Storj: A Decentralized Cloud Storage Network Framework

Storj Labs, Inc.

https://github.com/storj/whitepaper

v3.0, August 23, 2018

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 many traditional data failures and outages while simultaneously increasing security and privacy. Decentralization also allows market forces to optimize on cheaper ways to provide storage at a greater rate than any single entity could afford. While 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 building our distributed storage network. Additionally, we describe an initial concrete implementation for the entire framework.

TODO Remaining big todo items:

- Datascience: write an appendix about repair bandwidth based on node churn and uptime
- Datascience: Write an appendix about how we select RS numbers
- Eng/marketing: Quality control/branding section - talk about how we plan to ensure satellite quality by quality control evaluations, formal partnerships, relationships, and not letting poor quality satellites use the brand.

Remaining medium todo items:

- Anyone: Clean up distributed consensus positioning in the Metadata section.
- Eng: Describe how path encryption works
- Eng: Describe payer (satellite) ids and why they're needed
- Eng: General data repair framework description
- Eng: more macaroon details
- Eng: future work section about hot files
- get a better latex theme

Remaining little todo items:

- Eng: Write a future work section about eliminating TLS handshakes
- Eng: Write about how piece ids are generated
- Eng: Future work section about reputation sharing
- Eng: Future work section about finding a better data structure than a bloom filter for gc

1 Introduction

TODO Outline of this intro (and the entire paper) should be:

- Decentralized storage solves some big problems.
- Decentralized storage addresses a specific market we're interested in.
- To best address this market, our product must satisfy specific constraints.
- We believe a specific framework is required to best satisfy these constraints. We will not need a v4 as this framework is fairly fundamental to our space.
- Our concrete implementation is just one potential solution within our framework. We plan to improve our implementation within the framework over time.
- We have made a product out of these concrete framework choices.

Decentralized storage has emerged as a potential solution to the challenge of providing an economical cloud storage solution. It can address the rapidly expanding amount of data for which current solutions are cost-prohibitive. 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, we are convinced that decentralized cloud storage has the potential to address several key segments within that market today, particularly related to long term archival storage. As the capabilities within decentralized storage mature, decentralized cloud storage will be able to address a much wider range of use cases from basic object storage to content delivery networks (CDN).

There are a handful of storage-based companies emerging that aim to create vast networks to store data. Storage products such as Wuala, Allmydata, Tahoe-LAFS, Space Monkey, Sia, Maidsafe, Filecoin, Crashplan, Mozy, Hadoop Distributed File System (HDFS), Amazon S3, Google File System (GFS), and others believe that a single computer is not as powerful or secure as these networks. While all the above mentioned companies share a common goal of building robust and secure networks, they also share common principles of how this can be achieved. Companies generate redundancy for data in case of system failures, store this redundancy in locations with varying degrees of failure isolation, and then keep track of where the data was placed. However, the technical implementations, designs, and other considerations vary greatly with each organization.

Decentralized cloud storage is still rapidly advancing through the maturity cycle to address these challenges, but its evolution is subject to a specific set of design constraints which define the overall requirements and implementation of the network software components. When designing a distributed storage system, there are many parameters to be optimized: speed, capacity, trustlessness, byzantine fault tolerance, cost, etc. But unlike optimizing cost or speed, data needs to be persistent throughout the network with no loss. One way to achieve this goal is to have nodes communicate with each other - creating functional systems that are seamlessly integrated to store and retrieve data.

We propose a framework that scales horizontally serving hundreds of millions of people across the globe. Our system–the Storj Network–is a robust object store that encrypts, shards, and distributes data to our global community for storage. Data is served with low latency in a manner purposefully designed to prevent breaches that have continuously plagued many companies. 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 with each other to implement a secure, robust, and reliable storage system.

We organized the rest of this paper into five sections. Section 2 discusses the design space in which Storj operates and elaborates on specific design constraints which we are focusing our optimization efforts on. Section 3 covers our framework, while 4 proposes a simple concrete implementation of each component. Section 5 covers specific details about how we will deliver this implementation to users. Finally, Section 6 covers future areas of research.

2 Storj design constraints and considerations

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

2.1 S3 compatibility

At the time of publishing this paper, the flagship cloud storage product is Amazon's Simple Storage Service, or S3 for short. Most other currently available-cloud storage products provide some form of compatibility with the S3 API.

Until a decentralized cloud storage protocol becomes the *lingua franca* of storage protocols, we should create a graceful transition path from centralized providers for our users. This will alleviate any switching costs for users with data currently stored on a centralized provider. Our objective is for Storj to compete successfully in the wider cloud storage industry and bring decentralized cloud storage into the mainstream, enabling more people to enjoy greater security with less centralized control. To achieve this, the Storj implementation should allow applications previously built against S3 be adjusted to work with Storj with minimal friction or changes. This adds strong requirements on feature set, performance, and durability.

2.2 Device failure and churn

For all storage systems, but especially distributed ones, component failure is a guarantee. All hard drives fail after enough wear [1], and servers providing network access to these hard drives will eventually fail, too. Network links may die, power failures could cause havoc sporadically, and storage media become unreliable with time. For data to outlast individual component failures, data must be stored with enough redundancy to recover from these failure scenarios. Perhaps more importantly, no data should be assumed to be stationary as all data must eventually be moved. In such an environment, redundancy, data maintenance, repair, and replacement of lost redundancy must be considered facts of life, 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 such a network will fail to scale. As a result, a scalable, highly durable storage system must give preference to stable nodes and endeavor to keep the churn rate as low as possible.

Churn rates affect 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 [2]. 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 to minimize churn.

See Appendix C for a discussion about how repair bandwidth varies as a function of node churn.

2.3 Latency

A decentralized, distributed storage system has the potential to capitalize on massive opportunities for parallelism in transfer rates, processing, and a number of other factors. Parallelism by itself is a great way to increase overall throughput even when individual network links are slow. However, parallelism cannot by itself improve *latency.* If an individual network link has fixed latency and is a required part of an operation, the latency of the required network link will be the lower bound for the overall operation. Therefore, a distributed system intended for high performance applications must aggressively optimize for low latency, both at the individual process scale and at the overall architecture scale.

In addition, we emphasize an architectural strategy aimed at achieving low latency by focusing on eliminating the need to wait for long tails [3]. 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 turns what could be a potential liability (highly variable performance from individual actors) into a great source of strength for a distributed storage network.

2.4 Bandwidth

Global bandwidth availability is increasing year over year; however, access to high-bandwidth internet connections is unevenly distributed in different parts of the world. While users in some countries can easily access symmetric, high-speed, unlimited bandwidth, users in other countries may have significant difficulty in obtaining access to the same.

In the United States, the way many residential internet service providers (ISPs) operate presents two specific challenges for designers of a decentralized network protocol. The first challenge is that the internet connection is often asymmetric. 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 traffic per month. For example, in many US markets, Comcast imposes a one terabyte per month bandwidth cap with stiff fines for customers who go over this limit. 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 cap, even if the internet connection advertises speeds of 10 MB/s or higher. Such caps impose significant limitations on the bandwidth available at any given moment.

With device failure and churn guaranteed, any decentralized system will have a corresponding amount of repair traffic. It is therefore important to make sure there is enough headroom for the bandwidth required for data maintenance, over and above that required for data storage and retrieval. Designing a storage system that is careless with bandwidth usage would be to give undue preference to storage providers with access to unlimited high-speed bandwidth, recentralizing the system to some degree. In order to keep the storage system as decentralized as possible while working in as many environments as possible, bandwidth usage must be aggressively minimized.

Please see Appendix TODO for a discussion on how available bandwidth combined with required repair traffic limits usable space.

2.5 Security and privacy

One of the primary focuses of the Storj network is to ensure that its users data privacy is protected. This is not addressed as an after thought but is built into the design of the clients, network and other related services.

Any object storage platform must ensure both the privacy and security of data stored. Distributed storage platforms include an additional layer of complexity and risk associated with the storage of data on inherently untrusted nodes.

First and foremost, the object storage platform must address the security of the data stored. Data security encompasses a wide range of concerns including identity and access management, prevention of tampering or unauthorized modification, prevention of ransomware attacks, system vulnerabilities, and malicious insiders.

