8:       return nil
9:     chunkProofs[nonce ‖ chunkID] ← H(nonce ‖ chunk)
10:  return chunkProofs[nonce ‖ chunkID]
```

5 Implementation

We have implemented SUP as a package in Swarm Bee version 1.7.0 (released 24 July 2022). The implementation consists of about 700 lines of Go code, plus about 600 lines of code for benchmarking, testing and metrics collection. We have made the source code available at: (`https://github.com/relab/sup`).

5.1 Adapting SUP to Swarm

We presented SUP in Section 4 as a protocol suitable for use in a generalized decentralized storage network. To implement SUP in Swarm, we had to deviate slightly from the protocol specification. The following paragraphs explain how we implemented SUP in Swarm.

1) Our implementation sends *one chunk identifier per challenge* instead of multiple identifiers batched together in a single challenge. In Swarm, each chunk should be stored by its *storer peers*, as defined in Section 2.1. The challenger cannot accurately predict which chunks will share the same storer peers, due to its incomplete view of the network. Furthermore, the probability that any two chunks will share the same storer peers decreases as the network size increases. A possible way to implement batching is to split the batches if a forwarder notices that some chunk identifiers in the batch have different storer peers. Nevertheless, SUP significantly improves bandwidth use even without this optimization, compared to Data Stewardship. Therefore, we leave it to future work to implement this optimization.

2) We introduce a forwarder role to relay challenges from challengers to provers that are not directly connected. A challenger sends a challenge to the directly connected peer closest to the challenge's chunk identifier. A peer receiving a challenge becomes a forwarder of that challenge if it is directly connected to a peer in even closer proximity to the target chunk identifier; otherwise, the peer becomes the prover. Proofs are returned to the challenger through each forwarder who helped deliver the challenge.

3) The challenger is unlikely to be directly connected to the prover. A mechanism is needed for the challenger to verify that the proof comes from the correct part of the network, that is, the chunk's storer peers. As discussed above, the challenger cannot accurately predict the storer peers for a chunk. However, it can make an informed guess about the storer peers' proximity to the chunk. Peer addresses are uniformly distributed in Swarm. Challengers can expect storer peers to have at least the same proximity to the chunk identifier as the challenger's proximity to its nearest neighbors. Therefore, we require provers to include a *block hash* with each chunk proof. This should be the block hash used to generate the prover's peer address. The verifier can then recover the public key from the signature of the proof and combine it with this block hash to recover the peer address of the prover. The verifier can compute the prover's proximity to the chunk identifier using the recovered peer address. Based on the verifier's view of the network, the verifier can ensure that the prover is at least as close as the chunk's storer peers *should* be.

4) Finally, our implementation omits signatures of challenges. We note that the *request* messages in other Swarm protocols, such as push-sync and the retrieval protocol, do not include such signatures. We believe that including signatures would compromise *sender anonymity*, as the prover must identify the signer to verify a signature. Sender anonymity is a "crucial feature of Swarm" [43]. Therefore, as explained above, we chose to omit signatures on challenges, but they are still necessary on proofs. We believe an incentive system would encourage forwarder peers to forward challenges without modification. However, implementing such an incentive system is beyond the scope of this paper.

5.2 Structure

In our SUP implementation in Swarm Bee, the same peer acting on behalf of the client fulfills the roles of challenger and verifier. Other peers may assume the role of forwarder or prover, depending on their proximity to the target chunk in a challenge.

We expose a private HTTP API endpoint in the Swarm Bee peer as a direct replacement for Data Stewardship. When invoking the API, the client specifies the identifier of the chunk, or the root chunk of a file, to be re-uploaded. This chunk identifier is input into a `Reupload` function, which performs the roles of the challenger and verifier.

SUP registers a new protocol with the *libp2p runtime*. libp2p [25], is the networking library used by Swarm Bee. Whenever a peer receives a new connection using this protocol, the libp2p runtime invokes a handler function specified by SUP. This handler performs the functions of the forwarder and prover.

5.3 Challenger and Verifier

When SUP is invoked through the API, the challenger computes the list of chunks to re-upload. The client can specify an entire file by inputting the identifier of the file's root chunk. In this case, the challenger traverses down the entire Merkle tree that composes the file to collect all the chunk identifiers. Then, for each chunk identifier, the challenger invokes the `reuploadChunk` function.

The `reuploadChunk` function creates a challenge, consisting of a nonce and the chunk identifier, and then sends it to the closest peer and waits for a response. If a proof is received in response, the challenger verifies it and checks the address of the prover, as explained in Section 5.1. If no response was received, or the proof was invalid, `reuploadChunk` invokes push-sync to re-upload the chunk to the network.

5.4 Forwarder and Prover

SUP's libp2p handler function implements the forwarder and prover roles. The handler's job is to receive a challenge and then respond with a proof, if possible. As explained in Section 5.1, it takes on the forwarder role if it is directly connected to a peer closer to the chunk identifier. Otherwise, it takes on the role of prover

and generates a storage proof for the chunk if it has the chunk. In either case, the handler eventually sends the proof back to the peer from whom it received the challenge.

