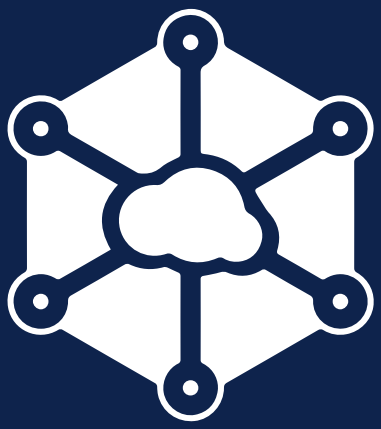


STORJ

Storj: A Decentralized Cloud Storage Network Framework

Storj Labs, Inc.

v3.0, September 24, 2018
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 on less expensive ways to provide storage at a greater rate than any single entity 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, 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 especially thank the other authors and contributors of the previous Storj v2 whitepaper: Tome Boshevski, Josh Brandoff, Vitalik Buterin, Braydon Fuller, Gordon Hall, Chris Pollard, and James Prestwich.

We would like to acknowledge the efforts, whitepapers, 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 appendix, but we would like to provide particular acknowledgement to the teams that built Allmydata, Ceph, Cleversafe, Crashplan, Ethereum, Filecoin, Gluster, IPFS, Kademlia, Maidsafe, Minio, Siacoin, SONM, 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.

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 currently 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 data 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, cost less than current storage methods. Another benefit of this method is that the ability to access data is not dependent on the current availability of any single data center.

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 across the globe 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 secure, performant, and reliable.

We have organized the rest of this paper into five additional chapters. Chapter 2 dis-

cusses 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 proposes a simple concrete implementation of each component, while chapter 5 explains what happens during each operation in the network. Finally, chapter 6 covers future work.

As with Google's Chubby lock service [3], the Storj Network is an engineering effort required to fill the needs mentioned above and below; the concrete implementation detailed here was not research. We claim no novel algorithms or techniques. The purpose of this paper is to describe what we did, what we plan to do, and why.

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; however, decentralized storage platforms include 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. 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. Open source software 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.

One of the main motivations for preferring decentralization is to drive down infrastructure costs for maintenance, utilities, and bandwidth. We believe that there are significant 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 has access to less expensive electricity than standard data centers or another small operator has 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 rural hydro-electric dam operator has enough excess electricity to run ten servers, but no 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.

Operating a decentralized network requires node operators to contribute resources (e.g. network bandwidth, compute, storage, and energy) in exchange for compensation (i.e. the opportunity to earn revenue). In most instances, an increase in demand for a decentralized service drives increase in adoption and proliferation of storage nodes. This effectively increases the available capacity while at the same time driving costs down.

Our decentralization goals are also driven by our desire to provide a viable alternative for fundamental infrastructure, such as storage, to the few major 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 addresses the core limitations, risks, and cost factors that result from centralized cloud storage. Within this context, decentralization results in a globally distributed network that can serve a wide range of storage use cases from archival to CDN. Whereas, centralized storage systems require different architectures, implementations, and infrastructure to address each of those same use cases.

2.3 Amazon S3 compatibility

At the time of this paper's publication, the most widely deployed cloud storage protocol is Amazon Web Services' Simple Storage Service protocol (Amazon S3). 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, we are creating a graceful transition path from cen-

```
// Bucket operations
CreateBucket(bucketName string)
DeleteBucket(bucketName string)
ListBuckets() []BucketInfo

// Object operations
GetObject(bucketName, objectPath string, offset, length int64) (
    ObjectData, ObjectInfo)
PutObject(bucketName, objectPath string, data ObjectData,
    metadata map[string]string) ObjectInfo
DeleteObject(bucketName, objectPath string)
ListObjects(bucketName, prefix, startKey string,
    limit int, delimiter string) ([]ObjectInfo, more bool)
```

Figure 2.1: Minimum S3 API

tralized providers for our users. This will alleviate many switching costs for users with data currently stored on a centralized provider. 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.

2.4 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 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 it 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 options 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 compel 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 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. Since node availability and reliability has a large impact on network availability, cost, and reliability, 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 must be able to retain a reasonable profit. They 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 accommodate 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.

2.5 Device failure and churn

For all devices, component failure is a guarantee. All hard drives fail after enough wear [5] and servers providing network access to these hard drives will also eventually fail. Network

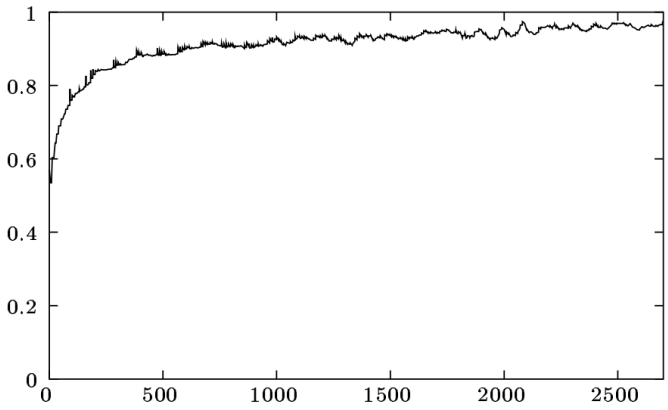


Figure 2.2: Probability of remaining online an additional hour as a function of uptime. The x axis represents minutes. The y axis shows the fraction of Nodes that stayed online at least x minutes that also stayed online at least x+60 minutes. Source: Maymounkov et al. *[6]*

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. A network with a high churn rate will use large amounts of bandwidth just to ensure durability of the data and will fail to scale. As a result, a scalable, highly durable storage network must encourage stable nodes and keep the churn rate as low as possible.

Churn rate affects the network much more than hardware failure. As an illustration, Maymounkov *et al.* found that in decentralized systems, the probability of a node staying connected to the network for an additional hour is an *increasing* function of uptime (Figure 2.2 [6]). In other words, the longer a node is a participant in the network, the more likely it is to continue participating. This gives our system a strong incentive to prefer long-lived, stable nodes.

Node churn heavily affects overall system bandwidth usage. See appendix C for a discussion of how repair bandwidth varies as a function of node churn.

2.6 Latency

Decentralized storage systems can potentially capitalize on massive opportunities for parallelism. Some of these opportunities include increased transfer rates, processing capabil-

ities, 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 designers often encounter is the asymmetric internet connections offered by 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 [7]. 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. 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 appendix A.2 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 filesystem 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 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.

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 [8] 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 one that optimizes for a utility function that is independent of the one given for the suggested protocol. Inevitable hardware failures aside, *Altruistic* nodes participate in a proposed protocol even if the rational choice is to deviate. *Rational* nodes participate or deviate only when it is in their net best interest.

Some distributed storage systems (e.g. Amazon S3, Microsoft Azure Blob Storage) operate in an environment where all nodes are considered *altruistic*. For example, sans 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. Therefore, incentives that allow for or encourage *byzantine* behavior must also be eliminated.

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 [9–16]. 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 [13]. 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 significant cost.

While many types of operations in a network may require coordination (e.g., operations that require linearizability[1] [12, 17, 18]), choosing strategies that avoid coordination (such as Highly Available Transactions [12]) 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 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 [14, 19]. Not all coordination can be avoided, but new frameworks (such as Invariant Confluence [13]) 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 [20] or even Raft [21]) 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 that is entirely controllable by each user, it is a fundamental requirement to limit hotpath coordination domains to small spheres.

[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, the next goal is to design a framework consisting of relatively fundamental components. The framework should describe 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 now and Storj in 10 years from now. While there will be significant 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 relatively independent components and then combine them to form the desired framework.

The individual components are:

1. Storage nodes
2. Peer-to-peer communication and discovery
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.

Because storage nodes are selected via changing variables external to the protocol, node selection is an explicit, nondeterministic 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 [22]. 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 and discovery

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, UPnP, NAT-PMP, etc.
- provides authentication, 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 the bandwidth allocation protocol (see section 4.15.1), 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 network address, any other peer can connect to it. This responsibility is similar to the internet's standard domain name system (DNS), which is a mapping of an identifier to an ephemeral connection address, but unlike DNS, there can be no centralized registration process. To achieve this, a network overlay can be built on top of our peer-to-peer communication protocol that provides this functionality. See Section 4.5 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 avail-

able 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, 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. Importantly, *erasure codes* allow changes in durability without changes in expansion factor. Erasure codes are an encoding scheme for manipulating data durability without tying it to bandwidth usage. 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! Intuitively, 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.

By being able to tweak the durability independently of the expansion factor, very high durabilities can 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. It also results in storage nodes getting paid more. High expansion factors dilute the incoming funds per byte across more storage nodes; whereas, low expansion factors allow for a much more direct passthrough of income to storage node operators.

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)
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%

To see how this table was 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:

$$P(D) = e^{-\lambda} \sum_{i=0}^{n-k} \frac{\lambda^i}{i!}. \tag{3.1}$$

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

k	n	Exp. factor	P(D)
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%

3.4.1 Erasure codes' effect on streaming

Erasure codes are used in many streaming contexts such as audio CDs and satellite communications, 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) more challenging. Whatever erasure code is chosen for our framework, 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 4.7 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 [23]. 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 use the fastest peers to download the starting pieces and slower nodes to help with the later pieces.

