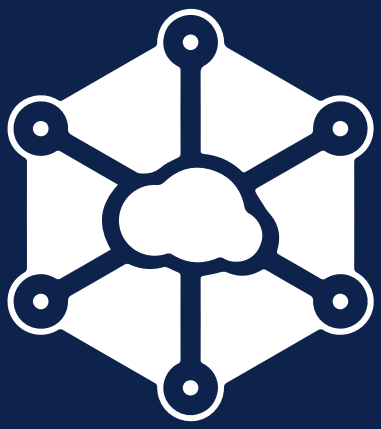


STORJ

Storj: A Decentralized Cloud Storage Network Framework

Storj Labs, Inc.

October 30, 2018
v3.0
March 7, 2024
v3.1
https://github.com/storj/whitepaper

Contents

0.1 Abstract

Decentralized cloud storage represents a fundamental shift in the efficiency and economics of large-scale storage. Eliminating central control allows users to store and share data without reliance on a third-party storage provider. Decentralization mitigates the risk of data failures and outages while simultaneously increasing the security and privacy of object storage. It also allows market forces to optimize for less expensive storage at a greater rate than any single provider could afford. Although there are many ways to build such a system, there are some specific responsibilities any given implementation should address. Based on our experience with petabyte-scale storage systems, we introduce a modular framework for considering these responsibilities and for building our distributed storage network. Additionally, we describe an initial concrete implementation for the entire framework.

0.2 Contributors

This paper represents the combined efforts of many individuals. Contributors affiliated with Storj Labs, Inc. include but are not limited to: Tim Adams, Kishore Aligeti, Cameron Ayer, Atikh Bana, Alexander Bender, Stefan Benten, Maximillian von Briesen, Paul Cannon, Gina Cooley, Dennis Coyle, Egon Elbre, Nadine Farah, Patrick Gerbes, John Gleeson, Ben Golub, James Hagans, Jens Heimbürge, Faris Huskovic, Philip Hutchins, Brandon Iglesias, Viktor Ihnatiuk, Jennifer Johnson, Kevin Leffew, Alexander Leitner, Richard Littauer, Dylan Lott, JT Olio, Kaloyan Raev, Garrett Ransom, Matthew Robinson, Jon Sanderson, Benjamin Sirb, Dan Sorensen, Helene Unland, Natalie Villasana, Bryan White, and Shawn Wilkinson.

We'd also like to thank the other authors and contributors of the previous Storj and Metadisk white papers: Tome Boshevski, Josh Brandoff, Vitalik Buterin, Braydon Fuller, Gordy Hall, Jim Lowry, Chris Pollard, and James Prestwich.

We'd like to especially thank Petar Maymounkov, Anand Babu Periasamy, Tim Kosse, Roberto Galoppini, Steven Willoughby, and Aaron Boodman for their helpful review of and contributions to an early draft of this paper.

We would like to acknowledge the efforts, white papers, and communications of others in the distributed computing, blockchain, distributed storage, and decentralized storage space, whose work has informed our efforts. A more comprehensive list of sources is in the bibliography, but we would like to provide particular acknowledgement for the guidance and inspiration provided by the teams that designed and built Allmydata, Ceph, CoralCDN, Ethereum, Farsite, Filecoin, Freenet, Gluster, GFS, Hadoop, IPFS, Kademlia, Lustre, Maidsafe, Minio, MojoNation, OceanStore, Scality, Siacoin, and Tahoe-LAFS.

Finally, we extend a huge thank you to everyone we talked to during the design and architecture of this system for their valuable thoughts, feedback, input, and suggestions.

Please address correspondence to paper@storj.io.

0.3 Changelog

This section describes updates from the past editions of this white paper. Beyond a few trivial wording tweaks, we changed the following aspects in version 3.1:

- Clarified *encryption blocks* in section 4.1.2.
- Replaced Kademlia for storage node discovery with a direct node-to-satellite indication of network participation (section 4.6).
- Simplified the Audits service and containment mode (section 4.13).
- Added the ability for storage node operators to select the Satellites they would like to work with, eliminating the need for Satellite vetting and opt-out (section 4.18).
- Removed section 4.21 *Quality Control and Branding* about obsolete branding ideas.
- Updated appendix B about anticipated attacks to reflect the removal of Kademlia.

With these changes in mind, we expect that this paper once again matches our in-production service at the time of publication.

For more a detailed changelog, please see https://github.com/storj/whitepaper/compare/v3.0-merged...v3.1.

1. Introduction

The Internet is a massive decentralized and distributed network consisting of billions of devices which are not controlled by a single group or entity. Much of the data currently available through the Internet is quite centralized and is stored with a handful of technology companies that have the experience and capital to build massive data centers capable of handling this vast amount of information. A few of the challenges faced by data centers are: data breaches, periods of unavailability on a grand scale, storage costs, and expanding and upgrading quickly enough to meet user demand for faster data and larger formats.

Decentralized storage has emerged as an answer to the challenge of providing a performant, secure, private, and economical cloud storage solution. Decentralized storage is better positioned to achieve these outcomes as the architecture has a more natural alignment to the decentralized architecture of the Internet as a whole, as opposed to massive centralized data centers.

News coverage of data breaches over the past few years has shown us that the frequency of such breaches has been increasing by as much as a factor of 10 between 2005 and 2017 [1]. Decentralized storage's process of protecting data makes data breaches more difficult than current methods used by data centers while, at the same time, costing less than current storage methods.

This model can address the rapidly expanding amount of data for which current solutions struggle. With an anticipated 44 zettabytes of data expected to exist by 2020 and a market that will grow to $92 billion USD in the same time frame [2], we have identified several key market segments that decentralized cloud storage has the potential to address. As decentralized cloud storage capabilities evolve, it will be able to address a much wider range of use cases from basic object storage to content delivery networks (CDN).

Decentralized cloud storage is rapidly advancing in maturity, but its evolution is subject to a specific set of design constraints which define the overall requirements and implementation of the network. When designing a distributed storage system, there are many parameters to be optimized such as speed, capacity, trustlessness, Byzantine fault tolerance, cost, bandwidth, and latency.

We propose a framework that scales horizontally to exabytes of data storage across the globe. Our system, the Storj Network, is a robust object store that encrypts, shards, and distributes data to nodes around the world for storage. Data is stored and served in a manner purposefully designed to prevent breaches. In order to accomplish this task, we've designed our system to be modular, consisting of independent components with task-specific jobs. We've integrated these components to implement a decentralized object storage system that is not only secure, performant, and reliable but also significantly more economical than either on-premise or traditional, centralized cloud storage.

We have organized the rest of this paper into six additional chapters. Chapter 2 discusses the design space in which Storj operates and the specific constraints on which our optimization efforts are based. Chapter 3 covers our framework. Chapter 4 describes the concrete implementation of the framework, while chapter 5 explains what happens during each operation in the network. Chapter 6 covers future work. Finally, chapter 7 covers selected calculations.

2. Storj design constraints

Before designing a system, it's important to first define its requirements. There are many different ways to design a decentralized storage system. However, with the addition of a few requirements, the potential design space shrinks significantly. Our design constraints are heavily influenced by our product and market fit goals. By carefully considering each requirement, we ensure the framework we choose is as universal as possible, given the constraints.

2.1 Security and privacy

Any object storage platform must ensure both the privacy and security of data stored regardless of whether it is centralized or decentralized. Decentralized storage platforms must mitigate an additional layer of complexity and risk associated with the storage of data on inherently untrusted nodes. Because decentralized storage platforms cannot take many of the same shortcuts data center based approaches can (e.g. firewalls, DMZs, etc.), decentralized storage must be designed from the ground up to support not only end-to-end encryption but also enhanced security and privacy at all levels of the system.

Certain categories of data are also subject to specific regulatory compliance. For example, the United States legislation for the Health Insurance Portability and Accountability Act (HIPAA) has specific requirements for data center compatibility. European countries have to consider the General Data Protection Regulation (GDPR) regarding how individual information must be protected and secured. Many customers outside of the United States may feel they have significant geopolitical reasons to consider storing data in a way that limits the ability for US-based entities to impact their privacy [3]. There are many other regulations in other sectors regarding user's data privacy.

Customers should be able to evaluate that our software is implemented correctly, is resistant to attack vectors (known or unknown), is secure, and otherwise fulfills all of the customers' requirements. The code for the Storj network is open source software and provides the level of transparency and assurance needed to prove that the behaviors of the system are as advertised.

2.2 Decentralization

Informally, a decentralized application is a service that has no single operator. Furthermore, no single entity should be solely responsible for the cost associated with running the service or be able to cause a service interruption for other users.

One of the main motivations for preferring decentralization is to drive down infrastructure costs for maintenance, utilities, and bandwidth. We believe that there are sig-

nificant underutilized resources at the edge of the network for many smaller operators. In our experience building decentralized storage networks, we have found a long tail of resources that are presently unused or underused that could provide affordable and geographically distributed cloud storage. Conceivably, some small operator might have access to less-expensive electricity than standard data centers or another small operator could have access to less-expensive cooling. Many of these small operator environments are not substantial enough to run an entire datacenter-like storage system. For example, perhaps a small business or home Network Attached Storage (NAS) operator has enough excess electricity to run ten hard drives but not more. We have found that in aggregate, enough small operator environments exist such that their combination over the internet constitutes significant opportunity and advantage for less-expensive and faster storage.

Our decentralization goals for fundamental infrastructure, such as storage, are also driven by our desire to provide a viable alternative to the few major centralized storage entities who dominate the market at present. We believe that there exists inherent risk in trusting a single entity, company, or organization with a significant percentage of the world's data. In fact, we believe that there is an implicit cost associated with the risk of trusting any third party with custodianship of personal data. Some possible costly outcomes include changes to the company's roadmap that could result in the product becoming less useful, changes to the company's position on data collection that could cause it to sell customer metadata to advertisers, or even the company could go out of business or otherwise fail to keep customer data safe. By creating an equivalent or better decentralized system, many users concerned about single-entity risk will have a viable alternative. With decentralized architecture, Storj could cease operating and the data would continue to be available.

We have decided to adopt a decentralized architecture because, despite the trade-offs, we believe decentralization better addresses the needs of cloud storage and resolves many core limitations, risks, and cost factors that result from centralization. Within this context, decentralization results in a globally distributed network that can serve a wide range of storage use cases from archival to CDN. However, centralized storage systems require different architectures, implementations, and infrastructure to address each of those same use cases.

2.3 Marketplace and economics

Public cloud computing, and public cloud storage in particular, has proven to be an attractive business model for the large centralized cloud providers. Cloud computing is estimated to be a $186.4 billion dollar market in 2018, and is expected to reach $302.5 billion by 2021 [4].

The public cloud storage model has provided a compelling economic model to end users. Not only does it enable end users to scale on demand but also allows them to avoid the significant fixed costs of facilities, power, and data center personnel. Public

cloud storage has generally proven to be an economical, durable, and performant option for many end users when compared to on-premise solutions.

However, the public cloud storage model has, by its nature, led to a high degree of concentration. Fixed costs are born by the network operators, who invest billions of dollars in building out a network of data centers and then enjoy significant economies of scale. The combination of large upfront costs and economies of scale means that there is an extremely limited number of viable suppliers of public cloud storage (arguably, fewer than five major operators worldwide). These few suppliers are also the primary beneficiaries of the economic return.

We believe that decentralized storage can provide a viable alternative to centralized cloud. However, to encourage partners or customers to bring data to the network, the price charged for storage and bandwidth—combined with the other benefits of decentralized storage—must be more compelling and economically beneficial than competing storage solutions. In our design of Storj, we seek to create an economically advantageous situation for four different groups:

End users - We must provide the same economically compelling characteristics of public cloud storage with no upfront costs and scale on demand. In addition, end users must experience meaningfully better value for given levels of capacity, durability, security, and performance.

Storage node operators - It must be economically attractive for storage node operators to help build out the network. They must be paid fairly, transparently, and be able to make a reasonable profit relative to any marginal costs they incur. It should be economically advantageous to be a storage node operator not only by utilizing underused capacity but also by creating new capacity, so that we can grow the network beyond the capacity that currently exists. Since node availability and reliability has a large impact on network availability, cost, and durability, it is required that storage node operators have sufficient incentive to maintain reliable and continuous connections to the network.

Demand providers - It must be economically attractive for developers and businesses to drive customers and data onto the Storj network. We must design the system to fairly and transparently deliver margin to partners. We believe that there is a unique opportunity to provide open-source software (OSS) companies and projects, which drive over two-thirds of the public cloud workloads today without receiving direct revenue, a source of sustainable revenue.

Network operator - To sustain continued investment in code, functionality, network maintenance, and demand generation, the network operator, currently Storj Labs, Inc., must be able to retain a reasonable profit. The operator must maintain this profit while not only charging end users less than the public cloud providers but also margin sharing with storage node operators and demand providers.

Additionally, the network must be able to account for ensuring efficient, timely billing and payment processes as well as regulatory compliance for tax and other reporting. To be as globally versatile as possible with payments, our network must be robust to accom-

modate several types of transactions (such as cryptocurrency, bank payments, and other forms of barter).

Lastly, the Storj roadmap must be aligned with the economic drivers of the network. New features and changes to the concrete implementations of framework components must be driven by applicability to specific object storage use cases and the relationship between features and performance to the price of storage and bandwidth relative to those use cases.

2.4 Amazon S3 compatibility

At the time of this paper's publication, the most widely deployed public cloud is Amazon Web Services [5]. Amazon Web Services not only is the largest cloud services ecosystem but also has the benefit of first mover advantage. Amazon's first cloud services product was Amazon Simple Storage Service, or Amazon S3 for short. Public numbers are hard to come by but Amazon S3 is likely the most widely deployed cloud storage protocol in existence. Most cloud storage products provide some form of compatibility with the Amazon S3 application program interface (API) architecture.

Our objective is to aggressively compete in the wider cloud storage industry and bring decentralized cloud storage into the mainstream. Until a decentralized cloud storage protocol becomes widely adopted, Amazon S3 compatibility creates a graceful transition path from centralized providers by alleviating many switching costs for our users. To achieve this, the Storj implementation allows applications previously built against Amazon S3 to work with Storj with minimal friction or changes. S3 compatibility adds aggressive requirements for feature set, performance, and durability. At a bare minimum, this requires the methods described in Figure 2.1 to be implemented.

```
1  // Bucket operations
2  CreateBucket(bucketName)
3  DeleteBucket(bucketName)
4  ListBuckets()
5
6  // Object operations
7  GetObject(bucketName, objectPath, offset, length)
8  PutObject(bucketName, objectPath, data, metadata)
9  DeleteObject(bucketName, objectPath)
10 ListObjects(bucketName, prefix, startKey, limit, delimiter)
```

Figure 2.1: Minimum S3 API

The Storj service supports the majority of the S3 protocol via two different integration patterns that emerged as specific customer requirements:

Single tenant gateway - a self-hosted, end-to-end encrypted S3 compatible gateway where the customer's equipment is responsible for encryption, erasure coding and

direct peer-to-peer transmission of data to storage nodes, including data expansion from erasure coded redundancy and long tail mitigation.

Multi-tenant gateway - a hosted S3 compatible gateway where a trusted provider is responsible for encryption, erasure coding, and transmission of data to storage nodes. This gateway requires a key from the customer as part of every request, since the customer's encryption keys are not preserved in the hosted environment.

2.5 Durability, device failure, and churn

A storage platform is useless unless it also functions as a retrieval platform. For any storage platform to be valuable, it must be careful not to lose the data it was given, even in the presence of a variety of possible failures within the system. Our system must store data with high durability and have negligible risk of data loss.

For all devices, component failure is a guarantee. All hard drives fail after enough wear [6] and servers providing network access to these hard drives will also eventually fail. Network links may die, power failures could cause havoc sporadically, and storage media become unreliable over time. Data must be stored with enough redundancy to recover from individual component failures. Perhaps more importantly, no data can be left in a single location indefinitely. In such an environment, redundancy, data maintenance, repair, and replacement of lost redundancy must be considered inevitable, and the system must account for these issues.

Furthermore, decentralized systems are susceptible to high churn rates where participants join the network and then leave for various reasons, well before their hardware has actually failed. For instance, Rhea *et al.* found that in many real world peer-to-peer systems, the median time a participant lasts in the network ranges from hours to mere minutes [7]. Maymounkov *et al.* found that the probability of a node staying connected to a decentralized network for an additional hour is an *increasing* function of uptime (Figure 2.2 [8]). In other words, nodes that have been online for a long time are less likely to contribute to overall node churn.

Churn could be caused by any number of factors. Storage nodes may go offline due to hardware or software failure, intermittent internet connectivity, power loss, complete disk failure, or software shutdown or removal. The more network churn that exists, the more redundancy is required to make up for the greater rate of node loss. The more redundancy that is required, the more bandwidth is needed for correct operation of the system. In fact, there is a tight relationship between network churn, additional redundancy, and bandwidth availability [9]. To keep background bandwidth usage and redundancy low, our network must have low network churn and a strong incentive to favor long-lived, stable nodes.

See section 7.3.3 and Blake *et al.* [9] for a discussion of how repair bandwidth varies as a function of node churn.

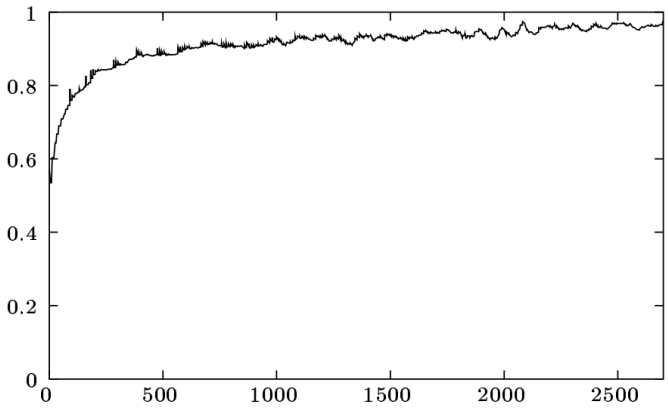


Figure 2.2: Probability of remaining online an additional hour as a function of uptime. The x axis represents minutes. The y axis shows the fraction of nodes that stayed online at least x minutes that also stayed online at least x + 60 minutes. Source: Maymounkov et al. *[8]*

2.6 Latency

Decentralized storage systems can potentially capitalize on massive opportunities for parallelism. Some of these opportunities include increased transfer rates, processing capabilities, and overall throughput even when individual network links are slow. However, parallelism cannot, by itself, improve *latency*. If an individual network link is utilized as part of an operation, its latency will be the lower bound for the overall operation. Therefore, any distributed system intended for high performance applications must continuously and aggressively optimize for low latency not only on an individual process scale but also for the system's entire architecture.

2.7 Bandwidth

Global bandwidth availability is increasing year after year. Unfortunately, access to high-bandwidth internet connections is unevenly distributed across the world. While some users can easily access symmetric, high-speed, unlimited bandwidth connections, others have significant difficulty obtaining the same type of access.

In the United States and other countries, the method in which many residential internet service providers (ISPs) operate presents two specific challenges for designers of a decentralized network protocol. The first challenge is the asymmetric internet connections offered by many ISPs. Customers subscribe to internet service based on an advertised download speed, but the upload speed is potentially an order of magnitude or two slower. The second challenge is that bandwidth is sometimes "capped" by the ISP at a fixed amount of allowed traffic per month. For example, in many US markets, the ISP Comcast imposes a one terabyte per month bandwidth cap with stiff fines for customers

who go over this limit [10]. An internet connection with a cap of 1 TB/month cannot average more than 385 KB/s over the month without exceeding the monthly bandwidth allowance, even if the ISP advertises speeds of 10 MB/s or higher. Such caps impose significant limitations on the bandwidth available to the network at any given moment.

With device failure and churn guaranteed, any decentralized system will have a corresponding amount of repair traffic. As a result, it is important to account for the bandwidth required not only for data storage and retrieval but also for data maintenance and repair [9]. Designing a storage system that is careless with bandwidth usage would not only give undue preference to storage node operators with access to unlimited high-speed bandwidth but also centralize the system to some degree. In order to keep the storage system as decentralized as possible and working in as many environments as possible, bandwidth usage must be aggressively minimized.

Please see section 7.1.1 for a discussion on how bandwidth availability and repair traffic limit usable space.

2.8 Object size

We can broadly classify large storage systems into two groups by average object size. To differentiate between the two groups, we classify a “large” file as a few megabytes or greater in size. A database is the preferred solution for storing many small pieces of information, whereas an object store or file system is ideal for storing many large files.

The initial product offering by Storj Labs is designed to function primarily as a decentralized object store for larger files. While future improvements may enable database-like use cases, object storage is the predominant initial use case described in this paper. We made protocol design decisions with the assumption that the vast majority of stored objects will be 4MB or larger. While smaller files are supported, they may simply be more costly to store.

It is worth noting that this will not negatively impact use cases that require reading lots of files smaller than a megabyte. Users can address this with a packing strategy by aggregating and storing many small files as one large file. The protocol supports seeking and streaming, which will allow users to download small files without requiring full retrieval of the aggregated object.

2.9 Byzantine fault tolerance

Unlike centralized solutions like Amazon S3, Storj operates in an untrusted environment where individual storage providers are not necessarily assumed to be trustworthy. Storj operates over the public internet, allowing anyone to sign up to become a storage provider.

We adopt the Byzantine, Altruistic, Rational (BAR) model [11] to discuss participants in the network.