If a proof could not be obtained, the handler closes the communication stream instead. This causes a cascade where every previous forwarder in the chain between the challenger and the handler closes their communication streams also. Eventually, the challenger detects that its communication stream with its closest peer was closed and re-uploads the chunk.

6 Evaluation

In this section, we present our evaluation of SUP. We measured and compared the message sizes of SUP and Data Stewardship to estimate SUP's theoretical bandwidth savings. We ran experiments with both protocols on a network of 1000 Swarm peers to measure the bandwidth use and re-upload duration in several different scenarios. Our results show that SUP saves up to 94 % bandwidth and uses up to 82 % less time than Data Stewardship. Its bandwidth use and re-upload duration scale linearly with the number of lost chunks.

6.1 Experimental Setup

To run our experiments, we used a cluster of 30 physical machines. Each machine is installed with Ubuntu 18.04.4 LTS and has 32 GB RAM, an Intel Xeon E-2136 3.30 GHz CPU, a 1.5 TB SSD disk, and 1 Gbit/s NIC. To orchestrate the cluster and manage 1000 Swarm Bee peers, we used Kubernetes [23] and Helm [19]. We distributed the Swarm Bee peers on 28 of the machines, used one to host a private Ethereum network, and the last one to manage the experiment execution. In our setup, we used version 1.7.0 [38] of Swarm Bee with our modifications and SUP implementation.

6.2 Evaluation Framework

To measure the practical bandwidth savings of SUP in comparison with Data Stewardship, we ran the two re-upload protocols on files with different chunk loss

rates. We developed an evaluation framework that facilitates the uploading of files to Swarm and the random removal of a percentage of chunks.

Our evaluation framework improves upon previous work evaluating Swarm [28]. The previous work required all peers to be terminated before the evaluation tool could modify the state of each peer. Our framework, however, can operate directly on online peers. We accomplish this by adding new features to Swarm Bee's debug API and integrating our framework with the API.

We used the framework to write a program for running our experiments. The program is given a set of file addresses to re-upload, a range of chunk-loss percentages, and snapshots listing the chunks stored by each peer. For each file, it applies a modified snapshot with a percentage of the chunk identifiers belonging to the file removed. Then, it performs a re-upload for the file using one of the two re-upload protocols. Lastly, it measures the re-upload duration and gathers metrics from each peer to calculate the bandwidth usage.

The evaluation program uses the Kubernetes port-forwarding API to connect to peers from outside the Kubernetes cluster. When running experiments using our framework, we discovered and reported a memory leak in the Kubernetes port-forwarding API client (`https://github.com/kubernetes/kubernetes/issues/112032`).

6.3 Message Sizes

We measured the message sizes for the request and response messages in SUP and push-sync (used by both SUP and Data Stewardship). SUP's requests (challenges) are 68 bytes, and its responses (proofs) are 205 bytes. Push-sync's requests (chunk deliveries) are 4256 bytes, of which 4104 bytes is the chunk itself, and its responses (receipts) are 135 bytes. In Table 1, we use these message sizes to estimate the bandwidth usage of a single instance of SUP and Data Stewardship when the chunk is available and unavailable. We do not consider the unexpected case where an invalid proof is received, in which the transmission of the invalid proof comes in addition to the challenge, delivery, and receipt.
Based on the estimates in Table 1 we derive a linear expression for bandwidth usage in SUP: $273 + 4186l$ bytes, given chunk loss rate $l \in [0, 1]$. From this expression, we calculate that SUP should use less bandwidth than Data Stewardship until 98.38 % chunk loss.

Table 1: Estimated bandwidth usage for one chunk.

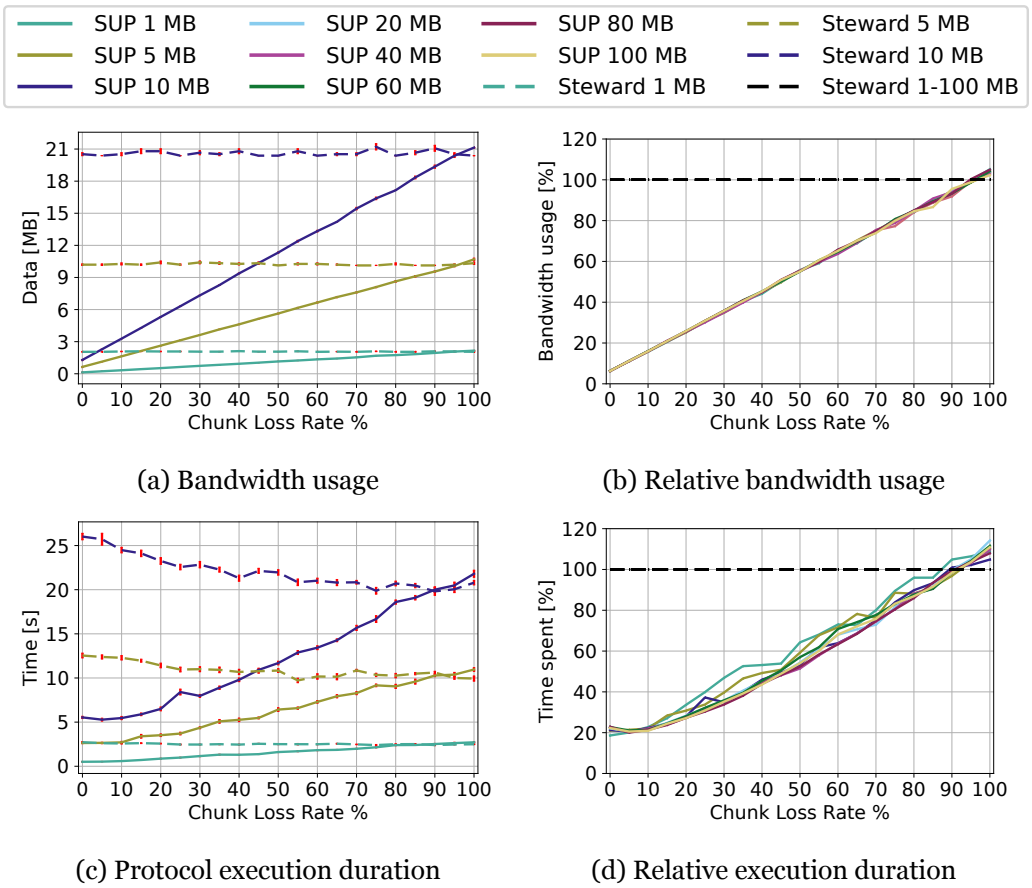
Chunk Availability	SUP	Data Stewardship	SUP Savings
Available	273 B	4391 B	93.78 %