The goal is a protocol that allows for every request to be satisfiable by the fastest nodes participating in any given transaction, without needing to wait for a slower subset. Focusing on operations where the result is only dependent on the fastest nodes of a random

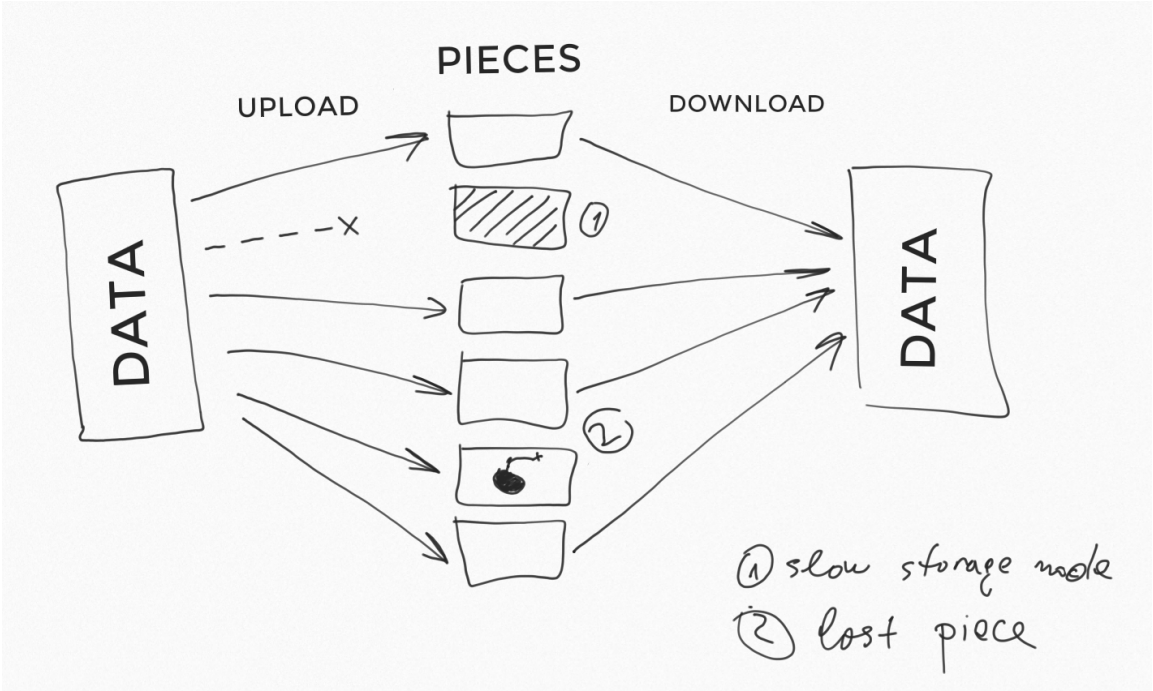


Figure 3.1: Various outcomes during upload and download

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. Moreover, 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. This implies 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. Metadata values should be able to be stored and retrieved by arbitrary key; in addition, deterministic iteration over those keys will be required.

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 this system stores 1 total exabyte of data, where the average object size is 50MB and our erasure code is selected such that $n = 40$. 1 exabyte of 50MB objects is 20 billion objects. This metadata 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, this metadata can be heavily partitioned by the user. A user storing 100 terabytes of 50MB objects will only incur a metadata overhead of 5.5 gigabytes. It's worth pointing out that these numbers vary heavily with average object size: the larger the object size, the less the metadata overhead.

An additional framework focus is enabling this component – metadata storage – to be interchangeable per user. 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).

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).

3.6 Encryption

Regardless of storage system, our design constraints require total security and privacy. All data or metadata will be encrypted. Data should 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 as well as store metadata about that encryption scheme to allow them to recover their data using the appropriate decryption mechanism.

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 similar to BIP32 [24] will allow subtrees to be shared without sharing their parents and will allow some files to be shared without sharing other files. See section 4.10 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.

Many storage systems use audits as a way of determining when and where to repair files. Our storage system does not use auditing for this purpose. Instead, we are extending the probabilistic nature of common per-file *proofs of retrievability* [25] to range across all possible files.

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 future. Storage Node uptime and overall health are the primary metrics used to determine which files need repair. 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 to help calculate the future usefulness of a given Storage Node.

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 appendix B). 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 loose requirements on the use of such a system, but see section 4.14 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 is the leading risk. As evidenced by the findings of Maymounkov *et al.*, expected node availability is an increasing function

of uptime [6], implying that node churn is the dominant effect. Storage nodes may go offline due to hardware or software failure, intermittent internet connectivity, or operator choices. Because audits are validating that conforming nodes store data correctly, all that remains is to detect when a storage node goes offline or becomes bad and then repair at-risk data.

Assuming that probabilistic audits are enough to estimate the likelihood that a node will have the data it should represents a huge shortcut in terms of work. We can use overall node status, which is much more efficient than file-by-file audits, to calculate when a file is at risk. We don't consider any types of availability other than node availability when determining which files to repair.

There are many other ways data might get lost in the network besides node churn, such as corruption, malicious behavior, bad hardware, software error, or user initiated space reclamation. However, these issues are less serious than full node churn issues such as power loss, internet connectivity intermittency, complete disk failure, and software shutdown or removal. Our spot-check based audits will incentivize storage nodes to reliably store data while estimating the rate at which data is actually stored reliably. Therefore, our repair system only seeks to solve the node churn problem. We expect to account for varying amounts of other issues by configuring erasure code parameters according to differing network conditions.

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 avoid naively placing a blockchain-based solution in the storage hot path. 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 (see section 2.10).

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.

The Storj network is payment agnostic. The protocol does not require a specific payment type. The network assumes the Ethereum-based STORJ token as the default payment medium, but many other payment types could be implemented, including Bitcoin,

Ether, credit or debit card, ACH transfer, or physical transfer of live goats.

In addition to the payment for services rendered to those contributing storage and bandwidth, the system also tracks and aggregates the value of the consumption of those services by those who own the data stored on the network. In this way, the framework is able to support the end-to-end economics of the storage marketplace ecosystem.

4. Concrete implementation

We believe the framework we've described above to be relatively fundamental given our design constraints. However, within the framework there remains a significant degree of 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 over time, but 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.

As with our previous network, we will publish changes to this concrete architecture through our Storj Improvement Proposal process [26].

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 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*. The Storj Network allows the different peer classes and associated functions to be run separately and independently by different operators.

Storage Node This peer class participates in the node discovery system, stores data for others, and gets paid for storage and bandwidth (via the bandwidth allocation protocol in section 4.15.1).

Uplink This peer class represents any application or service that wants to store data. Applications can store data via an Amazon S3-compatible API or through our *libstorj* C-bindings. This peer class is not expected to remain online like the other two classes and is otherwise relatively lightweight. This peer class performs encryption, erasure encoding, and coordinates with the other peer classes on behalf of the customer.

Satellite This peer class participates in the node discovery system, caches node address information, stores per-object metadata, keeps storage node reputation, pays Storage Nodes, aggregates billing data, performs audits and repair, and manages authorization and user accounts. 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 a friend, group, or workplace.

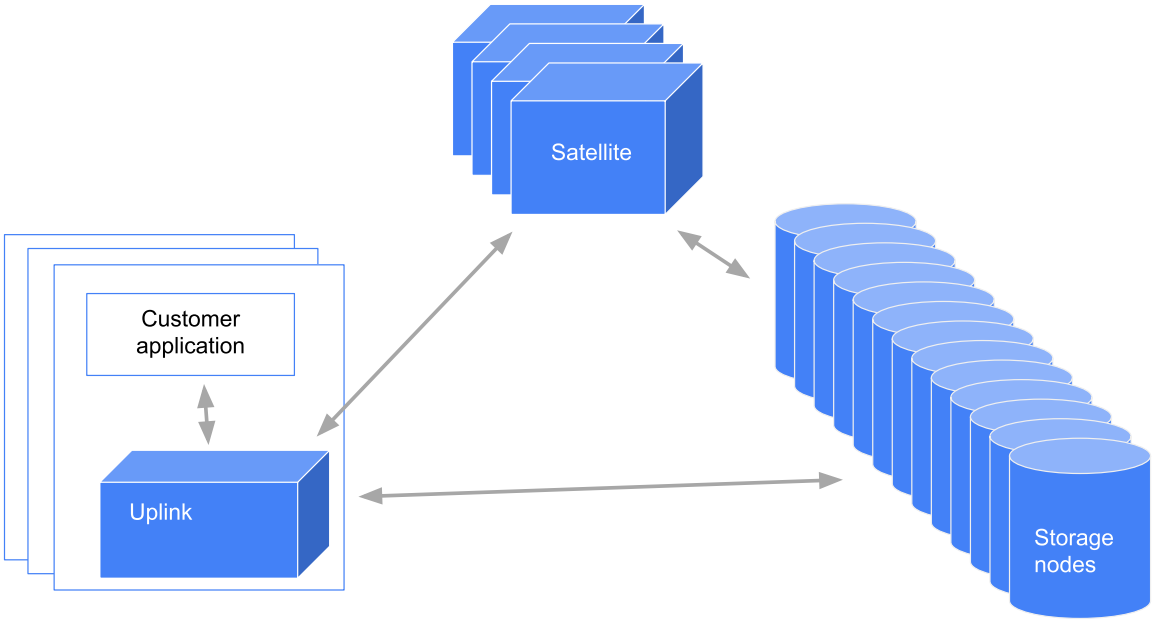


Figure 4.1: The three different peer classes

4.1.2 Data

Bucket A *bucket* is an unbounded, but named collection of files identified by paths. Each path represents one file, and every file has a unique path within a bucket.

Path A *path* is a unique identifier for a file within a bucket. A *path* is a string of bytes that begins with a forward slash byte and ends with something besides a forward slash byte. Forward slashes (referred to as the path separator) separate path components. An example *path* might be */etc/hosts*, where the path components are *etc* and *hosts*. We encrypt paths before they ever leave the customer's application's computer.

File or Object A *file* (or object) is an ordered collection of one or more segments. Segments have a fixed maximum size. The more bytes the *file* has, the more segments there tend to be.

