Storj
A Decentralized Cloud Storage Network Framework

Alex Bender (bender@storj.io), Alex Leitner (alex@storj.io), Benjamin Sirb (bens@storj.io), Braydon Fuller (braydon@storj.io), Bryan White (bryan@storj.io), Chris Pollard (cpollard1001@gmail.com), Dennis Coyle (dennis@storj.io), Dylan Lott (dylan@storj.io), Garrett Ransom (garrett@storj.io), Gordon Hall (gordonhall@openmailbox.org), James Hagans (jhagans@storj.io), James Prestwich (james@storj.io), John Gleeson (jg@storj.io), Josh Brandoff (josh.brandoff@gmail.com), JT Olio (jt@storj.io), Kaloyan Raev (kaloyan@storj.io), Kishore Aligeti (kishore@storj.io), Nadine Farah (nadine@storj.io), Natalie Villasana (nat@storj.io), Patrick Gerbes (patrick@storj.io), Philip Hutchins (philip@storj.io), Shawn Wilkinson (shawn@storj.io), Tome Boshevski (tome@storj.io)

April 26, 2018
v3.0

Abstract

Decentralized cloud storage is attractive for a number of reasons. 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 it simultaneously increases security and privacy. It allows market forces to innovate on cheaper ways to provide storage at a greater innovation rate than any single entity can afford. While there are many ways to build such a system, there are many specific responsibilities every system must address. Based on our experience with petabyte-scale storage systems, we introduce a general framework for considering these responsibilities and building such a system, along with concrete implementations for each responsibility of the framework.

1 Introduction

Many storage products have been created based on distributed storage techniques. Products such as Wuala, Allmydata, Tahoe-LAFS, Space Monkey, Sia, Maidsafe, Filecoin, Crashplan, Mozy, HDFS, Storj, all share one thing in common: a single computer is not as powerful or as robust as a network. These products and others attempt to solve many different use cases and have many different requirements, but at a high level, they all operate on the same principles. They generate redundancy for data in case of failure, store this redundancy in multiple locations, then keep track of where the data was placed.

There are many different primary focuses one could choose when creating a distributed storage system. Speed, capacity, simplicity, trustlessness, security, cost, etc., are all desirable traits in a storage system, but independently of anything else, data must be maintained to prevent data loss, nodes in the system must be able to be communicated with, and metadata must be kept track of, implicitly or explicitly.

While the Storj design space is different from systems such as HDFS or Wuala, we propose a framework that will allow us to choose some reasonable tradeoffs and then iterate on improvements to components of the system without changing the overall system.

Besides this introduction, this paper is divided into 5 main sections. Section 2 covers the framework at a high level. Section 3 talks about the design space Storj operates in and covers some specific design constraints. Section 4 proposes a concrete implementation of the framework. Section 5 covers specific details about how we will ship section 4 to users. Section 6 covers future areas of research.

2 Framework

3 Storj design constraints

Before designing a system, it's important to understand the requirements of said system, so we'll begin with a discussion of Storj's design constraints.

3.1 S3 compatibility

The flagship cloud storage product is Amazon's Simple Storage Service, or S3 for short. Most cloud storage products provide some form of compatibility with the S3 API.

Until a decentralized cloud storage protocol is the *lingua franca* of storage protocols, a graceful transition must be allowed for users with data currently on a centralized provider who are interested in the benefits of decentralized cloud storage but have low tolerance for switching costs.

For Storj to compete successfully in the wider cloud storage industry and bring decentralized cloud storage to the mainstream, thus enabling more people greater security and less centralized control, applications built against S3 should be able to be made to work with Storj with minimal friction and changes. This adds strong requirements on feature set, performance, and durability.

3.2 Latency

Decentralized, distributed storage has massive opportunities for parallelism with transfer rates, processing, and number of other factors. Parallelism by itself is a great way to increase overall throughput even when individual network links are slow.

Even though parallelism can improve *throughput* performance, parallelism cannot by itself improve *latency*. If an individual network link has fixed latency and is a required part of an operation, the overall operation will have at least the latency of that individual link.

A distributed system interested in high performance must aggressively and ruthlessly optimize for low latency, both at the individual process scale and at the overall architecture scale.

Besides attempting to reduce latency in the small scale directly in as many components as possible, one overall architecture strategy emphasized in this paper is to optimize latency by focusing on eliminating the need to wait for long tails.[1] In other words, no system request should require waiting on the slowest peers out of a set. Every request should be able to be satisfied by the fastest nodes out of the set participating, without waiting for a slow 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.

3.3 Device failure and churn

For any storage system, but especially a distributed storage system, component failure is a guarantee. All hard drives fail after enough wear[2], and the servers providing the network access to these hard drives will eventually fail, too. Network links die, power is lost, and storage mediums become unreliable. For data to outlast individual component failure, data must be stored with enough redundancy to recover from failure, and perhaps most importantly, no data is stationary and all data must eventually be moved.

In such an environment, redundancy, data maintanance, repair, and replacement of lost redundancy are facts of life, and the system must account for these issues.

Decentralized systems are additionally susceptible to high churn rates, where potential participants join the network and then leave for various reasons well before their hardware has actually failed. Despite the issues with failure, in decentralized systems, Maymounkov et al. found that the probability of a node staying a member of the network for an additional hour only *increases* with uptime.[3] In other words, the longer a node is a participant in the network, the more likely it is to continue to participate.

As a result, a highly durable storage system must strive to keep the churn rate as low as possible. Too high of a churn rate and the system will simply be moving data around constantly, using up all available bandwidth. See Appendix TODO for a discussion about how repair bandwidth varies as a function of node churn and uptime.

3.4 Bandwidth

Even though bandwidth availability globally is increasing year over year, access to high bandwidth internet connections is unevenly distributed, and while some countries have good access to symmetric, high-speed, unlimited bandwidth, other countries have significant widespread problems.