Unavailable	4459 B	4391 B	-1.55 %

6.4 Cost-effectiveness of SUP

We demonstrate that SUP is cost-effective by comparing it against Data Stewardship in Swarm Bee. The evaluations were made on our cluster with 1000 Swarm Bee peers. We varied the file sizes from 1 to 100 MB, and the chunk loss rates from 0 to 100 %. The chunk loss rate is defined as the percentage of chunks that are missing in the network. Each experiment was repeated 22 times, and the results are presented in Figure 5. As expected, the results show that the benefit of SUP deteriorates as the chunk loss rate increases. Our results show that when the chunk loss rate reaches around 90 %, it is more cost-effective to re-upload the chunk, without checking if it is already stored. A previous study of file availability in Swarm [29] shows that even with a high replication degree, the storage system breaks down long before reaching such extreme rates of chunk loss. Our experiment on the data availability in the public Swarm network conducted over four weeks, presented in Section 2.4, shows that the chunk loss rate peaked at 12 % on a single day and was less than 10 % on all other days.

As forecasted in Section 6.3, we observe in Figure 5a that bandwidth usage in SUP scales linearly for files of 1 MB, 5 MB, and 10 MB. As expected, bandwidth usage is not affected by chunk loss in Data Stewardship for the same file sizes. We observe the same effect in Figure 5b, which shows the relative bandwidth used by SUP compared to Data Stewardship for files ranging from 1 to 100 MB. The eight lines representing SUP are more or less completely overlapping and range from 6.3 % bandwidth usage at 0 % chunk loss to 104 % bandwidth usage at 100 % chunk loss. The bandwidth usage of SUP and Data Stewardship is similar when around 95 % of the network's chunks are missing.

Figure 5c shows our results when evaluating the protocol execution duration of SUP and Data Stewardship for files of 1 MB, 5 MB, and 10 MB. By protocol execution duration, we mean the total time elapsed since the client initiated the protocol until all missing chunks have been re-uploaded. We see that SUP has a lower execution duration than Data Stewardship until the chunk loss rate reaches 90 %.



(a) Bandwidth usage

(b) Relative bandwidth usage

(c) Protocol execution duration

(d) Relative execution duration

Figure 5: Cost-effectiveness for different chunk-loss rates. (a,c) The red vertical lines show the standard error.

Interestingly, Data Stewardship is slower when the chunk loss rate is low. We believe this effect is due to storage peers updating their prioritized list of chunks to garbage collect when receiving a chunk they already have. When comparing the relative protocol execution duration, we see in Figure 5d, that SUP only uses 20 % of the time that Data Stewardship uses when the chunk loss rate is low, and that SUP remains superior for all file sizes until between 85 to 90 % chunk loss rate.

7 Related Work

This section will discuss how other proof-of-storage (PoS) algorithms relate to the one presented in SUP. The earliest relevant works were published in 2007 [21, 4,

31]. Since that time, there has been considerable work to construct schemes with additional features and improved properties [30, 15, 11, 44, 45, 3, 22, 5, 52, 18, 51, 48, 46, 47, 6, 8, 9, 13, 50, 33, 41, 32, 10, 1, 2, 12, 16].

Proof-of-storage algorithms provide a way to outsource storage to a remote server while being able to verify that the peer is correctly storing the data. The verifications are done via three actors, a challenger, a prover, and a verifier. PoS algorithms are closely related to proof-of-retrievability, which not merely asserts that the data is stored but also can be retrieved. The variety of PoS algorithms differs in performance, as summarized in [49]. The three actors must share the computational burden of the PoS algorithm. However, the algorithms divide the computational share differently. Some algorithms also require some pre-processing and additional metadata at one or more of the actors and thus has some storage overhead. Lastly, the number of bits required to transmit a challenge or a proof differs between the algorithms.