A *file* also supports a limited amount of key/value user-defined fields called extended attributes.

Like paths, the data contained in a *file* is encrypted before it ever leaves the client computer.

Extended attribute An *extended attribute* is a user defined key/value field that is associated with a file. *Extended attributes* are stored encrypted.

Segment A *segment* represents a single array of bytes, between 0 and a user-configurable maximum *segment* size.

Remote Segment A *remote segment* is a larger segment that will be encoded and distributed across the network. A *remote segment* is larger than the metadata required to keep track of its bookkeeping, such as the Nodes it is stored on.

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

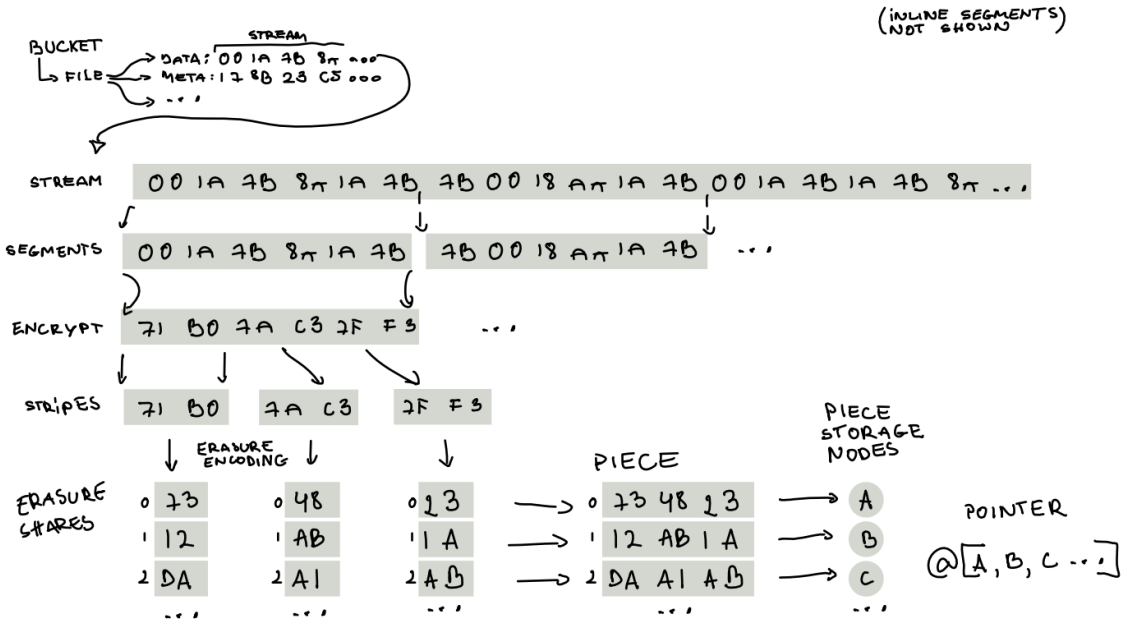


Figure 4.2: Files, segments, stripes, erasure shares, and pieces

need to keep track of which Nodes had the data. In these cases, the data is stored "inline" instead of being stored on Nodes.

Stripe A *stripe* is a further subdivision of a segment. A *stripe* is a fixed amount of bytes that is used as an encryption and erasure encoding boundary size. Erasure encoding happen on *stripes* individually; whereas, encryption may happen on a small multiple of *stripes at a time*. All segments are encrypted, but only remote segments are erasure encoded. A *stripe* is the unit on which audits are performed.

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, but 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, there are *n pieces* after processing a remote segment. The *i*th *piece* is the concatenation of all of the *i*th erasure shares from that segment's stripes.

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.

4.2 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 bandwidth allowance. During node discovery, Storage Nodes will advertise how much bandwidth and hard drive space is available, and what is their designated STORJ token wallet address.

Storage Nodes also keep track of optional per-piece "time-to-live", or TTL. 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 4.15.1) to send to Satellites for later settlement and payment. This also requires a small database. Both TTL and bandwidth allocations are stored in a SQLite [27] database.

Storage Nodes can choose which Satellites to work with. 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. Bandwidth payment is made payable after the Storage Node sends in signed bandwidth allocation messages (section 4.15.1). 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 removed from the pool 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. They are also not paid for node discovery or other maintenance traffic.

Storage Nodes will support three methods: *get*, *put*, and *delete*. Each method will take a *piece ID*, an ID and signature of the associated satellite instance, and the other metadata required by the bandwidth allocation protocol (see section 4.15.1).

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* operation is received, whichever comes first. Storage Nodes will *not* be penalized for rejecting initial *put* operations.

Storage Nodes will allow administrators to configure maximum allowed disk space

[1] Registration with a US-1099 tax form service may be required. See section 4.19.

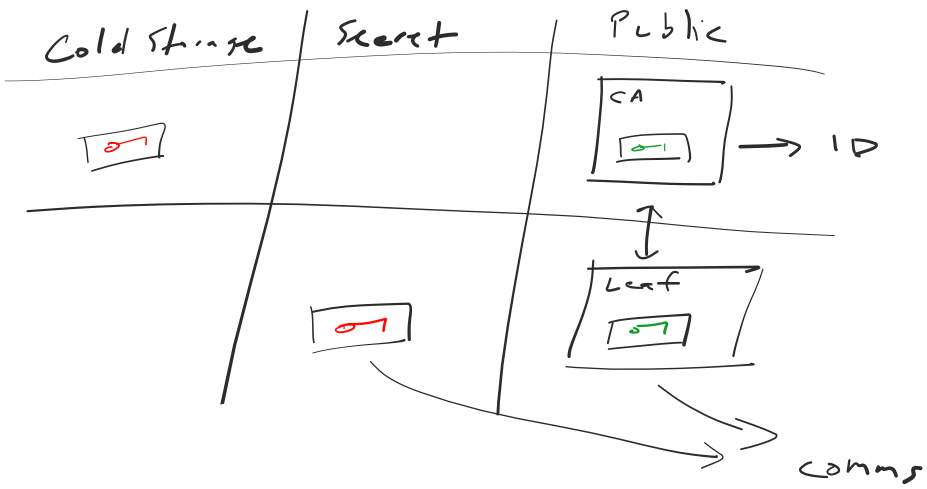


Figure 4.3: The different keys and certificates that compose a Storage Node's overall identity.

and bandwidth usage over the last rolling 30 days. They will keep track of how much is remaining of both, and reject operations that do not have a valid signature from the appropriate Satellite.

4.3 Node identity

During setup, Nodes generate their own identity and certificates for use in the network. This node ID is used for node discovery and routing.

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.

As in S/Kademlia [28], the *node ID* will be the hash of a public key and will serve as a proof of work for joining the network. Unlike Bitcoin's proof of work [20], the work will be dependent on how many *trailing* zero bits one can find in the hash output. This means that the node ID will still be usable in a balanced Kademlia [6] tree. 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.

Nodes use leaves from their root certificate for communication with other Nodes in the network, allowing for secure node-to-node communication in the network.

4.4 Peer-to-peer communication

Initially, we are using the gRPC [29] protocol on top of of the Transport Layer Security protocol (TLS) on top of the μTP [30] transport protocol with added Session Traversal Utilities for NAT (STUN) functionality. STUN provides NAT traversal; μTP provides reliable, ordered delivery (like TCP would); TLS provides privacy and authentication; and gRPC provides multiplexing and a convenient programmer interface. Over time, we will replace TLS with a more flexible secure transport framework (such as the Noise Protocol Framework [31]) 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 0 bits at the end of the ID. Satellites can configure a minimum proof of work required to pass an audit such that, over time, the network will require greater proofs of work due to natural user intervention.

For the few cases where a Node cannot achieve a successful connection through a NAT or firewall (via STUN, uPnP, NATPmP, or similar techniques), manual intervention and port forwarding will be required. In the future, Nodes unable to create a connection through their firewalls may rely on traffic proxying from other, more available Nodes, for a fee.[2] Nodes can also provide assistance to other Nodes for initial STUN setup, public address validation, and so forth.

4.5 Node discovery

At this point, we have Storage Nodes and we have a 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 provide a means to look up a node's latest address by node ID, somewhat similar to the role DNS provides for the public internet.

The Kademlia distributed hash table (DHT) is a key/value store with a built-in node lookup protocol. We utilize Kademlia as our primary source of truth for DNS-like functionality for node lookup, while ignoring the key/value storage aspects of Kademlia. Using

[2] See the bandwidth allocation protocol (4.15.1) for a description of how fees work.

only Kademlia for node lookup, eliminates the need for some other functionality Kademlia would otherwise require (such as owner-based key republishing) neighbor-based key republishing, storage and retrieval of values, and so forth. Furthermore, we avoid a number of known attacks by using the S/Kademlia [28] extensions where appropriate.

Unfortunately, DHTs such as Kademlia require multiple network round trips for many operations, which makes it difficult to achieve millisecond-level response times. To solve this problem, we introduce a caching service on top of Kademlia.

The caching service will attempt to talk to every Storage Node in the network on an ongoing basis, perhaps once per hour. We expect this to scale for the reasonable future, as ping operations are inexpensive, but admit a new solution may be necessary in the future. The caching service will then cache the last known good address for each Node, and evict Nodes that it hasn't talked to after a certain period of time. Fortunately, space requirements are negligible. For instance, caching addresses for a network of 80k nodes can be done with only 5MB of memory[3].

Based on this design, the cache will not be expected to be a primary source of truth, and results in the cache may be stale. Due to our redundant storage strategy, the storage network will be resilient against an expected degree of node churn and staleness. Therefore, the system will be robust even if some lookups in the cache fail or return incorrect addresses. Furthermore, because our peer-to-peer communication system already provides peer authentication, a node discovery cache that sometimes returns faulty or deliberately misleading address lookup responses can only cause a loss of performance but not correctness.