Separate but related from security issues are privacy concerns. Privacy-related issues considered for purposed of the design constraints include storage of data without providing access to the underlying data and its metadata by Storj Labs, third-party operators of system components, or malicious actors.

2.6 Object size

Large storage systems can broadly be classified into two groups by average object size. When storing a large amount of small pieces of information, a database is generally the preferred solution. On the other hand, when storing many large files, an object store or filesystem is ideal. We classify a "large" file as a few megabytes or greater in size.

The initial product offering by Storj Labs is designed to function primarily as an object store for larger files. While future improvements may enable database-like use cases, the predominant use case described in this paper is object storage. Protocol design decisions are made with the assumption that the vast majority of objects stored will be a couple of megabytes or larger.

It is worth pointing out that this will not negatively impact use cases that require reading lots of files smaller than a megabyte. Such cases can be addressed with a packing strategy, where many small files are aggregated and stored together as one large file. As the protocol supports seeking and streaming, small files can even be downloaded without requiring full retrieval of the aggregated object they were packed into.

2.7 Decentralization

TODO why even try for decentralization?

2.8 Byzantine Fault Tolerance

Unlike datacenter-based 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 BAR (Byzantine, Altruistic, Rational) model [4] 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 to do so.

Some distributed storage systems (e.g. Amazon S3) operate in an environment where all nodes are altruistic. Amazon owns all of their nodes directly and rogue operations engineers are not generally coopting node behavior for their own purposes. On the other hand, 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. Any potential implementation must account for this distinction.

2.9 Coordination contention

TODO explain why avoiding coordination is good, coordination contention is bad, and reference calm conjecture [5, 6], i-confluence [7], hat [8], and anna [9]

2.10 Marketplace and economics

TODO john g

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 replaced without affecting other components.

3.1 Framework overview

At a high level, every design within our framework will do the following things:

Store data When data is stored with the network, a client will encrypt it and break it up into multiple pieces. It will distribute the pieces to peers in the network and generate and store some metadata about where to find the data again.

Retrieve data When data is retrieved from the network, the client will first recover the metadata about where to find the pieces. Then the pieces will be retrieved and the original data will be reconstructed on the client's local machine.

Maintain data Data is maintained in the network by replacing missing pieces when the amount of redundancy drops below a certain threshold. The data is reconstructed, and the missing pieces are regenerated and replaced.

Pay for usage A form of currency should be sent in exchange for services rendered.

To make this framework feasible while satisfying our design constraints, we will need to solve a number of complex challenges. Inspired by Raft [10], we break up the design into a collection of relatively independent concerns 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 most fundamental part of this network is the storage node. 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, disk space, geographic location, uptime, history of responding accurately to audits, etc. In return for their service, nodes are rewarded for their participation via payments.

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 [11]. This decision implies the requirement of a separate metadata storage system to keep track of selected nodes.

3.3 Peer-to-peer communication and discovery

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

- provides peer reachability, even in the face of firewalls and NATs. This may require techniques like STUN, UPnP, NAT-PMP, etc.
- provides authentication, where each participant knows exactly the identity of the peer with whom they are speaking to avoid man-in-the-middle attacks.

- provides privacy, where only the two peers know what transfers between them.

Additionally, the framework requires a way to look up peer network addresses by node ID so that, given a peer's network address, any other peer can connect to it. This responsibility is similar to the service the internet's standard domain name system (DNS) provides, which is a mapping of an identifier to an ephemeral connection address. To achieve this, a network overlay can be built on top of our peer-to-peer communication protocol that provides this functionality. See Section TODO ref for implementation details.

3.4 Redundancy

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 offline. To achieve a certain level of *durability* (the probability that data will remain available in the face of failures), many products in this space use simple replication. Unfortunately, this ties durability to the network *expansion factor*, which is the storage overhead for reliably storing data.

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.

Instead, *erasure codes* are a more general and flexible scheme for manipulating data durability without tying it to bandwidth usage. Importantly, erasure codes allow changes in durability without changes in expansion factor!

TODO fix flow:

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.

With the simplifying assumption that p is the monthly node birth/death TODO pick one rate (the proportion that describes the number of nodes leaving/joining the network per month), we can model file durability as the 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:

$$P(D) = e^{-\lambda} \sum_{i=0}^{n-k} \frac{\lambda^i}{i!}. \tag{1}$$

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, eliminating replication as a strategy entirely and using erasure codes only for redundancy causes a drastic decrease in bandwidth footprint. It further causes a significant increase in the funds available per byte to storage nodes due to the decreased dilution of incoming funds to storage node payment relative to larger expansion factors.

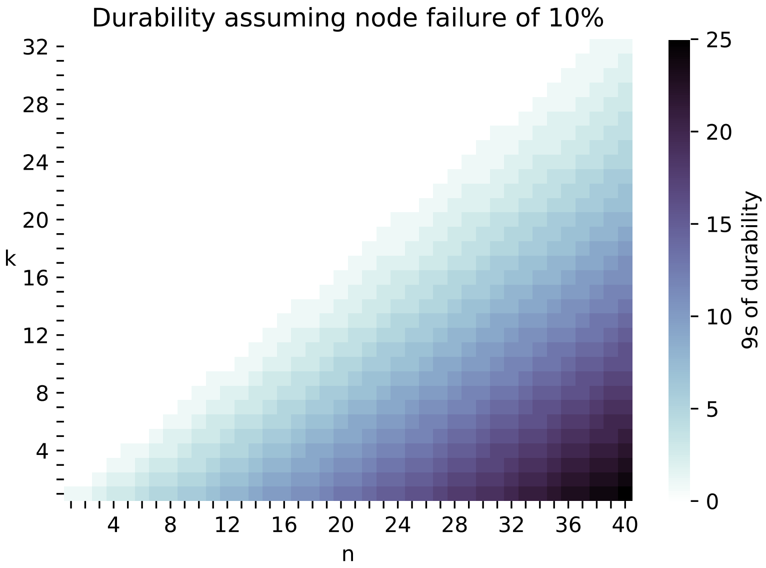
k	n	Exp. factor	P(D)
2	4	2	99.207366813274616391%
4	8	2	99.858868985411326445%
8	16	2	99.995462406878260756%
16	32	2	99.999994620652776179%
32	64	2	99.999999999990544376%

Table 1: $P(D)$ for various choices of k and n, assuming $p = 0.1$.

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 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 the structured file storage section for more details. add reference to the section

TODO fix graphic title to make more english sense



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 [3]. 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 cancelled, allowing the upload to continue as fast as the fastest nodes in a set, instead of waiting for the slowest nodes. Downloads are similarly improved. Since more redundancy exists than is needed, downloads can be served from the fastest peers, eliminating a wait for temporarily slow or offline peers.

3.5 Metadata

Once we split an object up and selected storage nodes on which to store the new pieces, we now must keep track of which storage nodes we selected. Moreover, to maintain 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.

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 on. Metadata values should be able to be stored, retrieved, and removed by arbitrary key, and deterministic iteration over those keys will also 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 sizeable.

To provide some examples, 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 such that $n = 40$. This metadata will need to keep track of which 40 nodes were selected for each object. 1 exabyte of 50MB objects is 20 billion objects. If each metadata element is roughly 40*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 user. A user storing a 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.

Aside from scale requirements, the desired API is straightforward and simple: `Put` (store a pointer given a path), `Get` (retrieve a pointer given a path), `List` (paginated, deterministic listing of existing paths), and `Delete` (remove a path).

One of our framework's focuses is enabling this component – metadata storage – to be interchangeable per user. Specifically, we expect to ship with multiple implementations of metadata storage that users will be allowed to choose between. TODO reference i-confluence and coordination minimization

3.6 Encryption

Regardless of storage system, our design constraints require total security, so any such 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 S3-compatible interface or appropriate similar client library should run colocated on the same computer as the user's application.

Ideally encryption uses 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, such that users can 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 way that is secure and reliable. This metadata will be stored in the previously discussed metadata storage system, itself encrypted by a deterministic, hierarchical encryption scheme. A hierarchical encryption scheme similar to BIP32 [12] will allow subtrees to be shared without sharing their parents, and will allow some files to be shared without sharing other files.