- *Byzantine* nodes may deviate arbitrarily from the suggested protocol for any reason. Some examples include nodes that are broken or nodes that are actively trying to sabotage the protocol. In general, a *Byzantine* node is a bad actor, or one that optimizes for a utility function that is independent of the one given for the suggested protocol.
- Inevitable hardware failures aside, *Altruistic* nodes are good actors and participate in a proposed protocol even if the rational choice is to deviate.
- *Rational* nodes are neutral actors and participate or deviate only when it is in their net best interest.

Some distributed storage systems (e.g. datacenter-based cloud object storage systems) operate in an environment where all nodes are considered *altruistic*. For example, absent hardware failure or security breaches, Amazon's storage nodes will not do anything besides what they were explicitly programmed to do, because Amazon owns and runs all of them.

In contrast, Storj operates in an environment where every node is managed by its own independent operator. In this environment, we can expect that a majority of storage nodes are *rational* and a minority are *Byzantine*. Storj assumes no *altruistic* nodes.

We must include incentives that encourage the network to ensure that the rational nodes on the network (the majority of operators) behave as similarly as possible to the expected behavior of altruistic nodes. Likewise, the effects of Byzantine behavior must be minimized or eliminated.

Note that creating a system that is robust in the face of Byzantine behavior does not require a Byzantine fault tolerant consensus protocol—we avoid Byzantine consensus. See sections 4.9, 6.2, and appendix A for more details.

2.10 Coordination avoidance

A growing body of distributed database research shows that systems that avoid coordination wherever possible have far better throughput than systems where subcomponents are forced to coordinate to achieve correctness [12–19]. We use Bailis *et al.*'s informal definition that coordination is the requirement that concurrently executing operations synchronously communicate or otherwise stall in order to complete [16]. This observation happens at all scales and applies not only to distributed networks but also to concurrent threads of execution coordinating within the same computer. As soon as coordination is needed, actors in the system will need to wait for other actors, and waiting—due to coordination issues—can have a significant cost.

While many types of operations in a network may require coordination (e.g., opera-

tions that require linearizability[1] [15, 20, 21]), choosing strategies that avoid coordination (such as Highly Available Transactions [15]) can offer performance gains of two to three orders of magnitude over wide area networks. In fact, by carefully avoiding coordination as much as possible, the Anna database [17] is able to be 10 times faster than both Cassandra and Redis in their corresponding environments and 700 to 800 times faster than performance-focused in-memory databases such as Masstree or Intel's TBB [22]. Not all coordination can be avoided, but new frameworks (such as Invariant Confluence [16] or the CALM principle [18,19]) allow system architects to understand when coordination is required for consistency and correctness. As evidenced by Anna's performance successes, it is most efficient to avoid coordination where possible.

Systems that minimize coordination are much better at scaling from small to large workloads. Adding more resources to a coordination-avoidant system will directly increase throughput and performance. However, adding more resources to a coordination-dependent system (such as Bitcoin [23] or even Raft [24]) will not result in much additional throughput or overall performance.

To get to exabyte scale, minimizing coordination is one of the key components of our strategy. Surprisingly, many decentralized storage platforms are working towards architectures that require significant amounts of coordination, where most if not all operations must be accounted for by a single global ledger. For us to achieve exabyte scale, it is a fundamental requirement to limit hotpath coordination domains to small spheres which are entirely controllable by each user. This limits the applicability of blockchain-like solutions for our use case.

[1] Linearizable operations are atomic operations on a specific object where the order of operations is equivalent to the order given original "wall clock" time.

3. Framework

After having considered our design constraints, this chapter outlines the design of a framework consisting of only the most fundamental components. The framework describes all of the components that must exist to satisfy our constraints. As long as our design constraints remain constant, this framework will, as much as is feasible, describe Storj both now and ten years from now. While there will be some design freedom within the framework, this framework will obviate the need for future rearchitectures entirely, as independent components will be able to be replaced without affecting other components.

3.1 Framework overview

All designs within our framework will do the following things:

Store data When data is stored with the network, a client encrypts and breaks it up into multiple pieces. The pieces are distributed to peers across the network. When this occurs, metadata is generated that contains information on where to find the data again.

Retrieve data When data is retrieved from the network, the client will first reference the metadata to identify the locations of the previously stored pieces. Then the pieces will be retrieved and the original data will be reassembled on the client's local machine.

Maintain data When the amount of redundancy drops below a certain threshold, the necessary data for the missing pieces is regenerated and replaced.

Pay for usage A unit of value should be sent in exchange for services rendered.

To improve understandability, we break up the design into a collection of eight independent components and then combine them to form the desired framework.

The individual components are:

1. Storage nodes
2. Peer-to-peer communication
3. Redundancy
4. Metadata
5. Encryption
6. Audits and reputation
7. Data repair
8. Payments

3.2 Storage nodes

The storage node's role is to store and return data. Aside from reliably storing data, nodes should provide network bandwidth and appropriate responsiveness. Storage nodes are selected to store data based on various criteria: ping time, latency, throughput, bandwidth caps, sufficient disk space, geographic location, uptime, history of responding accurately to audits, and so forth. In return for their service, nodes are paid for both data egress and data at rest.

Because storage nodes are selected via changing variables external to the protocol, node selection is an explicit, non-deterministic process in our framework. This means that we must keep track of which nodes were selected for each upload via a small amount of metadata; we can't select nodes for storing data implicitly or deterministically as in a system like Dynamo [25]. As with GFS [26], HDFS [27], or Lustre [28], this decision implies the requirement of a metadata storage system to keep track of selected nodes (see section 3.5).

3.3 Peer-to-peer communication

All peers on the network communicate via a standarized protocol. The framework requires that this protocol:

- provides peer reachability, even in the face of firewalls and NATs where possible. This may require techniques like STUN [29], UPnP [30], NAT-PMP [31], etc.
- provides authentication as in S/Kademlia [32], where each participant cryptographically proves the identity of the peer with whom they are speaking to avoid man-in-the-middle attacks.
- provides complete privacy. In cases such as bandwidth measurement (see section 4.17), the client and storage node must be able to communicate without any risk of eavesdroppers. The protocol should ensure that all communications are private by default.

Additionally, the framework requires a way to look up peer network addresses by a unique identifier so that, given a peer's unique identifier, any other peer can connect to it. This responsibility is similar to the internet's standard domain name system (DNS) [33], which is a mapping of an identifier to an ephemeral connection address, but unlike DNS, there can be no centralized registration process. A network overlay can be built on top of our chosen peer-to-peer communication protocol to achieve these goals. See Section 4.6 for implementation details.

3.4 Redundancy

We assume that at any moment, any storage node could go offline permanently. Our redundancy strategy must store data in a way that provides access to the data with high probability, even though any given number of individual nodes may be in an offline state. To achieve a specific level of *durability* (defined as the probability that data remains available in the face of failures), many products in this space use simple replication. Unfortunately, this ties durability to the network *expansion factor*, which is the storage overhead for reliably storing data. This significantly increases the total cost relative to the stored data.

For example, suppose a certain desired level of durability requires a replication strategy that makes eight copies of the data. This yields an expansion factor of 8x, or 800%. This data then needs to be stored on the network, using bandwidth in the process. Thus, more replication results in more bandwidth usage for a fixed amount of data. As discussed in the protocol design constraints (section 2.7) and Blake *et al.* [9], high bandwidth usage prevents scaling, so this is an undesirable strategy for ensuring a high degree of file durability.

As an alternative to simple replication, *erasure codes* provide a much more efficient method to achieve redundancy. Erasure codes are well-established in use for both distributed and peer-to-peer storage systems [34–40]. Erasure codes are an encoding scheme for manipulating data durability without tying it to bandwidth usage, and have been found to improve repair traffic significantly over replication [9]. Importantly, they allow changes in durability without changes in expansion factor.

An erasure code is often described by two numbers, k and n. If a block of data is encoded with a (k, n) erasure code, there are n total generated *erasure shares*, where only any k of them are required to recover the original block of data. If a block of data is s bytes, each of the n erasure shares is roughly s/k bytes. Besides the case when $k = 1$ (replication), all erasure shares are unique.

Interestingly, the durability of a $(k = 20, n = 40)$ erasure code is better than a $(k = 10, n = 20)$ erasure code, even though the expansion factor (2x) is the same for both. This is because the risk is spread across more nodes in the $(k = 20, n = 40)$ case. These considerations make erasure codes an important part of our general framework.

To better understand how erasure codes increase durability without increasing expansion factors, the following table shows various choices of k and n, along with the expansion factor and associated durability:

k	n	Exp. factor	$P(D \mid p = 10\%)$
2	4	2	99.207366813274616%
4	8	2	99.858868985411326%
8	16	2	99.995462406878260%
16	32	2	99.999994620652776%
20	40	2	99.999999807694154%
32	64	2	99.999999999990544%

In contrast, replication requires significantly higher expansion factors for the same durability. The following table shows durability with a replication scheme:

k	n	Exp. factor	$P(D \mid p = 10\%)$
1	1	1	90.483741803595962%
1	2	2	98.247690369357827%
1	3	3	99.640050681691051%
1	10	10	99.999988857452166%
1	16	16	99.999999998036174%

To see how these tables were calculated, we'll start with the simplifying assumption that p is the monthly node churn rate (that is, the fraction of nodes that will go offline in a month on average). Mathematically, time-dependent processes are modeled according to the Poisson distribution, where it is assumed that λ events are observed in the given unit of time. As a result, we model durability as the cumulative distribution function (CDF) of the Poisson distribution with mean $\lambda = pn$, where we expect λ pieces of the file to be lost monthly. To estimate durability, we consider the CDF up to $n–k$, looking at the probability that at most $n – k$ pieces of the file are lost in a month and the file can still be rebuilt. The CDF is given by:

$$P(D) = e^{-\lambda} \sum_{i=0}^{n-k} \frac{\lambda^i}{i!}$$

The expansion factor still plays a big role in durability, as seen in the following table:

k	n	Exp. factor	$P(D \mid p = 10\%)$
4	6	1.5	97.688471224736705%
4	12	3	99.999514117129605%
20	30	1.5	99.970766304935266%
20	50	2.5	99.999999999999548%
100	150	1.5	99.999999999973570%

By being able to tweak the durability independently of the expansion factor, erasure coding allows very high durability to be achieved with surprisingly low expansion factors. Because of how limited bandwidth is as a resource, completely eliminating replication

as a strategy and using erasure codes only for redundancy causes a drastic decrease in bandwidth footprint.

Erasure coding also results in storage nodes getting paid more. High expansion factors dilute the incoming funds per byte across more storage nodes; therefore, low expansion factors, such as those provided by erasure coding, allow for a much more direct passthrough of income to storage node operators.

3.4.1 Erasure codes' effect on streaming

Erasure codes are used in many streaming contexts such as audio CDs and satellite communications [36], so it's important to point out that using erasure coding in general does not make our streaming design requirement (required by Amazon S3 compatibility, see section 2.4) more challenging. Whatever erasure code is chosen for our framework, as with CDs, streaming can be added on top by encoding small portions at a time, instead of attempting to encode a file all at once. See section 4.8 for more details.

3.4.2 Erasure codes' effect on long tails

Erasure codes enable an enormous performance benefit, which is the ability to avoid waiting for "long-tail" response times [41]. A long-tail response occurs in situations where a needed server has an unreasonably slow operation time due to a confluence of unpredictable factors. Long-tail responses are so-named due to their rare average rate of occurrence but highly variable nature, which in a probability density graph looks like a "long tail." In aggregate, long-tail responses are a big issue in distributed system design.

In MapReduce, long-tail responses are called "stragglers." MapReduce executes redundant requests called "backup tasks" to make sure that if specific stragglers take too long, the overall operation can still proceed without waiting. If the backup task mechanism is disabled in MapReduce, basic operations can take 44% longer to complete, even though the backup task mechanism is causing duplicated work [42].

By using erasure codes, we are in a position to create MapReduce-like backup tasks for storage [37, 38]. For uploads, a file can be encoded to a higher (k, n) ratio than necessary for desired durability guarantees. During an upload, after enough pieces have uploaded to gain required redundancy, the remaining additional uploads can be canceled. This cancellation allows the upload to continue as fast as the fastest nodes in a set, instead of waiting for the slowest nodes.

Downloads are similarly improved. Since more redundancy exists than is needed, downloads can be served from the fastest peers, eliminating a wait for temporarily slow or offline peers.

The outcome is that every request is satisfiable by the fastest nodes participating in any given transaction, without needing to wait for a slower subset. Focusing on opera-

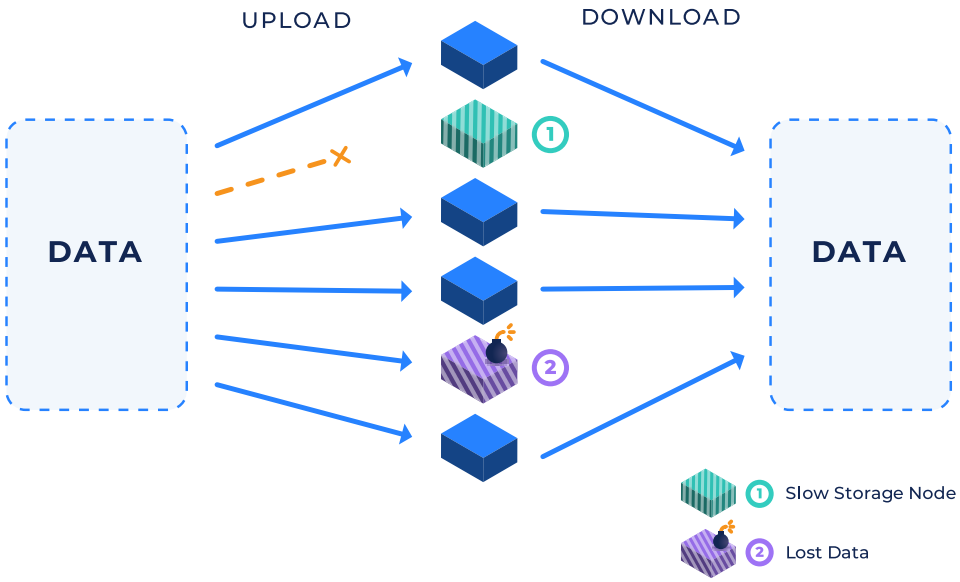


Figure 3.1: Various outcomes during upload and download

tions where the result is only dependent on the fastest nodes of a random subpopulation turns what could be a potential liability (highly variable performance from individual actors) into a great source of strength for a distributed storage network, while still providing great load balancing characteristics.

This ability to over-encode a file greatly assists dynamic load balancing of popular content on the network. See section 6.1 for a discussion on how we plan to address load balancing very active files.

3.5 Metadata

Once we split an object up with erasure codes and select storage nodes on which to store the new pieces, we now need to keep track of which storage nodes we selected. We allow users to choose storage based on geographic location, performance characteristics, available space, and other features. Therefore, instead of implicit node selection such as a scheme using consistent hashing like Dynamo [25], we must use an explicit node selection scheme such as directory-based lookups [43]. Additionally, to maintain Amazon S3 compatibility, the user must be able to choose an arbitrary key, often treated like a path, to identify this mapping of data pieces to node. These features imply the necessity of a metadata storage system.

Amazon S3 compatibility once again imposes some tight requirements. We should support: hierarchical objects (paths with prefixes), per-object key/value storage, arbitrarily large files, arbitrarily large amounts of files, and so forth. Objects should be able to be

stored and retrieved by arbitrary key; in addition, deterministic iteration over those keys will be required to allow for paginated listing.

Every time an object is added, edited, or removed, one or more entries in this metadata storage system will need to be adjusted. As a result, there could be heavy churn in this metadata system, and across the entire userbase the metadata itself could end up being sizable.

For example, suppose in a few years the network stores one total exabyte of data, where the average object size is 50MB and our erasure code is selected such that $n = 40$. One exabyte of 50MB objects is 20 billion objects. This metadata system will need to keep track of which 40 nodes were selected for each object. If each metadata element is roughly $40 \cdot 64 + 192$ bytes (info for each selected node plus the path and some general overhead), there are over 55 terabytes of metadata of which to keep track.

Fortunately, the metadata can be heavily partitioned by the user. A user storing 100 terabytes of 50 megabyte objects will only incur a metadata overhead of 5.5 gigabytes. It's worth pointing out that these numbers vary heavily with object size: the larger the average object size, the less the metadata overhead.

An additional framework focus is enabling this component—metadata storage—to be interchangeable. Specifically, we expect the platform to incorporate multiple implementations of metadata storage that users will be allowed to choose between. This greatly assists with our design goal of coordination avoidance between users (see section 2.10).

As metadata retrieval is a prerequisite to data retrieval in any operation, availability of services responsible for metadata storage and retrieval are designed and implemented to be distributed over multiple regions within a geography to provide maximum resistance to any single point of failure, whether is it a device, server, network connection, or entire region within a geography.

Aside from scale requirements, to implement Amazon S3 compatibility, the desired API is straightforward and simple: *Put* (store metadata at a given path), *Get* (retrieve metadata at a given a path), *List* (paginated, deterministic listing of existing paths), and *Delete* (remove a path). See Figure 2.1 for more details.

3.6 Encryption

Regardless of storage system, our design constraints require total security and privacy. All data or metadata will be encrypted. Data must be encrypted as early as possible in the data storage pipeline, ideally before the data ever leaves the source computer. This means that an Amazon S3-compatible interface or appropriate similar client library should run colocated on the same computer as the user's application.

Encryption should use a pluggable mechanism that allows users to choose their desired encryption scheme. It should also store metadata about that encryption scheme to

allow users to recover their data using the appropriate decryption mechanism in cases where their encryption choices are changed or upgraded.

To support rich access management features, the same encryption key should not be used for every file, as having access to one file would result in access to decryption keys for all files. Instead, each file should be encrypted with a unique key. This should allow users to share access to certain selected files without giving up encryption details for others.

Because each file should be encrypted differently with different keys and potentially different algorithms, the metadata about that encryption must be stored somewhere in a manner that is secure and reliable. This metadata, along with other metadata about the file, including its path, will be stored in the previously discussed metadata storage system, encrypted by a deterministic, hierarchical encryption scheme. A hierarchical encryption scheme based on BIP32 [44] will allow subtrees to be shared without sharing their parents and will allow some files to be shared without sharing other files. See section 4.11 for a discussion of our path-based hierarchical deterministic encryption scheme.

3.7 Audits and reputation

Incentivizing storage nodes to accurately store data is of paramount importance to the viability of this whole system. It is essential to be able to validate and verify that storage nodes are accurately storing what they have been asked to store.

When storage nodes initially join the network, they generate a unique identity via a small proof of work function. Storage nodes build up an initial reputation score during a vetting period during which they are subject to an increased level of audit and uptime checks to ensure the nodes are properly operated. During the first 9 months of operation, a portion of storage node earnings are held by a Satellite as an offset against the possibility that a node may leave the network with data stored on the storage node such that some portion of that data may require repair. At all times while a storage node is operating on the network, it is subject to an audit process for each Satellite for which it stores data.

Many storage systems use probabilistic per-file audits, called *proofs of retrievability*, as a way of determining when and where to repair files [45, 46]. We are extending the probabilistic nature of common per-file proofs of retrievability to range across all possible files stored by a specific node. Audits, in this case, are probabilistic challenges that confirm, with a high degree of certainty and a low amount of overhead, that a storage node is well-behaved, keeping the data it claims, and not susceptible to hardware failure or malintent. Audits function as “spot checks” [47] to help calculate the future usefulness of a given storage node.

In our storage system, audits are simply a mechanism used to determine a node’s degree of stability. Failed audits will result in a storage node being marked as bad, which will result in redistributing data to new nodes and avoiding that node altogether in the fu-

ture. Storage node uptime and overall health are the primary metrics used to determine which files need repair.

As is the case with proofs of retrievability [45, 46], this auditing mechanism does not audit all bytes in all files. This can leave room for false positives, where the verifier believes the storage node retains the intact data when it has actually been modified or partially deleted. Fortunately, the probability of a false positive on an individual partial audit is easily calculable (see section 7.2). When applied iteratively to a storage node as a whole, detection of missing or altered data becomes certain to within a known and modifiable error threshold.

A reputation system is needed to persist the history of audit outcomes for given node identities. Our overall framework has flexible requirements on the use of such a system, but see section 4.15 for a discussion of our initial approach.

3.8 Data repair

Data loss is an ever-present risk in any distributed storage system. While there are many potential causes for file loss, storage node churn (storage nodes joining and leaving the network) is the largest leading risk by a significant degree compared to other causes. As discussed in section 2.5, network session time in many real world systems range from hours to mere minutes [7]. While there are many other ways data might get lost, such as corruption, malicious behavior, bad hardware, software error, or user initiated space reclamation, these issues are less serious than full node churn. We expect node churn to be the dominant cause of data loss in our network.

Because audits are validating that conforming nodes store data correctly, all that remains is to detect when a storage node stops storing data correctly or goes offline and then repair the data it had to new nodes. To repair the data, we will recover the original data via an erasure code reconstruction from the remaining pieces and then regenerate the missing pieces and store them back in the network on new storage nodes.

It is vital in our system to incentivize storage node participants to remain online for much longer than a few hours. To encourage this behavior, our payment strategy will involve rewarding storage node operators that keep their nodes participating for months and years at a time.

3.9 Payments

Payments, value attribution, and billing in decentralized networks are a critical part of maintaining a healthy ecosystem of both supply and demand. Of course, decentralized payment systems are still in their infancy in a number of ways.