The PoS algorithm in SUP stands out in a few ways. First, there is no storage overhead on any of the actors. Second, the proofs generated by our algorithm verify the entire chunk, as opposed to a few specific bits or random samples. Third, our algorithm targets decentralized storage systems where peers are untrusted and unreliable. Lastly, as we have targeted the re-uploading use case, there are several features that our algorithm does not require.

The first feature is *public verifiability*, allowing anyone, not just the data owner, to query the remote server with storage challenges. Such schemes require that the data owner generates some *proving metadata* that other peers can use to generate challenges and verify proofs. Other definitions for public verifiability, sometimes called *public auditability*, allows a peer to assert the correctness of a challenge or proof to a third-party. Typically, such a feature is desired in protocols where peers want to prove the misbehavior of other peers. One example is FileCoin [24], where storage peers must periodically prove data possession and integrity. Other peers can verify the storage peer's proofs, and if found invalid, they can punish the storage peer by excluding it from the network or taking its collateral.

The next feature is *updatable*, which allows metadata to be partially modified on the storage peers. An updatable scheme is well suited for use cases where data updates are frequent and computing the metadata needed for proving is expensive. As previously mentioned, SUP does not need any additional metadata.

Moreover, the decentralized storage systems that we target are immutable.

Lastly, some schemes can detect the data's replication degree. Typically, these schemes work by encoding the data differently for each storage peer. For our use case, this is not sufficient, as storage peers are untrusted and may collude in answering storage challenges. In addition, such a design requires additional metadata, which is undesirable for SUP.

8 Conclusion

This paper presents SUP, a protocol for cost-effective data upkeep in decentralized storage networks. The need for data upkeep is well documented in both IPFS [36] and Swarm [20]. We monitored the data availability in the public Swarm network over four weeks. We found that clients can be expected to re-upload their files 6 days after the previous upload to keep the file available. Current protocols for re-upload waste resources, as they require clients to upload the entire file, even though only a few chunks may have been lost. SUP employs a novel proof-of-storage algorithm, which is used to determine what is already stored in the network before unnecessarily uploading existing data. In addition, SUP does not incur additional storage overhead at the peers.

We have demonstrated a working solution in a large P2P network with 1000 peers running a recent version of Ethereum Swarm. SUP saves up to 94 % bandwidth and reduces re-uploading time by up to 82 %. This reduction benefits the client as bandwidth consumption is linked to monetary cost, and it also improves the entire network resource utilization. The source code is made available to support the adoption of SUP in other decentralized storage networks.

Acknowledgments

We would like to thank members of the Swarm team for helpful discussions and technical assistance. This work is partially funded by the BBChain and Credence projects under grants 274451 and 288126 from the Research Council of Norway.

References

[1] Frederik Armknecht et al. “Outsourced Proofs of Retrievability”. In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. New York, NY, USA: Association for Computing Machinery, 2014, pp. 831–843.

[2] Frederik Armknecht et al. “Sharing Proofs of Retrievability across Tenants”. In: *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*. New York, NY, USA: Association for Computing Machinery, 2017, pp. 275–287.

[3] Giuseppe Ateniese, Seny Kamara, and Jonathan Katz. “Proofs of Storage from Homomorphic Identification Protocols”. In: *Advances in Cryptology – ASIACRYPT 2009*. Berlin, Heidelberg: Springer, 2009, pp. 319–333.

[4] Giuseppe Ateniese et al. “Provable Data Possession at Untrusted Stores”. In: *Proceedings of the 14th ACM conference on Computer and communications security*. New York, NY, USA: Association for Computing Machinery, 2007, pp. 598–609.

[5] Giuseppe Ateniese et al. “Remote Data Checking Using Provable Data Possession”. In: *ACM Transactions on Information and System Security (TISSEC)* 14.1 (2011), pp. 1–34.

[6] Giuseppe Ateniese et al. “Scalable and Efficient Provable Data Possession”. In: *Proceedings of the 4th International Conference on Security and Privacy in Communication Netowrks*. New York, NY, USA: Association for Computing Machinery, 2008, pp. 1–10.

[7] Juan Benet. “IPFS - Content Addressed, Versioned, P2P File System”. In: *arXiv* abs/1407.3561 (2014).

[8] Kevin D Bowers, Ari Juels, and Alina Oprea. “HAIL: A High-Availability and Integrity Layer for Cloud Storage”. In: *Proceedings of the 16th ACM Conference on Computer and Communications Security*. New York, NY, USA: Association for Computing Machinery, 2009, pp. 187–198.

[9] Kevin D Bowers, Ari Juels, and Alina Oprea. “Proofs of Retrievability: Theory and Implementation”. In: *Proceedings of the 2009 ACM Workshop on Cloud Computing Security*. New York, NY, USA: Association for Computing Machinery, 2009, pp. 43–54.

[10] David Cash, Alptekin Küpçü, and Daniel Wichs. “Dynamic Proofs of Retrievability Via Oblivious RAM”. In: *Journal of Cryptology* 30.1 (2017), pp. 22–57.

[11] Reza Curtmola et al. “MR-PDP: Multiple-Replica Provable Data Possession”. In: *2008 The 28th International Conference on Distributed Computing Systems*. New York, NY, USA: IEEE, 2008, pp. 411–420.