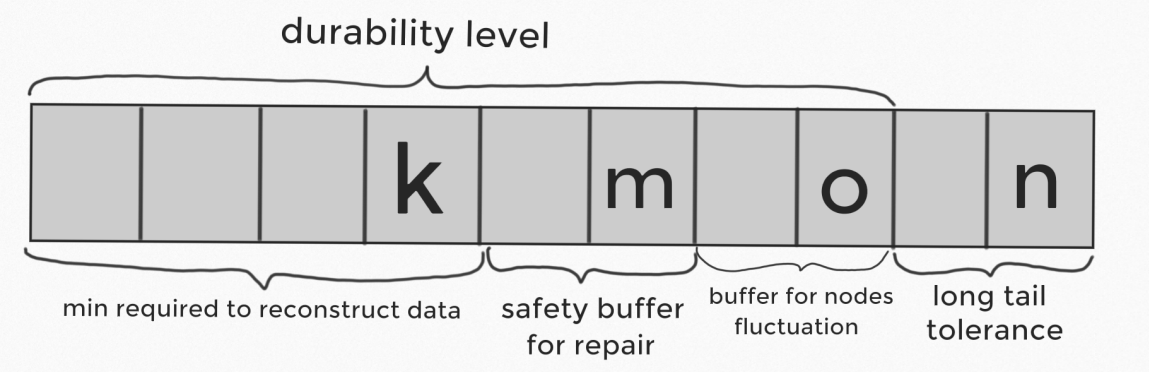
We plan to host and help set up some well-known community-run overlay caches. These caches will perform the duty of quickly returning address information for a given node ID if the node has been online recently. Kademlia will be the long-lived source of truth and will be used directly in many situations that have less stringent performance demands.

With each message shared on the network, Nodes will include their available disk space, bandwidth availability, STORJ wallet address, and any other metadata the network needs. The network overlay cache will collect information provided by the Nodes during refresh periods, allowing faster lookups for this information.

4.6 Redundancy

We use the Reed-Solomon erasure code [32]. For each object that we store we choose 4 numbers, 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.

[3] This is assuming an ordered in-memory list of 4-tuples of node ID (32 bytes), IP address (16 bytes for IPv6), port (2 bytes), and timestamp (4 bytes). $80000 \cdot (32\text{B} + 16\text{B} + 2\text{B} + 4\text{B}) \approx 4.12\text{MB}$



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. 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 appendix C for how we select our Reed-Solomon numbers. Also see section 4.13 for a discussion about how we repair data as its durability drops over time.

4.7 Structured file storage

4.7.1 Files with extended attributes

Many applications benefit from being able to keep metadata alongside files. Amazon S3 supports "object metadata" [33] to assist with this need. This functionality is called "extended attributes" in our system. Every file will include a limited set of user-specified key-value pairs (extended attributes) that will be stored alongside other metadata about the file.

4.7.2 Files as Segments

The highest level subdivision of a file is called a "segment." 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. 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. The last segment of a file stored remotely contains the extended attributes of the file and other useful information, such as the number of segments in the file, the size of the segments, and file encryption details.

Segmenting large files (e.g. videos) and distributing the segments across the network reduces the impact of content delivery on any given Node, as bandwidth demands are distributed more evenly across the network. 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 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.

Segments are represented as pointers, so files are actually collections of segment pointers. The metadata database that stores pointers keeps track of segment pointers by path, so we adjust the path a small amount to help us keep track of files instead of segments.

4.7.3 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. To support these operations, it's useful to be able to decode and decrypt small parts of a segment.

A stripe defines a subset of a segment and should be no more than a couple of kilobytes large. Erasure encoding a single stripe at a time allows us to read small portions of a large segment without retrieving the entire segment first. It also allows us to stream data into the network without staging it beforehand, and it enables a number of other useful features.

4.7.4 Stripes as Erasure Shares

As discussed in sections 3.4 and 4.6, 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. 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.

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

4.7.5 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. 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 $1/k$th of a stripe, each piece is $1/k$th of a segment, and only k pieces are needed to recover the full segment. A piece is what we finally 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. 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.

4.7.6 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, and the owner can secure the pointer as they wish.

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.

4.8 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, SQLite, Cassandra [34], Spanner [35], 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.

There are a few downsides with 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 here is that availability can be made arbitrarily good with existing trusted distributed solutions, such as Cassandra, Spanner, or CockroachDB, and with an appropriate amount of effort 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 just encryption keys. Fortunately, the metadata itself can be periodically backed up into the Storj network, such that only needing to keep track of metadata-metadata further decreases 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. Like Mastodon [36], 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. 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 [12], 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 D for more about why we've chosen to currently avoid trying to solve the problem of Byzantine distributed consensus. See section 6.4 for a discussion of future plans.

4.9 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. The satellite service has been developed and 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, and so forth. 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.

The satellite instance is made up of these components:

- A full node discovery cache (section 4.5)
- A per-object metadata database indexed by encrypted path (section 4.8)
- An account management and authorization system (section 4.11)
- A storage node reputation, statistics, and auditing system (section 4.12)
- A data repair service (section 4.13)
- A storage node payment service (section 4.15)

Future releases will see significant improvements to and potentially rearchitectures of these systems.

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.

4.10 Encryption

Our encryption choice is authenticated encryption, with support for both the AES-GCM cipher and the Salsa20 and Poly1305 combination NaCl calls "Secretbox" [37]. Authenticated encryption is used so that the user can know if anything has tampered with the data. Data encryption keys are chosen by the Uplink.

Data is encrypted in blocks of small batches of stripes, recommended to be 4KB or less [38]. While the same encryption key is used for every stripe 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. The nonce of the first segment is chosen at random and is stored with the file's metadata. To prevent reordering attacks, the starting nonce of each subsequent segment is deterministically chosen based on the segment number. When multiple segments are uploaded in parallel, such as in the case of Amazon S3 multipart upload, 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 with authenticated encryption, but the nonce and key must be deterministic, determined entirely from a root secret combined with the unencrypted path. Each path component is encrypted separately. The first path component is encrypted with a key and nonce derived from the root secret. A new secret is derived from the root secret and the unencrypted name of the first path component. This new secret is used to derive the encryption key and nonce for encrypting the second path component.

For every next path component a new secret is derived from, the secret and unencrypted name of the previous path component and the respective encryption key and nonce are derived from this new secret. HMAC-SHA512 is used for all key derivations.

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. In the future, we may consider an order-preserving encryption scheme to try and achieve the best of both worlds (see section 6.3).

4.11 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. 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 chosen metadata service, which will allow them access to various operations according to their authorization configuration.

Our initial metadata authorization scheme uses macaroons [39]. Each account has a root macaroon and operations are validated against a supplied macaroon's set of caveats. 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. Additionally, they provide the property that anyone can add restrictions in a way where those restrictions cannot later be removed. 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 sharing to specific path prefixes, path separation boundaries between path components must remain across encryption.

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 or append only. Our macaroons are further caveated with revocation tokens and optional expirations, which allow users to revoke macaroons programmatically.

Once authorized with a metadata service, that metadata service has an associated *satellite ID* and is able to sign operations. All operations with Storage Nodes require a specific satellite ID and associated signature. A Storage Node will reject operations not

signed by the appropriate satellite ID. The Uplink must retrieve valid signatures from the metadata service prior to operations with Storage Nodes. Storage Nodes will not allow operations signed by one Satellite to apply to objects owned by another, unless explicitly granted by the owning Satellite.

4.12 Audits

In a network with untrusted nodes, validating that 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 to prove it has the data.

Some distributed storage systems, including the previous release of Storj [40], discuss *Merkle tree proofs*, in which audit challenges and expected responses are generated at the time of storage as a form of compact proof of retrievability [25]. By using a Merkle tree [41], the amount of metadata needed to store these pre-generated challenges and responses can be negligible.

Unfortunately, in such a scheme, the challenges and expected responses must be pre-generated. Without a periodic regeneration of these challenges, a Storage Node can begin to pass most audits without storing all of the requested data. Finding a solution that avoids pre-generated challenges is preferred.

An assumption in our storage system is that most Storage Nodes are reasonably well-behaved, and 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 [42]. Not only are we are already using Reed-Solomon erasure coding [32] on small ranges, *stripes*, but also we use it to issue challenges and verify 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 [42] 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, incorrect, or correct. In the case of a Node being offline, the audit failure may be due to the address in the node discovery cache being stale, so another, fresh lookup will be attempted. If the Node still appears to be offline, the Satellite will continue to try the same audit with that Node periodically until the Node either responds successfully, actively fails the audit, or is disqualified from being offline too long. These audit failures will be stored and 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 appendix B for a discussion on how many audits are required to be confident data is stored correctly.

4.13 Data repair

As Storage Nodes go offline – taking their file pieces with them – it will be necessary for the missing pieces to be rebuilt once the entire file'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 has caches in place that 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 appendix C 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 [6].

The ingress bandwidth demands of the audit and repair system are large, but given standard configuration, the egress demands are relatively small. A large amount of data comes in to the system for audits and repairs, but only the formerly missing pieces get 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. See section 6.2 for a planned distributed repair method that does not rely on the favorable pricing model of current hosting providers.

4.13.1 Merkle trees

Repairs are one of the few places latency matters little. The data repair system needs to process as many files as possible, but it doesn't matter if a specific file takes longer. Throughput is much more important than latency during repair. However, repair is still a costly operation due to significant bandwidth and CPU usage impacting a single operator, so repair work will be minimized. As a result, when repairing a segment, only the minimum number of pieces required will be downloaded. Without full redundancy, erasure codes will be less effective at catching errors. Furthermore, the fallback safety mechanism that the user has for detecting errors (authenticated encryption) is unavailable to the repair system (no decryption keys). Because full segments are repaired at a time, hashes of each *piece* will be stored in the system via a Merkle tree [41], storing the root of the tree in the *pointer*. This allows the repair system to correctly assess whether or not repair has been completed successfully without using extra redundancy for the same task.