For our framework to achieve low latency and high throughput, we must not have

transactional dependencies on a blockchain (see section 2.10). This means that an adequately performant storage system cannot afford to wait for blockchain operations. When operations should be measured in milliseconds, waiting for a cluster of nodes to probabilistically come to agreement on a shared global ledger is a non-starter.

Our framework instead emphasizes game theoretic models to ensure that participants in the network are properly incentivized to remain in the network and behave rationally to get paid. Many of our decisions are modeled after real-world financial relationships. Payments will be transferred during a background settlement process in which well-behaved participants within the network cooperate. Storage nodes in our framework should limit their exposure to untrusted payers until confidence is gained that those payers are likely to pay for services rendered.

In addition, the framework also tracks and aggregates the value of the consumption of those services by those who own the data stored on the network. By charging for usage, the framework is able to support the end-to-end economics of the storage marketplace ecosystem.

Although the Storj network is payment agnostic and the protocol does not require a specific payment type, the network assumes the Ethereum-based STORJ token as the default mechanism for payment. While we intend for the STORJ token to be the primary form of payment, in the future other alternate payment types could be implemented, including Bitcoin, Ether, credit or debit card, ACH transfer, or even physical transfer of live goats.

4. Concrete implementation

We believe the framework we've described to be relatively fundamental given our design constraints. However, within the framework there still remains some freedom in choosing how to implement each component.

In this section, we lay out our initial implementation strategy. We expect the details contained within this section to change gradually over time. However, we believe the details outlined here are viable and support a working implementation of our framework capable of providing highly secure, performant, and durable production-grade cloud storage.

4.1 Definitions

The following defined terms are used throughout the description of the concrete implementation that follows:

4.1.1 Actors

Client A user or application that will upload or download data from the network.

Peer class A cohesive collection of network services and responsibilities. There are three different peer classes that represent services in our network: *storage nodes*, *Satellites*, and *Uplinks*.

Storage node This peer class stores data for others and gets paid for storage and bandwidth. It registers itself directly with Satellites it trusts.

Uplink This peer class represents any application or service that implements *libuplink* and wants to store and/or retrieve data. This peer class is not expected to remain online like the other two classes and is relatively lightweight. This peer class performs encryption, erasure encoding, and coordinates with the other peer classes on behalf of the customer/client.

- **libuplink** A library which provides all necessary functions to interact with *storage nodes* and *Satellites* directly. This library will be available in a number of different programming languages.
- **Gateway** A service which provides a compatibility layer between other object storage services such as Amazon S3 and *libuplink* exposing an Amazon S3-compatible API.
- **Uplink CLI** A command line interface for uploading and downloading files from the network, managing permissions and sharing, and managing accounts.

Satellite This peer class allows storage nodes to register with it, caches node address information, stores per-object metadata, maintains storage node reputation, aggregates billing data, pays storage nodes, performs audits and repair, and manages authorization and user accounts. Users have accounts on and trust specific Satel-

lites. Any user can run their own Satellite, but we expect many users to elect to avoid the operational complexity and create an account on another Satellite hosted by a trusted third party such as Storj Labs, a friend, group, or workplace.

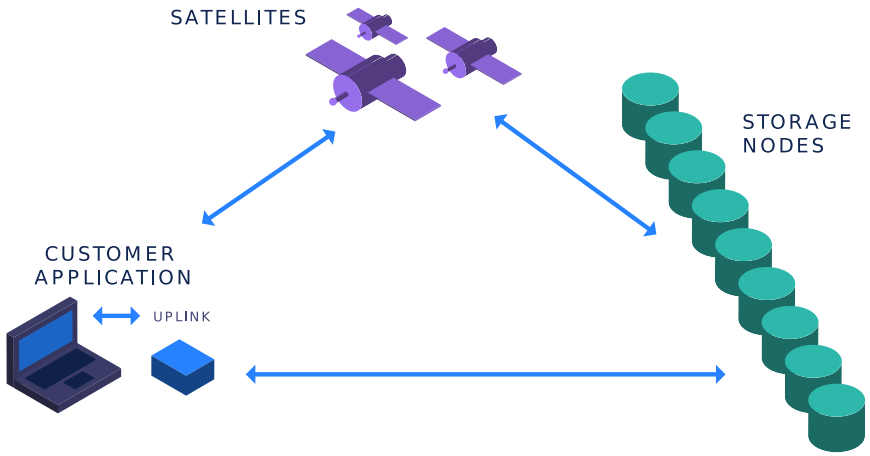


Figure 4.1: The three different peer classes

4.1.2 Data

Bucket A *bucket* is an unbounded but named collection of objects (or files) identified by object keys. Every object has a unique key within a bucket.

Object Key An *object key* is a unique identifier for an object (or file) within a bucket. An *object key* is an arbitrary string of bytes. Object keys, like traditional filesystem paths, contain forward slashes at access control boundaries. Forward slashes (referred to as the path separator) separate object key path components. An example *object key* might be *videos/carlsagan/gloriousdawn.mp4*, where the path components are *videos*, *carlsagan*, and *gloriousdawn.mp4*. Unless otherwise requested, we encrypt paths before they ever leave the customer's application's computer.

Object (or File) An *object* (or *file*) is the main data type in our system. An object is referred to by an *object key*, contains an arbitrary amount of bytes, and has no minimum or maximum size. An object is represented by an ordered collection of one or more segments. Segments have a fixed maximum size. An object also supports a limited amount of key/value user-defined fields called object metadata. Like object keys, the data contained in an object is encrypted before it ever leaves the client computer.

Object metadata *Object metadata* are user defined key/value fields that are associated with an object. Object metadata are stored encrypted.

Segment A *segment* represents a single array of bytes, between 0 and a Satellite-defined maximum *segment* size. See section 4.8.2 for more details.

Remote Segment A *remote segment* is a segment that will be erasure encoded and distributed across the network. A *remote segment* is larger than the metadata required to keep track of its bookkeeping, which includes information such as the IDs of the nodes that the data is stored on.

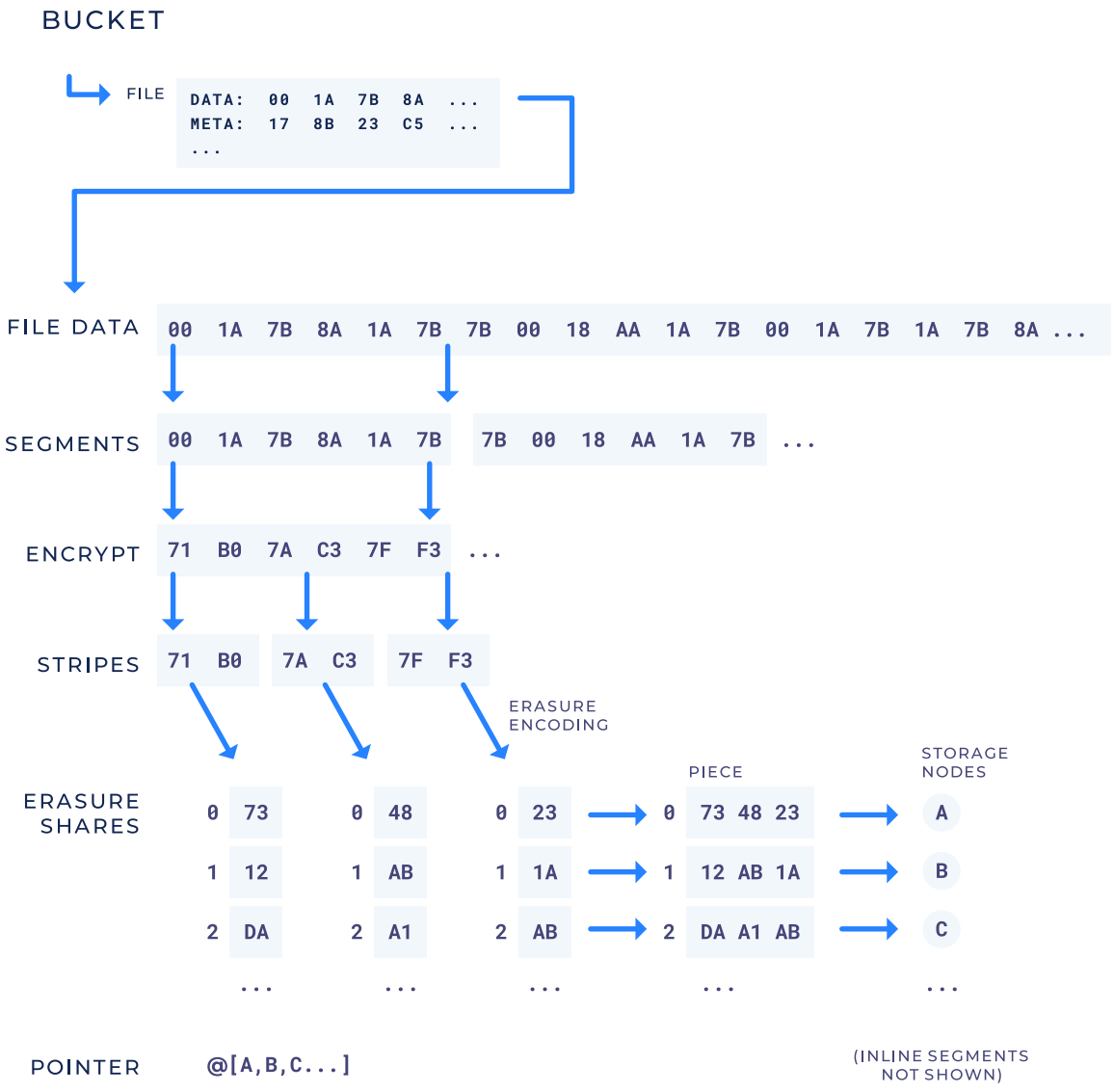


Figure 4.2: Files, segments, stripes, erasure shares, and pieces

Inline Segment An *inline segment* is a segment that is small enough where the data it represents takes less space than the corresponding data a remote segment will need to keep track of which nodes had the data. In these cases, the data is stored "inline" instead of being stored on nodes.

Encryption block An encryption block is a small fixed amount of bytes that is used as an encryption boundary size. Authenticated encryption happens on *encryption blocks* individually (with monotonically increasing nonces). All segments are encrypted in encryption blocks. Encryption blocks are often an integer multiple of the stripe size for alignment reasons.

Stripe A *stripe* is a further subdivision of a segment. A *stripe* is a fixed amount of bytes that is used as an erasure encoding boundary size. Erasure encoding happens on *stripes* individually. Inline segments do not have erasure encoding, and thus only remote segments erasure encode stripes. A *stripe* is the unit on which audits are performed. See section 4.8.3 for more details.

Erasure Share When a stripe is erasure encoded, it generates multiple pieces called *erasure shares*. Only a subset of the *erasure shares* are needed to recover the original stripe. Each *erasure share* has an index identifying which *erasure share* it is (e.g., the first, the second, etc.).

Piece When a remote segment's stripes are erasure encoded into erasure shares, the erasure shares for that remote segment with the same index are concatenated together, and that concatenated group of erasure shares is called a *piece*. If there are n erasure shares after erasure encoding a stripe, then there are n *pieces* after processing a remote segment. The ith *piece* is the concatenation of all of the ith erasure shares from that segment's stripes. See section 4.8.5 for more details.

Pointer A *pointer* is a data structure that either contains the inline segment data, or keeps track of which storage nodes the pieces of a remote segment were stored on, along with other per-file metadata.

4.2 Peer classes

Our overall strategy extends from our previous version [35] and also heavily mirrors distributed storage systems such as the Google File System [26] (and other GFS-like systems [27, 48, 49]) and the Lustre distributed file system [28]. In every case, there are three major actors in the network: metadata servers, object storage servers, and clients. Object storage servers hold the bulk of the data stored in the system. Metadata servers keep track of per-object metadata and where the objects are located on object storage servers. Clients provide a coherent view and easy access to files by communicating with both the metadata and object storage servers.

Lustre's architecture is proven for high performance. The majority of the top 100 fastest supercomputers use Lustre for their high-performance, scalable storage [28]. While we don't expect to achieve equal performance over a wide-area network, we expect dramatically better performance than other architectures. Any limitation, if any, we experience in performance will be due to factors besides our overall architecture.

Our previous version used different names for each component. What we previously referred to as Storj Share, we now refer to as simply storage nodes. Our formerly centralized single Bridge instance can now be run by anyone and is referred to as a Satellite. Our *libstorj* library will be made to be backwards compatible where possible, but we now refer to client software as Uplinks.

4.3 Storage node

The main duty of the storage node is to reliably store and return data. Node operators are individuals or entities that have excess hard drive space and want to earn income by renting their space to others. These operators will download, install, and configure Storj software locally, with no account required anywhere.[1] They will then configure disk space and per-Satellite bandwidth allowance. During Satellite registration and check-in, storage nodes will advertise how much bandwidth and hard drive space is available, and their designated STORJ token wallet address.

To simplify lifecycle management for ephemeral files, storage nodes also keep track of optional per-piece "time-to-live", or TTL, designations. Pieces may be stored with a specific TTL expiry where data is expected to be deleted after the expiration date. If no TTL is provided, data is expected to be stored indefinitely. This means storage nodes have a database of expiration times and must occasionally clear out old data.

Storage nodes must additionally keep track of signed bandwidth allocations (see section 4.17) to send to Satellites for later settlement and payment. This also requires a small database.

Storage nodes can choose with which Satellites to work. If they work with multiple Satellites (the default behavior), then payment may come from multiple sources on varying payment schedules. Storage nodes are paid by specific Satellites for (1) returning data when requested in the form of egress bandwidth payment, and for (2) storing data at rest. Storage nodes are expected to reliably store all data sent to them and are paid with the assumption that they are faithfully storing all data. Storage nodes that fail random audits will be "disqualified" and thus removed from the pool, can lose held funds to cover additional costs, and will receive limited to no future payments. Storage nodes are *not* paid for the initial transfer of data to store (ingress bandwidth). This is to discourage storage nodes from deleting data only to be paid for storing more, which became a problem with our previous version [35]. While storage nodes are paid for repair egress bandwidth usage, some Satellites may opt to pay less than normal retrieval egress bandwidth usage. Storage nodes are not paid for Satellite registration operations or any other maintenance traffic.

Storage nodes will support three primary methods: *get*, *put*, and *delete*. Each method will take a *piece ID*, a *Satellite ID*, a signature from the associated Satellite instance, and a

[1] Registration with a US-1099 tax form service may be required.

bandwidth allocation (see section 4.17). The *Satellite ID* forms a namespace. An identical *piece ID* with a different *Satellite ID* refers to a different *piece*.

The *put* operation will take a stream of bytes and an optional TTL and store the bytes such that any subrange of bytes can be retrieved again via a *get* operation. *Get* operations are expected to work until the TTL expires (if a TTL was provided) or until a *delete* or garbage collection (4.19) operation is received, whichever comes first.

Storage nodes will allow administrators to configure maximum allowed disk space. They will keep track of how much is remaining, and reject operations that do not have a valid signature from the appropriate Satellite.

Storage nodes also support a garbage collection system, where they can retrieve a probabilistic data structure called a Bloom filter [50] that indicates which pieces are no longer tracked and can be safely deleted.

The storage node has been released as open source software.

4.4 Node identity

During setup, storage nodes, Satellites, and Uplinks all generate their own identity and certificates for use in the network.

Each node will operate its own certificate authority, which requires a public/private key pair and a self-signed certificate. The certificate authority's private key will ideally be kept in cold storage to prevent key compromise. It's important that the certificate authority private key be managed with good operational security because key rotation for the certificate authority will require a brand new node ID.

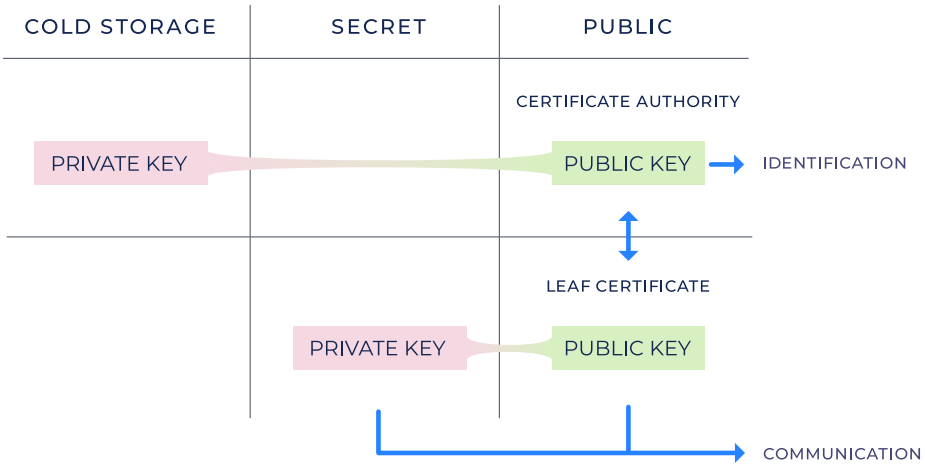


Figure 4.3: The different keys and certificates that compose a storage node's overall identity. Each row represents a private/public key pair.

The public key of the node's certificate authority determines its *node ID*. As in S/Kademlia [32], the node ID will be the hash of the public key and will serve as a proof of work for joining the network. Unlike Bitcoin's proof of work [23], the proof of work will be dependent on how many *trailing* zero bits one can find in the hash output. This cost is meant to make Sybil attacks prohibitively expensive and time consuming.

Each node will have a revocable leaf certificate and key pair that is signed by the node's certificate authority. Nodes use the leaf key pair for communication. Each leaf has a signed timestamp that Satellites keep track of per node. Should the leaf become compromised, the node can issue a new leaf with a later timestamp. Interested peers will make note of newly seen leaf timestamps and reject connections from nodes with older leaf certificates. As an optimized special case, peers will not need to make a note when the leaf certificate and certificate authority share the same timestamp.

4.5 Peer-to-peer communication

We are using a self-made gRPC-like [51] application protocol (DRPC). For security, it runs on top of either Transport Layer Security (TLS) [52] or the Noise Protocol Framework in IK mode [53]. That layer then runs on top of TCP or QUIC (over UDP). TCP and QUIC provide reliable, ordered delivery; TLS and Noise provide privacy and authentication; and DRPC provides multiplexing and a convenient programmer interface. Noise is used in certain cases to reduce round trips due to connection handshakes in situations where the data is already encrypted and forward secrecy isn't necessary.

When using authenticated communication such as TLS or Noise, every peer can ascertain the ID of the node with which it is speaking by validating the certificate chain and hashing its peer's certificate authority's public key. It can then be estimated how much work went into constructing the node ID by considering the number of trailing zero bits at the end of the ID. Satellites can configure a minimum proof of work required to pass an audit (section 4.13) such that, over time, the network will require greater proofs of work due to natural user intervention.

4.6 Node discovery

At this point, we have storage nodes and we have the means to identify and communicate with them if we know their address. We must account for the fact that storage nodes will often be on consumer internet connections and behind routers with constantly changing IP addresses. Therefore, the node discovery system's goal is to implement a means to look up a node's latest address by node ID, somewhat similar to the role DNS provides for the public internet.

In our per-Satellite node discovery cache, each Satellite stores information to be able to communicate with the nodes in its network, as well as data needed to select nodes on

which to store data. This caching service will live independently in each Satellite and is our primary source of truth for DNS-like functionality for node lookups.

When a node joins the network, it reaches out to each Satellite in its Satellite trustlist (see 4.18 for details) indicating that it is available and willing to accept data from the Satellite. In return, the Satellite confirms or denies the success of the connection. If it is successful, the Satellite stores the node's available disk space, STORJ wallet address, IP address, and any other metadata the Satellite needs in its node discovery cache.

The node will subsequently check-in with each of its Satellites on an ongoing basis, perhaps once per hour, to notify the Satellites of any updates. If a node has not communicated with a Satellite after a certain amount of time, the Satellite will reach out to the node to check its status. If it can no longer successfully reach the node, the Satellite will stop suggesting that node to clients.

4.7 Redundancy

We use the Reed-Solomon erasure code [54]. To implement our solution for reducing the effects of long-tails (see section 3.4.2), we choose 4 numbers for each object that we store, k, m, o, and n, such that $k \leq m \leq o \leq n$. The standard Reed-Solomon numbers are k and n, where k is the minimum required number of pieces for reconstruction, and n is the total number of pieces generated during creation.

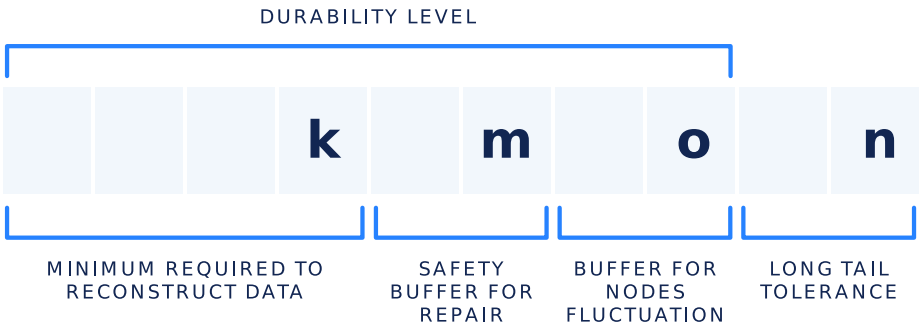


Figure 4.4: The relationship between k, m, o, and n.