[12] Ivan Damgård, Chaya Ganesh, and Claudio Orlandi. “Proofs of Replicated Storage Without Timing Assumptions”. In: *Advances in Cryptology – CRYPTO 2019*. Cham: Springer, 2019, pp. 355–380.

[13] Yevgeniy Dodis, Salil Vadhan, and Daniel Wichs. “Proofs of Retrievability via Hardness Amplification”. In: *Theory of Cryptography*. Berlin, Heidelberg: Springer, 2009, pp. 109–127.

[14] Thai Duong and Juliano Rizzo. *Flickr’s API Signature Forgery Vulnerability*. Accessed: 2022-12-29. 2009. URL: `https://dl.packetstormsecurity.net/0909-advisories/flickr_api_signature_forgery.pdf`.

[15] C Chris Erway et al. “Dynamic Provable Data Possession”. In: *ACM Transactions on Information and System Security (TISSEC)* 17.4 (2015), pp. 1–29.

[16] Chaowen Guan et al. “Symmetric-Key Based Proofs of Retrievability Supporting Public Verification”. In: *Computer Security – ESORICS 2015*. Cham: Springer, 2015, pp. 203–223.

[17] Janoš Guljaš. *Network Statistics - Swarm Scan*. Accessed: 2022-12-28. 2022. URL: `https://swarmscan.resenje.org`.

[18] Zhuo Hao, Sheng Zhong, and Nenghai Yu. “A Privacy-Preserving Remote Data Integrity Checking Protocol with Data Dynamics and Public Verifiability”. In: *IEEE Transactions on Knowledge and Data Engineering* 23.9 (2011), pp. 1432–1437.

[19] Helm. *The package manager for Kubernetes*. Accessed: 2022-12-29. 2022. URL: https://helm.sh/.

[20] Rinke Hendriksen. *Data stewardship #1508*. Accessed: 2022-12-29. 2021. URL: https://github.com/ethersphere/bee/issues/1508.

[21] Ari Juels and Burton S Kaliski Jr. "PORs: Proofs of Retrievability for Large Files". In: *Proceedings of the 14th ACM Conference on Computer and Communications Security*. New York, NY, USA: Association for Computing Machinery, 2007, pp. 584–597.

[22] Aniket Kate, Gregory M Zaverucha, and Ian Goldberg. "Constant-Size Commitments to Polynomials and Their Applications". In: *Advances in Cryptology - ASIACRYPT 2010*. Berlin, Heidelberg: Springer, 2010, pp. 177–194.

[23] Kubernetes. *Production-Grade Container Orchestration*. Accessed: 2022-12-29. 2022. URL: https://kubernetes.io/.

[24] Protocol Labs. *Filecoin: A Decentralized Storage Network*. Accessed: 2022-12-29. 2017. URL: https://filecoin.io/filecoin.pdf.

[25] libp2p. *libp2p - A modular network stack*. Accessed: 2022-12-29. 2022. URL: https://libp2p.io.

[26] Petar Maymounkov and David Mazières. "Kademlia: A Peer-to-Peer Information System Based on the XOR Metric". In: *International Workshop on Peer-to-Peer Systems*. Berlin, Heidelberg: Springer, 2002, pp. 53–65.

[27] Arvind Narayanan et al. *Bitcoin and Cryptocurrency Technologies: A Comprehensive Introduction*. Princeton, NJ, USA: Princeton University Press, 2016.

[28] Racin Nygaard. "Lessons Learned from a Bare-metal Evaluation of Erasure Coding Algorithms in P2P Networks". In: *arXiv* abs/2208.12360 (2022).

[29] Racin Nygaard, Vero Estrada-Galiñanes, and Hein Meling. "Snarl: Entangled Merkle Trees for Improved File Availability and Storage Utilization". In: *Proceedings of the 22nd International Middleware Conference*. Middleware '21. Québec city, Canada: Association for Computing Machinery, 2021, pp. 236–247.

[30] Hovav Shacham and Brent Waters. “Compact Proofs of Retrievability”. In: *Advances in Cryptology - ASIACRYPT 2008*. Berlin, Heidelberg: Springer, 2008, pp. 90–107.

[31] Mehul A Shah et al. “Auditing to Keep Online Storage Services Honest”. In: *Proceedings of the 11th USENIX Workshop on Hot Topics in Operating Systems*. USA: USENIX Association, 2007.

[32] Jian Shen et al. “An Efficient Public Auditing Protocol With Novel Dynamic Structure for Cloud Data”. In: *IEEE Transactions on Information Forensics and Security* 12.10 (2017), pp. 2402–2415.

[33] Elaine Shi, Emil Stefanov, and Charalampos Papamanthou. “Practical Dynamic Proofs of Retrievability”. In: *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*. New York, NY, USA: Association for Computing Machinery, 2013, pp. 325–336.

[34] William Stallings. *Cryptography And Network Security: Principles and Practice, 7th Edition*. Upper Saddle River, NJ, USA: Pearson Education, 2017.