Like all other metadata, paths themselves can be encrypted using a hierarchical encryption scheme. TODO ref to hd scheme section

3.7 Audits and Reputation

Incentivizing storage nodes to accurately store data is of paramount importance to the viability of this whole system. As such, it is important 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 to do repair and which files to repair. Our storage system does not. In our storage system, audits are simply a mechanism by which a node's degree of stability is determined. Failed audits will result in marking a storage node as bad, which could result in shuffling data to new nodes and avoiding that node altogether in the future. File repair needs are detected via another mechanism.

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, is keeping the data it claims, and is not susceptible to hardware failure or malintent. An audit functions as a spot check to help calculate a storage node's future usefulness.

This partial auditing mechanism does not audit all bytes in all files and leaves room for false positives, where the verifier believes the storage node retains the intact data, when it has actually been modified or partially deleted. Fortunately, the probability of a false positive on an individual partial audit is easily calculable (see Section TODO). When applied iteratively to a storage node as a whole, detection of unexpected behavior becomes certain to within a known and modifiable error threshold. We are extending the probabilistic nature of common per-file *proofs of retrievability* [13] to range across all files.

TODO discuss reputation

3.8 Data repair

An ever-present risk in any distributed storage system is file loss. While there are many potential causes for file loss, storage node churn is the leading risk, as evidenced by the findings of Maymonkov et al. that expected node availability is an increasing function of uptime [2]. Storage nodes may go offline due to hardware 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 bad and repair at-risk data.

We're taking a huge shortcut with the assumption that probabilistic audits are enough for us to estimate the likelihood that a node will have the data it should have; we can use that along with node uptime (which is much more efficient than audits) to calculate when a file is at risk. We *only* consider *node* availability and configured repair thresholds when determining which *files* to repair.

There are many other ways data might get lost in the network besides node churn: corruption, malicious behavior, bad hardware, software error, user space reclaimation, etc., but these issues are less serious than full node churn (power loss, internet connectivity intermittency, software shutdown or removal). Our spot-check-based audits will incentivize storage nodes to reliabily 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, and we expect to account for varying amounts of other issues by configuring Reed-Solomon erasure code parameters according to differing network conditions.

3.9 Payments

Payments 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, one must avoid naively placing a blockchain-based solution in the storage hotpath. By this we mean 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 is a non-starter. TODO i-confluence

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 that well behaved participants in the network will cooperate in. Storage nodes in our framework should limit their exposure to untrusted payers until confidence is gained that those payers are likely to pay their bills.

The Storj network is payment agnostic. The protocol does not require a specific payment type. The network assumes STORJ as the default payment medium, but many other payment types could be implemented, including Bitcoin, Ether, ACH transfer, or physical transfer of live goats.

4 Concrete implementations

The framework we've described above we believe to be relatively fundamental given our design constraints. However, within the framework there remains a significant amount 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.

As with our previous network, we will publish changes to this concrete architectures through our Storj Improvement Proposal process [14].

4.1 Definitions

Client A user that would like to 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. Peer classes tend to be run separately by different operators.

Uplink This peer class represents any application or service that wants to store data. Applications can store data via the 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 between the other peer classes on behalf of the customer.

Storage Node This peer class participates in the DHT, stores data for others, and gets paid for storage and bandwidth (via a bandwidth allocation protocol).

Satellite This peer class participates in the DHT, caches DHT lookups, stores per-object metadata, keeps storage node reputation, pays storage nodes, performs audits and repair, and manages authorization and user accounts. Any user can run their own satellite, but we expect many users will elect to avoid the operational complexity and create an account on another satellite hosted by a trusted party like a friend, group, or workplace.

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

Path A `path` is a unique identifier for a `file` within a `bucket`. A `path` is a string of UTF8 codepoints that begins with a forward slash and ends with something besides a forward slash. More than one forward slash (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/Object A `file` (or `object` or `stream`) is an ordered collection of 0 or more `segments`. `segment`s have a fixed maximum size, so the more bytes the `file` represents through `segment`s, the more `segment`s there are.

A `file` also support a limited amount of key/value user-defined fields to support extended attributes.

Like `path`s, the data contained in a `file` is encrypted before it ever leaves the client computer.

Segment A `segment` represents a single array of bytes, between 0 and a user-configurable maximum `segment` size. Breaking large `file`s into multiple `segment`s provides a number of security and scalability advantages.

Inline Segment An `inline segment` is a `segment` that is small enough it makes sense to store it "inline" with the metadata that keeps track of it, such as a `pointer`.

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 book keeping.

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 `stripe`s 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 `segment` is a `remote segment`, its `stripe`s will get erasure encoded. When a `stripe` is erasure encoded, it generates multiple pieces called `erasure share`s. Only a subset of the `erasure share`s 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 `stripe`s are erasure encoded into `erasure share`s, the `erasure share`s for that `remote segment` with the same index are concatenated together, and that concatenated group of `erasure share`s is called a `piece`. If there are n `erasure share`s after erasure encoding a `stripe`, there are n `piece`s after processing a `remote segment`. The ith `piece` is the concatenation of all of the ith `erasure shares` from that `segment`'s `stripe`s.

Pointer A `pointer` is a data structure that keeps track of which `piece storage nodes` a `remote segment` was stored on, or the `inline segment` data directly if applicable.

4.2 Storage Node

The main duty of a storage node is to reliably store and return data. Storage node operators are individuals or entities that have excess hard drive space and want to earn compensation for lending their space to others. Storage node operators will download, install, and configure Storj software locally, with no account required anywhere. Storage node operators will select what disk space and bandwidth usage is allowed during configuration. Storage nodes will advertise during DHT communications what hard drive space is still available, how much bandwidth is available, and what their desired STORJ token wallet address is.

Because Storj is optimized for larger files, storage nodes have no reason to do anything more complex than store `pieces` directly on disk. As a result, unlike the previous release of Storj that used KFS [15], Storj no longer has a restriction on the maximum amount of data a storage node can store.

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 was 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 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 [16] database.

Storage nodes can choose which satellites to work with. If storage nodes 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 returning data when requested in the form of egress bandwidth payment. Bandwidth payment is made payable after the storage node sends in signed bandwidth allocation messages. Storage nodes are also paid for data at rest. Storage nodes are expected to reliably store all data sent to them and are paid with the assumption that they are faithfully doing so. 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. Storage nodes are not paid for DHT or other maintenance traffic.

Storage nodes will support three methods: `get`, `put`, and `delete`. They store *pieces*. Each method will take a *piece ID*, a *payer ID* (the ID of the associated Satellite instance) and signature, an optional TTL, and the other metadata required by the bandwidth allocation protocol TODO clean up the list of arguments, add ref to bap .

The `put` operation will take a stream of bytes 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.

The *payer ID* forms a namespace. An identical *piece ID* with a different *payer ID* refers to a different *piece*.

Storage nodes should allow administrators to configure maximum allowed disk space usage and maximum allowed bandwidth usage over the last rolling 30 days. They should keep track of how much is remaining of both, and reject operations that do not have a valid signature from the appropriate payer.

4.3 Satellite

As should be apparent, the data owner has to shoulder significant burdens to maintain availability and integrity of data on the Storj network. Because nodes cannot be trusted, data owners are responsible for selecting good storage nodes, issuing and verifying audits, providing payments, managing file state and object metadata, etc. Many of these functions require high uptime and significant infrastructure, especially for an active set of files. User run applications, like a file syncing application, cannot be expected to efficiently manage files on the network.

To enable simple access to the network from the widest possible array of client applications, Storj implements a thin-client model that delegates trust to a dedicated server that manages data ownership. The burdens of the data owner can be split across the client and the server in a variety of ways. This sort of dedicated server, called the satellite, has been developed and released as Free Software. Any individual or organization can run their own satellite to facilitate network access.

With respect to customer data, the satellite is designed to store only metadata. It 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 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.

In cases where the cost of delegating trust is not excessively high, clients may use third-party satellites. Because satellites do not store data and have no access to keys, this is still 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. 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.