The *minimum safe* and *optimal* values, respectively, are m and o. The value m is chosen such that if a Satellite notices the amount of available pieces has fallen below m, it triggers a repair immediately in an attempt to make sure we always maintain k or more pieces (m is called r_0 in Giroire *et al.* [34]). To achieve our long-tail performance improvements [37,38,41,42], the value o is chosen such that during uploads and repairs, as soon as o pieces have finished uploading, remaining pieces up to n are canceled. Furthermore, o is chosen such that storing o pieces is all that is needed to achieve the desired durability goals; n is thus chosen such that storing n pieces will be excess durability.

The amount of long tail uploads we can tolerate is $n-o$, and thus is the amount of slow nodes to which we are immune. The amount of nodes that can go temporarily offline at

the same time without triggering a repair is $o - m$. The safety buffer to avoid losing the data between the time we recognize the data requires a repair and the actual repair is executed is $m - k$.

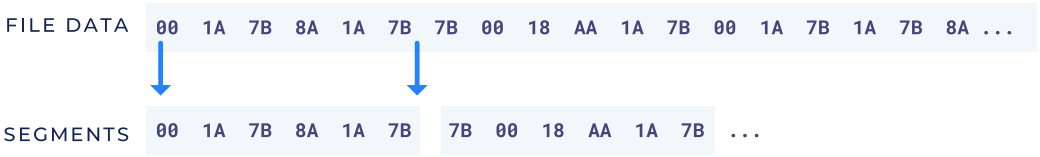
See section 7.3 for how we select our Reed-Solomon numbers. Also see section 4.14 for a discussion about how we repair data as its durability drops over time.

4.8 Structured file storage

4.8.1 Object metadata

Many applications benefit from being able to keep metadata alongside files. Amazon S3 supports "object metadata" [55] to assist with this need. This functionality is called "extended attributes" in many POSIX compatible systems. Every object will include a limited set of user-specified key-value pairs that will be stored alongside other metadata about the object.

4.8.2 Objects as Segments



In our previous version [35], the term *shard* referred to *pieces* on storage nodes, whereas *sharding* referred to segmenting a file into smaller chunks for easier processing. With the addition of erasure coding in our previous version, these terms became somewhat confusing, so we have decided to distinguish each meaning with new words.

The sharding process is now called *segmenting*, and the highest level subdivision of an object's stream of data is called a *segment*. Unfortunately, there is general inconsistency using these terms in the literature. GFS refers to segments as chunks [26]. Lustre refers to segments as stripes [28], but we use the term *stripes* for a further subdelineation.

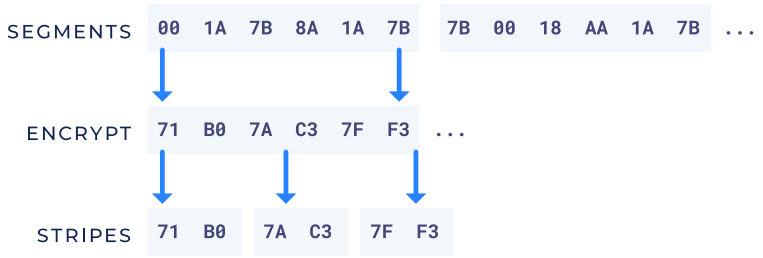
A file may be small enough that it consists of only one segment. If that segment is smaller than the metadata required to store it on the network, the data will be stored inline with the metadata.[2] We call this an *inline segment*.

For larger files, the data will be broken into one or more large remote segments. Segmenting in this manner offers numerous advantages to security, privacy, performance, and availability. As in other distributed storage systems [26–28, 48, 49], segmenting large files (e.g. videos) and distributing the segments across the network reduces the impact of

[2] The Linux file system Ext4 performs the same optimization with inline *inodes* [56].

content delivery on any given node, as bandwidth demands are distributed more evenly across the network. As with our previous version [35], standardized sizes help frustrate attempts to determine the content of a given segment and can help obscure the flow of data through the network. In addition, the end user can take advantage of parallel transfer, similar to BitTorrent [57] or other peer-to-peer networks. Lastly, capping the size of segments allows for more uniform storage node filling. Thus, a node only needs enough space to store a segment to participate in the network, and a client doesn't need to find nodes that have enough space for a large file.

4.8.3 Segments as Stripes

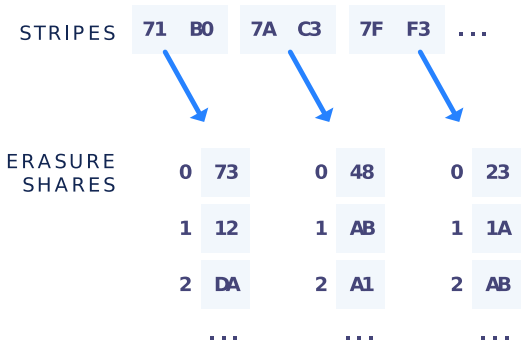


In many situations, it's important to access a subsection of a larger piece of data. Some file formats, such as video files or disk images, support seeking, where only a subset of the data is needed for read operations. As the creators of audio CDs discovered, it's useful to be able to decode small parts of a segment to support these operations [36].

For this purpose, a stripe defines a subset of a segment and should be no more than a couple of kilobytes in size. Encryption happens on an *encryption block* boundary, which may be a small multiple of stripes, whereas erasure encoding happens on a single stripe at a time. Because we use authenticated encryption, every encryption block has a slight overhead, so slightly larger encryption sizes are preferred. However, audits happen on stripes, and we want audit bandwidth usage to be small.

For the reader familiar with the *zfec* library, in *filefec* mode, *zfec* refers to a stripe as a chunk [40].

4.8.4 Stripes as Erasure Shares



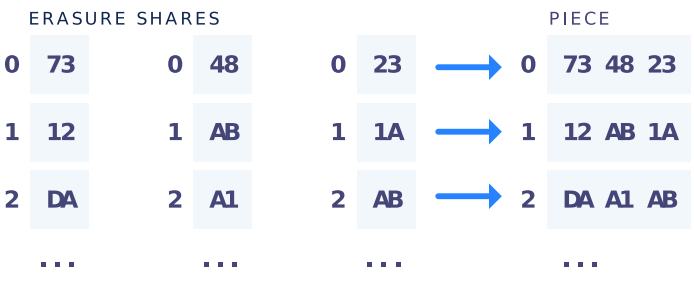
As discussed in sections 3.4 and 4.7, erasure codes give us the chance to control network durability in the face of unreliable storage nodes.

Stripes are the boundary by which we perform erasure encoding. In a (k, n) erasure code scheme, n erasure shares are generated for every stripe [54]. For example, perhaps a *stripe* is broken into 40 *erasure shares* ($n = 40$), where any 20 ($k = 20$) are needed to reconstruct the *stripe*. Each of the 40 *erasure shares* will be 1/20th the size of the original *stripe*.

Erasure encoding a single stripe at a time allows us to read small portions of a large segment without retrieving the entire segment first [36]. It also allows us to stream data into the network without staging it beforehand, and it enables a number of other useful features.

See section 7.3.3 for a breakdown of how varying the erasure code parameters affects availability and redundancy.

4.8.5 Erasure Shares as Pieces



Because stripes are already small, erasure shares are often much smaller, and the metadata to keep track of all of them separately will be immense relative to their size. All n erasure shares have a well-defined index associated with them. More specifically, for a fixed stripe and any given n, the ith share of an erasure code will always be the same. As with the *zfec* library's *filefec* mode [40], instead of keeping track of all of the erasure

shares separately, we pack all of the erasure shares with the same index into a *piece*. In a (k, n) scheme, there are n pieces, where each piece i is the ordered concatenation of all of the erasure shares with index i. As a result, where each erasure share is 1/kth of a *stripe*, each piece is 1/kth of a *segment*, and only k pieces are needed to recover the full segment. A piece is what we store on a storage node.

Satellites generate a brand-new, randomly chosen *root piece ID* each time a new upload begins. The Uplink will keep the *root piece ID* secret and send a *node-specific piece ID* to each storage node, formed by taking the Hash-based Message Authentication Code (HMAC) of the root piece ID and the node's ID. This serves to obscure what pieces belong together from storage nodes. The root piece ID is stored in the pointer.

Storage nodes namespace pieces by Satellite ID. If a piece ID used by one Satellite is reused by another Satellite, each Satellite can safely assume the shared piece ID refers to a different piece than the other Satellite, with different content and lifecycle.

4.8.6 Pointers

The data owner will need knowledge of how a *remote segment* is broken up and where in the network the *pieces* are located to recover it. This is contained in the *pointer* data structure.

A pointer includes: which nodes are storing the pieces, encryption information, erasure coding details, the repair threshold amount that determines how much redundancy a segment must lose before triggering a repair, the amount of pieces that must be stored to consider a repair to be successful, and other details. If the segment is an inline segment, the pointer contains the entire segment's binary data instead of which nodes store the pieces.

In our previous version [35], we used two data structures to keep track of the aforementioned kinds of information: *frames* and *pointers*. In this version, we have combined these data structures into a single data structure and elected to call the new combined data structure a *pointer*.

4.9 Metadata

The metadata storage system in the Storj network predominantly stores *pointers*. Other individual components of the Storj network communicate with the pointer database to store and retrieve pointers by path to perform actions.

The most trivial implementation for the metadata storage functionality we require will be to simply have each user use their preferred trusted database, such as MongoDB, MariaDB, Couchbase, PostgreSQL, SQLite [58], Cassandra [59], Spanner [60], or CockroachDB, to name a few. In many cases, this will be acceptable for specific users, provided those

users are managing appropriate backups of their metadata. Indeed, the types of users who have petabytes of data to store can most likely manage reliable backups of a single relational database storing only metadata.

On one hand, there are a few downsides to letting clients manage their metadata in a traditional database system, such as:

- **Availability** - The availability of the user's data is tied entirely to the availability of their metadata server. The counterpoint is that availability can be made arbitrarily good with existing trusted distributed solutions, such as Cassandra, Spanner, or CockroachDB, assuming an appropriate amount of effort is put into maintaining operations. Furthermore, any individual metadata service downtime does not affect the rest of the network. In fact, the network as a whole can never go down.
- **Durability** - If the metadata server suffers a catastrophic failure without backups, all of the user's data will be lost. This is already true with encryption keys, but a traditional database solution considerably increases the risk area from using encryption keys. Fortunately, the metadata itself can be periodically backed up into the Storj network. This in turn allows us to only keep track of the metadata of this metadata, further decreasing the amount of critical information that must be stored elsewhere.
- **Trust** - The user has to trust the metadata server.

On the other hand, there are a few upsides:

- **Control** - The user is in complete control of all of their data. There is no organizational single point of failure. The user is free to choose whatever metadata store with whatever trade-offs they prefer and can even run their own. Like Mastodon [61], this solution is still decentralized. Furthermore, in a catastrophic scenario, this design is no worse than most other technologies or techniques application developers frequently use (databases).
- **Simplicity** - Other projects have spent multiple years on shaky implementations of Byzantine-fault tolerant consensus metadata storage, with expected performance and complexity trade-offs (see appendix A). We can get a useful product to market without doing this work at all. This is a considerable advantage.
- **Coordination Avoidance** - Users only need to coordinate with other users on their Satellite. If a user has high throughput demands, they can set up their own Satellite and avoid coordination overhead from any other user. By allowing Satellite operators to select their own database, this will allow a user to choose a Satellite with weaker consistency semantics, such as Highly Available Transactions [15], that reduce coordination overhead within their own Satellite and increase performance even further.

Our launch goal is to allow customers to store their metadata in a database of their choosing. We expect and look forward to new systems and improvements specifically in this component of our framework.

Please see appendix A for more about why we've chosen to currently avoid trying to

solve the problem of Byzantine distributed consensus. See section 6.2 for a discussion of future plans.

4.10 Satellite

The collection of services that hold this metadata is called the *Satellite*. Users of the network will have accounts on a specific Satellite *instance*, which will: store their file metadata, manage authorization to data, keep track of storage node reliability, repair and maintain data when redundancy is reduced, and issue payments to storage nodes on the user's behalf. Notably, a specific Satellite instance does not necessarily constitute one server. A Satellite may be run as a collection of servers and be backed by a horizontally scalable trusted database for higher uptime.

Storj implements a thin-client model that delegates trust around managing files' location metadata to the Satellite service which manages data ownership. *Uplinks* are thus able to support the widest possible array of client applications, while Satellites require high uptime and potentially significant infrastructure, especially for an active set of files. Like storage nodes, the Satellite service has been developed and is released as open source software. Any individual or organization can run their own Satellite to facilitate network access.

The Satellite is, at its core, one of the most complex and yet straightforward components of our initial release that fulfills our framework. Notwithstanding future framework-conforming releases, the initial Satellite is a standard application server that wraps a trusted database, such as PostgreSQL, Cassandra, or whichever solution the metadata system chooses (section 4.9). Users sign in to a specific Satellite with account credentials. Data available through one Satellite instance is not available through another Satellite instance, though various levels of export and import are planned (section 6.2).

With respect to customer data, the Satellite is never given data unencrypted and does not hold encryption keys. The only knowledge of an object that the Satellite is able to share with third parties is its existence, rough size, and other metadata such as access patterns. This system protects the client's privacy and gives the client complete control over access to the data, while delegating the responsibility of keeping files available on the network to the Satellite.

Clients may use Satellites run by a third-party. Because Satellites store almost no data and have no access to keys, this is a large improvement over the traditional data-center model. Many of the features Satellites provide, like storage node selection and reputation, leverage considerable network effects. Reputation data sets grow more useful as they increase in size, indicating that there are strong economic incentives to share infrastructure and information in a Satellite.

Providers may choose to operate public Satellites as a service. Application developers then delegate trust regarding the location of their data on the network to a specific

Satellite, as they would to a traditional object store but to a lesser degree. Future updates will allow for various distributions of responsibilities, and thus levels of trust, between customer applications and Satellites.

The Satellite instance is made up of these components:

- A full node discovery cache (section 4.6)
- A per-object metadata database indexed by encrypted path (section 4.9)
- An account management and authorization system (section 4.12)
- A storage node reputation, statistics, and auditing system (section 4.13)
- A data repair service (section 4.14)
- A storage node payment service (section 4.16)

While our launch goal of many Satellites is a step ahead of our previous system's Bridge implementation [35], this is still just one point on our decentralization journey and we expect to continue to find ways to decentralize our components further.

4.11 Encryption

Our encryption choice is authenticated encryption, with support for both the AES-GCM cipher and the Salsa20 and Poly1305 combination NaCl calls "Secretbox" [62]. Authenticated encryption is used so that the user can know if anything has tampered with the data.

Data is encrypted in blocks that may be small batches of stripes, recommended to be 4KB or less [63]. While the same encryption key is used for every encryption batch in a segment, segments may have different encryption keys. However, the nonce for each encryption batch must be monotonically incrementing from the previous batch throughout the entire segment. The nonce wraps around to 0 if the counter reaches the maximum representable nonce. To prevent reordering attacks, the starting nonce of each segment is deterministically chosen based on the segment number. When multiple segments are uploaded in parallel, such as in the case of Amazon S3's multipart-upload feature, the starting nonce for each segment can be calculated from the starting nonce of the file and the segment number. This scheme protects the content of the data from the storage node housing the data. The data owner retains complete control over the encryption key, and thus over access to the data.

Paths are also encrypted. Like BIP32 [44], the encryption is hierarchical and deterministic, and each path component is encrypted separately. To explain how we do this, we start with a scheme for determining a secret value for each path component. Let's say a given path $\boldsymbol{p}$ has unencrypted path components $p_1, p_2, \ldots, p_n$ and we want to determine an encrypted path $\boldsymbol{e}$ with path components $e_1, e_2, \ldots, e_n$. We assume a predetermined root secret, s_0. This root secret is chosen by the user and, like all other encryption secrets, never leaves the client computer. We recursively define $s_i = \text{HMAC}(s_{i-1}, p_i)$. A key $K(s_i)$ can be deterministically generated from s_i. We then define the encrypted path component

$e_i = \text{enc}(K(s_{i-1}), p_i)$, such that the new path $\boldsymbol{e}$ is $e_1, e_2, \ldots, e_n$. HMAC-SHA256 or HMAC-SHA512 are used for key derivations.

This construction allows a client to share access to some subtree of the path without access to its parents or other paths of the same depth. For example, suppose a client would like to share access to all paths with the same prefix p_1, p_2, p_3 with another client. The client would give the other client e_1, e_2, e_3 and s_3. This allows the client to decrypt and access any arbitrary e_4, as $K(s_3)$ is known to them, without allowing the client to decrypt e_3 or earlier. More generally in this case, the client could decrypt and access any arbitrary e_i, if and only if $i > 3$.

Path encryption is enabled by default but is otherwise optional, as encrypted paths make efficient sorted path listing challenging. When path encryption is in use (a per-bucket feature), objects are sorted by their encrypted path name, which is deterministic but otherwise relatively unhelpful when the client application is interested in sorted, unencrypted paths. For this reason, users can opt out of path encryption. When path encryption is disabled, unencrypted paths are only revealed to the user's chosen Satellite, but not to the storage nodes. Storage nodes continue to have no information about the path and metadata of the pieces they store.

4.12 Authorization

Encryption protects the privacy of data while allowing for the *identification* of tampering, but authorization allows for the *prevention* of tampering by disallowing clients from making unauthorized edits. Users who are authorized will be able to add, remove, and edit files, while users who are not authorized will not have those abilities. Metadata operations will be authorized. Users will authenticate with their Satellite, which will allow them access to various operations according to their authorization configuration.

Our initial metadata authorization scheme uses macaroons [64]. Macaroons are a type of bearer token that authorizes the bearer to some restricted resources. Macaroons are especially interesting in that they allow for rich contextual decentralized delegation. In other words, they provide the property that anyone can add restrictions in a way in which those restrictions cannot later be removed, without coordination with a central party.

We use macaroons to restrict which operations can be applied and to which encrypted paths they can be applied. In this way, macaroons provide a mechanism to restrict delegated access to specific encrypted path prefixes, specific files, and specific operations, such as read only access or perhaps append only access. Each account has a root macaroon and operations are validated against a supplied macaroon's set of caveats. Our macaroons are further caveated with optional expirations and revocation tokens, which allow users to revoke macaroons programmatically.

Because we want to restrict Satellite operations, and Satellites only have access to encrypted paths, our authorization scheme must work on encrypted paths. For access del-

egation to specific path prefixes, path separation boundaries between path components must remain across encryption. This implies reduced functionality and/or performance for path delimiters other than a forward slash.

Once the Uplink is authorized with the Satellite, the Satellite will approve and sign for operations to storage nodes, including bandwidth allocations (section 4.17). The Uplink must retrieve valid signatures from the Satellite prior to operations with storage nodes. All operations on a storage node require a specific Satellite ID and associated signature. A storage node will reject operations not signed by the appropriate Satellite ID. Storage nodes will not allow operations signed by one Satellite to apply to objects owned by another, unless explicitly granted by the owning Satellite.

Our initial implementation does not detect or attempt to mitigate unexpected file removal or rollback by a misbehaving Satellite. Our trust model expects that a user's Satellite is well-behaved and stores and repairs data reliably. If a Satellite cannot be trusted, it is unlikely to repair data on a client's behalf anyway. However, a future implementation could add more thorough detection for Satellite-based file system tampering, via a scheme as in systems such as SUNDR, SiRiUS, or Plutus [65–67].

4.13 Audits

In a network with untrusted nodes, validating that those nodes are returning data accurately and otherwise behaving as expected is vital to ensuring a properly functioning system. Audits are a way to confirm that nodes have the data they claim to have. Auditors, such as Satellites, will send a *challenge* to a storage node and expect a valid response. A challenge is a request to the storage node in order to prove it has the expected data.

Some distributed storage systems, including the previous version of Storj [35], discuss *Merkle tree proofs*, in which audit challenges and expected responses are generated at the time of storage as a form of *proof of retrievability* [45]. By using a Merkle tree [68], the amount of metadata needed to store these challenges and responses is negligible.

Proofs of retrievability can be broadly classified into *limited* and *unlimited* schemes [47]. The Merkle tree variety used in our previous version is one such limited scheme. Unfortunately, in such a scheme, the challenges and expected responses must be pregenerated. As we learned with our previous version, without a periodic regeneration of these challenges, a storage node can begin to pass most audits without storing all of the requested data by keeping track of which challenges exist and then saving only the expected responses. During our previous version, we began to consider Reed-Solomon erasure coding to help us solve this problem.

An assumption in our storage system is that most storage nodes behave rationally, and incentives are aligned such that most data is stored faithfully. As long as that assumption holds, Reed-Solomon is able to detect errors and even correct them, via mechanisms such as the Berlekamp-Welch error correction algorithm [37, 69]. We are already using

Reed-Solomon erasure coding [54] on small ranges (stripes), so as discussed in the HAIL system [39], we use erasure coding to read a single stripe at a time as a challenge and then validate the erasure share responses. This allows us to run arbitrary audits without pre-generated challenges.

To perform an audit, we first choose a stripe. We request that stripe's erasure shares from all storage nodes responsible. We then run the Berlekamp-Welch algorithm [37,69] across all the erasure shares. When enough storage nodes return correct information, any faulty or missing responses can easily be identified.