[35] IPFS team. *”ipfs add” calculate CID and check for it *before* uploading to API server*. Accessed: 2022-12-29. 2020. URL: `https://github.com/ipfs/kubo/issues/7586`.

[36] IPFS team. *Garbage collection*. Accessed: 2022-12-29. 2022. URL: `https://docs.ipfs.tech/concepts/persistence/#garbage-collection`.

[37] IPFS team. *Persistence, permanence, and pinning*. Accessed: 2022-12-29. 2022. URL: `https://docs.ipfs.io/concepts/persistence/#persistence-permanence-and-pinning`.

[38] Swarm team. *Bee is a Swarm client implemented in Go*. Accessed: 2022-12-29. 2022. URL: `https://github.com/ethersphere/bee`.

[39] Swarm team. *Storage and Communication Infrastructure for a Self-Sovereign Digital Society*. Accessed: 2022-12-29. 2021. URL: `https://www.ethswarm.org/swarm-whitepaper.pdf`.

[40] Swarm team. *Swarm Gateway*. Accessed: 2022-12-29. 2022. URL: `https://gateway.ethswarm.org/`.

[41] Hui Tian et al. “Dynamic-Hash-Table Based Public Auditing for Secure Cloud Storage”. In: *IEEE Transactions on Services Computing* 10.5 (2017), pp. 701–714.

[42] Dennis Trautwein et al. “Design and Evaluation of IPFS: A Storage Layer for the Decentralized Web”. In: *Proceedings of the ACM SIGCOMM 2022 Conference*. New York, NY, USA: Association for Computing Machinery, 2022, pp. 739–752.

[43] Viktor Trón. *The Book of Swarm - v1.0 pre-release 7 November 17, 2020*. Accessed: 2022-12-29. 2020. URL: `https://www.ethswarm.org/The-Book-of-Swarm.pdf`.

[44] Boyang Wang, Baochun Li, and Hui Li. “Oruta: Privacy-Preserving Public Auditing for Shared Data in the Cloud”. In: *IEEE Transactions on Cloud Computing* 2.1 (2014), pp. 43–56.

[45] Cong Wang et al. “Privacy-Preserving Public Auditing for Secure Cloud Storage”. In: *IEEE Transactions on Computers* 62.2 (2013), pp. 362–375.

[46] Cong Wang et al. “Toward Publicly Auditable Secure Cloud Data Storage Services”. In: *IEEE Network* 24.4 (2010), pp. 19–24.

[47] Cong Wang et al. “Toward Secure and Dependable Storage Services in Cloud Computing”. In: *IEEE Transactions on Services Computing* 5.2 (2012), pp. 220–232.

[48] Hao Yan et al. “A Novel Efficient Remote Data Possession Checking Protocol in Cloud Storage”. In: *IEEE Transactions on Information Forensics and Security* 12.1 (2017), pp. 78–88.

[49] Anjia Yang et al. “Lightweight and Privacy-Preserving Delegatable Proofs of Storage with Data Dynamics in Cloud Storage”. In: *IEEE Transactions on Cloud Computing* 9.1 (2021), pp. 212–225.

[50] Kan Yang and Xiaohua Jia. “Data storage auditing service in cloud computing: challenges, methods and opportunities”. In: *World Wide Web* 15.4 (2012), pp. 409–428.

[51] Jiawei Yuan and Shucheng Yu. “Proofs of Retrievability with Public Verifiability and Constant Communication Cost in Cloud”. In: *Proceedings of the 2013 International Workshop on Security in Cloud Computing*. New York, NY, USA: Association for Computing Machinery, 2013, pp. 19–26.

[52] Yan Zhu et al. “Dynamic Audit Services for Integrity Verification of Outsourced Storages in Clouds”. In: *Proceedings of the 2011 ACM Symposium on Applied Computing*. New York, NY, USA: Association for Computing Machinery, 2011, pp. 1550–1557.

Paper 4

Lessons Learned from a Bare-metal Evaluation of Erasure Coding Algorithms in P2P Networks

Racin Nygaard

Prestented during Sigmetrics 2021 [50]

Abstract

We have built a bare-metal testbed in order to perform large-scale, reproducible evaluations of erasure coding algorithms. Our testbed supports at least 1000 Ethereum Swarm peers running on 30 machines. Running experimental evaluation is time-consuming and challenging. Researchers must consider the experimental software's limitations and artifacts. If not controlled, the network behavior may cause inaccurate measurements. This paper shares the lessons learned from a bare-metal evaluation of erasure coding algorithms and how to create a controlled-environment in a cluster consisting of 1000 Ethereum Swarm peers.

1 Background and Motivation

We are designing algorithms to protect data in decentralized networks, such as the Ethereum Swarm peer-to-peer network. The Merkle tree [6] is a widely-used hash tree to authenticate data. In content-addressed storage, a hash tree can be used to locate the chunks in the network. That permits to locate a large number of chunks by knowing only the tree's root. Due to the tree structure, it is critical to protect chunks in the path between the root and the data chunks (leaves).