Applications using object stores delegate significant amounts of trust to the storage providers. Providers may choose to operate public satellites as a service. Application developers then delegate trust 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. This shifts significant operational burdens from the application developer to the service-provider. This would also allow developers to pay for storage with standard payment mechanisms, like credit cards, rather than managing a cryptocurrency wallet. Storj Labs Inc. currently provides this service.

A specific satellite *instance* does not necessarily constitute one server. A satellite may be run by a collection of servers and be backed by a horizontally scalable trusted database for higher uptime.

The satellite is, at its core, one of the most complex and yet straightforward components of our initial release that fulfills our framework. Future framework-conforming releases nonwithstanding, the initial satellite is a standard application server that wraps a trusted database such as PostgreSQL, Cassandra, or something else. Users sign in to a specific satellite with account credentials. The satellite is responsible for keeping track of accounts and authorization, storage node contact information and reputation, and object metadata. The satellite is also responsible for payments and data repair. Data available through one satellite instance is not available through another satellite instance, though various levels of export and import are planned.

The satellite is made up of components discussed earlier:

- A full DHT cache.
- An account management and authorization system
- A per-object metadata database indexed by encrypted path
- A reputation and storage node statistics database
- A storage node payment service
- A data audit and data repair service

4.4 Uplink

The uplink provides an S3-compatible drop-in interface for 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 (via libstorj) first encrypts data, erasure encodes it, then streams it out to storage nodes, all while coordinating with a chosen satellite for metadata and tracking.

The uplink should run co-located with wherever data is generated, and will communicate directly with storage nodes so as to avoid central bandwidth costs.

4.5 Peer-to-peer communication

Initially, we'll be using the gRPC [17] protocol on top of of the Transport Layer Security protocol (TLS) on top of the μTP [18] 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'll be replacing TLS to reduce round trips due to connection handshakes in situations where the data is already encrypted and forward secrecy isn't necessary. TODO See the Future Work section for more details. Gateways will be provided in appropriate places that allow for interoperability with web browsers.

Each node will operate its own certificate authority, which requires a public/private keypair and a self-signed certificate. The certificate authority's private key should 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 as key rotation for the certificate authority will require a brand new node ID.

The *node ID* will be determined from the certificate authority by hashing the DER-encoded public key.

As in S/Kademlia [19], 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 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 [2] tree.

Each node will also have revokable leaf key pair and certificate, signed by the node's certificate authority. Nodes will use this leaf keypair for actual communication. Each leaf will have a signed timestamp that Satellites should keep track of per node. Should the leaf become compromised, the node can issue a new leaf with a later timestamp. Interested peers should make note of newly seen leaf timestamps and reject connections from nodes with earlier leaves.

When using TLS, 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. TODO ref for future work

For the few cases where a node cannot achieve a successful hole punch through a NAT or firewall via STUN, uPnP, NATPmP, or a similar technique, manual intervention and port forwarding will be required. In the future, nodes unable to punch a port through their firewalls may rely on traffic proxying from other, more available nodes, for a fee. See the bandwidth allocation protocol TODO ref for a description of how fees work.

4.6 Node discovery

At this point, we have storage nodes and we have a means to communicate with them if we know their address. We must account for the fact that storage nodes will often be on consumer internet behind routers with constantly changing IP addresses, so the overlay network'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 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. Only using Kademlia for node lookup eliminates the need for some other features Kademlia would otherwise require, such as owner-based key republishing, neighbor-based key republishing, store and retrieval of values, etc. Furthermore, we avoid a number of known attacks by using the S/Kademlia [19] 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 TODO future work . The caching service will then cache the last known good address for each node, evicting nodes that it hasn't talked to after a certain period. Fortunately, caching address information for an entire network of 80k nodes (for example) can be done with 3MB of memory, so the space requirements are negligible.

Based on this design, the cache should not be expected to be a primary source of truth and results in the cache may be stale. Luckily, due to redundant storage, the storage network will be resilient against an expected degree of node churn, so the system will be robust even if some lookups in the the cache fail or return incorrect addresses. Further, because our peer-to-peer communication system already provides peer authentication, an overlay network cache that sometimes returns faulty or perhaps 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 can be used directly if correctness is valued more highly than performance by certain customer classes.

TODO talk about how we're using kademlia to advertise disk space and bandwidth availability, and then storing that information in the cache

4.7 Redundancy

We use the Reed-Solomon erasure code [21]. For each object that we store we choose 4 numbers, k, m, o, and n, such that $k \leq m \leq o \leq n$. k and n are the standard Reed Solomon numbers, where k is the minimum required number of pieces for reconstruction, and n is the total number of pieces generated during creation.

m and o are the *minimum safe* and *optimal* values, respectively. m is chosen such that if the amount of available pieces falls below m, a repair is triggered immediately in an attempt to make sure we always maintain k or more pieces. o is chosen such that during uploads, as soon as o pieces have finished uploading, remaining pieces up to n are canceled as described above. 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 would be excess durability. $n - o$ is the amount of long tail uploads we can tolerate and is thus the amount of slow nodes we are immune to.

Our durability story does not end with our selection of these numbers. Please see section 4.13 for a discussion about how we repair data as its durability drops over time.

See Appendix C for how we select our Reed-Solomon numbers.

TODO future work: better erasure code algorithm than rs

4.8 Structured file storage

4.8.1 Files

Many applications benefit from being able to keep metadata alongside files. For example, NTFS supports ”alternate data streams” for each file, HFS supports resource forks, EXT4 supports ”extended attributes,” and more importantly for our purposes, AWS S3 supports ”object metadata” [22]. Being able to support arbitrarily named sets of keys/values dramatically improves compatibility with other storage platforms. Every `file` will support a limited set of arbitrary key/value metadata to support object metadata.

4.8.2 Files as Segments

`Files` should be designed both for small data and large data. A `file` may be small enough it consists of only one segment. If that `segment` is smaller than the metadata required to store it on the network, the `segment` will be an `inline segment` and the data will be stored directly inline with the metadata.

For larger `files`–`files` past a certain size–the data will be broken into multiple large `remote segments`. Segmenting in this manner has a number of advantages to security, privacy, performance, and availability.

Maximum `segment` size is a configurable parameter. To preserve privacy, it is recommended that `segment` sizes be standardized as a byte multiple, such as 8 or 32 MB. Smaller `segments` may be padded with zeroes or random data. Standardized sizes help frustrate attempts to determine the content of a given `segment` and can help obscure the flow of data through the network.

Segmenting large files like video content and distributing the `segments` across the network separately reduces the impact of content delivery on any given node. Bandwidth demands are distributed more evenly across the network. In addition, the end-user can take advantage of parallel transfer, similar to BitTorrent or other peer-to-peer networks. Last, capping the size of segments allows for more uniform storage node filling–a node must only have enough space to store a segment to participate in the network, and clients don't have to find nodes that have enough space for their specific large file.

4.8.3 Segments as Stripes

In many situations it's important to access just a portion of a piece of data. Some large file formats such as large video files, disk images, or file archives support the concept of seeking, where only a partial subset of the data is needed for correct access or read operation. To support these uses, it's useful to be able to decode and decrypt only parts of a file.

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

4.8.4 Stripes as Erasure Shares

Erasure encoding gives us the chance to control network durability in the face of unreliable `piece storage nodes`. Erasure encoding schemes often are described as (k, n) schemes, where k `erasure shares` are needed for reconstruction out of n total. For every `stripe`, n `erasure shares` are generated, where the network has an expansion factor of n/k.

For example, let's say 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 $\frac{1}{20}$th the size of the original `stripe`.

All n `erasure shares` have a well defined index associated with them. More specifically, for any given n, the ith share of an erasure code will always be the same.

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

4.8.5 Erasure Shares as Pieces

Because `stripe`s are already small, `erasure share`s are often much smaller, and the metadata to keep track of all of them separately would be immense relative to their size. Instead of keeping track of all of the erasure shares separately, we pack all of the `erasure share`s together into a few `piece`s. In a (k, n) scheme, there are n `piece`s, where each `piece` i is the ordered concatenation of all of the `erasure share`s 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 `piece`s are needed to recover the full `segment`.