Given a specific storage node, an audit might reveal that it is offline or incorrect, or unavailable. If a node is offline, it is simply marked as offline. A node is moved into what we call "containment mode" if it responds to the initial audit service's dial but fails to send the requested data, perhaps because it's busy fulfilling other requests and doesn't respond to the audit in time. In this mode, the Satellite will calculate and save the expected response, then continue to try the same audit with that node until the node either responds successfully, actively fails the audit, or is disqualified for being offline too long. Once the node responds successfully, it leaves containment mode.

All audit failures will be stored and saved in the reputation system. Audits additionally serve as opportunity to test storage node latency, throughput, responsiveness, and uptime. This data will also be saved in the reputation system.

It is important that every storage node has a frequent set of random audits to gain statistical power on how well-behaved that storage node is operating. However, as discussed in section 3.7, it is not a requirement that audits are performed on every byte, or even on every file. Additionally, it is important that every byte stored in the system has an equal probability of being checked for a future audit to every other byte in the system. See section 7.2 for a discussion on how many audits are required to be confident data is stored correctly.

4.14 Data repair

As storage nodes go offline—taking their pieces with them—it will be necessary for the missing pieces to be rebuilt once each segment's pieces fall below the predetermined threshold, *m*. If a node goes offline, the Satellite will mark that nodes' file pieces as missing.

The node discovery system's caches have reasonably accurate and up-to-date information about which storage nodes have been online recently. When a storage node changes state from recently online to offline, this can trigger a lookup in a reverse index within a user's metadata database, identifying all segment pointers that were stored on that node.

For every segment that drops below the appropriate minimum safety threshold, *m*,

the segment will be downloaded and reconstructed, and the missing pieces will be regenerated and uploaded to new nodes. Finally, the *pointer* will be updated to include the new information.

Users will choose their desired durability with their Satellite which may impact price and other considerations. This desired durability (along with statistics from ongoing audits) will directly inform what Reed-Solomon erasure code choices will be made for new and repaired files, and what thresholds will be set for when uploads are successful and when repair is needed. See sections 3.4 and 7.3 for how we calculate these values given user inputs.

A direct implication of this design is that, for now, the Satellite must constantly stay running. If the user's Satellite stops running, repairs will stop, and data will eventually disappear from the network due to node churn. This is similar to the design of how value storing and republishing works in Kademlia [8], which requires the owner to stay online.

The *ingress* (or inbound) bandwidth demands of the audit and repair system are large, but given standard configuration, the *egress* (or outbound) demands are relatively small. A large amount of data comes into the system for audits and repairs, but only the formerly missing pieces are sent back out. While the repair and audit system can run anywhere, the bandwidth usage asymmetry means that hosting providers which offer free ingress make for an especially attractive hosting location for users of this system.

4.14.1 Piece hashes

Data repair is an ongoing, costly operation that will use significant bandwidth, memory, and processing power, often impacting a single operator. As a result, repair resource usage should be aggressively minimized as much as possible.

For repairing a segment to be effective at minimizing bandwidth usage, only as few pieces as needed for reconstruction should be downloaded. Unfortunately, Reed-Solomon is insufficient on its own for correcting errors when only a few redundant pieces are provided. Instead, piece hashes provide a better way to be confident that we're repairing the data correctly.

To solve this problem, hashes of every piece will be stored alongside each piece on each storage node. A validation hash that the set of hashes is correct will be stored in the pointer. During repair, the hashes of every piece can be retrieved and validated for correctness against the pointer, thus allowing each piece to be validated in its entirety. This allows the repair system to correctly assess whether or not repair has been completed successfully without using extra redundancy for the same task.

4.15 Storage node reputation

Reputation metrics on decentralized networks are a critical part of enabling cooperation between nodes where progress would be challenging otherwise. Reputation metrics are used to ensure that bad actors within the network are eliminated as participants, improving security, reliability, and durability.

Storage node reputation can be divided into four subsystems. The first subsystem is a proof of work identity system, the second subsystem is the initial vetting process, the third subsystem is a filtering system, and finally, the fourth system is a preference system.

The goal of the first system is to require a short proof that the storage node operator is invested, through time, stake, or resources. Initially, we are using proof of work. As mentioned in section 4.3, storage nodes require a proof of work as part of identity generation. This helps the network avoid some Sybil attacks [70], but we glossed over how proof of work difficulty is set. We will let Satellite operators set per-Satellite minimum difficulty required for new data storage. If a storage node has an identity generated with a lower difficulty than the Satellite's configured minimum, that storage node will not be a candidate for new data. We expect Satellite operators to naturally increase the minimum proof of work difficulty requirements over time until a reasonable balance is found. In the case of a changing difficulty configuration, Satellites will leave existing data on existing nodes where possible. Other investment proof schemes are possible, such as a form of proof of stake as we proposed in our previous work [71].

The second subsystem slowly allows nodes to join the network. When a storage node first joins the network, its reliability is unknown. As a result, it will be placed into a vetting process until enough data is known about it. We propose the following way to gather data about new nodes without compromising the integrity of the network. Every time a file is uploaded, the Satellite will select a small number of additional unvetted storage nodes to include in the list of target nodes. The Reed-Solomon parameters will be chosen such that these unvetted storage nodes will not affect the durability of the file, but will allow the network to test the node with a small fraction of data until we are sure the node is reliable. After the storage node has successfully stored enough data for a long enough period (at least one payment period), the Satellite will then start including that storage node in the standard selection process used for general uploads. It will also give the node a signed message claiming that the vetting process is completed. Importantly, storage nodes get paid during this vetting period, but don't receive as much data.

The filtering system is the third subsystem; it blocks bad storage nodes from participating. In addition to simply not having done a sufficient proof of work, certain actions a storage node can take are disqualifying events. The reputation system will be used to filter these nodes out from future uploads, regardless of where the node is in the vetting process. Actions that are disqualifying include: failing too many audits; failing to return data, with reasonable speed; and failing too many uptime checks.

If a storage node is disqualified, that node will no longer be selected for future data

storage and the data that node stores will be moved to new storage nodes. Likewise, if a client attempts to download a piece from a storage node that the node should have stored and the node fails to return it, the node will be disqualified. Importantly, storage nodes will be allowed to reject and fail *put* operations without penalty, as nodes will be allowed to choose which Satellite operators to work with and which data to store.

It's worth reiterating that failing too many uptime checks is a disqualifying event. Storage nodes can be taken down for maintenance, but if a storage node is offline too much, it can have an adverse impact on the network. If a node is offline during an audit, that specific audit should be retried until the node responds successfully or is disqualified, to prevent nodes from selectively failing to respond to audits.

After a storage node is disqualified, the node must go back through the entire vetting process again. If the node decides to start over with a brand-new identity, the node must restart the vetting process from the beginning (in addition to generating a new node ID via the proof of work system). This strongly disincentivizes storage nodes from being cavalier with their reputation.

The last subsystem is a preference system. After disqualified storage nodes have been filtered out, remaining statistics collected during audits will be used to establish a preference for better storage nodes during uploads. These statistics include performance characteristics such as throughput and latency, history of reliability and uptime, geographic location, and other desirable qualities. They will be combined into a load-balancing selection process, such that all uploads are sent to qualified nodes, with a higher likelihood of uploads to preferred nodes, but with a non-zero chance for any qualified node. Initially, we'll be load balancing with these preferences via a randomized scheme, such as the Power of Two Choices [72], which selects two options entirely at random, and then chooses the more qualified between those two.

On the Storj network, preferential storage node reputation is only used to select where new data will be stored, both during repair and during the upload of new files, unlike disqualifying events. If a storage node's preferential reputation decreases, its file pieces will not be moved or repaired to other nodes.

There is no process planned in our system for storage nodes to contest their reputation scores. It is in the best interest of storage nodes to have good uptime, pass audits, and return data. Storage nodes that don't do these things are not useful to the network. Storage nodes that are treated by Satellites unfairly will not accept future data from those Satellites.

Initially, storage node reputation will be individually determined by each Satellite. If a node is disqualified by one Satellite, it may still store data for other Satellites. Reputation will not initially be shared between Satellites. Over time, reputation will be determined globally.

4.16 Payments

In the Storj network, payments are made by clients who store data on the platform to the Satellite they utilize. The Satellites then pays storage nodes for the amount of storage and bandwidth they provide on the network. Payments by clients may be through any mechanism (STORJ, credit card, invoice, etc.), but payments to storage nodes are via the Ethereum-based ERC20 [73] STORJ token.

Previous distributed systems have handled payments as hard-coded contracts. For example, the previous Storj network utilized 90-day contracts to maintain data on the network. After that period of time, the file was deleted. Other distributed storage platforms use 15-day renewable contracts that delete data if the user does not login every 15 days. Others use 30-day contracts. We believe that the most common use case is indefinite storage. To best solve this use case, our network will no longer use contracts to manage payments and file storage durations. The default assumption is that data will last indefinitely.

Satellites will pay storage nodes for the data they store and for piece downloads. Storage nodes will not be paid for the initial transfer of data, but they will be paid for storing the data month-by-month. At the end of the payment period, a Satellite will calculate earnings for each of its storage nodes. Provided the storage node hasn't been disqualified, the storage node will be paid by the Satellite for the data it has stored over the course of the month, per the Satellite's records.

Satellites have a strong incentive to prefer long-lived storage nodes. If storage node churn is too high, Satellites will hold back a portion of a storage node's payment until the storage node has maintained good participation and uptime for some minimum amount of time, on the order of greater than half a year. If a storage node leaves the network prematurely, the Satellite will reclaim held payments to it.

If a storage node misses a delete command due to the node being offline, it will be storing more data than the Satellite credits it for. Storage nodes are not paid for storing such file pieces, but will eventually be cleaned up through the garbage collection process (see section 4.19). This means that storage nodes who maintain higher availability can maximize their profits by deleting files on request, which minimizes the amount of garbage data they store.

The Satellite maintains a database of all file pieces it is responsible for and the storage nodes it believes are storing these pieces. Each day, the Satellite adds another day's worth of accounting to each storage node for every file piece it will be storing. Satellites will track utilized bandwidth (see section 4.17). At the end of the month, each Satellite adds up all bandwidth and storage payments each storage node has earned and makes the payments to the appropriate storage nodes.

Satellites will also earn revenue from account holders for executing audits, repairing segments, and storing metadata. Satellites charge a per-segment and per-byte cost, in addition to charging for access and retrieval. Per-segment charges cover the cost of

pointer metadata, whereas per-byte charges cover the cost of data maintenance on the network. Every day, each Satellite will execute a number of audits across all of its storage nodes on the network. The Satellite will charge for both completing audits and repairs, once segments fall below the piece threshold needed for repair.

When it is detected that a storage node acts maliciously and does not store files properly or maintain sufficient availability, it will not be paid for the services rendered, and the funds allocated to it will instead be used to repair any missing file pieces and to pay new storage nodes for storing the data.

To reduce transaction fees and other overhead as much as possible, payments must be worth at least some minimum value. Certain Satellites may elect to use a portion of the storage nodes' payout to cover transaction fees in part or in whole.

See the Satellite reputation section (4.18) for details on how storage nodes will be able to choose which Satellites they trust.

4.17 Bandwidth allocation

A core component of our system requires knowing how much bandwidth is used between two peers.

In our previous version [35,74], we used *exchange reports* to gather information about what transpired between two peers. At the end of an operation, both peers would send reports to a central collection service for settlement. When both peers mutually agreed, it was straightforward to determine how much bandwidth had been used. When they disagreed, however, we resorted to data analysis and regression to determine which peer had a greater propensity for dishonesty in an effort to catch "cheaters" (or, self-rational nodes). With our new version, we want to make cheating impossible from the protocol level.

To solve this problem, we turn to Neuman's *Proxy-based authorization and accounting for distributed systems* [75]. This accounting protocol more correctly measures resource usage in a delegated and decentralized way.

In Neuman's accounting protocol, if an account holder has enough funds to cover the operation, an account server will create a signed, digital check and transfer it to the account holder. The protocol refers to this check as a *proxy*, but we refer to it as a *bandwidth allocation* in this paper and as an *order limit* in our code. This check contains information identifying the account server, the payer, the payee, the maximum amount of resources available to be used in the operation, a check number to prevent any double spending problems [76], and an expiration date.

In our case, the account server is the Satellite, the payer is the Uplink, the payee is the storage node, and the resource in question is bandwidth. The Satellite will only create a bandwidth allocation if the Uplink is authorized for the request. At the beginning of a

storage operation, the Uplink can transfer bandwidth allocation to a storage node. The storage node can validate the Satellite's signature and perform the requested operation up to the allowed bandwidth limit, storing and later sending the bandwidth allocation to the Satellite for payment.

We're further inspired by Filecoin's off-chain retrieval market, wherein only small amounts of data are transferred at a time [77]. Instead of allowing the storage node to cheat and save the bandwidth allocation without performing the requested operation, we break each operation into smaller requests such that if either the storage node or Uplink stop participating in the protocol prematurely, neither peer class is exposed to too much loss. This is similar to an optimistic, gradual-release, fair-exchange protocol [76].

To support this with Neuman's accounting protocol and little Satellite overhead, we use *restricted bandwidth allocations* (referred to as *restricted proxies* in [75] and referred to as *orders* in our codebase, contrasted with the Satellite's *order limits*). Neuman's restricted proxies work much like Macaroons [64] in which further caveats can be added in a way that can't be removed, limiting the capabilities of the proxy. Proxies can use public/private key cryptography, which means that anyone can validate the proxy, instead of just the original issuer. Because each Uplink already has a key pair as part of its identity (section 4.4), we use the existing key pair instead of creating a new key pair for every restriction.

Restricted bandwidth allocations, in our case, are restricted by the Uplink to limit the bandwidth allocation's value to only what has transferred so far. In this way, the storage node will only keep the largest bandwidth allocation it has received up to that point, and the Uplink will only send bandwidth allocations that are slightly larger than what it has received. The storage node has no incentive to keep more than the largest allocation, as they all share the same "check number," which can only be cashed once.

In the case of a *Get* operation, assume the Satellite-signed bandwidth allocation allows up to x bytes total. The Uplink will start by sending a restricted allocation for some small amount (y bytes), perhaps only a few kilobytes, so the storage node can verify the Uplink's authorization. If the allocation is signed correctly, the storage node will transfer up to the amount listed in the restricted allocation (y bytes) before awaiting another allocation. The Uplink will then send another allocation where y is larger, continuing to send allocations for data until y has grown to the full x value. For each transaction, the storage node only sends previously-unsent data, so that the storage node only sends x bytes total. As seen in Figure 4.6, we pipeline these requests to avoid pipeline stall performance penalties.

If the request is terminated at any time, either planned or unexpectedly, the storage node will keep the largest restricted bandwidth allocation it has received. This largest restricted bandwidth allocation is the signed confirmation by the Uplink that the Uplink agreed to bandwidth usage of up to y bytes, along with the Satellite's confirmation of the Uplink's bandwidth allowance x. The storage node will periodically send the largest

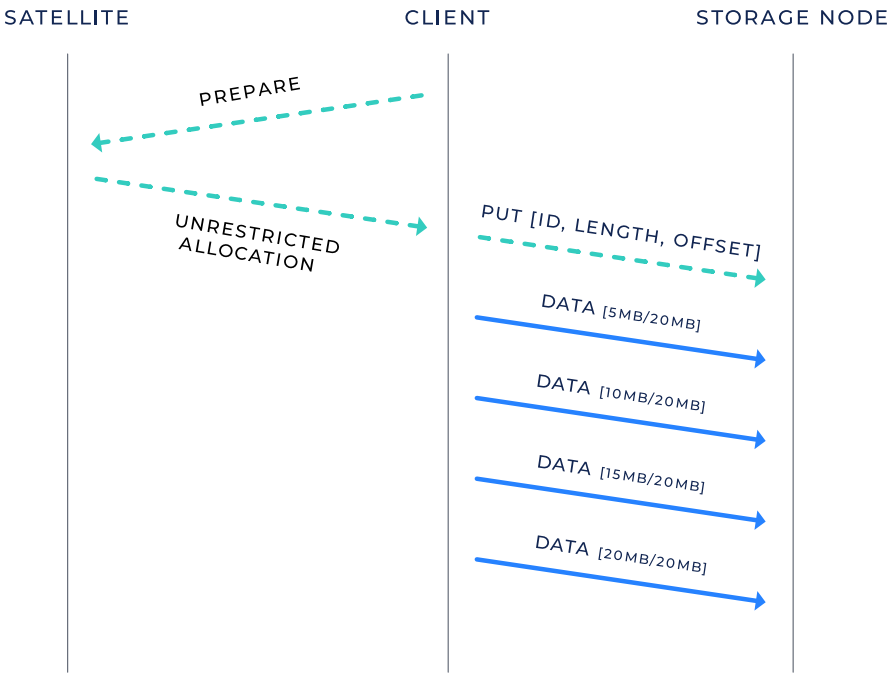


Figure 4.5: Diagram of a put operation

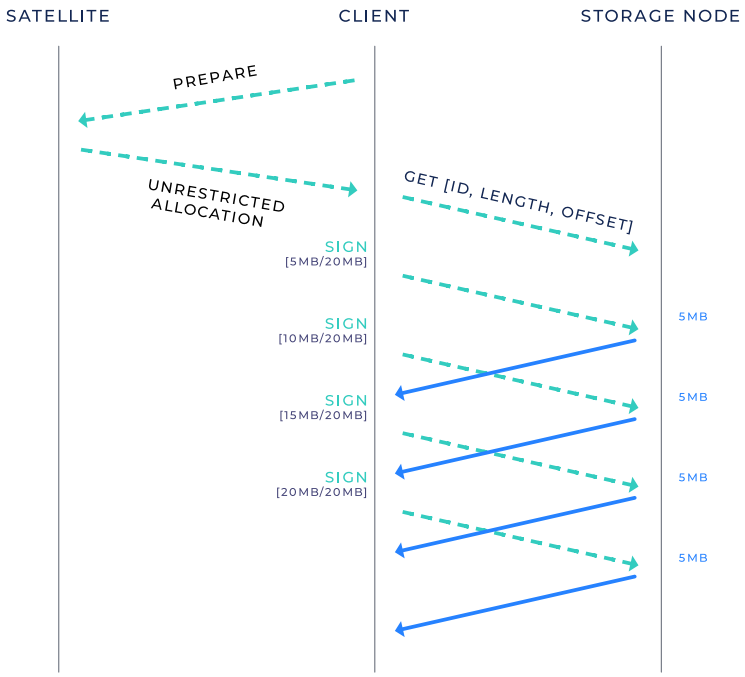


Figure 4.6: Diagram of a get operation

restricted bandwidth allocations it has received to appropriate Satellites, at which point Satellites will add to the overall owed totals for later payment.

If the Uplink can't afford the bandwidth usage, the Satellite will not sign an bandwidth allocation, protecting the Satellite's reputation. Likewise, if the Uplink tries to use more bandwidth than allocated, the storage node will decline the request. The storage node can only get paid for the maximum amount a client has agreed to, as it otherwise has no valid bandwidth allocations to return for payment.

As before, we don't measure all peer-to-peer traffic. This bandwidth traffic measurement system only tracks bandwidth used during storage operations (storage and retrievals of pieces). However, it does not apply to node discovery traffic or other generic maintenance overhead.

4.18 Satellite reputation

Whenever a Satellite on the Storj network has a less than stellar payment, demand generation, or performance history, there is a strong incentive for the storage nodes to avoid accepting its data.

Storage nodes can select which Satellites they want to work with and can remove those that they don't. If a Satellite misbehaves, storage nodes can indicate they no longer trust that Satellite.

Storage node operators can elect to automatically trust a Storj Labs provided collection of recommended Satellites that adhere to a strict set of quality controls and payment service level agreements (SLAs). To protect storage node operators, if a Satellite operator wants to be included in the Storj-provided approved list, the Satellite operator may be required to adhere to a set of operating, payment, and pricing parameters and to sign a business arrangement with Storj Labs.

4.19 Garbage collection

When clients move, replace, or delete data, Satellites, or clients on behalf of Satellites, will notify storage nodes that they are no longer required to store that data. In configurations where delete messages are issued by the client, the metadata system will require proof that deletes were issued to a configurable minimum number of storage nodes. In such a configuration, every time data is deleted, storage nodes that are online and reachable will receive notifications right away.

Storage nodes will sometimes be temporarily unavailable and will miss delete messages. In these cases, and in configurations where delete messages are not issued by the client, unneeded data is considered *garbage*. Satellites only pay for data that they expect to be stored. Storage nodes with lots of garbage will earn less than they otherwise would

unless a garbage collection system is employed. For this reason, we introduce garbage collection to free up space on storage nodes.

A garbage collection algorithm is a method for freeing no-longer used resources. A *precise* garbage collector collects all garbage exactly and leaves no additional garbage. A *conservative* garbage collector, on the other hand, may leave some small proportion of garbage around given some other trade-offs, often with the aim of improving performance. As long as a conservative garbage collector is used in our system, the payment for storage owed to a storage node will be high enough to amortize the cost of storing the garbage.

For the nodes that miss initial delete messages if they are sent, our first release will start with a conservative garbage collection strategy, though we anticipate a precise strategy in the future. Periodically, storage nodes will request a data structure to detect differences. In the simplest form, it can be a hash of stored keys, which allows efficient detection of out-of-sync state. After detecting out-of-sync state, collection can use another structure, such as a Bloom filter [50], to find out what data has not been deleted. By returning a data structure tailored to each node on a periodic schedule, a Satellite can give a storage node the ability to clean up garbage data to a configurable tolerance. Satellites will reject overly frequent requests for these data structures.