Large-scale experimental evaluation of erasure codes is rarely observed in the literature. We want to implement algorithms and understand their impact on distributed storage networks. Typically, the metrics of interest are latency, throughput, network- and storage overhead. Many conditions can affect measurements, including individual peers' behavior and the need to run hundreds or thousands of concurrent instances. To obtain meaningful results, we run our experimental evaluations in a controlled environment and, at the same time, close to a real-world scenario. Our bare-metal cluster consists of 30 machines, running up to 1000 Swarm peers [8].

This paper shares insight from our evaluation environment and lessons learned by observing the peers' behavior. Our evaluation techniques can help others improve the evaluation of erasure codes in large networks. We discuss our experimental setup, the main challenges, and the tools developed to solve them. The source code for the tools will be made available [7].

2 Ethereum Swarm

Swarm is a decentralized storage and communication system. Swarm's test network is relatively large; a 3-month study found 6,500 unique peers [3]. The basic unit of storage is a *chunk* limited to 4K bytes.

Swarm splits a file into chunks for upload and computes a cryptographic hash of each chunk. This hash is also known as the *content address* and is necessary to access the chunk later.

Chunks that belong to the same file are organized in a Merkle tree. During retrieval, the client initially queries the network with the root's content address. It then deciphers the root chunk to retrieve its children's content address, for which it again queries the network. This process continues until all leaves are

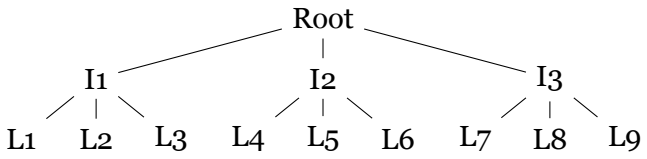


Figure 1: Merkle tree illustrated with branching factor 3, resulting in 9 leaves, 3 internal nodes and 1 root.

retrieved and the original file has been rebuilt.

3 Adding Redundancy

The current Swarm deployment can support traditional erasure coding techniques on a per-chunk basis. For example, by using Reed-Solomon coding [9], a file consisting of k chunks can be encoded using $n > k$ chunks so that any k-out-of-n chunks can be used for retrieval.

Unfortunately, erasure coding the original file would not be sufficient, as this would only apply to the leaves of the Merkle tree, while the internal nodes would be vulnerable to data loss.

Our research revolves around finding better ways to add redundancy to the Merkle tree. To that end, we have used several coding schemes and developed novel algorithms for use with Merkle trees. We have evaluated our algorithms experimentally. Next, we describe our setup and how we were able to evaluate our algorithms in a cluster of 1000 peers.

4 Experimental Setup

Our cluster consists of 30 machines running Ubuntu. We use Helm [2] and Kubernetes [4] to distribute the load and quickly scale up to 1000 Swarm peers. We avoid overloading the cluster machines to facilitate restarting the experiment if our algorithms cause a crash or get stuck while debugging.

Helm uses a single configuration file as input to a dozen configuration files during the cluster's initialization. In the single configuration file, we specify parameters such as storage capacity, cluster placement, scaling, run-time parameters of Swarm, and much more. We base our Helm scripts on those provided by

the Ethereum Swarm organization [1].

Peers running in Kubernetes are ephemeral by default, with limited options for persistent storage. The before-mentioned Helm scripts only supported persistent storage options for cloud providers, but not for private clusters. To provide persistent storage to the peers from the local SSD, we had to create a *Persistent Volume* (PV). Each peer has a dedicated PV, and each PV is linked to a physical location on the SSD. We elegantly used the modulus operator in the Helm configuration file to create the link so that peer y's data was allocated to a PV assigned to machine $y \mod 29$.

5 Dealing with Replication

Each peer in Swarm is assigned a unique identifier in a Kademlia [5] overlay network, and peers are grouped into neighborhoods based on the similarity of their identifiers. Inside neighborhoods, peers attempt to replicate their data to each other. Peers with the most similar identifier to a chunk is deemed responsible for persisting that chunk.

As peers have their own view of the overlay network, some "superpeers" may exist in multiple neighborhoods. This results in some chunks being more replicated than others, and in our particular setup, this ranged from 9 replicas to 154 replicas. Figure 2 shows how the chunks of a 100 MB file were replicated, and Figure 3 shows how the chunks were distributed to the peers. To ensure fair comparisons between algorithms, we need to replicate chunks exactly once.

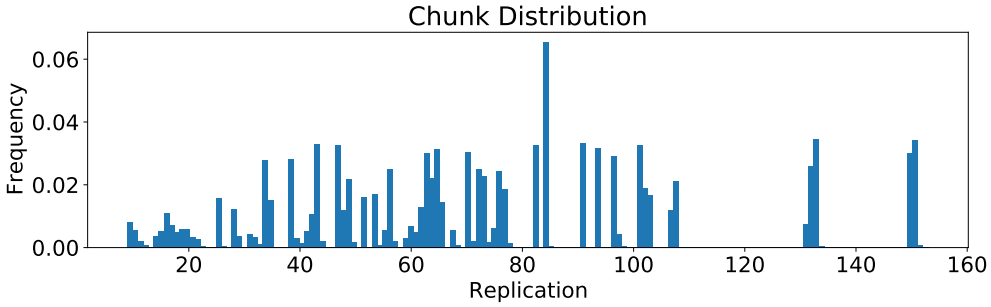


Figure 2: Chunk replication with 1000 Swarm peers.