A full copy of the leaves of the Merkle tree of pieces (enough to generate the full tree) will be stored alongside each *piece* on each Storage Node. Therefore, the only additional central metadata storage required is just for the root of the Merkle tree.

Each repair will validate the tree before the *pointer* is updated to point to new locations.

4.14 Storage node reputation

Reputation metrics on decentralized networks are a critical part of enabling cooperation between Nodes where progress will 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 three subsystems. The first subsystem is the initial vetting process, the second subsystem is a filtering system, and the third system is a preference system.

The first 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 system 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 system will then start including that storage node in the standard selection process used for general uploads. Importantly, Storage Nodes get paid during this vetting period, but don't receive as much data.

While new nodes require a proof of work to avoid some Sybil attacks [43], additional effort may be required to prevent malicious and determined new nodes from overwhelming the vetting process and preventing well-behaved new nodes from getting enough data to progress past it. Satellite operators will be able to choose as a configuration parameter the minimum proof of work required from storage nodes for new data. Additionally, other schemes are possible, such as a form of proof of stake as we proposed in our previous work [44].

The filtering system is the second subsystem; it blocks bad storage nodes from participating. 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 by failing too many audits, 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 and the Node fails to return it, the Node will be disqualified. Importantly, Storage Nodes will be allowed to reject and fail uploads 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.

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 third subsystem is a preference system. After disqualified storage nodes have been eliminated, 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 [45], 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. See the section 4.19 about quality control for how we plan to ensure Satellites are incentivized to treat Storage Nodes fairly.

Initially, Storage Node reputation will be individually determined by each Satellite. If a Node is disqualified by one Satellite, it will still store data for other Satellites. Reputation will not be shared between Satellites. Over time, as we plan to eliminate Satellites, reputation will then be determined globally.

4.15 Payments

In the Storj network, payments are made by clients who store data on the platform to the Satellite they utilize. The Satellite then pays Storage Nodes for the amount of storage and bandwidth they provide on the network.

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 will be 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. Moving forward, our network will not use contracts at all to manage payments and file storage durations.

Satellites will pay Storage Nodes for the data they store and for object downloads. Storage Nodes will not be paid for the initial storage 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 escrow 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 escrowed 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 they will eventually be cleaned up through the garbage collection process (see section 4.17). 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 on their nodes.

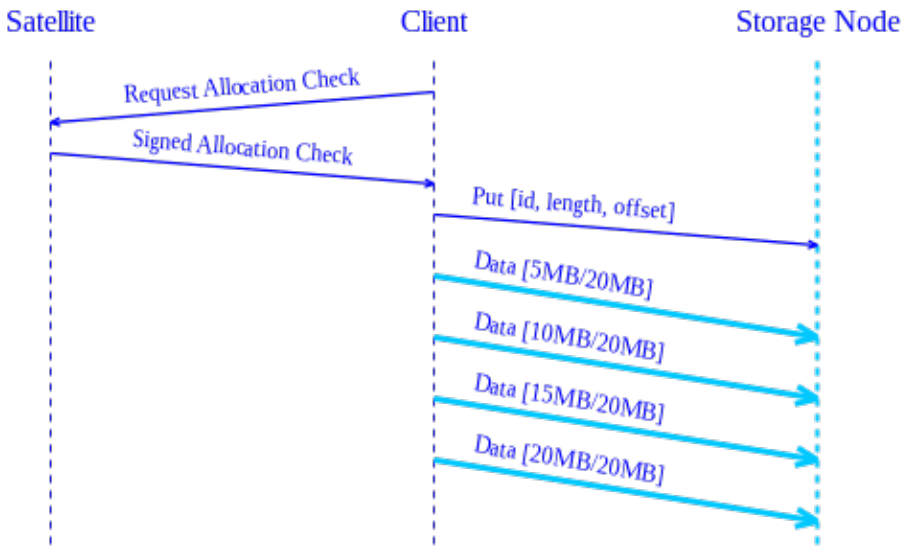


Figure 4.4: Diagram of a put operation with the bandwidth allocation protocol

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 credits to each Storage Node for every file piece it will be storing. Satellites will track utilized bandwidth through a bandwidth allocation protocol (see section 4.15.1. 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 files, and storing metadata. Every day, each Satellite will execute a number of audits across all of its Storage Nodes on the network. During an audit, if a Storage Node does not have the file it should be storing, it will immediately be disqualified and the Satellite will flag that Storage Node's file pieces for repair in the system. The Satellite will be paid for both completing the audit and for the repair, once that file falls below the file piece threshold needed for repair.

See the satellite reputation section for details on how Storage Nodes will know to trust Satellites.

If a Storage Node acts maliciously and does not store files properly or maintain sufficient availability, they 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 will be recipient-initiated and must be worth at least some minimum value.

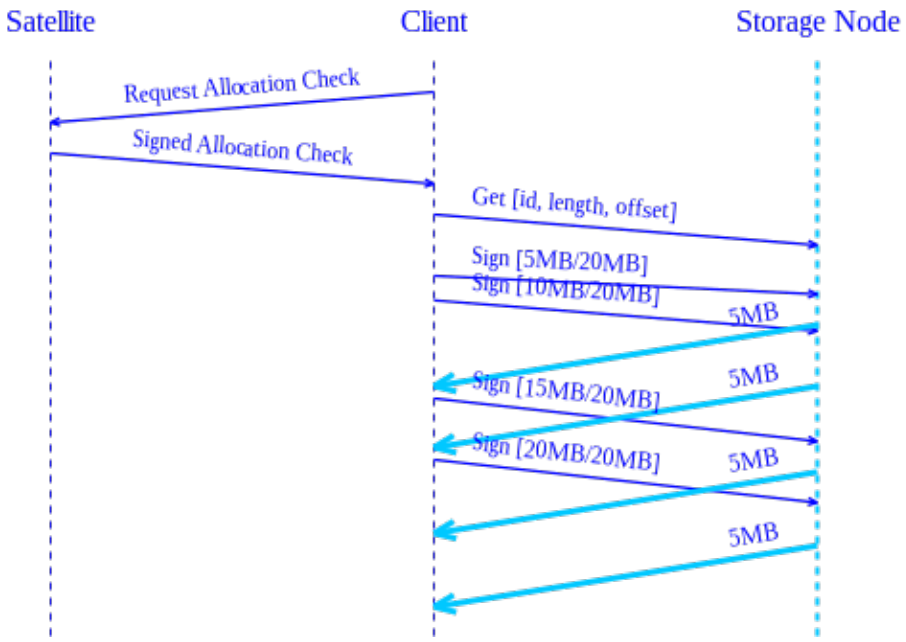


Figure 4.5: Diagram of a get operation with the bandwidth allocation protocol

4.15.1 Bandwidth allocation protocol

A core component of our system requires knowing how much bandwidth is used between two peers. Therefore, we introduce a protocol we call the Bandwidth Allocation Protocol for correctly verifying that a certain amount of bandwidth was used between two peers with incentives. We don't measure all peer-to-peer traffic; this bandwidth traffic measurement only tracks bandwidth used during storage operations, storage and retrievals of pieces, and does not apply to overlay traffic (Kademlia DHT) or other generic maintenance overhead.

When a client wants to perform an operation for *x* bytes of bandwidth, it must first get authorization from a Satellite that it has enough funds and is authorized to perform that operation. The Satellite will return an *unrestricted bandwidth allocation* message. This message will include the satellite ID, the identity of the client, an expiration timestamp, a serial number, the maximum amount of bytes authorized *x*, and the direction the bytes will flow (whether or not the data will be transferred from or to the client). The message will be signed by the Satellite.

Once the client has an unrestricted bandwidth allocation, the client will then create *restricted bandwidth allocations*, indicating *y* bytes have been transferred so far. In the case of a *Get* operation, the client will start by sending a restricted allocation for some small amount, perhaps only a few kilobytes, so the Storage Node can verify the client'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 client 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 *y* bytes total.

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 client that the client agreed to bandwidth usage of up to *y* bytes, along with the Satellite's confirmation of the client's bandwidth allowance. The Storage Node will periodically send the largest restricted bandwidth allocations it has received to appropriate Satellites, at which point Satellites will pay the Storage Node for that bandwidth.

If the client can't afford the bandwidth usage, the Satellite will not sign an unrestricted bandwidth allocation, protecting the satellite's reputation. Likewise, if the client 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.

4.16 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.

When a new Satellite joins the network, the participating Storage Nodes will commence their own vetting process. This process limits their exposure to the new and unknown Satellite, while building trust over time to highlight which of the Satellites have the best payment record. Storage Nodes will be able to configure the maximum amount of data they will store for an untrusted Satellite, and will build historical data on whether that Satellite will be trusted further in the future. storage node operators will also retain manual control on what Satellites they will trust, or don't trust, if desired.

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 "Tardigrade" 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. See section 4.19 for more details.

4.17 Garbage collection

When clients move or delete data, clients, on behalf of Satellites, will notify Storage Nodes that they are no longer required to store that data. However, Storage Nodes will sometimes be temporarily unavailable and will miss delete messages. In these cases, 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 will otherwise 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 cost of storage owed to a Storage Node will be high enough to amortize the cost of storing the garbage.

When delete messages are issued via the client, the metadata system (and thus a Satellite, with satellite reputation on the line) will require proof that deletes were issued to a configurable minimum number of Storage Nodes. This means that every time data is deleted, Storage Nodes that are online and reachable will receive notifications right away.