4.20 Uplink

Uplink is the term which we use to identify any software or service that invokes *libuplink* in order to interact with Satellites and storage nodes. It comes in a few forms:

Libuplink - *libuplink* is a library that provides access to storing and retrieving data in the Storj network.

Gateways - Gateways act as compatibility layers between a service or application and the Storj network. They run as a service co-located with wherever data is generated, and will communicate directly with storage nodes so as to avoid central bandwidth costs. The Gateway is a simple service layer on top of *libuplink*.
Our first gateway is an Amazon S3 gateway. It provides an S3-compatible, drop-in interface for users and applications that need to store data but don't want to bother with the complexities of distributed storage directly.

Uplink CLI - The Uplink CLI is a command line application which invokes *libuplink*, allowing its user to upload and download files, create and remove buckets, manage file permissions, and other related tasks. It aims to provide an experience familiar to what you might expect when using Linux/UNIX tools such as *scp* or *rsync*.

Like storage nodes and Satellites, the Uplink software in all three forms has been developed and released as open source software.

5. Walkthroughs

The following is a collection of common use case examples of different types of transactions of data through the system.

5.1 Upload

When a user wants to upload a file, the user first begins transferring data to an instance of the Uplink.

- The Uplink chooses an encryption key and starting nonce for the first segment and begins encrypting incoming data with authenticated encryption as it flows to the network.
- The Uplink buffers data until it knows whether the incoming segment is short enough to be an inline segment or a remote segment. Inline segments are small enough to be stored on the Satellite itself.

The rest of this walkthrough will assume a remote segment because remote segments involve the full technology stack.

- The Uplink sends a request to the Satellite to prepare for the storage of this first segment. The request object contains API credentials, such as macaroons, and identity certificates.

Upon receiving the request, the Satellite will:

- Confirm that the Uplink has appropriate authorization and funds for the request. The Uplink must have an account with this specific Satellite already.
- Make a selection of nodes with adequate resources that conform to the bucket's configured durability, performance, geographic, and reputation requirements.
- Return a list of nodes, along with their contact information and unrestricted bandwidth allocations (section 4.17), and a chosen root piece ID.

Next, the Uplink will take this information and begin parallel connections to all of the chosen storage nodes while measuring bandwidth (section 4.17).

- The Uplink will begin breaking the segment into stripes and then erasure encode each stripe.
- The generated erasure shares will be concatenated into *pieces* as they transfer to each storage node in parallel.
- The erasure encoding will be configured to over-encode to more pieces than needed. This will eliminate the long tail effect and lead to a significant improvement of visible performance by allowing the Uplink to cancel the slowest uploads.

- The data will continue to transfer until the maximum segment size is hit or the stream ends, whichever is sooner.
- All of the hashes of every piece will be written to the end of each piece stream.

After that, the storage node will store: the largest restricted bandwidth allocation (section 4.17); the TTL of the segment, if one exists; and the data itself. The data will be identified by the storage node-specific piece ID and the delegating Satellite ID.

If the upload is aborted for any reason, the storage node will keep the largest restricted bandwidth allocation it received from the client Uplink on behalf of the Satellite, but will throw away all other relevant request data.

Assuming success:

- The Uplink encrypts the random encryption key it chose for this file, utilizing a deterministic hierarchical key.
- The Uplink will upload a *pointer* object back to the Satellite, which contains the following information:
  - which storage nodes were ultimately successful
  - what encrypted path was chosen for this segment
  - which erasure code algorithm was used
  - the chosen piece ID
  - the encrypted encryption key and other metadata
  - the hash of the piece hashes
  - a signature

Finally, the Uplink will then proceed with the next segment, continuing to process segments until the entire stream has completed. Each segment gets a new encryption key, but the segment's starting nonce monotonically increases from the previous segment.

Once the last segment stored, the Uplink will send additional metadata:

- how many segments the stream contained
- how large the segments are, in bytes
- the starting nonce of the first segment
- any other object metadata

Periodically, the storage nodes will later send the largest restricted bandwidth allocation (section 4.17) they received as part of the upload to the appropriate Satellite for payment.

If an upload happens via the Amazon S3 multipart-upload interface, each *part* is uploaded as a segment individually.

5.2 Download

When a user wants to download an object, first the user sends a request for data to the Uplink. The Uplink then tries to reduce the number of round trips to the Satellite by speculatively requesting metadata about the object along with the first segment's pointer. If the Uplink knows about the requested byte range, the Uplink may just tell the Satellite which byte ranges are needed and the Satellite can respond with the appropriate segment pointers.

For every segment pointer requested, the Satellite will:

- Validate that the Uplink has access to download the segment pointer and has enough funds to pay for the download.
- Generate an unrestricted bandwidth allocation (section 4.17) for each piece that makes up the segment.
- Look up the contact information for the storage nodes listed in the pointer.
- Return the requested segment pointer, the bandwidth allocations, and node contact info for each piece.

The Uplink will determine whether more segments are necessary for the data request it received, and will request the remaining segment pointers if needed.

- Once all necessary segment pointers have been returned, if the requested segments are not inline, the Uplink will initiate parallel requests while measuring bandwidth (section 4.17) to all appropriate storage nodes for the appropriate erasure share ranges inside of each stored piece.
- Because not all erasure shares are necessary for recovery, long tails will be eliminated and a significant and visible performance improvement will be gained by allowing the Uplink to cancel the slowest downloads.
- The Uplink will combine the retrieved erasure shares into stripes and decrypt the data.

If the download is aborted for any reason, each storage node will keep the largest restricted bandwidth allocation (section 4.17) it received, but it will throw away all other relevant request data. Either way, the storage nodes will later send the largest restricted bandwidth allocation they received as part of the download to the appropriate Satellite for later payment.

5.3 Delete

When a user wants to delete a file, the delete operation is first received by the Uplink. The Uplink then requests all of the segment pointers for the file.

If the client and Satellite are configured to issue direct deletes to nodes, for every segment pointer, the Satellite will:

- Validate that the Uplink has access to delete the segment pointer.
- Generate a signed agreement for the deletion of the segment, so the storage node knows the Satellite is expecting the delete to proceed.
- Look up the contact information for the storage nodes listed in the pointer.
- Return the segments, the agreements, and contact information.

For all of the remote segments, the Uplink will initiate parallel requests to all appropriate storage nodes to signal that the pieces are being removed.

- The storage nodes will return a signed message indicating either that the storage node received the delete operation and will delete both the file and its bookkeeping information or that it was already removed.
- The Uplink will upload all of the signed messages that it received from working storage nodes back to the Satellite. The Satellite will require an adjustable percent of the total storage nodes to successfully sign messages to ensure that the Uplink did its part in notifying the storage nodes that the object was deleted.
- The Satellite will remove the segment pointers and stop charging the customer and stop paying the storage nodes for them.
- The Uplink will return a success status.

Some Satellite and Uplink configurations may elect to simply delete the metadata directly and let the garbage collection system recover the free space on storage nodes.

Periodically, storage nodes will ask the Satellite for generated garbage collection messages that will update storage nodes who were offline during the main deletion event. Satellites will reject requests for garbage collection messages that happen too frequently. See section 4.19 for more details.

5.4 Move

When a user wants to move a file, first, the Uplink receives a request for moving the file to a new path. Then, the Uplink requests all of the segment pointers of that file.

For every segment pointer, the Satellite:

- Validates that the Uplink has access to download it.
- Returns the requested segment metadata.

For every segment pointer, the Uplink:

- Decrypts the metadata with an encryption key derived from the path.
- Calculates the path at the new destination.
- Re-encrypts the metadata with a new encryption key derived from the new path.

The Uplink requests that the Satellite add all modified segment pointers and remove all old segment pointers in an atomic compare-and-swap operation.

The Satellite will validate that:

- The Uplink has appropriate authorization to remove the old path and create the new path.
- The content of the old path hasn't changed since the overall operation started.

If the validation is successful, the Satellite will perform the operation. No storage node will receive any request related to the file move.

Because of the complexity around atomic pointer batch modifications, efficient move operations may not be implemented in the first release of this network.

5.5 List

When a user wants to list files:

- First, a request for listing a page of objects is received by the Uplink.
- Then, the Uplink will translate the request on unencrypted paths to encrypted paths.
- Next, the Uplink will request from the Satellite the appropriate page of encrypted paths.
- After that, the Satellite will validate that the Uplink has appropriate access and then return the requested list page.
- Finally, the Uplink will decrypt the results and return them.

5.6 Audit

Each Satellite has a queue of segment stripes that will be audited across a set of storage nodes. This queue is filled when the Satellite chooses a stripe to audit by identifying storage nodes that have had fewer recent audits than other storage nodes. The Satellite will select a stripe at random from the data contained by that storage node. Because the audit must also retrieve data from the other nodes in the erasure coded set, many other nodes are audited as well. Because segments have a maximum size, this also sufficiently approximates our goal of choosing a byte to audit uniformly at random.

Satellites will then work to process the queue and report errors.

- For each stripe request, the Satellite will perform the entire download operation for that small stripe range, filtering out nodes that are in containment mode. Unlike standard downloads, the stripe request does not need to be performant. The Satellite will attempt to download all of the erasure shares for the stripe and will wait for slow storage nodes.
- After receiving as many shares as possible within a generous timeout, the erasure shares will be analyzed to discover which, if any, are wrong. Satellites will take note of storage nodes that return invalid data, and if a storage node returns too much invalid

data, the Satellite will disqualify the storage node from future exchanges. In the case of a disqualification, the Satellite will not pay the storage node going forward, and it will not select the storage node for new data.
- For storage nodes that did not respond, a cryptographic checksum of the expected audit result will be created and stored, placing the unresponsive nodes in containment. While in containment, a node will continue to be given only the audit it was unresponsive for until it passes or is disqualified.
- For storage nodes that do respond but the response is judged to be incorrect, the full piece will be requested from the storage node and then compared with the expected hash of that piece 4.14.1. This allows the Satellite to differentiate between dishonest storage nodes and dishonest Uplinks.

5.7 Data repair

The repair process has two parts. The first part detects unhealthy files, and the second part repairs them. Detection is straightforward.

- Each Satellite will periodically ping every storage node it knows about, either as part of the audit process or via standard node discovery ping operations.
- The Satellite will keep track of nodes that fail to respond and mark them as down.
- When a node is marked down or is marked bad via the audit process, the pointers that point to that storage node will be considered for repair. Pointers keep track of their minimum allowable redundancy. If a pointer is not stored on enough good, online storage nodes, it will be added to the repair queue.

A worker process will take segment pointers off the repair queue. When a segment pointer is taken off the repair queue:

- The worker will download enough pieces to reconstruct the entire segment, along with the piece hashes stored with those pieces (see section 4.14.1). Unlike audits, only enough pieces for accurate repair are needed. Unlike streaming downloads, the repair system can wait for the entire segment before starting.
- The piece hashes are validated against the signature in the pointer, and then the downloaded pieces are validated against the validated piece hashes. Incorrect pieces are thrown away and count against the source as failed audits.
- Once enough correct pieces are recovered, the missing pieces are regenerated.
- The Satellite selects some new nodes and uploads the new pieces to those new nodes via the normal upload process.
- The Satellite updates the pointer's metadata.

5.8 Payment

The payment process works as follows:

- First, the Satellite will choose a rollup period. This is a period of time—defaulting to a day—that payment for data at rest is calculated. This is purely a period chosen for accounting; actual payments will happen on a less frequent schedule.
- During each roll-up period, a Satellite will consider all of the files it believes are currently stored on each storage node. Satellites will keep track of payments owed to each storage node for each rollup period, based on the data kept on each storage node.
- Storage nodes will periodically send in bandwidth allocation reports (section 4.17).
- Finally, when Satellite are ready to pay storage nodes, the Satellite will calculate the owed funds due to bandwidth allocation along with the outstanding data at rest calculations. It then sends the funds to the storage node's requested wallet address.

6. Future work

Storj is a work in progress, and many features are planned for future versions. In this chapter, we discuss a few especially interesting areas in which we want to consider improvements to our concrete implementation.

6.1 Hot files and content delivery

Occasionally, users of our system may end up delivering files that are more popular than anticipated. While storage node operators might welcome the opportunity to be paid for more bandwidth usage for the data they already have, demand for these popular files might outstrip available bandwidth capacity, and a form of dynamic scaling is needed.

Fortunately, Satellites already authorize all accesses to pieces, and can therefore meter and rate limit access to popular files. If a file's demand starts to grow more than current resources can serve, the Satellite has an opportunity to temporarily pause accesses if necessary, increase the redundancy of the file over more storage nodes, and then continue allowing access.

Reed-Solomon erasure coding has a very useful property. Assume a (k, n) encoding, where any k pieces are needed of n total. For any non-negative integer number x, the first n pieces of a $(k, n + x)$ encoding are the exact same pieces as a (k, n) encoding. This means that redundancy can easily be scaled with little overhead.

As a practical example, suppose a file was encoded via a $(k = 20, n = 40)$ scheme, and a Satellite discovers that it needs to double bandwidth resources to meet demand. The Satellite can download any 20 pieces of the 40, generate just the last 40 pieces of a new $(k = 20, n = 80)$ scheme, store the new pieces on 40 new nodes, and—without changing any data on the original 40 nodes—store the file as a $(k = 20, n = 80)$ scheme, where any 20 out of 80 pieces are needed. This allows all requests to adequately load balance across the 80 pieces. If demand outstrips supply again, only 20 pieces are needed to generate even more redundancy. In this manner, a Satellite could temporarily increase redundancy to $(20, 250)$, where requests are load balanced across 250 nodes, such that every piece of all 250 are unique, and any 20 of those pieces are all that is required to regenerate the original file.

On one hand, the Satellite will need to pay storage nodes for the increased redundancy, so content delivery in this manner has increased at-rest costs during high demand, in addition to bandwidth costs. On the other hand, content delivery is often desired to be highly geographically redundant, which this scheme provides naturally.

6.2 Improving user experience around metadata

In our initial concrete implementation, we place significant burdens on the Satellite operator to maintain a good service level with high availability, high durability, regular payments, and regular backups.

Over time, clients of Satellites will want to reduce their dependence on Satellite operators and enjoy more efficient data portability between Satellites besides downloading and uploading their data manually. We plan to spend significant time on improving this user experience in a number of ways.

In the short term, we plan to build a metadata import/export system, so users can make backups of their metadata on their own and transfer their metadata between Satellites.

In the medium term, we plan to reduce the size of these exports considerably and make as much of this backup process as automatic and seamless as possible. We expect to build a system to periodically back up the major portion of the metadata directly to the network.

In the long term, we plan to architect the Satellite out of the platform. We hope to eliminate Satellite control of the metadata entirely via a viable Byzantine-fault tolerant consensus algorithm, should one arise. The biggest challenge to this is finding the right balance between coordination avoidance and Byzantine fault tolerant consensus, where storage nodes can interact with one another and share encoded pieces of files while still operating within the performance levels users will expect from a platform that is competing with traditional cloud storage providers. Our team will continue to research viable means to achieve this end.

See section 2.10 and appendix A for discussions on why we aren't tackling the Byzantine fault tolerant consensus problem right away.

7. Selected calculations

7.1 Object repair costs

A fundamental challenge in our system is how to not only choose the system parameters that keep the expansion factor and repair bandwidth to a minimum but also provide an acceptable level of durability.

Fortunately, we are not alone in wondering about this, and there is a good amount of prior research on the problem. “Peer-to-Peer Storage Systems: a Practical Guideline to be Lazy” [34] is an excellent guide, and much of our work follows from their conclusions. The end result is a mathematical framework which determines network durability and repair bandwidth given Reed-Solomon parameters, average node lifetime, and reconstruction rate.

The following is a summary of results and explanation of their implications.

Variable	Description
$MTTF$	Mean time to failure
α	$1/MTTF$
MRT	Mean reconstruction time
γ	$1/MRT$
D	Total bytes on the network
n	Total number of pieces per segment (RS encoding)
k	Pieces needed to rebuild a segment (RS encoding)
m	Repair threshold
LR	Loss rate
$1\text{-}LR$	Durability
E_D	Expansion factor
B_R	Ratio of data that is repair bandwidth
BW_R	Total repair bandwidth in the network

$$LR = \frac{1}{(m+1)\ln(n/m)} \frac{m!}{(k-1)!} \left(\frac{\alpha}{\gamma}\right)^{m-k+2}$$

$$E_D = n/k$$

$$B_R = \frac{\alpha(n-m+k)}{k\ln(n/m)}$$

$$BW_R = D \cdot B_R$$

The equations demonstrate that repair bandwidth is impacted by node churn linearly, which is expected. Lower mean time to node failure triggers more frequent rebuilds and,

therefore, more bandwidth usage. Loss rate is much more sensitive to high node churn, as it increases exponentially with α. This necessitates very stable nodes, with lifetimes of several months, to achieve acceptable network durability. See section 7.3 for a more in-depth discussion of how node churn affects erasure code parameters.

7.1.1 Bandwidth limits usable space

Repair affects storage nodes' participation beyond their bandwidth usage; it also constrains the amount of usable disk space. Consider a storage node with 1 TB of available space, with a stated monthly bandwidth limit of 500 GB. If it's known (via the above framework) that a storage node can expect to repair 50% of its data in a given month, and assuming each stored object is served at least once, then we can store no more than 333 GB on this node since anything more than that causes more bandwidth than allowed. In other words, paid bandwidth plus repair bandwidth must always be less than or equal to the bandwidth limit.

Higher repair rates equal lower effective storage size, but nodes serving paid data more frequently are more sensitive to this effect. In practice, the paid bandwidth rate will vary with the type of data being stored on each node. These ratios must be monitored closely to determine appropriate usable space limits as the network evolves over time.

7.2 Audit false positive risk

We rely on a Bayesian approach to determine the probability that a storage node is maintaining stored pieces faithfully. At a high level, we seek to answer the following question: how do consecutive successful audits change our estimate of the probability that a node will continue to return successful audits?

We model the audit process as being a binomial random variable with an unknown probability of success $p \in [0,1]$, with each audit being an independent Bernoulli trial. It is well-known that the conjugate prior of the binomial distribution is the beta distribution $\beta(a,b)$, and that the posterior also follows the beta distribution. As in [78], we use the mean of the posterior distribution as our Bayes estimator, which is given by $P = (a+x)/(a+b+n)$ where a, b are the parameters of the prior distribution, and x is the number of successes observed in n audits. Under our assumption that each audit is successful, we arrive at the Bayes estimate of the success probability $P = (a+n)/(a+b+n)$.

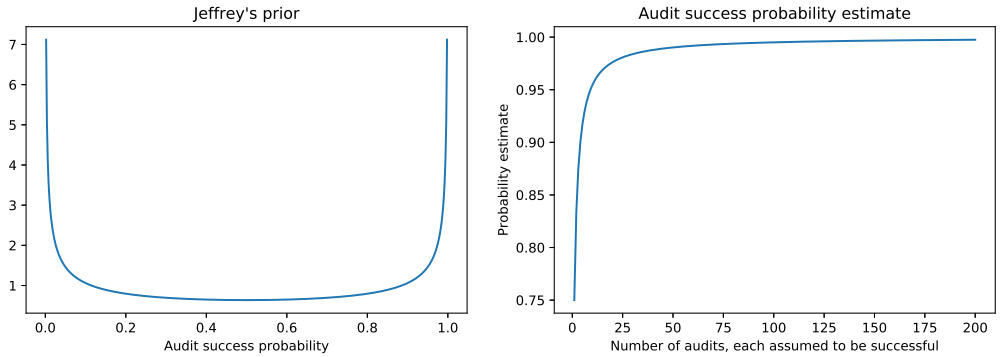


Figure 7.1: In Jeffrey's prior, we see the estimate for audit success probability is heavily weighted to be near 0 or near 1.

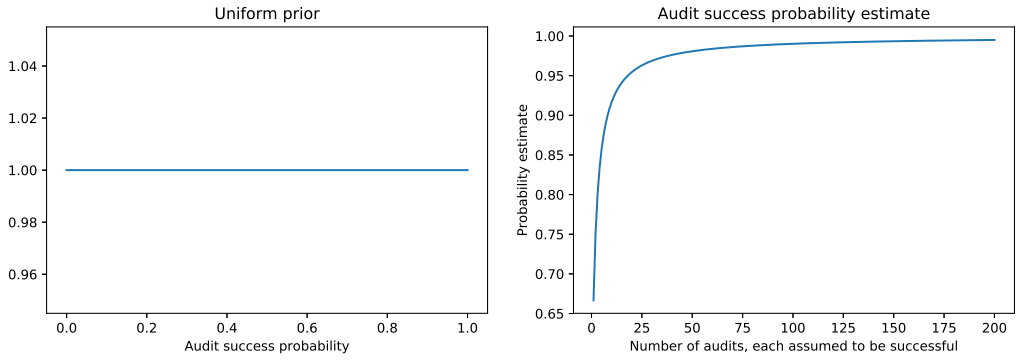


Figure 7.2: Using a Uniform prior, there is no assumption placed on the estimated audit success probability, and all probabilities are assumed to be equally likely.

We now choose a prior to derive a numerical estimate of the audit success probability based on the number of audits performed. There are many reasonable choices of Bayesian priors, but we restrict our attention to two popular choices: the Uniform prior

and Jeffrey's prior [79]. Using the Uniform prior $\beta(1,1)$ initializes the experiment by assigning an equal probability to all possible outcomes; that is, the probability of success is drawn from the uniform distribution on $(0,1)$. Under Jeffrey's prior $\beta(0.5,0.5)$, it is assumed that the probability of success falls towards either extreme, so that a node will return a successful audit either with probability near 0 or with probability near 1.