Swarm offers no tools to control each peer's storage, so the only way to achieve this was to develop our own tools. Our first tool is named *listchunks* (1), and its

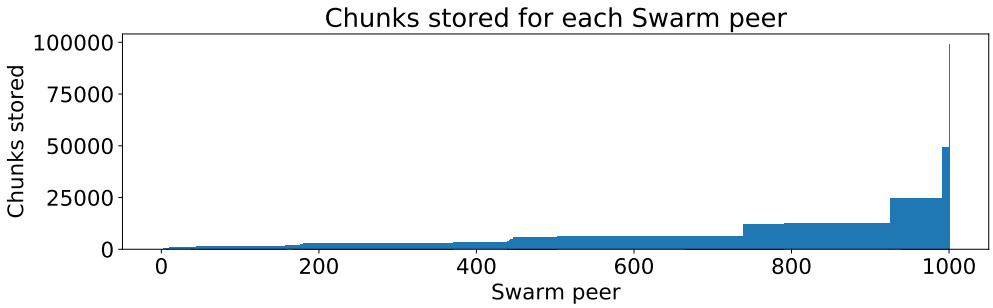


Figure 3: Chunks stored by each Swarm peer.

main objective is to list the content address of all the chunks belonging to each file in our storage network. The listchunks tool achieves this by merely appending each new chunk's content address it traverses from the root chunk when retrieving the file.

The second tool is *bakedeletion* (2), which is responsible for figuring out which chunks must be deleted from which peer. It generates the deletion list by iteratively simulating the removal of chunks from peers while following four basic rules; (A) All peers must have some chunks, (B) Cardinality of unique chunks must be equal after deletion, (C) Peers can not get new chunks, (D) Replication factor for each chunk must be uniform.

To run experiments on different file sizes, we need to run tools (1) and (2) for each file we want to evaluate, as rule (A) must be respected for each file. Thus, we need to aggregate all deletion lists, and this is done in the third tool *combinestorage* (3).

Finally, the aggregated deletion list is passed to *deletechunks* (4), which goes through the storage network, peer by peer, and deletes the mapped chunks.

6 Dealing with Syncing

To ensure an identical system state across test iterations, we had to control the chunk distribution, or *syncing*. The *push-syncing* and *pull-syncing* can both be disabled by the command-line option *no-sync* when starting the peer.

However, the syncing process that occurs when chunks are delivered through the Kademlia DHT can not be disabled. Therefore we had to create our own procedures to create snapshots of the storage states and recover from a snapshot

between each test iteration.

The snapshot of a peer is created by copying all the chunks stored by the peer immediately after we have made replication uniform for all chunks, as discussed in Section 5. Before initiating the copying process, we must first terminate the peers to avoid data corruption.

To ensure an identical system state between test iterations, we (A) replace the local storage with the snapshots for each of the peers, (B) ensure that the *no-sync* option is turned on and (C) ensure that each peer is well-connected. When the test iteration concludes, we (D) shut down the peers, and for the next iteration, we continue from step (A) again.

We achieve step (A) by merely copying the snapshot to the peers using *rsync*. To reach sufficient connectivity required for step (C), the peers must first discover each other. We monitor this process by periodically polling the inter-process communication file *bzzd.ipc*. As soon as the desired connectivity is reached, we can continue with the experimental evaluation, e.g., file availability is given peer failures.

7 Conclusion

In this paper, we have shared our experiences with running a 1000 peer cluster of Ethereum Swarm instances. Without these tools to manage the peers, the configuration would be a nightmare scenario and ensuring they are all running correctly, even worse. With the assistance of these tools, we can make fair comparisons of redundancy algorithms in a P2P storage system.

References

[1] Helm. *Helm charts to deploy Swarm and Geth*. https://github.com/ethersphere/helm-charts.

[2] Helm. *The package manager for Kubernetes*. https://helm.sh/.

[3] Seoung Kyun Kim et al. “Measuring ethereum network peers”. In: *Proceedings of the Internet Measurement Conference 2018*. 2018, pp. 91–104.

[4] Kubernetes. *Production-Grade Container Orchestration*. https://kubernetes.io/.

[5] Petar Maymounkov and David Mazieres. “Kademlia: A peer-to-peer information system based on the xor metric”. In: *International Workshop on Peer-to-Peer Systems*. Springer. 2002, pp. 53–65.

[6] Ralph C Merkle. “A digital signature based on a conventional encryption function”. In: *Conference on the theory and application of cryptographic techniques*. Springer. 1987, pp. 369–378.

[7] Racin Nygaard. *Snarl Tools*. https://github.com/racin/snarl_evaluation_tools.

[8] Swarm. *Ethereum Swarm*. https://swarm.ethereum.org/.

[9] Stephen B Wicker and Vijay K Bhargava. *Reed-Solomon codes and their applications*. John Wiley & Sons, 1999.

University of Stavanger

4036 Stavanger
Tel: +47 51 83 10 00
E-mail: post@uis.no
www.uis.no

Cover Photo: Racin Nygaard

ISBN: 978-82-8439-158-8
ISSN: 1890-1387