TODO piece ids are generated as the hmac of a root piece id and the storing node id

4.8.6 Pointers

The data owner will need knowledge of how a `remote segment` is broken up and where in the network the `piece`s are located to recover it. This is contained in the `pointer` data structure, and the owner can secure the `pointer` as they wish.

TODO code-based pointer definition. protobuf?

4.8.7 Paths

TODO

4.9 Metadata

The most trivial implementation for the metadata storage functionality we require would be to simply have each user use their preferred trusted database such as PostgreSQL, SQLite, MongoDB, Cassandra [23], Spanner [24], 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 this punt-to-the-user approach, however, 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 the availability can be made arbitrarily good with existing trusted distributed solutions such as Cassandra, Spanner, or CockroachDB. Further, any individual metadata service downtime does not affect the entire 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 anyway, but a punt-to-the-user solution increases the risk area from just encryption keys considerably. 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:

- **Use cases** - even in a catastrophic scenario, this design still covers all required use cases.
- **Control** - the user is in complete control of all of their data. There is still no organizational single point of failure. The user is free to choose whatever metadata store with whatever tradeoffs they like. Like Mastodon [25], this solution is still decentralized. Further, 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. We can get a useful product to market without doing this work at all. This is a considerable advantage.

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 TODO about why we've chosen to avoid trying to solve the problem of Byzantine distributed consensus for now. See Future work TODO for a discussion of medium to long term plans.

4.10 Encryption

Our encryption is authenticated encryption, with support for either the AES-GCM cipher and the Salsa20 and Poly1305 combination NaCl calls "Secretbox" [26]. Authenticated encryption is used so that the user can know if the data has been tampered with. Encryption keys are chosen by clients randomly.

Data is encrypted in small batches of `stripes`, recommended to be 4KB or less [27]. While the same encryption key is used for every `stripe` in a `segment`, `segments` may have different encryption keys. On the other hand, the nonce for each `stripe` batch must be monotonically increasing from the previous batch throughout the entire `stream`. The nonce wraps around to 0 if the counter reaches the maximum representable nonce. The first nonce is chosen at random and is stored with the `stream`'s metadata. TODO consider multipart upload

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. TODO explain path encryption scheme

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.

TODO describe path encryption

Path encryption is optional, as encrypted paths make efficient sorted path listing challenging. When path encryption is enabled (a per-bucket feature), objects are sorted by their encrypted path name, which is relatively unhelpful when interested in unencrypted paths. For this reason, users can opt in to disabling path encryption. When path encryption is disabled, unencrypted paths are only revealed to the user's chosen metadata storage system.

Order-preserving path encryption is left for future work.

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 should be able to add, remove, and edit files, while users who are not authorized should not be able to do so.

First, metadata operations should be authorized. Users should authenticate with their chosen metadata service, which should allow them given their authorization configuration access to various operations.

Our initial metadata authorization scheme uses macaroons [28]. Each account has a root macaroon and operations are validated against a supplied macaroon's set of caveats.

TODO discuss macaroon-based path restrictions

Once authorized with a metadata service, that metadata service has an associated *payer ID* TODO discuss payer IDs and is able to sign operations. All operations with storage nodes require a specific payer ID and associated signature. A storage node should reject operations not signed by the appropriate payer ID. The client must retrieve valid signatures from the metadata service prior to operations with storage nodes.

4.12 Audits

Some distributed storage systems (including the previous release of Storj [15]) discuss *Merkle tree proofs*, in which audit challenges and expected responses are generated ahead of time as a form of compact proof of retrievability [13]. By using a Merkle tree [29], the amount of metadata needed to store these pre-generated challenges and responses can be made to be negligible.

Unfortunately, in such a scheme, the challenges and responses must be pregenerated, and without a periodic regeneration of these challenges, a storage node can begin to pass most audits without storing all of the requested data.

We do something else. 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 [30]. We are already using Reed-Solomon erasure coding [21] on small ranges (`stripes`), so we use it to issue challenges and verify responses as well. This feature can be used for arbitrary audits without pregenerated challenges.

To perform an audit, we first choose a `stripe` to audit. We request that `stripe`'s `erasure shares` from all storage nodes responsible. We then run the Berlekamp-Welch algorithm [30] across all the `erasure shares`. When enough storage nodes return correct information, any faulty or missing response can easily be identified. 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, but 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. Audits should happen uniformly at random by byte with replacement.

4.13 Data repair

The node discovery system already has caches in place that have 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 in 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 should be downloaded and reconstructed, and the missing pieces should be regenerated and uploaded to new nodes. Finally, the `pointer` should be updated to include the new information.

As storage node 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. Once enough file pieces are lost, the satellite will download the remaining file pieces from their corresponding storage nodes, using those pieces to rebuild the file's missing, encrypted, erasure encoded pieces. Once the repair process is complete, the satellite will send the recovered pieces to new storage nodes.

Users will choose their desired durability with their chosen metadata service (which may impact price, among other things). This desired durability, along with statistics from ongoing audits, will directly inform what Reed-Solomon erasure code choices should be made for new and repaired files, and what thresholds should be set for when uploads are successful and when repair is needed. See Appendix TODO for how we calculate these things given user inputs.

A practical upshot 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 fall out of the network due to node churn. This is similar to the design of how value storing and republishing works in Kademlia [2].

The ingress bandwidth demands of the audit and repair system are large, but the egress demands are relatively small. A large amount of data comes in to the system for audits and repairs, but just 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 that offer free ingress make for an especially attractive hosting location for users of this system. We will describe a Distributed Repair method in the Future Works section TODO ref 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 doesn't matter. The data repair system just needs to get through 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. Furthermore, repair is still a costly operation due to significant bandwidth and CPU usage impacting a single operator, so repair work should be minimized. As a result, when repairing a segment, only the minimum number of pieces required should be downloaded. Unfortunately, this means that without full redundancy, erasure codes will be less effective at catching errors. Further, 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` should be stored in the system via a Merkle tree [29], storing the root of the tree in the `pointer`. This allows the repair system to correctly assess whether or not repair has 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) should be stored alongside each `piece` on each storage node, with the root in the `pointer`, such that the only additional central metadata storage required is just for the root.

Each repair should 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 reasonable trust TODO Can we eliminate the word trust here? between nodes where there would otherwise be none. 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 amount of 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 (potentially months), 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 [31], 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 [32].

The filtering system is the second subsystem, and blocks bad storage nodes from participating. Certain actions a storage node can take are disqualifying events, and 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 should be allowed to choose which data to store and which satellite operators to work with.

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. See Appendix TODO for why uptime is so important in our storage system.

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 [33], 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 should be stored, both during repair and during the upload of new files, unlink 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 should not accept future data from those payers. See the section TODO about quality control on how we plan to ensure payers 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 could still store data for other satellites. Reputation will not be shared between satellites. Over time, as we plan to eliminate satellites, reputation would then be determined globally.

TODO future work section about reputation sharing

4.15 Payments

In the Storj network, payments are made by uplink users 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 would 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. Moving forward, our network will not use contracts to manage payments and file storage durations.

Satellites will pay storage nodes for the data they store long-term 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 node hasn't been blacklisted, 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.

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 would eventually be cleaned up through the garbage collection process. Because of the way delete commands are issued, and because storage nodes are not expected to be online at all times, storage nodes may be storing file pieces that were slated for deletion. This is factored into the storage node payment amounts, meaning storage nodes are paid more than they should for the file pieces they store, offsetting the lost revenue due to storing garbage data. 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.

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 should be storing. The satellite also tracks file downloads in its database. 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 track utilized bandwidth through a bandwidth allocation protocol. To download a file, an uplink user connects to the satellite to identify where its file pieces are stored and to provide a promise to pay for the file download. The satellite sends a confirmation of this promise to the uplink, along with file piece storage node node locations. The console then sends the promise to pay directly to the storage node nodes, along with the details on the file pieces it needs. Each storage node then accepts or rejects this operation. If a storage node accepts this operation, it confirms and retains a copy of this promise to pay, sending the client the file piece it needs. Later, the storage node sends the promise to pay to the satellite, and the satellite credits that storage node as having successfully delivered the file piece. TODO ref to bandwidth allocation protocol section

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 be immediately blacklisted 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.