Number of audits	Audit success estimate given uniform prior	Audit success estimate given Jeffrey's prior
0	0.5	0.5
20	0.9545	0.9762
40	0.9762	0.9878
80	0.9878	0.9938
200	0.99505	0.99751

Table 7.1: Estimate of audit success probability by number of audits, each assumed to be successful. We find that the estimated probability of success begins at 0.5 when there is no information known about the node (no audits have been performed), with the estimate quickly jumping to above 99% in as few as 80 audits using Jeffrey's prior.

In Table 7.1, we present results obtained from using both priors. We remark that the well-established Bayesian approach allows us to rapidly gain more confidence in a node's ability to return a successful audit, given that the success probability estimate tends closer to 1 with each consecutive audit success.

7.3 Choosing erasure parameters

In the context of storing an erasure-coded segment on a decentralized network, we consider the loss of a *piece* from two different perspectives.

7.3.1 Direct piece loss

With direct piece loss, we assume that for a specific segment, its erasure pieces are lost according to a certain rate. We point out that modeling this is straightforward: if pieces are lost at a rate $0 < p < 1$ and we start with n pieces, then piece decay follows an exponential decay pattern of the form $n(1-p)^t$, with t being the time elapsed according to the units used for the rate.[1] To account for a multiple checks per month, we may extend this to $n(1-p/a)^{at}$. If m is the rebuild threshold which controls when a segment is rebuilt, we may solve $n(1-p/a)^{at} = m$ for t (taking the ceiling when necessary) to determine how long it will take for the n pieces of a segment to decay to less than m pieces. This works out to the smallest t for which $t > \frac{\ln(m/n)}{a\ln(1-p/a)}$. Thus it becomes clear, given parameters n, m, a and p, how long we expect a segment to last between repairs.

7.3.2 Indirect piece loss

When modeling indirect piece loss, we suppose that a fixed rate of nodes drop out of the network each month,[2] whether or not they are holding pieces of the segment under consideration. To describe the probability that d of the dropped nodes were each storing one of the n pieces of a specific segment, we turn to the hypergeometric probability distribution. Suppose c nodes are replaced per month out of C total nodes on the network. Then the probability that d nodes were each storing a piece of the segment is given by

$$P(X = d) = \frac{\binom{n}{d}\binom{C-n}{c-d}}{\binom{C}{c}} \tag{7.1}$$

which has mean nc/C. We then determine how long it will take for the number of pieces to fall below the desired threshold m by iterating, holding the overall churn c fixed but reducing the number of existing pieces by the distribution's mean in each iteration and counting the number of iterations required. For example, after one iteration, the number of existing pieces is reduced by nc/C, so instead of n pieces on the network (as the parameter in (7.1)), there are $n - nc/C$ pieces, changing both the parameter and the mean for (7.1) in iteration 2.

We may extend this model by considering multiple checks per month (as in the direct piece loss case), assuming that c/a nodes are lost every $1/a$-th of a month instead of assuming that c nodes are lost per month, where a is the number of checks per month. This yields an initial hypergeometric probability distribution with mean nc/aC.

[1] So if we assume a proportion of $p = 0.1$ pieces are lost per month, t is given in months.
[2] Though the rate may be taken over any desired time interval.

In either of these two cases (single or multiple segment integrity checks per month), we track the number of iterations until the number of available pieces fall below the repair threshold. This number may then be used to determine the expected number of rebuilds per month for any given segment.

7.3.3 Numerical simulations for indirect piece loss

We produce decision tables (Table 7.2) showcasing worst-case mean segment rebuild outcomes based on simulating piece loss for segments encoded with varying Reed-Solomon parameters. We assume a (k, n) RS encoding scheme, where n pieces are generated, with k pieces needed for reconstruction, using three different values for n. We also assume that a segment undergoes the process of repair when less than m pieces remain on the network, using three different values of m for each n. For the initial table, we use a simplifying assumption that pieces on the network are lost at a constant rate per month,[3] which may be due to node churn, data corruption, or other problems.

To arrive at the value for mean rebuilds per month, we consider a single segment that is encoded with n pieces which are distributed uniformly randomly to nodes on the network. To simulate conditions leading to a rebuild, we uniformly randomly select a subset of nodes from the total population and designate them as failed. We do this multiple times per (simulated) month, scaling the piece loss rate linearly according to the number of segment integrity checks per month.[4]

Once enough nodes have failed to bring the number of pieces above the repair threshold m, the segment is rebuilt, and we track the number of rebuilds over the course of 24 months. We repeat this simulation for 1000 iterations, simulating 1000 two-year periods for a single segment. We then take the number of rebuilds at the 99th percentile (or higher) of the number of rebuilds occurring over these 1000 iterations. In other words, we choose the value for which the value of the observed cumulative distribution function (CDF), describing the number of rebuilds over this two-year period, is at least 0.99. This value is then divided by the number of months to arrive at the mean rebuilds/month value. An example of the approach is shown in Figure 7.3. We perform the experiment on a network of 10,000 nodes, observing that the network size will not directly impact the mean rebuilds/month value for a single segment under our working assumption of a constant rate of loss per month.[5]

In forming the decision tables, we consider as part of our calculations how different choices of k, n, m, and mean time to failure affect durability and repair bandwidth. What

[3] This constant rate may be viewed as the mean of the Poisson distribution modeling piece loss per month.

[4] For example, if the monthly network piece loss rate is assumed to be 0.1 of the network size (or 10%), and if 10 segment integrity checks are performed per month, we assume that, on average, 1% of pieces are lost between checks.

[5] We represent piece loss as a proportion of nodes selected uniformly randomly from the total network. The proportion scales directly with network size, so the overall number of pieces lost stays the same for networks of different sizes.

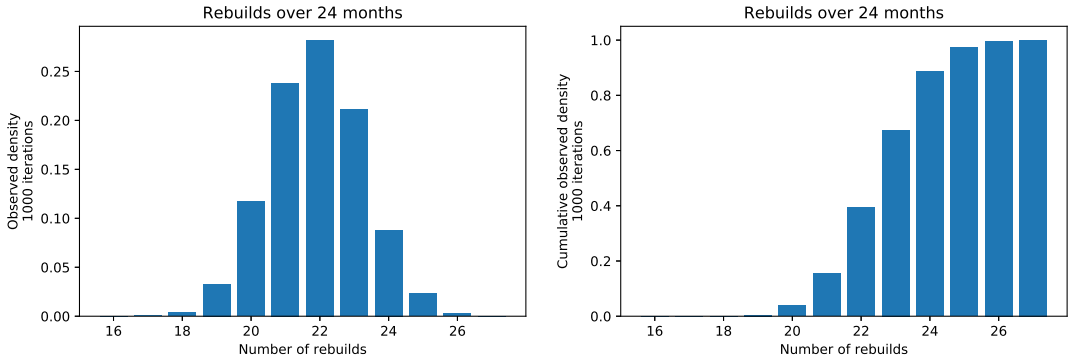


Figure 7.3: Left: Density for the number of rebuilds over a 24 month period, repeated for 1000 iterations. Right: CDF of the number of rebuilds. In this case, the mean rebuilds/month value would be taken as $26/24 \approx 1.083$*, with there being a 99.7% chance that a segment is rebuilt at most 26 times over the course of 24 months.*

we are looking for is the lowest repair bandwidth that also meets our durability requirements.

MTTF (months)	k	n	m	Repair Bandwidth Ratio	Durability (# nines)
1	20	40	35	9.36	0.9999 (8)
6	20	40	30	0.87	0.9999 (17)
12	20	40	25	0.31	0.9999 (13)
1	30	60	35	3.40	0.9999 (4)
6	30	70	40	0.60	0.9999 (15)
12	30	80	45	0.31	0.9999 (25)
1	40	80	60	5.21	0.9999 (4)
6	40	120	50	0.52	0.9999(14)
12	40	120	45	0.24	0.9999 (11)

Table 7.2: Decision tables showing the relationship between churn (MTTF), Reed-Solomon parameters (k, n, m), repair bandwidth ratio, and durability

7.3.4 Conclusion

We conclude by observing that these models may be tuned to target specific network scenarios and requirements. One network may require one set of Reed-Solomon parameters, while a different network may require another. In general, the closer m/n is to 1, the more rebuilds per month should be expected under a fixed churn rate. While having a larger ratio for m/n increases file durability for any given churn rate, it comes at the expense of more bandwidth used since repairs are triggered more often. To maintain a low mean rebuilds/month value while also maintaining a higher file durability, the aim should be to increase the value of n as much as feasible given other network conditions (latency, download speed, etc.), which allows for a lower relative value of m while still not jeopardizing file durability.

Informally, it takes longer to lose more pieces under a given fixed network size and churn rate. Therefore, to maximize durability while minimizing repair bandwidth usage, n should be as large as existing network conditions allow. This allows for a value of m that is relatively closer to k, reducing the mean rebuilds/month value, which in turn lowers the amount of repair bandwidth used.

For example, assume we have a network with a mean time to failure of six months. Suppose we consider the same file encoded with two different RS parameters: one under a $(20, 40)$ schema and the other as an $(30, 80)$ schema. If we set m so that $m = k + 10$ for both cases, we observe from the above table that the bandwidth repair ratio is 0.87 in the $(20, 40)$ case and is 0.60 in the $(40, 80)$ case. Both encoding schemes have similar durability, as a repair in both cases is triggered when there are $k + 10$ pieces left; even though the mean of rebuilds per month is empirically and theoretically lower for the $(40, 80)$ case using $m = k + 10$.

A. Distributed consensus

To explain why we are not trying to solve Byzantine distributed consensus, it's worth a brief discussion of the history of distributed consensus.

A.1 Non-Byzantine distributed consensus

Computerized data storage systems began by necessity with single computers storing and retrieving data on their own. Unfortunately, in environments where the system must continue operating at all times, a single computer failure can grind an important process to a halt. As a result, researchers have often sought ways to enable groups of computers to manage data without any specific computer being required for operation. Spreading ownership of data across multiple computers could increase uptime in the face of failures, increase throughput by spreading work across more processors, and so forth. This research field has been long and challenging; but, fortunately, it has led to some really exciting technology.

The biggest issue with getting a group of computers to agree is that messages can be lost. How this impacts decision making is succinctly described by the "Two Generals' Problem" [80],[1] in which two armies try to communicate in the face of potentially lost messages. Both armies have already agreed to attack a shared enemy, but have yet to decide on a time. Both armies must attack at the same time or else failure is assured. Both armies can send messengers, but the messengers are often captured by the enemy. Both armies must know what time to attack and that the other army has also agreed to this time.

Ultimately, a generic solution to the two generals' problem with a finite number of messages is impossible, so engineering approaches have had to embrace uncertainty by necessity. Many distributed systems make trade-offs to deal with this uncertainty. Some systems embrace *consistency*, which means that the system will choose downtime over inconsistent answers. Other systems embrace *availability*, which means that the system chooses potentially inconsistent answers over downtime. The widely-cited CAP theorem [12, 13] states that every system must choose only two of consistency, availability, and partition tolerance. Due to the inevitability of network failures, partition tolerance is non-negotiable, so when a partition happens, every system must choose to sacrifice either consistency or availability. Many systems sacrifice both (sometimes by accident).

In the CAP theorem, consistency (specifically, linearizability) means that every read receives the most recent write or an error, so an inconsistent answer means the system returned something besides the most recent write without obviously failing. More generally, there are a number of other *consistency models* that may be acceptable by making various trade-offs. Linearizability, sequential consistency, causal consistency, PRAM con-

[1] earlier described as a problem between groups of gangsters [81]

sistency, eventual consistency, read-after-write consistency, etc., are all models for discussing how a history of events appears to various participants in a distributed system.[2]

Amazon S3 generally provides *read-after-write consistency*, though in some cases will provide *eventual consistency* instead [84]. Many distributed databases provide eventual consistency by default, such as Dynamo [25] and Cassandra [59].

Linearizability in a distributed system is often much more desirable than more weakly consistent models, as it is useful as a building block for many higher level data structures and operations (such as distributed locks and other coordination techniques). Initially, early efforts to build linearizable distributed consensus centered around two-phase commit, then three-phase commit, which both suffered due to issues similar to the two generals' problem. In 1985, the FLP-impossibility paper [85] proved that no algorithm could reach linearizable consensus in bounded time. Then in 1988, Barbara Liskov and Brian Oki published the Viewstamped Replication algorithm [86] which was the first linearizable distributed consensus algorithm. Unaware of the VR publication, Leslie Lamport set out to prove linearizable distributed consensus was impossible [87], but instead in 1989 proved it was possible by publishing his own Paxos algorithm [88], which became significantly more popular, even though it wasn't officially published in a journal until 1998. Ultimately, both algorithms have a large amount in common.

Despite Lamport's claims that Paxos is simple [89], many papers have been published since then challenging that assertion. Google's description of their attempts to implement Paxos are described in Paxos Made Live [90], and Paxos Made Moderately Complex [91] is an attempt to try and fill in all the details of the protocol. The entire basis of the Raft algorithm is rooted in trying to wrangle and simplify the complexity of Paxos [24]. Ultimately, after an upsetting few decades, reliable implementations of Paxos, Raft, Viewstamped Replication [92], Chain Replication [93], and Zab [94] now exist, with ongoing work to improve the situation further [95, 96]. Arguably, part of Google's early success was in spending the time to build their internal Paxos-as-a-service distributed lock system, Chubby [97]. Most of Google's famous early internal data storage tools, such as Bigtable [98], depend on Chubby for correctness. Spanner [60]—perhaps one of the most incredible distributed databases in the world—is largely just two-phase commit on top of multiple Paxos groups.

A.2 Byzantine distributed consensus

As mentioned in our design constraints, we expect most nodes to be *rational* and some to be *Byzantine*, but few-to-none to be *altruistic*. Unfortunately, all of the previous algorithms we discussed assume a collection of altruistic nodes. Reliable distributed consensus algorithms have been game-changing for many applications requiring fault-tolerant

[2] If differing consistency models are new to you, it may be worth reading about them in Kyle Kingbury's excellent tutorial [82]. If you're wondering why computers can't just use the current time to order events, keep in mind it is exceedingly difficult to get computers to even agree on that [83].

storage. However, success has been much more mixed in the Byzantine fault tolerant world.

There have been a number of attempts to solve the Byzantine fault tolerant distributed consensus problem. The field exploded after the release of Bitcoin [23], and is still in its early stages. Of note, we are particularly interested in PBFT [99] (Barbara Liskov once again with the first solution), Q/U [100], FaB [101] (but see [102]), Bitcoin, Zyzzyva [103] (but also see [102]), RBFT [104], Tangaroa [105], Tendermint [106], Aliph [107], Hashgraph [108], HoneybadgerBFT [109], Algorand [110], Casper [111], Tangle [112], Avalanche [113], PARSEC [114], and others [115].

Each of these algorithms make additional trade-offs, that non-Byzantine distributed consensus algorithms don't require, to deal with the potential for uncooperative nodes. For example, PBFT [99] causes a significant amount of network overhead. In PBFT, every client must attempt to talk to a majority of participants, which must all individually reply to the client. Bitcoin [23] intentionally limits the transaction rate with changing proof-of-work difficulty. Many other post-Bitcoin protocols require all participants to keep a full copy of all change histories.

A.3 Why we're avoiding Byzantine distributed consensus

Ultimately, all of the existing solutions fall short of our goal of minimizing coordination (see section 2.10). Flexible Paxos [96] does significantly better than normal Paxos in the steady-state for avoiding coordination, but is completely unusable in a Byzantine environment. Distributed ledger or "tangle-like" approaches suffer from an inability to prune history and retain significant global coordination overhead.

We are excited about and look forward to a fast, scalable Byzantine fault tolerant solution. The building blocks of one may already be listed in the previous discussion. Until it is clear that one has arisen, we are reducing our risk by avoiding the problem entirely.

B. Attacks

As with any distributed system, a variety of attack vectors exist. Many of these are common to all distributed systems. Some are storage-specific and will apply to any distributed storage system.

B.1 Spartacus

Spartacus attacks, or identity hijacking, can occur when any node assumes the identity of another node and receive some fraction of messages intended for that node by simply copying its node ID. This allows for targeted attacks against specific nodes and data. Spartacus attacks can be mitgated by implementing node IDs as public key hashes and requiring messages to be signed [32]. A Spartacus attacker in this system would be unable to generate the corresponding private key, and thus unable to sign messages and participate in the network.

B.2 Sybil

Sybil attacks [70] involve the creation of large amounts of nodes in an attempt to disrupt network operation by hijacking or dropping messages. Our adoption of proof of work identity generation (section 4.4) reduces the vulnerability to a degree [32].

Further, our storage node reputation system involves a prolonged initial vetting period nodes must complete before they are trusted with significant amounts of data. This vetting system, discussed more in section 4.15, prevents a large influx of new nodes from taking incoming data from existing reputable storage nodes without first proving their longevity.

B.3 Eclipse

An eclipse attack attempts to isolate a node or set of nodes in the network graph by ensuring that all outbound connections reach malicious nodes. Eclipse attacks can be hard to identify, as malicious nodes can be made to function normally in most cases, only eclipsing certain important messages or information. Storj addresses eclipse attacks by using public key hashes as node IDs and signatures based on those public keys [32].

The larger the network is, the harder it will be to prevent a node from finding a portion of the network uncontrolled by an attacker. As long as a storage node or Satellite has been introduced to a portion of the network that is not controlled by the attacker at any

point, the public key hashes and signatures ensure that man-in-the-middle attacks are impossible.

B.4 Honest Geppetto

In this attack, the attacker operates a large number of "puppet" storage nodes on the network, accumulating reputation and data over time. Once a certain threshold is reached, she pulls the strings on each puppet to execute a hostage attack with the data involved, or simply drops each storage node from the network. The best defense against this attack is to create a network of sufficient scale that this attack is ineffective. In the meantime, this can be partially prevented by relatedness analysis of storage nodes. Bayesian inference across downtime, latency, network route, and other attributes can be used to assess the likelihood that two storage nodes are operated by the same organization. Satellites can and should attempt to distribute pieces across as many unrelated storage nodes as possible.

B.5 Hostage bytes

The hostage byte attack is a storage-specific attack where malicious storage nodes refuse to transfer pieces, or portions of pieces, in order to extort additional payments from clients. The Reed-Solomon encoding ought to be sufficient to defeat attacks of this sort (as the client can simply download the necessary number of pieces from other nodes) unless multiple malicious nodes collude to gain control of many pieces of the same file. The same mitigations discussed under the Honest Geppetto attack can apply here to help avoid this situation.

B.6 Cheating storage nodes, Uplinks, or Satellites

Measuring bandwidth with signatures minimizes the risk for Uplink and storage nodes. The Uplink can only interact with the storage node by sending a signed restricted bandwidth allocation (section 4.17). The restriction limits the risk to a very low level. The storage node has to comply with the protocol as expected in order to get more restricted allocations. Storage nodes and Satellites will commence a vetting process that limits their exposure. Storage nodes are allowed to decline requests from untrusted Satellites.

B.7 Faithless storage nodes and Satellites

While storage nodes and Satellites are built to require authentication via signatures before serving download requests, it is reasonable to imagine a modification of the storage

node or Satellite that will provide downloads to any paying requestor. Even in a network with a faithless Satellite, data privacy is not significantly compromised. Strong client-side encryption protects the contents of the file from inspection. Storj is not designed to protect against compromised clients.

B.8 Defeated audit attacks

A typical Merkle proof verification requires pre-generated challenges and responses. Without a periodic regeneration of these challenges, a storage node can begin to pass most audits without storing all of the requested data. Instead, we request a random stripe of erasure shares from all storage nodes. We run the Berlekamp-Welch algorithm [69] across all the erasure shares. When enough storage nodes return correct information, any faulty or missing responses can easily be identified. New storage nodes will be placed into a vetting process until enough audits have passed. See section 4.15 for more details.

C. Primary user benefits

We have designed the Storj network to provide users better security, availability, performance, and economics—across a wide variety of use cases—than either on-premise storage solutions or traditional, centralized cloud storage. While the bulk of this paper describes the design considerations to overcome the challenges of a highly decentralized system, this appendix describes why the end result should be a significant improvement over traditional approaches.

C.1 Security

We have designed our system to be the equivalent of spreading encrypted sand on an encrypted beach. All data is encrypted client-side before reaching our system. Data is sharded and distributed across a large number of independently operated disk drives which are part of a much larger network of independently operated storage nodes.

In a typical scenario (with a 20/40 Reed-Solomon setup), each file is distributed across 40 different disk drives in a global network of over 100,000 independently operated nodes. (The previous version of the Storj network had over 150,000 independently operated nodes.) To compromise an individual file, a would-be bad actor would have to locate and compromise roughly 40 different drives, each operated by a different provider, in a network of over 100,000 drives. Even if the actor were somehow able to compromise those drives, to reconstruct the file, the would-be bad actor would then have to decrypt 256-bit AES encrypted data, with keys that are only held by the end user. And, the would-be bad actor would have to repeat this process with an entirely different set of potential drives for the next file they wish to obtain.

By design, it is not possible for Storj, Satellite operators, storage node operators, or would-be bad actors to mine or compromise end user data. The level of decentralization on the network creates powerful disincentives for malicious actors, as there is no centralized trove of data to target.