For the Nodes that miss initial delete messages, we propose a conservative garbage collection strategy. 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 [46], 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.

4.18 Uplink

The Uplink provides an Amazon 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. The Uplink is a simple service layer on top of *libstorj*, which is a library that provides access to storing and retrieving data in the Storj network.

The Uplink will run co-located with wherever data is generated, and will communicate directly with Storage Nodes so as to avoid central bandwidth costs.

4.19 Quality control and branding

The Storj Network has two essential software components that serve two distinct target markets:

1. creating storage supply for the network via recruiting storage node operators and
2. creating demand for cloud storage with paying users.

Storj will differentiate these software components and the experience design for each segment by separating the supply side of our business from the demand side through two brands, *Storj* and *Tardigrade*.

We will retain storj.io as the place for contributing extra storage and bandwidth to the Storj Network. This includes Storage Node setup, documentation, frequently asked questions (FAQs), and tutorials. Users of both brands will also be able to access our source code and community through storj.io. The demand side of our business will be directed through tardigrade.io. This experience will be focused toward our partners and customers who purchased decentralized storage and bandwidth from the network with the expectation of high durability, resilience, and reliability, backed by an industry-leading service level agreement (SLA). This includes any offers, free trials, Satellite selection, documentation, FAQs, tutorials, and so forth.

The "Tardigrade" brand will additionally serve as a satellite quality credentialing system. Anyone can set up a Satellite via storj.io, but to have a Satellite listed as an official Tardigrade Satellite and be considered "Tardigrade quality," and benefit directly from Storj Labs' demand generation activities, an operator must pass certain compliance and quality requirements. These quality controls will continuously audit and rank Satellites on their behavior, durability, compliance, and performance. In addition, the satellite operator will have to adhere to particular business policies around pricing, storage node recruitment, SLAs, storage node payments, etc. Satellite operators in the Tardigrade network will have a business relationship with Storj Labs that defines, among other things, franchise fees and revenue sharing between the entities. Storj Labs will also assume responsibilities including demand generation, brand enforcement, satellite operator support, end user support, United States Form 1099 tax filing compliance,[4] insurance, and maintenance of overall network quality.

These compliance and quality controls will be implemented to ensure that Storage Nodes and Satellites are able to continuously meet all SLAs of the Tardigrade products.

[4] US Form 1099 is required by law for any payments to an individual in a given year exceeding a total of $600.

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 through it.
- 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 that process involves 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 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 signed unrestricted bandwidth allocation messages, and a chosen root piece ID.

Next, the Uplink will take this information and begin parallel connections to all of the chosen Storage Nodes via the bandwidth allocation protocol (section 4.15.1).

- 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 Merkle tree leaves (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; 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. It will throw away all other relevant request data.

Next:

- 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
  - 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 nonce monotonically increases from the previous segment.

- The last segment stored in the stream will contain additional metadata:
  - how many segments the stream contained
  - how large the segments are, in bytes
  - the starting nonce of the first segment
  - file extended attributes and other metadata

Periodically, the Storage Nodes will later send the largest restricted bandwidth allocation 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 a file, 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 the pointers for the first few segments in addition to the pointer for the last segment. The Uplink needs the last segment to learn the size of the object, the size and

number of segments, and how to decrypt the data.

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 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 via the bandwidth allocation protocol (section 4.15.1) 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.
- If the download is aborted for any reason, each Storage Node will keep the largest restricted bandwidth allocation it received, but it will throw away all other relevant request data.
- The Uplink will combine the retrieved erasure shares into stripes and decrypt the data.

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 recieved by the Uplink. The Uplink then requests all of the segment pointers for the file.

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 appropri-

ate Storage Nodes to signal that the pieces are being removed.

- The Storage Nodes will return a signed message indicating the Storage Node received the delete operation and will delete both the file and its bookkeeping information.
- 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.

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 4.17 for more details.

5.4 Move

When a user wants to move a file, first, the Uplink receives a request for moving a 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 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 right away.

5.5 Copy

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

For every segment pointer, the Satellite:

- validates that the Uplink has access to download it
- looks up the contact information for the Storage Nodes listed in the pointer
- returns the requested segment metadata, a new root piece ID, and contact information

For every segment pointer, the Uplink:

- decrypts the metadata with an encryption key derived from the path
- changes the path to the new destination
- invokes a copy operation on each of the Storage Nodes from the pointer to duplicate the piece with a new piece ID
- waits for the Storage Nodes to respond that they have duplicated the data
- re-encrypts the metadata with a new encryption key derived from the new path

Finally, the Uplink uploads all modified segment pointers to the Satellite.

Importantly, it is okay if the Storage Nodes de-duplicate storage, or only store one actual copy of the data. All that matters is that the Storage Node can identify the data by both the old and new piece ID. If one of the piece IDs receives a delete operation, the other piece ID will continue working. Only after both pieces are deleted will the Node free the space.

5.6 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 return results and return them.

5.7 Audit

Each Satellite has a queue of segment stripes that will be audited across a set of Storage Nodes. The queue is filled via two mechanisms.

- In the first mechanism, the Satellite populates the queue periodically by selecting segments randomly, and then stripes within those segments also at random. Because segments have a maximum size, this sufficiently approximates our goal of choosing a byte to audit uniformly at random.
- In the second mechanism, 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.

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. 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. The Satellite will not pay the storage node going forward, and it will not select the storage node for new data.

5.8 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 overlay 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. Unlike audits, only enough pieces for accurate repair are needed. Unlike streaming downloads, the repair system can wait for the entire segment before starting. As a result,

pieces are compared against a Merkle tree of hashes for correctness prior to repair, where the Merkle root is stored in the pointer.
- 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.

5.9 Payment

The payment process works as follows:

- First, the Satellite will choose a roll-up period. This is a period of time – defaulting to a day – that payment for data at rest is calculated.
- 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.
- Finally, Storage Nodes will periodically send in bandwidth allocation reports.

When a Satellite receives a bandwidth allocation report, it calculates the owed funds 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 potential 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 authorize all accesses to files via the bandwidth allocation protocol, 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 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 (20,1000), where requests are load balanced across 1,000 nodes, such that every piece of all 1,000 are unique, and any 20 of those pieces are all that is required to regenerate the original file.

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. However, content delivery is often desired to be highly geographically redundant, which this scheme provides naturally.

6.2 Distributed repair

As mentioned, the audit process checks continually for files whose Reed-Solomon erasure-encoded pieces have fallen below a certain threshold. When such a file is found, it must be repaired, with the new pieces being stored on new Storage Nodes. Currently, this repair process takes place on the Satellite. The Satellite downloads all the file fragments needed to repair the file, the file is rebuilt, and the previously missing pieces are sent to new Storage Nodes selected by the Satellite. Long term, it would be better to create a technique where file repair takes place in a distributed manner on Storage Nodes, putting their excess processor cycles to work.

The system would need more checks and balances to ensure the Storage Node is correctly executing a repair and that the data inside the encrypted file is accurate. Luckily, our existing repair Merkle tree root greatly assists with this new distributed repair scheme.

A basic outline is as follows. First, a Storage Node would volunteer to do a repair for a Satellite, potentially in exchange for later payment. The Satellite would choose a segment that needed repair, determine which pieces need to be replaced, and indicate to the volunteer repairing node which pieces should be regenerated and on which new Storage Nodes they should be placed. By having the Satellite choose the new Storage Nodes, collusion risk is limited. Then, the repairing node would download the segment from the existing Storage Nodes with the blessing of the Satellite, repair the missing pieces, and send the replacement pieces to the desired new Storage Nodes. The new Storage Nodes would return signed statements detailing the hash of the contents they received. The repairing node would return these signed statements to the Satellite, and the Satellite could then use the signed statements along with the piece Merkle tree root to confirm that the data was stored correctly.

6.3 Order-preserving encryption

Our protocol currently encrypts paths by default, increasing security of the network and anonymity of stored data. However, this security comes with the trade-off of having file paths that are not listed in lexicographic order. To list paths in lexicographic order in the current implementation, a client may opt-out of using encrypted file paths, though this has the potential to decrease security and/or anonymity of the stored data. As part of our ongoing work, we are researching order-preserving encryption (OPE) schemes [47] so that a user may have the ability to sort file paths lexicographically while still maintaining data anonymity and security. We believe that a good starting point for research on OPE's should include [48], and [49]. The reader is directed to those resources for further information on the benefits and drawbacks of OPE as it is known currently. Another approach is to consider the fully homomorphic encryption (FHE) schemes found in [50] which may provide a higher level of security than OPE's. This added security may come with other drawbacks, such as higher computational complexity. Part of our future work in this area will require us to consider the benefits and drawbacks of various OPE- or

FHE-type schemes when deciding on which schemes to implement, if any.

6.4 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. We expect a large degree of variation in quality of Satellites, which led us to our quality control program (see section 4.19).

Over time, clientele 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 program 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 achieving fast Byzantine fault tolerant consensus, where Storage Nodes can interact with one another, share encoded pieces of files, and still operate 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 appendix D for a discussion on why we aren't tackling the Byzantine fault tolerant consensus problem right away.

A. 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" [51] 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.

A.1 Repair Bandwidth and Loss Rate Framework

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-LR	Durability
E_D	Expansion factor
B_R	Ratio of data that is repair bandwidth

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

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

$$E_D = n/k$$

$$B_R = BW_R/D$$

A.2 Node churn is bad for durability

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; it increases exponentially with α. This necessitates very stable nodes, with lifetimes of several months, to achieve acceptable network durability. See appendix C for a more in depth discussion of how node churn affects erasure code parameters.

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 cap 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 cap.