TODO users pay satellites TODO payers roll up payments every day, but pay every month

TODO Payment automation?

TODO Payment wallets vs payment addresses.

TODO

See the payer reputation section for details on how storage nodes will know to trust payers.

Payments to storage nodes will be calculated on a daily basis based on the bandwidth utilized and files stored, and will be paid at the end of each month. 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.

4.15.1 Bandwidth allocation protocol

A core component of our system requires knowing how much bandwidth is used between two peers, so 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; some operations are simply considered to be the cost of doing business. This bandwidth traffic measurement only applies to storage operations (storage and retrievals of pieces) and does not apply to overlay traffic (Kademlia DHT) or other generic maintenance overhead.

TODO diagram, gory detail, protobufs, update references

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 payer will return an *unrestricted bandwidth allocation* message. This message will include the identity of the payer, the identity of the client, an expiration timestamp, a serial number, the maximum amount of bytes authorized, and the direction the bytes will flow (whether or not the data will be transfered from or to the client). The message will be signed by the payer.

Once the client has an unrestricted bandwidth allocation, the client will then create *restricted bandwidth allocations*, indicating y bytes have been transfered so far. 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 clients 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 seen. 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 payer'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 own reputation. If the client tries to use more bandwidth than allocated, the storage node will shut down 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

Storage nodes have a strong incentive to avoid accepting data assigned to satellites that don't have a good history of paying.

Initially, storage nodes will put satellites through a vetting process where storage nodes limit their exposure to unknown satellites and build up trust over time with specific payers that are likely to pay their bills. Storage nodes will have a configurable maximum amount of data that they will store for an unknown satellite, and can use whether or not they get paid for that data as input into whether or not that satellite should be trusted for more data in the future.

Storage node operators will be able to opt in and out of working with specific satellites they already trust or distrust. Storj Labs will ship a list of recommended satellites that they have already vetted for quality control that node operators can elect to use.

If a satellite operator wants their satellite included on the Storj-provided inclusion list, the satellite operator may be required to pay Storj for insurance such that Storj can pay storage nodes on the satellite operator's behalf if the satellite goes down.

TODO future work - shared reputation

4.17 Garbage collection

When data is moved or deleted, it's important to inform impacted storage nodes that they are no longer required to store that data. Unfortunately, sometimes storage nodes will be temporarily unavailable and delete messages will be missed. In these cases, data that is no longer needed is considered *garbage*. Payers only pay for data they expect to be stored, so storage nodes with lots of garbage will find less earnings than they would otherwise be entitled to unless a garbage collection system is employed.

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, whereas a *conservative* garbage collector may leave some small proportion of garbage around given some other tradeoffs, often with the aim of improving performance. As long as a conservative garbage collector is used in our system, it should be assumed that the cost of storage owed to a storage node is high enough to amortize the cost of storing the garbage.

When data is deleted through 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 get notification right away.

For the nodes that miss initial delete messages, we propose a conservative garbage collection strategy. Periodically, storage nodes will request a highly-compressible data structure such as a *Bloom filter* [34] from satellites that contains hints about what pieces a node is expected to continue storing. A Bloom filter is a mechanism that can answer certain set membership questions, describing whether an element *isn't contained* or *maybe contained*, but can not determine whether an element *is contained* in the set. Satellites will reject requests for these data structures that happen too frequently. 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.

Because Bloom filters are probabilistic and their collision risk is configurable, the conservative garbage collector can be tuned to eliminate garbage down to an acceptable tolerance, given the tradeoff of additional bandwidth for these larger, more exact cleanup messages. Further, each time a Bloom filter is generated, it can be generated with a new hashing seed, lowering the probability that a specific piece of garbage consistently gets missed by the garbage collector.

Because this garbage collection system is not precise, storage nodes have a strong incentive to stay online to witness as many delete messages as possible. If a storage node misses a handful of delete messages due to an outage, the garbage will eventually get cleaned up with enough Bloom filter based cleanups. On the other hand, because this garbage collection system is not precise, bandwidth overhead for negotiating the list of pieces a storage node must store will be efficient and small.

TODO See our future work section about undeletes in the case of bugs or mistaken file removal.

TODO future work: is a bloom filter the best data structure?

5 Product details

5.1 Quality control and branding

5.2 Durability

TODO discuss quality control and branding TODO insurance

5.3 Detailed walkthroughs

5.3.1 Uploads

When a user uploads a file:

- First, data begins transfer to the console.
- The console chooses an encryption key and starting nonce for this segment and begins encrypting incoming data with authenticated encryption as it flows through.
- The console buffers data until it knows whether the incoming file is short enough to be an inline segment or a remote segment. We'll assume a remote segment.
- The console sends a request to the satellite to prepare for the storage of this first segment. The satellite will:
  - Confirm that the console has appropriate authorization and funds for the request. The console must have an account with this specific satellite already.
  - Make a selection of nodes that conform to the bucket's configured durability, performance, geographic and reputation requirements that have enough resources.
  - Return a list of nodes, along with their contact information and signed unrestricted bandwidth allocation messages, and a chosen root piece id.

- The console will take this information and begin parallel connections to all of the chosen storage nodes via the bandwidth allocation protocol.

- The console 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 allow for the elimination of long tails and the significant improvement of visible performance by allowing the console 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.

- The storage node will store the largest restricted bandwidth allocation, the TTL of the segment (if any), and the data itself by the storage node-specific piece id (the HMAC of the root piece id and the storage node's id).

- If the upload is aborted for any reason, the storage node will keep the largest bandwidth allocation it received but otherwise will throw away all relevant request data.

- The console encrypts the random encryption key chosen for this file with a deterministic hierarchical key.

- The console will upload a `pointer` back to the satellite, which contains information on 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, and a signature.

- The console 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 about how many segments the stream contained, how large the segments were in bytes, and the starting nonce of the first segment.

- The storage nodes will later send the largest restricted bandwidth allocation they received as part of the upload to the appropriate satellite for later payment.

5.3.2 Download

When a user downloads a file:

- First, a request for data is received by the console.
- The console tries to reduce round trips to the satellite by speculatively requesting the pointers for the first few segments, in addition to the pointer for the last segment from the satellite. The last segment is needed to learn the size of the object, how many segments there are, and how big the segments are.
- For every segment pointer requested, the satellite will:
  - Validate that the console has access to download the segment pointer and funds to pay for its downloading.
  - Generate an unrestricted bandwidth allocation for the segment.
  - Look up the contact information for the storage nodes listed in the pointer.
  - Return the requested segment, the bandwidth allocations, and contact info.
- The console will calculate if more segments are necessary for the data request it received, requesting the remaining segment pointers if so.
- Once all necessary segment pointers have been returned, if the requested segments are not inline, the satellite will initiate parallel requests via the bandwidth allocation protocol 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 console to cancel the slowest downloads.
- If the download is aborted for any reason, the storage node will keep the largest bandwidth allocation it received but otherwise will throw away all relevant request data.
- The console will combine the retrieved erasure shares into stripes.
- 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.

5.3.3 Delete

When a user deletes a file:

- First, the delete operation is received by the console.
- The console requests all of the segment pointers for the file.
- For every segment pointer, the satellite will:
  - Validate that the console 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 info.
- For all of the segments that are not inline, the satellite will initiate parallel requests to all appropriate storage nodes to signal that the pieces are being removed.
- The storage nodes will return a signed message saying the storage node received the delete operation and will delete the file and its bookkeeping info.
- The console will upload back to the satellite all of the signed messages it received from working storage nodes. The satellite will require an adjustable percent of the total storage nodes to sign messages successfully to ensure that the console did its part in letting storage nodes know the object has been deleted.
- The satellite will remove the segment pointers and stop charging and paying for them.
- The console will return success.
- Periodically, storage nodes will ask the satellite for generated garbage collection messages that will help storage nodes who were offline during the main deletion event. The garbage collection messages will assist the storage node in pruning data that is no longer live. Initially, these garbage collection messages will be tunable Bloom filters to allow the storage node to probabilistically prune unneeded data without using much bandwidth. Satellites will reject requests for garbage collection messages that happen too frequently.