C.2 Availability

While most centralized cloud providers employ various strategies to provide protection against individual drive failures, they are not immune to system-wide events. Storms, power outages, floods, earthquakes, operator error, design flaws, network overload, or attacks can compromise entire data centers.

While the centralized providers may calculate and publish theoretically high availability numbers, these calculations depend on drive failures being uncorrelated. In fact, in

any data center, the chances of an individual drive failing is highly correlated with the chances of another drive failing.

In a decentralized system, by contrast, each node is operated by a different individual, in a different location, with separate personnel, power, network access, and so forth. Therefore, the chance of an individual node failing is almost entirely uncorrelated with the chances of other drives failing. As a result, the kinds of availability we obtain are not subject to storms, power outages, or other "black swan" events. Even if the chance of an individual drive failing in the Storj network is higher than in a centralized cloud, the chance of collective failure (e.g. losing 20 out of 40 independent drives) is vanishingly small. In addition, the chance of losing one file is not correlated with the chances of losing a second file.

C.3 Performance

For read-intensive use cases, the Storj network can deliver superior performance by taking advantage of parallelism. The storage nodes are located close to "the edge," reducing the latency experienced when recipients of data are physically far from the data center that houses the data. Read performance benefits from parallelism. The particular erasure coding scheme that we use ensures that slow drives, slow networks, or networks and drives experiencing temporarily high load do not limit throughput. We can adjust the k/n ratio so that we dramatically improve download and streaming speeds, without imposing the kinds of high costs associated with CDN networks.

C.4 Economics

While the amount of data created around the world has doubled every year, the price of cloud storage has only declined about 10% per year over the last three years. There are a number of potential explanations, both on the supply and demand side.

Public cloud storage operators must make large capital investments in building out a network of data centers and must incur significant costs for power, personnel, security, fire suppression, and so forth. Their pricing structure must allow them to recoup those costs. Moreover, the structure of the industry is such that it is inherently oligopolistic: there are only a handful of public cloud companies, and they comprise the largest companies by market cap on the planet (Microsoft, Google, Amazon, Alibaba). As any price decreases by one provider are quickly matched by the other providers, there has been little incentive for providers to drop prices to gain market share.

In a decentralized network, by contrast, there is little marginal cost to being a storage node operator. In our experience, the vast majority of operators are using existing live equipment with significant spare capacity. There is no additional cost to a storage node operator in terms of capital or personnel. Running a drive at full capacity does not

consume significantly more power than running a drive with excess space. And, with careful management relative to caps, most operators should not experience increased bandwidth costs. Consequently, operating a node represents nearly pure margin, and these supply cost savings can be passed on to end users.

We have designed market mechanisms on the demand side as well, to prevent any Satellite operator from cornering the market. Even after providing a healthy margin to farmers, demand partners, and Satellite operators, we believe we should be able to provide profitable storage services at a fraction of the cost of equivalent centralized cloud storage providers.

Bibliography

[1] Identity Theft Resource Center and CyberScout. Annual number of data breaches and exposed records in the United States from 2005 to 2018 (in millions). https://www.statista.com/statistics/273550/data-breaches-recorded-in-the-united-states-by-number-of-breaches-and-records-exposed/, 2018.

[2] Knowledge Sourcing Intelligence LLP. Cloud storage market - forecasts from 2017 to 2022. https://www.researchandmarkets.com/research/lf8wbx/cloud_storage, 2017.

[3] Dan Shearer. EU-US Cloud Privacy Crash. https://kopano.com/kopano-documents/EU-US-Cloud-Privacy.pdf, 2017.

[4] Gartner Inc. Gartner forecasts worldwide public cloud revenue to grow 21.4 percent in 2018. https://www.gartner.com/en/newsroom/press-releases/2018-04-12-gartner-forecasts-worldwide-public-cloud-revenue-to-grow-21-percent-in-2018, 2018.

[5] Synergy Research Group. Cloud Growth Rate Increased Again in Q1; Amazon Maintains Market Share Dominance. https://www.srgresearch.com/articles/cloud-growth-rate-increased-again-q1-amazon-maintains-market-share-dominance, 2018.

[6] Backblaze Inc. How Long Do Hard Drives Last: 2018 Hard Drive Stats. https://www.backblaze.com/blog/hard-drive-stats-for-q1-2018/, 2018.

[7] Sean Rhea, Dennis Geels, Timothy Roscoe, and John Kubiatowicz. Handling Churn in a DHT. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '04, page 10, Berkeley, CA, USA, 2004. USENIX Association.

[8] Petar Maymounkov and David Mazières. Kademlia: A Peer-to-Peer Information System Based on the XOR Metric. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, IPTPS '01, pages 53–65, London, UK, 2002. Springer-Verlag.

[9] Charles Blake and Rodrigo Rodrigues. High availability, scalable storage, dynamic peer networks: Pick two. In *Proceedings of the 9th Conference on Hot Topics in Operating Systems - Volume 9*, HOTOS'03, page 1, Berkeley, CA, USA, 2003. USENIX Association.

[10] Comcast Inc. XFINITY Data Usage Center–FAQ. https://dataplan.xfinity.com/faq/, 2018.

[11] Amitanand S. Aiyer, Lorenzo Alvisi, Allen Clement, Mike Dahlin, Jean-Philippe Martin, and Carl Porth. BAR Fault Tolerance for Cooperative Services. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP '05, pages 45–58, New York, NY, USA, 2005. ACM.

[12] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002.

[13] Seth Gilbert and Nancy Lynch. Perspectives on the CAP Theorem. *Computer*, 45(2):30–36, February 2012.

[14] Daniel Abadi. Consistency Tradeoffs in Modern Distributed Database System Design: CAP is Only Part of the Story. *Computer*, 45(2):37–42, February 2012.

[15] Peter Bailis, Aaron Davidson, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Highly available transactions: Virtues and limitations. *Proc. VLDB Endow.*, 7(3):181–192, November 2013.

[16] Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Coordination avoidance in database systems. *Proc. VLDB Endow.*, 8(3):185–196, November 2014.

[17] Chenggang Wu, Jose M. Faleiro, Yihan Lin, and Joseph M. Hellerstein. Anna: A KVS for any scale. *ICDE*, 2018.

[18] Joseph M. Hellerstein. The declarative imperative: Experiences and conjectures in distributed logic. *SIGMOD Rec.*, 39(1):5–19, September 2010.

[19] Peter Alvaro, Neil Conway, Joseph M. Hellerstein, and William R. Marczak. Consistency Analysis in Bloom: a CALM and Collected Approach. *CIDR*, 2011.

[20] Kyle Kingsbury. Consistency models clickable map. https://jepsen.io/consistency, 2018.

[21] Paolo Viotti and Marko Vukolić. Consistency in non-transactional distributed storage systems. *ACM Comput. Surv.*, 49(1):19:1–19:34, June 2016.

[22] Joseph Hellerstein. Anna: A crazy fast, super-scalable, flexibly consistent KVS. https://rise.cs.berkeley.edu/blog/anna-kvs/, 2018.

[23] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. https://bitcoin.org/bitcoin.pdf, 2008.

[24] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 305–320, Berkeley, CA, USA, 2014. USENIX Association.

[25] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM.

[26] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM.

[27] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[28] Lustre. Introduction to Lustre Architecture. http://wiki.lustre.org/images/6/64/LustreArchitecture-v4.pdf, 2017.

[29] J. Rosenberg, R. Mahy, P. Matthews, and D. Wing. Session Traversal Utilities for NAT (STUN). RFC 5389, RFC Editor, October 2008. http://www.rfc-editor.org/rfc/rfc5389.txt.

[30] ISO. ISO/IEC 29341-1:2011: Information technology — UPnP Device Architecture, 2011.

[31] S. Cheshire and M. Krochmal. NAT Port Mapping Protocol (NAT-PMP). RFC 6886, RFC Editor, April 2013. http://www.rfc-editor.org/rfc/rfc6886.txt.

[32] Ingmar Baumgart and Sebastian Mies. S/Kademlia: A practicable approach towards secure key-based routing. In *ICPADS*, pages 1–8. IEEE Computer Society, 2007.

[33] P. Mockapetris. Domain names - implementation and specification. STD 13, RFC Editor, November 1987. http://www.rfc-editor.org/rfc/rfc1035.txt.

[34] Frédéric Giroire, Julian Monteiro, and Stéphane Pérennes. Peer-to-peer storage systems: A practical guideline to be lazy. *IEEE Global Communications Conference (GlobeCom)*, 12 2010.

[35] Shawn Wilkinson, Tome Boshevski, Josh Brandoff, James Prestwich, Gordon Hall, Patrick Gerbes, Philip Hutchins, and Chris Pollard. Storj: A peer-to-peer cloud storage network v2.0. https://storj.io/storjv2.pdf, 2016.

[36] Vijay K. Bhargava, Stephen B. Wicker, IEEE Communications Society., and IEEE Information Theory Society. *Reed-Solomon codes and their applications / edited by Stephen B. Wicker, Vijay K. Bhargava ; IEEE Communications Society and IEEE Information Theory Society, co-sponsors*. IEEE Press Piscataway, NJ, 1994.

[37] Jeff Wendling and JT Olds. Introduction to Reed-Solomon. https://innovation.vivint.com/introduction-to-reed-solomon-bc264d0794f8, 2017.

[38] Netanel Raviv, Yuval Cassuto, Rami Cohen, and Moshe Schwartz. Erasure correction of scalar codes in the presence of stragglers. *CoRR*, abs/1802.02265, 2018.

[39] 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*, CCS '09, pages 187–198, New York, NY, USA, 2009. ACM.

[40] Zooko Wilcox. zfec: filefec.py's encode_file. https://github.com/tahoe-lafs/zfec/commit/2594d395923dd945cd62, 2007.

[41] Jeffrey Dean and Luiz André Barroso. The Tail at Scale. *Communications of the ACM*, 56:74–80, 2013.

[42] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[43] J. Paiva and L. Rodrigues. Policies for Efficient Data Replication in P2P Systems. In *2013 International Conference on Parallel and Distributed Systems*, pages 404–411, Dec 2013.

[44] Peter Wuille. BIP32: Hierarchical Deterministic Wallets. https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki, 2012.

[45] 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*, CCS '07, pages 584–597, New York, NY, USA, 2007. ACM.

[46] Hovav Shacham and Brent Waters. Compact proofs of retrievability. In *Proceedings of the 14th International Conference on the Theory and Application of Cryptology and Information Security: Advances in Cryptology*, ASIACRYPT '08, pages 90–107, Berlin, Heidelberg, 2008. Springer-Verlag.

[47] 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*, CCSW '09, pages 43–54, New York, NY, USA, 2009. ACM.

[48] Michael Ovsiannikov, Silvius Rus, Damian Reeves, Paul Sutter, Sriram Rao, and Jim Kelly. The Quantcast File System. *Proc. VLDB Endow.*, 6(11):1092–1101, August 2013.

[49] Salman Niazi, Mahmoud Ismail, Seif Haridi, Jim Dowling, Steffen Grohsschmiedt, and Mikael Ronström. HopsFS: Scaling Hierarchical File System Metadata Using newSQL Databases. In *Proceedings of the 15th Usenix Conference on File and Storage Technologies*, FAST'17, pages 89–103, Berkeley, CA, USA, 2017. USENIX Association.

[50] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, July 1970.

[51] Google Inc. What is gRPC? https://grpc.io/docs/guides/index.html, Accessed 2018.

[52] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246, RFC Editor, August 2008. http://www.rfc-editor.org/rfc/rfc5246.txt.

[53] Trevor Perrin. The Noise Protocol Framework. https://noiseprotocol.org/noise.pdf, 2018.

[54] Irving S. Reed and Gustave Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, 1960.

[55] Amazon Inc. Amazon Simple Storage Service - Object Metadata. https://docs.aws.amazon.com/AmazonS3/latest/dev/UsingMetadata.html, Accessed 2018.

[56] Tao Ma. ext4: Add inline data support. https://lwn.net/Articles/468678/, 2011.

[57] Bram Cohen. The BitTorrent Protocol Specification. http://www.bittorrent.org/beps/bep_0003.html, 2008.

[58] D. Richard Hipp et al. SQLite. https://www.sqlite.org/, 2000.

[59] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.

[60] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Dale Woodford, Yasushi Saito, Christopher Taylor, Michal Szymaniak, and Ruth Wang. Spanner: Google's globally-distributed database. In *OSDI*, 2012.

[61] Eugen Rochko and others. Mastodon: Your self-hosted, globally interconnected microblogging community. https://github.com/tootsuite/mastodon, 2016.

[62] Daniel J. Bernstein. Cryptography in NaCl. https://cr.yp.to/highspeed/naclcrypto-20090310.pdf, 2009.

[63] Daniel J. Bernstein. NaCl: Validation and verification. https://nacl.cr.yp.to/valid.html, 2016.

[64] Arnar Birgisson, Joe Gibbs Politz, Úlfar Erlingsson, Ankur Taly, Michael Vrable, and Mark Lentczner. Macaroons: Cookies with contextual caveats for decentralized authorization in the cloud. In *Network and Distributed System Security Symposium*, 2014.

[65] Jinyuan Li, Maxwell Krohn, David Mazières, and Dennis Shasha. Secure Untrusted Data Repository (SUNDR). In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 9–9, Berkeley, CA, USA, 2004. USENIX Association.

[66] Eu-Jin Goh, Hovav Shacham, Nagendra Modadugu, and Dan Boneh. SiRiUS: Securing Remote Untrusted Storage. In *NDSS*, volume 3, pages 131–145, 2003.

[67] Mahesh Kallahalla, Erik Riedel, Ram Swaminathan, Qian Wang, and Kevin Fu. Plutus: Scalable Secure File Sharing on Untrusted Storage. In *Proceedings of the 2Nd USENIX Conference on File and Storage Technologies*, FAST '03, pages 29–42, Berkeley, CA, USA, 2003. USENIX Association.

[68] Ralph C. Merkle. A digital signature based on a conventional encryption function. In Carl Pomerance, editor, *Advances in Cryptology — CRYPTO '87*, pages 369–378, Berlin, Heidelberg, 1988. Springer.

[69] Lloyd R. Welch and Elwyn R. Berlekamp. Error correction for algebraic block codes. US Patent US4633470A, 1986.

[70] John R. Douceur. The Sybil Attack. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, IPTPS '01, pages 251–260, London, UK, 2002. Springer-Verlag.

[71] Shawn Wilkinson and James Prestwich. SIP02: Bounding Sybil Attacks with Identity Cost, (2016). https://github.com/storj/sips/blob/main/sip-0002.md.

[72] Michael Mitzenmacher. The Power of Two Choices in Randomized Load Balancing. *IEEE Trans. Parallel Distrib. Syst.*, 12(10):1094–1104, October 2001.

[73] Fabian Vogelsteller and Vitalik Buterin. ERC-20 Token Standard, (2015). https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md.

[74] Braydon Fuller. SIP09: Bandwidth Reputation and Accounting, (2017). https://github.com/storj/sips/blob/main/sip-0009.md.

[75] B. C. Neuman. Proxy-based authorization and accounting for distributed systems. In *The 13th International Conference on Distributed Computing Systems*, pages 283–291, May 1993.

[76] Bruce Schneier. *Applied Cryptography (2nd Ed.): Protocols, Algorithms, and Source Code in C*. John Wiley & Sons, Inc., New York, NY, USA, 1995.

[77] Protocol Labs. Filecoin: A decentralized storage network. https://filecoin.io/filecoin.pdf, 2017.

[78] Asit P. Basu, David W. Gaylor, and James J. Chen. Estimating the probability of occurrence of tumor for a rare cancer with zero occurrence in a sample. *Regulatory Toxicology and Pharmacology*, 23(2):139 – 144, 1996.

[79] Harold Jeffreys. An invariant form for the prior probability in estimation problems. *Proceedings of the Royal Society of London. Series A, Mathematical and Physical Sciences*, 186(1007):453–461, 1946.

[80] Jim Gray. Notes on data base operating systems. In *Operating Systems, An Advanced Course*, pages 393–481, London, UK, 1978. Springer-Verlag.

[81] E. A. Akkoyunlu, K. Ekanadham, and R. V. Huber. Some constraints and tradeoffs in the design of network communications. In *Proceedings of the Fifth ACM Symposium on Operating Systems Principles*, SOSP '75, pages 67–74, New York, NY, USA, 1975. ACM.

[82] Kyle Kingsbury. Strong consistency models. https://aphyr.com/posts/313-strong-consistency-models, 2014.

[83] Justin Sheehy. There is no now. *Queue*, 13(3):20:20–20:27, March 2015.

[84] Amazon Inc. Amazon Simple Storage Service - Data Consistency Model. https://docs.aws.amazon.com/AmazonS3/latest/dev/Introduction.html#ConsistencyModel, Accessed 2018.

[85] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, April 1985.

[86] Brian M. Oki and Barbara H. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, PODC '88, pages 8–17, New York, NY, USA, 1988. ACM.

[87] Leslie Lamport. The part-time parliament website note. https://www.microsoft.com/en-us/research/publication/part-time-parliament/, Accessed 2018.

[88] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[89] Leslie Lamport. Paxos made simple. https://www.microsoft.com/en-us/research/publication/paxos-made-simple/, 2001.

[90] Tushar Deepak Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live - an engineering perspective (2006 invited talk). In *Proceedings of the 26th Annual ACM Symposium on Principles of Distributed Computing*, 2007.

[91] Robbert Van Renesse and Deniz Altinbuken. Paxos made moderately complex. *ACM Comput. Surv.*, 47(3):42:1–42:36, February 2015.

[92] Barbara Liskov and James Cowling. Viewstamped replication revisited. Technical Report MIT-CSAIL-TR-2012-021, MIT, July 2012.

[93] Robbert van Renesse and Fred B. Schneider. Chain replication for supporting high throughput and availability. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6*, OSDI'04, page 7, Berkeley, CA, USA, 2004. USENIX Association.

[94] Flavio Paiva Junqueira, Benjamin C. Reed, and Marco Serafini. Zab: High-performance broadcast for primary-backup systems. *IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN)*, pages 245–256, 2011.

[95] Iulian Moraru, David G. Andersen, and Michael Kaminsky. There is more consensus in egalitarian parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 358–372, New York, NY, USA, 2013. ACM.

[96] H. Howard, D. Malkhi, and A. Spiegelman. Flexible Paxos: Quorum intersection revisited. *ArXiv e-prints*, August 2016.

[97] Mike Burrows. The Chubby Lock Service for Loosely-coupled Distributed Systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 335–350, Berkeley, CA, USA, 2006. USENIX Association.

[98] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. In *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 205–218, 2006.

[99] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, pages 173–186, Berkeley, CA, USA, 1999. USENIX Association.

[100] Michael Abd-El-Malek, Gregory R. Ganger, Garth R. Goodson, Michael K. Reiter, and Jay J. Wylie. Fault-scalable byzantine fault-tolerant services. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP '05, pages 59–74, New York, NY, USA, 2005. ACM.

[101] Jean-Philippe Martin and Lorenzo Alvisi. Fast byzantine consensus. *IEEE Trans. Dependable Secur. Comput.*, 3(3):202–215, July 2006.

[102] I. Abraham, G. Gueta, D. Malkhi, L. Alvisi, R. Kotla, and J.-P. Martin. Revisiting Fast Practical Byzantine Fault Tolerance. *ArXiv e-prints*, December 2017.

[103] Ramakrishna Kotla. Zyzzyva: Speculative byzantine fault tolerance. *ACM Transactions on Computer Systems (TOCS)*, 27, Issue 4, Article No. 7, December 2009.

[104] P. L. Aublin, S. B. Mokhtar, and V. Quéma. RBFT: Redundant Byzantine Fault Tolerance. In *2013 IEEE 33rd International Conference on Distributed Computing Systems*, pages 297–306, July 2013.

[105] Christopher N. Copeland and Hongxia Zhong. Tangaroa: a Byzantine Fault Tolerant Raft, 2014.

[106] Jae Kwon. Tendermint: Consensus without mining. https://tendermint.com/docs/tendermint.pdf, 2014.

[107] Pierre-Louis Aublin, Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. The Next 700 BFT Protocols. *ACM Trans. Comput. Syst.*, 32(4):12:1–12:45, January 2015.

[108] Leemon Baird. The Swirlds hashgraph consensus algorithm: Fair, fast, byzantine fault tolerance, 2016.

[109] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The Honey Badger of BFT Protocols. Cryptology ePrint Archive, Report 2016/199, 2016. https://eprint.iacr.org/2016/199.

[110] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 51–68, New York, NY, USA, 2017. ACM.

[111] Vitalik Buterin and Virgil Griffith. Casper the friendly finality gadget. *CoRR*, abs/1710.09437, 2017.

[112] Serguei Popov. The Tangle. https://iota.org/IOTA_Whitepaper.pdf, 2018.

[113] Team Rocket. Snowflake to Avalanche: A Novel Metastable Consensus Protocol Family for Cryptocurrencies. https://ipfs.io/ipfs/QmUy4jh5mGNZvLkjies1RWM4YuvJh5o2FYopNPVYwrRVGV, 2018.

[114] Pierre Chevalier, Bartłomiej Kamiński, Fraser Hutchison, Qi Ma, and Spandan Sharma. Protocol for Asynchronous, Reliable, Secure and Efficient Consensus (PARSEC). http://docs.maidsafe.net/Whitepapers/pdf/PARSEC.pdf, 2018.

[115] James Mickens. The saddest moment. *;login: logout*, May 2013. https://scholar.harvard.edu/files/mickens/files/thesaddestmoment.pdf.