Higher repair rates equal lower effective storage size, but Nodes serving paid data more frequently are more sensitive to the 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.

B. 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 follows the beta distribution also. As in [52], 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 $P = (a+n)/(a+b+n)$.

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 [53]. 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. 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. In Figure B.1 we plot Jeffrey's prior, as well as the estimate of success probability obtained from using this prior, plotted as a function of the number of (successful) audits performed. In Figure B.2 we plot the Uniform prior, along with a plot of the resulting success probability estimate.

Number of audits	Uniform prior	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 B.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.

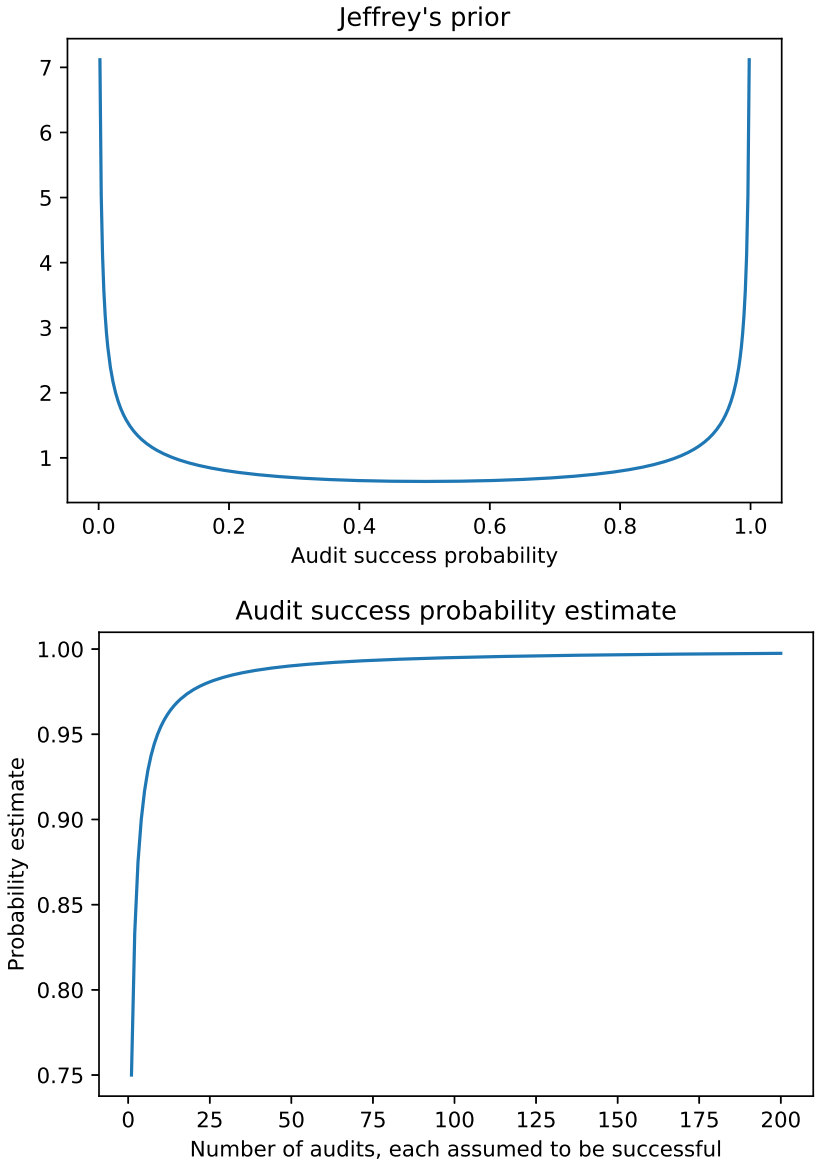


Figure B.1: In Jeffrey's prior, we see the estimate for audit success probability is heavily weighted to be near 0 or near 1.

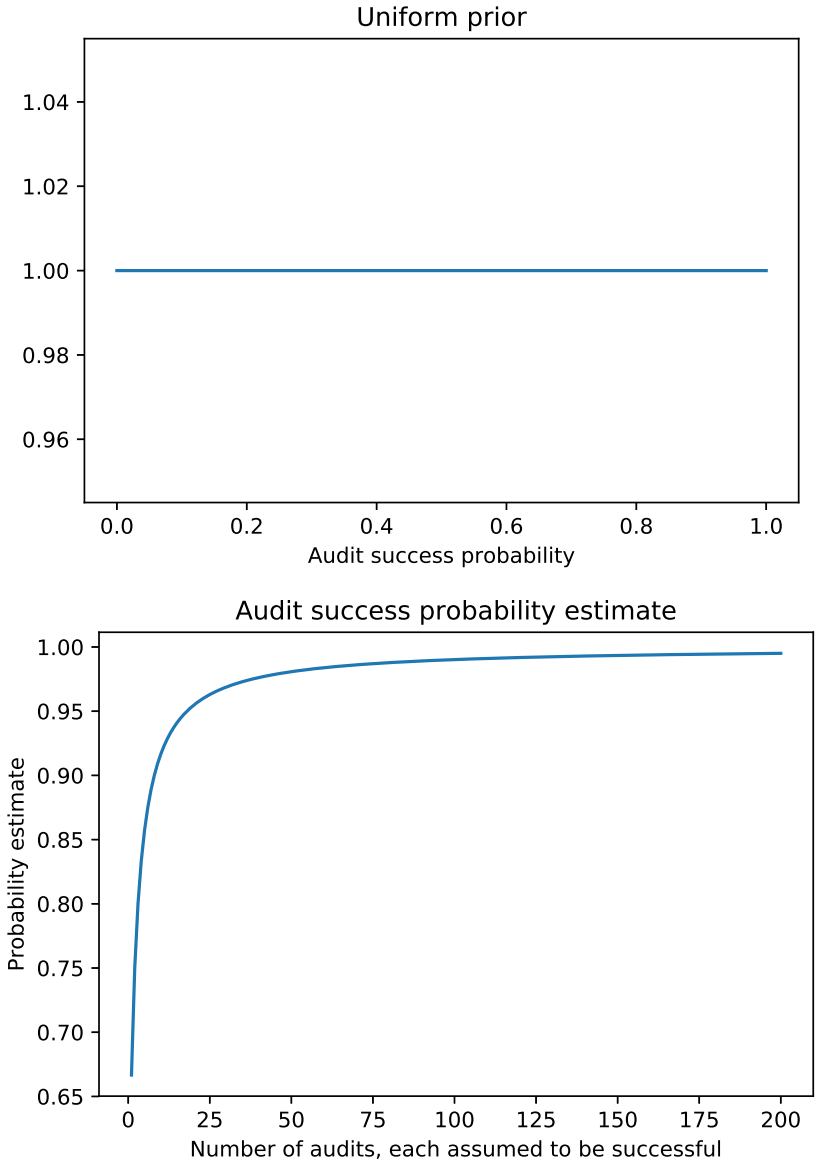


Figure B.2: Using a Uniform prior, there is no assumption placed on the estimated audit success probability, and all probabilities are assumed to be equally likely.

C. Choosing erasure parameters

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

C.1 Direct file piece loss

With direct file piece loss, we assume that for a specific file, its erasure pieces are lost according to a certain rate. We point out that modeling this is straightforward: if file pieces are lost at a rate $0 < p < 1$ and we start with n pieces, then file 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 file 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 file 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 file to last between repairs.

C.2 Indirect file piece loss

When modeling indirect file 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 file under consideration. To describe the probability that d of the dropped nodes were delegates for a specific file coded into n pieces, 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 delegates for the file is given by

$$P(X = d) = \frac{\binom{n}{d}\binom{C-n}{c-d}}{\binom{C}{c}} \tag{C.1}$$

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 (C.1)), there are $n - nc/C$ pieces, changing both the parameter and the mean for (C.1) in iteration 2.

[1] So if we assume a proportion of $p = .1$ pieces are lost per month, t is given in months.

[2] Though the rate may be taken over any desired time interval.

We may extend this model by considering multiple checks per month (as in the direct file 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.

In either of these two cases (single or multiple file 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 file.

C.3 Numerical simulations for indirect file piece loss

C.3.1 Introduction

We produce decision tables showcasing worst-case mean file rebuild outcomes based on simulating file piece loss for files encoded with varying Reed-Solomon parameters. We assume an (k,n) RS encoding scheme, where n pieces are generated, with k pieces needed for reconstruction, using three different values for n. We assume that a file undergoes the process of repair when less than r 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 file 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 file integrity checks ("checks") per month[4]. Once enough nodes have failed to bring the number of file pieces under the repair threshold, the file is rebuilt, and we track the number of rebuilds over the course of 24 months. We repeat this simulation for 1000 iterations, simulating 1000 2-year periods for a single file. We then take the number of rebuilds at the 99-th 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 2 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 C.1. We perform the experiment on a network of 10,000 nodes, observing that the network size

[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 file integrity checks are performed per month, we assume that, on average, 1% of pieces are lost between checks.

will not directly impact the mean rebuilds/month value for a single file under our working assumption of a constant rate of loss per month[5].