5.3.4 List

When a user wants to receive many files:

- First, a request for listing objects is received by the console.
- The console will translate the request on unencrypted paths to encrypted paths.
- The console will request from the satellite the appropriate list of encrypted paths.
- The satellite will validate that the console has appropriate access and then return the requested list.
- The console will decrypt the return results and return them.

It's worth pointing out that because the satellite stores paths in sorted order, the order returned to the customer is sorted by the encrypted path element, which means that unencrypted paths will be in random but deterministic order. If a customer wants sorted paths and doesn't mind the satellite operator having access to unencrypted paths, the customer can opt into unencrypted (and thus lexicographically sorted) paths.

TODO future work section about order-preserving encryption

5.3.5 Audits

The auditing process:

- Each satellite has a queue of audits, where an audit will entail validating a specific stripe of a segment across a set of storage nodes.
- Periodically, satellites will choose a stripe to audit by selecting an object uniformly at random, weighted by the number of bytes it has, and place that stripe into the audit queue.
- Similarly, satellites will choose a stripe to audit by identifying storage nodes that have had fewer recent audits than other storage nodes, and selecting a stripe at random from the data contained by that storage node. That stripe audit request will also be placed in the audit queue.
- Satellites will process elements from the queue.

- 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 storage node will be blacklisted by the satellite and marked as bad. The satellite will not pay the storage node going forward, nor will it select it for new data.

5.3.6 Repair

The repair process:

- Each satellite periodically will 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 and 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 entire segment will be downloaded. 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.

5.3.7 Payment

The payment process:

- First, a 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.
- 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.
- Periodically, storage nodes will send in bandwidth allocation reports. When a satellite receives these, it calculates the owed funds along with the outstanding data at rest calculations, and sends the funds to the storage node's requested destination.

5.4 Reliability

6 Future Areas of Research

TODO Storj is a work in progress, and many features are planned for future versions. There are relatively few examples of functional distributed systems at scale, and many areas of research are still open.

6.1 Improving user experience around metadata

TODO automatic exports, backups, distributed consensus

6.2 Fast Byzantine Consensus

Over time, we plan to program the satellite out of the platform. The satellite's role on the network means that the network could be prone to some centralization if others outside of the Storj Labs team do not run their own satellites. The biggest challenge is achieving fast byzantine consensus, where storage node 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 be researching ways to store lots of small pieces of metadata in a distributed manner, even when those pieces are constantly changing. There currently is not a way to achieve this without significant investment in time, compute, and bandwidth. A practical byzantine fault tolerance algorithm could work. They are generally faster and use less disk space than blockchain protocols, however there is significant trade off around network usage and coordination contention, as there could be problematic overlap with two storage nodes trying to communicate with one another at the same time.

6.3 Distributed Repair

The system can detect when a file's Reed Solomon erasure encoding pieces fall below a certain threshold. At that time, the file 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 shards 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 CPU cycles to work. This will be a first step to eliminating the satellite. This approach would also be more decentralized than file repair on satellites. It is also more efficient to execute this operation at the edge of the network.

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. Merkle tree roots will greatly help with distributed repair. The storage node executing the repair would get approval from the satellite to repair a file, the satellite would share its merkle tree root with the storage node and notify which storage nodes should store the restored file pieces. The storage node would then download the file pieces needed for the repair from the storage nodes where they reside. The repair node would execute the repair and run the shards through the merkle tree root to prove the data was correct and properly repaired. We are currently taking the steps needed to ensure the network and our data format will support merkle tree repair in the future.

6.4 Order-preserving encryption

TODO

7 Contributors

This paper took the combined contributions of many individuals. Contributors at Storj Labs, Inc. include but are not limited to: Alex Bender, Alex Leitner, Benjamin Sirb, Brandon Iglesias, Bryan White, Cameron Ayer, Dan Sorenson, Dennis Coyle, Dylan Lott, Garrett Ransom, James Hagans, Jennifer Johnson, Jens Heimbürge, John Gleeson, Jon Sanderson, JT Olio, Kaloyan Raev, Kishore Aligeti, Matt Robinson, Moby von Briesen, Nadine Farah, Natalie Villasana, Patrick Gerbes, Philip Hutchins, and Shawn Wilkinson.

We'd like to especially thank the other authors of the previous Storj v2 paper: Braydon Fuller, Chris Pollard, Gordon Hall, James Prestwich, Josh Brandoff, and Tome Boshevski.

We'd like to extend a huge thank you to everyone we talked to during the design and architecture of this system for your valuable thoughts, feedback, input, and suggestions.

A 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.

A.1 Sybil

Sybil attacks involve the creation of large amounts of nodes in an attempt to disrupt network operation by hijacking or dropping messages. Kademlia is reasonably resistant to Sybil attacks, because it relies on 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 three 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.

TODO discuss vetting period

A.1.1 Honest Geppetto

The Honest Geppetto attack is a storage-specific variant of the Google attack. The attacker operates a large number of 'puppet' 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 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 nodes. Bayesian inference across downtime, latency, and other attributes can be used to assess the likelihood that two nodes are operated by the same organization, and data owners can and should attempt to distribute shards across as many unrelated nodes as possible.

A.2 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. In order to eclipse any node in the network, the attacker must repeatedly generate key pairs until it finds three keys whose hashes are closer to the targeted node than its nearest non-malicious neighbor, and must defend that position against any new nodes with closer IDs. This is, in essence, a proof-of-work problem whose difficulty is proportional to the number of nodes in the network.

It follows that the best way to defend against eclipse attacks is to increase the number of nodes in the network. For large networks it becomes prohibitively expensive to perform an eclipse attack (see Section 6.2). Furthermore, any node that suspects it has been eclipsed may trivially generate a new keypair and node ID, thus restarting the proof-of-work challenge.

S/Kademlia additionally assists in preventing eclipse attacks by ensuring multiple concurrent disjoint lookup paths through the network.

A.3 Hostage Bytes

The hostage byte attack is a storage-specific attack where malicious storage nodes refuse to transfer shards, or portions of shards, in order to extort additional payments from data owners. Data owners should protect themselves against hostage byte attacks by storing shards redundantly across multiple nodes (see Section 2.7). As long as the client keeps the bounds of its erasure encoding a secret, the malicious storage node cannot know what the last byte is. Redundant storage is not a complete solution for this attack, but addresses the vast majority of practical applications of this attack. Defeating redundancy requires collusion across multiple malicious nodes, which is difficult to execute in practice.

B Selected use cases

C File piece loss model

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: the simple case first

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: it's not that much harder

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

$$P(X = d) = \frac{\binom{n}{d}\binom{C-n}{c-d}}{\binom{C}{c}} \tag{2}$$

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 piececs is reduced by nc/C, so instead of n pieces on the network (as the parameter in (2)), there are $n-nc/C$ pieces, changing both the parameter and the mean for (2) in iteration 2.

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

[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.

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 an alien megarace extracting a farmer's hard-drive to a higher dimension (amongst other possibilities).

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

[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.

rebuilds at the 99-th percentile (or higher) of the number of rebuilds occuring over these 1000 iterations. In other words, we choose the value for which the value of the observed 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 1. We perform the experiment on a network of 10,000 nodes, observing that the network size will not directly impact the mean rebuilds/month value for a single file under our working assumption of a constant rate of loss per month[5].

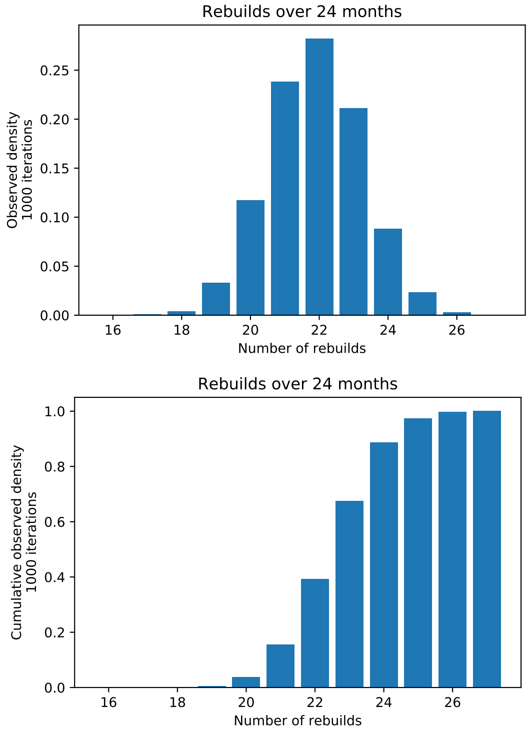


Figure 1: Top: Density for the number of rebuilds over a 24 month period, repeated for 1000 iterations. Bottom: CDF of the number of rebuilds. In this case, the mean rebuilds/month value would be taken as $26/24 \approx 1.083$*, with there being a 99.7% chance that a file is rebuilt at most 26 times 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 m affect durability under a fixed expansion factor of 2. That is, we consider how the size of the "safety margin" (the number of pieces between k and m) affects durability.

Churn rate	k	n	m	Mean rebuilds/month	Durability (# nines)
0.1	20	40	35	0.833	0.9999 (6)
0.1	20	40	30	0.416	0.9997 (3)
0.1	20	40	25	0.292	0.9580 (1)
0.2	20	40	35	1.500	0.9976 (2)
0.2	20	40	30	0.792	0.9574 (1)
0.2	20	40	25	0.500	0.6160 (0)
0.3	20	40	35	2.042	0.9317 (1)
0.3	20	40	30	1.083	0.7060 (0)
0.3	20	40	25	0.708	0.2414 (0)
0.5	20	40	30	1.750	0.1185 (0)

Table 2: Table for $n = 40$.

Churn rate	k	n	m	Mean rebuilds/month	Durability (# nines)
0.1	40	80	70	0.833	0.999 (10)
0.1	40	80	60	0.416	0.9999 (5)
0.1	40	80	50	0.292	0.9863 (1)
0.2	40	80	70	1.583	0.9999 (4)
0.2	40	80	60	0.792	0.9884 (1)
0.2	40	80	50	0.500	0.5830 (0)
0.3	40	80	70	2.125	0.9758 (1)
0.3	40	80	60	1.083	0.7307 (0)
0.3	40	80	50	0.708	0.1185 (0)
0.5	40	80	60	1.750	0.0353 (0)

Table 3: Table for $n = 80$.

Churn rate	k	n	m	Mean rebuilds/month	Durability (# nines)
0.1	100	200	160	0.500	0.999 (17)
0.1	100	200	140	0.333	0.9999 (8)
0.1	100	200	120	0.250	0.9884 (1)
0.2	100	200	160	1.000	0.9999 (5)
0.2	100	200	140	0.625	0.9875 (1)
0.2	100	200	120	0.458	0.2426 (0)
0.3	100	200	160	1.375	0.9605 (1)
0.3	100	200	140	0.875	0.4180 (0)
0.3	100	200	120	0.625	0.0027 (0)
0.5	100	200	140	1.458	0.0000 (0)

Table 4: Table for $n = 200$.

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 one should expect 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, one may aim 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, so to maximize durability while minimizing repair bandwidth usage, n should be as large as existing network conditions allow for. 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 fixed network size of $10,000$ nodes and a fixed churn rate of 0.1, so that a fixed number of $1,000$ pieces are lost on the network each month[6]. Suppose we consider the same file encoded with two different RS parameters: once under a $(20, 40)$ schema and the other as an $(40, 80)$ schema. If we want to set m so that $m = k + 10$ for both cases, we observe from Tables 2 and 3 that the expected mean rebuilds is 0.416 in the $(20, 40)$ case and is 0.292 in the $(40, 80)$ case. Both encoding schemes have similar durabilities, as a repair

[6] Technically, we assume this figure represents "churn", so while we assume $1,000$ nodes are lost each month, we also assume $1,000$ new nodes have come online.

in both cases is triggered when there are $k + 10$ pieces left, though the mean rebuilds per month is empirically and theoretically lower for the $(40, 80)$ case using $m = k + 10$.

D Distributed consensus

D.1 Non-byzantine

D.1.1 Aside about distributed consensus

A long and challenging area of research has been directed toward getting a group of computers to agree on a set of values what kind of values? principles? or numerical-type values?, with the goal of constructing a horizontally-scalable database that works in the face of expected failures (crash failures, for example: failures where a server simply shuts down). Fortunately, this research 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" [35] [7], 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 solution to the two generals' problem with a finite number of messages is readily seen to be impossible, so engineering approaches have had to embrace uncertainty by necessity. Many distributed systems make tradeoffs to deal with this uncertainty. Some systems embrace *consistency*, which means that the system will choose downtime over inconsistent answers. Other systems embrace *availability*, which means that the system chooses potentially inconsistent answers over downtime. The widely-cited CAP theorem [37] 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 means that every read receives the most

[7] earlier described as a problem between groups of gangsters [36]

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 *consistency models* that may be acceptable by making various tradeoffs. Linearizability, sequential consistency, causal consistency, PRAM consistency, 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.[8]

Amazon S3 generally provides *read-after-write consistency*, though in some cases will provide *eventual consistency* instead [40]. Arguably, there may be some flexibility here which allows for the selection of alternate consistency models that suit us better while still broadly providing S3 compatibility. Many distributed databases provide eventual consistency by default, such as Dynamo [11] and Cassandra [23].

Linearizability in a distributed system is often much more desirable than what?, 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 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 [41] proved that no algorithm could reach linearizable consensus in bounded time. Then in 1988, Barbara Liskov and Brian Oki published the Viewstamped Replication algorithm [42] which was the first linearizable distributed consensus algorithm. Unaware of the VR publication, Leslie Lamport set out to prove linearizable distributed consensus was impossible [43], but instead in 1989 proved it was possible by publishing his own Paxos algorithm [44], 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 [45], 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 [46], and Paxos Made Moderately Complex [47] 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 [10]. Ultimately, after an upsetting few decades, reliable implementations of Paxos, Raft, Viewstamped Replication [48], Chain Replication [49], and Zab [50] now exist, with ongoing work to improve the situation further [51, 52]. Arguably, part of Google's early

[8] If differing consistency models are new to you, it may be worth reading about them in Kyle Kingbury's excellent tutorial [38]. 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 [39].

success was in spending the time to build their internal Paxos-as-a-service distributed lock system, Chubby [53]. Most of Google's most famous internal data storage tools such as Bigtable [54] depend on Chubby for correctness. Spanner [24] – perhaps one of the most incredible distributed databases in the world – is mainly 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.

D.2 Byzantine

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 [20, 55–71]. Each of these algorithms make some additional tradeoffs that non-Byzantine distributed consensus algorithms don't require to deal with the potential for uncooperative nodes. For example, PBFT [55] causes a significant amount of network overhead. Bitcoin [20] intentionally limits the transaction rate with changing proof-of-work difficulty, in addition to requiring all participants to keep a full copy of all change histories (like other blockchain-based solutions).

TODO talk about merkle-dag, git-inspired approaches to metadata, and how they struggle with conflict resolution due to the lack of crdt-like options for file systems

References

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

[2] 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.

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

[4] 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.

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

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

[7] 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.

[8] 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.

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

[10] 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.

[11] 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.

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

[13] 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.

[14] Shawn Wilkinson. Sip purpose and guidelines, (2016). https://github.com/storj/sips/blob/master/sip-0001.md.

[15] 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.

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

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

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

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

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

[21] 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.

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

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

[24] 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.

[25] Matteo Zignani, Sabrina Gaito, and Gian Paolo Rossi. Follow the ”mastodon”: Structure and evolution of a decentralized online social network, 2018.

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

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

[28] 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.

[29] 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.

[30] Lloyd R. Welch and Elwyn R. Berlekamp. Error correction for algebraic block codes, 1986.

[31] 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).

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

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

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

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

[36] 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.

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

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

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

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

[41] 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.

[42] 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.

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

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

[45] Leslie Lamport. Paxos made simple. pages 51–58, December 2001.

[46] 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.

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

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

[49] 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.

[50] 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.

[51] 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.

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

[53] 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.

[54] 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.

[55] 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.

[56] 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.

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

[58] 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.

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

[60] 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.

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

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

[63] 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.

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

[65] 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`.

[66] 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.

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

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

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

[70] 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.

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