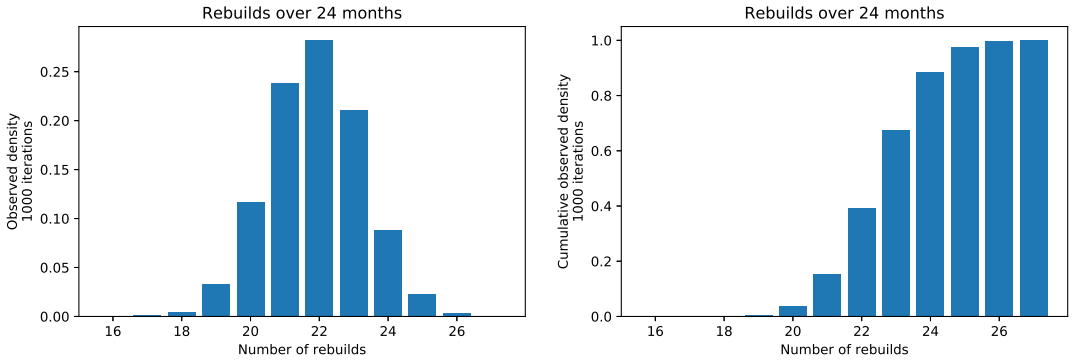


Figure C.1: Top: Density for the number of rebuilds over a 24 month period, repeated for 1000 iterations. Bottom: CDF of the number of rebuilds. In this case, the mean rebuilds/month value would be taken as $26/24 \approx 1.083$, with there being a 99.7% chance that a file is rebuilt at most 26 times over the course of 24 months.

[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.

C.3.2 The decision tables

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 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)

C.4 Making a decision

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 r 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 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 rebuilds per month is empirically and theoretically lower for the $(40,80)$ case using $m = k+10$.

D. Distributed consensus

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

D.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" [54] [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 down-time over inconsistent answers. Other systems embrace *availability*, which means that the system chooses potentially inconsistent answers over downtime. The widely-cited CAP theorem [9,10] 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 consis-

[1] earlier described as a problem between groups of gangsters [55]

tency, 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 [58]. Many distributed databases provide eventual consistency by default, such as Dynamo [22] and Cassandra [34].

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. Things were looking bad in 1985 when the FLP-impossibility paper [59] proved that no algorithm could reach linearizable consensus in bounded time. Then in 1988, Barbara Liskov and Brian Oki published the Viewstamped Replication algorithm [60] which was the first linearizable distributed consensus algorithm. Unaware of the VR publication, Leslie Lamport set out to prove linearizable distributed consensus was impossible [61], but instead in 1989 proved it was possible by publishing his own Paxos algorithm [62], which for some reason became significantly more popular. Ultimately both algorithms have a large amount in common.

Despite Lamport's claims that Paxos is actually simple [63], 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 [64], and Paxos Made Moderately Complex [65] 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 [21]. Ultimately, after an upsetting few decades, reliable implementations of Paxos, Raft, Viewstamped Replication [66], Chain Replication [67], and Zab [68] now exist, with ongoing work to improve the situation further [69, 70]. Arguably, part of Google's early success was in spending the time to build their internal Paxos-as-a-service distributed lock system, Chubby [3]. Most of Google's famous early internal data storage tools, such as Bigtable [71], depend on Chubby for correctness. Spanner [35] – perhaps one of the most incredible distributed databases in the world – is largely just two-phase commit on top of multiple Paxos groups.

Reliable distributed consensus algorithms have been game-changing for many applications requiring fault-tolerant storage. However, success has been much more mixed in the byzantine fault tolerant world.

[2] If differing consistency models are new to you, it may be worth reading about them in Kyle Kingbury's excellent tutorial [56]. 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 [57].

D.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.

There have been a number of attempts to solve the Byzantine fault tolerant distributed consensus problem. The field exploded after the release of Bitcoin [20], and is still in its early stages. Of note, we are particularly interested in PBFT [72] (Barbara Liskov again with the first solution out of the gate), Q/U [73], FaB [74] (but see [75]), Bitcoin, Zyzzyva [76] (but also see [75]), RBFT [77], Tangaroa [78], Tendermint [79], Aliph [80], Hashgraph [81], HoneybadgerBFT [82], Algorand [83], Casper [84], Tangle [85], Avalanche [86], PARSEC [87], and others [88]. 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 [72] 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 [20] 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.

D.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 [70] 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 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.

E. 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.

E.1 Spartacus

Spartacus attacks, or identity hijacking, are possible on unmodified Kademlia [6]. Any node may assume 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 attack mitigation is addressed in S/Kademlia [28] by implementing Node IDs as public key hashes and requiring messages to be signed. 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.

E.2 Sybil

Sybil attacks involve the creation of large amounts of nodes in an attempt to disrupt network operation by hijacking or dropping messages. Kademlia [6] is already somewhat resistant to Sybil attacks, because it relies on both message redundancy and a concrete distance metric. A node's neighbors in the network are selected by node ID from an evenly distributed pool, and most messages are sent to at least k neighbors. If a Sybil attacker controls 50% of the network, it successfully isolates only 12.5% of honest nodes. While reliability and performance will degrade, the network will still be functional unless a large portion of the network consists of colluding Sybil nodes.

As an additional defense against sybil attacks, S/Kademlia [28] extends Kademlia with a proof of work scheme, which we have adopted. See section 4.3 for more details.

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 4.14, prevents a large influx of new Nodes from taking incoming data from existing reputable Storage Nodes without first proving their longevity.

E.3 Eclipse

An eclipse attack attempts to isolate a node or set of node 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, signatures based on those public keys, and multiple disjoint network lookups as prescribed by S/Kademlia.

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, and multiple disjoint network lookups ensure that Kademlia routing is prohibitively expensive to bias.

To avoid an eclipse attack, all that remains is to make sure new Nodes are appropriately introduced to at least one well-behaved Node on the network during the bootstrapping process. To that end, Storj Labs will run some well-known, verified bootstrap nodes.

E.4 Honest Geppetto

In this attack, the attacker operates a large number of "puppet" storage nodes on the network, accumulating trust and contracts over time. Once a certain threshold is reached, he 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.

E.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.

E.6 Cheating Storage Nodes, clients, or Satellites

The bandwidth allocation protocol minimizes the risk for client and Storage Nodes. The client can only interact with the Storage Node by sending a signed restricted allocation. The restriction limits the risk to a very low amount. 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.

E.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.

E.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 [42] 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.

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] 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.

[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] Backblaze Inc. How long do hard drives last: 2018 hard drive stats. https://www.backblaze.com/blog/hard-drive-stats-for-q1-2018/, 2018.

[6] 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, UK, 2002. Springer-Verlag.

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

[8] 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.

[9] 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.

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

[11] Daniel Abadi. Consistency tradeoffs in modern distributed database system design: Cap is only part of the story. *Computer*, 45(2):37–42, February 2012.

[12] 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.

[13] 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.

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

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

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

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

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

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

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

[21] 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.

[22] 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.

[23] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56:74–80, 2013.

[24] Peter Wuille. BIP32: hierarchical deterministic wallets. https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki, 2012.

[25] 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.

[26] Shawn Wilkinson. SIP Purpose and Guidelines, (2016). https://github.com/storj/sips/blob/master/sip-0001.md.

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

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

[29] Google Inc. What is gRPC? https://grpc.io/docs/guides/index.html.

[30] Arvid Norberg. uTorrent transport protocol. http://www.bittorrent.org/beps/bep_0029.html, 2009.

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

[32] 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.

[33] Amazon Inc. Amazon simple storage service - object metadata. https://docs.aws.amazon.com/AmazonS3/latest/dev/UsingMetadata.html.

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

[35] 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.

[36] Matteo Zignani, Sabrina Gaito, and Gian Paolo Rossi. Follow the "Mastodon": Structure and Evolution of a Decentralized Online Social Network. *International AAAI Conference on Web and Social Media*, 2018.

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

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

[39] 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.

[40] 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/storj.pdf, 2016.

[41] 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 Berlin Heidelberg.

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

[43] Zied Trifa and Maher Khemakhem. Sybil nodes as a mitigation strategy against sybil attack. *Procedia Computer Science*, 32:1135 – 1140, 2014. The 5th International Conference on Ambient Systems, Networks and Technologies (ANT-2014), the 4th International Conference on Sustainable Energy Information Technology (SEIT-2014).

[44] Shawn Wilkinson and James Prestwich. Bounding sybil attacks with identity cost, (2016). https://github.com/Storj/sips/blob/master/sip-0002.md.

[45] Michael Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Trans. Parallel Distrib. Syst.*, 12(10):1094–1104, October 2001.

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

[47] Rakesh Agrawal, Jerry Kiernan, Ramakrishnan Srikant, and Yirong Xu. Order preserving encryption for numeric data. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, pages 563–574, New York, NY, USA, 2004. ACM.

[48] S. S. Moghadam, G. Gavint, and J. Darmonti. A secure order-preserving indexing scheme for outsourced data. In *2016 IEEE International Carnahan Conference on Security Technology (ICCST)*, pages 1–7, Oct 2016.

[49] Alexandra Boldyreva, Nathan Chenette, and Adam O'Neill. Order-preserving encryption revisited: Improved security analysis and alternative solutions. In Phillip Rogaway, editor, *Advances in Cryptology – CRYPTO 2011*, pages 578–595, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[50] Craig B Gentry. Fully homomorphic encryption, July 14 2015. US Patent 9,083,526.

[51] Frédéric Giroire, Julian Monteiro, and Stéphane Pérennes. Peer-to-peer storage systems: A practical guideline to be lazy. 12 2010.

[52] 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.

[53] 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.

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

[55] 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.

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

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

[58] Amazon Inc. Amazon simple storage service - data consistency model. https://docs.aws.amazon.com/AmazonS3/latest/dev/Introduction.html.

[59] 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.

[60] 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.

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

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

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

[64] 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.

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

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

[67] 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, pages 7–7, Berkeley, CA, USA, 2004. USENIX Association.

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

[69] 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.

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

[71] 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.

[72] 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.

[73] 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.

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

[75] 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.

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

[77] 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.

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

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

[80] 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.

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

[82] 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.

[83] 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.

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

[85] Serguei Popov. The tangle. https://iota.org/IOTA_Whitepaper.pdf, 2018.

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

[87] 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.

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