In the United States, many residential internet service providers provide internet with two specific problems. The first is that the internet connection is often asymmetric, where customers buy internet based on the advertised

download speed and the upload speed is potentially an order of magnitude or two slower. The second is that the bandwidth is sometimes "capped" at a fixed amount of traffic per month. An internet connection that gets 10 megabytes per second of throughput with a cap of one terabyte per month is actually only able to use 414 kilobytes on a consistent basis all month without going over the bandwidth cap. Bandwidth caps impose a significant limit to the amount of available bandwidth throughout the month.

With all of this guaranteed repair traffic, it's important to make sure there is enough headroom for the bandwidth required by data maintenance, over and above the bandwidth required for data storage and retrieval.

To design a storage system that is careless with bandwidth usage would be to abdicate that system to just the hands of the storage providers with access to unlimited, high-speed bandwidth, thus recentralizing the system to some degree. To keep the storage system as decentralized as possible and to work 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.

4 Concrete implementation

Storj is a protocol that creates a distributed network for the reliable storage of data and facilitates payment for successful data storage between peers. The Storj protocol enables peers on the network to transfer data, verify the integrity and availability of remote data, retrieve data, and pay other nodes for storing data. Each peer is an autonomous agent, capable of performing these actions without significant human interaction.

At a high level, there are three major operations in the system: storing, retrieving, and maintaining data.

Storing When data is stored with the network, the client encrypts it, breaks it up into multiple little pieces, distributes the pieces to peers in the network, and generates and stores some metadata about where to find the data again.

Retrieving When data is retrieved from the network, the metadata about where to find the pieces is recovered first, then the pieces are retrieved and the original data is reconstructed.

Maintaining 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 then the missing pieces are regenerated and replaced.

To make this system feasible while satisfying our design constraints, we will need to solve a number of complex challenges. Inspired by Raft[4], we will break the design up into a collection of relatively independent concerns and then bring them together.

Importantly, one large benefit of breaking up the system into this collection of concerns is it will be much easier to vastly improve individual components without rearchitecting the rest of the network. In the Future Work section TODO , we'll discuss a number of improvements each component might adopt in the near future, but for now, our goal is concrete proposals with clear implementation strategies for each required component.

- Farmers
- Peer-to-peer communication
- Overlay network
- Erasure encoding
- Encryption
- Structured file storage
- Network state
- Authorization
- Farmer Reputation
- Payments
- Payer Reputation
- Repair

4.1 Farmers

TODO

- Identity, proof of work (S/Kad), use ERC20 addresses
- Get
- Put
- Delete

4.2 Peer-to-peer communication

TODO

- Signed messages (who is talking)
- NAT traversal
- gRPC

TODO Due to the presence of NATs and other adverse network conditions, not all devices are publicly accessible. To enable non-public nodes to participate in the network, Storj implements a reverse tunnel system.

To facilitate this system, Storj extends Kademlia with three additional message types: PROBE, FIND_TUNNEL, and OPEN_TUNNEL The tunneling system also makes use of the publish/subscribe system detailed in section 2.6.

PROBE messages allow a node to determine whether it is publically addressable. The message is sent to a publicly addressable node, typically a known network seed. The receiving node issues a separate PING message. The receiving node then responds to the PROBE message with the result of the PING. Nodes joining the network should immediately send a PROBE to any known node.

Nodes that receive a negative response to their initial PROBE should issue a FIND_TUNNEL request to any known node. That node must respond with three contacts that have previously published a tunnel announcement via the publish/subscribe system. Tunnel providers must be publicly addressable.

Once the non-public node has received a list of tunnel providers, it issues OPEN_TUNNEL requests to the tunnel providers. The providers must provide a tunnel for that node if they are capable. To open a connection, the provider sends back an affirmative response with tunnel information. The tunneled node then opens a long-lived connection to the provider, and updates its own contact information to reflect the tunnel address.

Tunnels are operated over TCP sockets by a custom reverse-tunneling library, Diglet [5]. Diglet provides a simple and flexible interface for general-purpose reverse tunneling. It is accessible both by command-line and programmatically.

4.3 Overlay network

Storj is built on Kademlia[3], a distributed hash table (DHT). We have additionally implemented the S/Kademlia[6] extensions to mitigate a number of attack vectors. Kademlia and S/Kademlia typically provide key/value storage in addition to peer address lookup; however, we are only using Kademlia for peer address lookup.

TODO

4.4 Erasure encoding

TODO

TODO Cloud object stores typically own or lease servers to store their customers files. They use RAID schemes or a multi-datacenter approach to protect the file from physical or network failure. Because Storj objects exist in a distributed network of untrusted peers, farmers should not be relied upon to employ the same safety measures against data loss as a traditional cloud storage company. Indeed, farmers may simply turn off their node at any time. As such, it is strongly recommended that the data owner implement redundancy schemes to ensure the safety of their file. Because the protocol deals only with contracts for individual shards, many redundancy schemes may be used. Three are described below.

Storj will soon implement client-side Reed-Solomon erasure coding [7]. Erasure coding algorithms break a file into k shards, and programmatically create m parity shards, giving a total of $k + m = n$ shards. Any k of these n shards can be used to rebuild the file or any missing shards. Availability of the file is then $P = 1 - \prod_0^m a_m$ across the set of the $m + 1$ least available nodes. In the case of loss of individual shards, the file can be retrieved, the missing shard rebuilt, and then a new contract negotiated for the missing shard.

To prevent loss of the file, data owners should set shard loss tolerance levels. Consider a 20-of-40 erasure coding scheme. A data owner might tolerate the loss of 5 shards out of 40, knowing that the chance of 16 more becoming inaccessible

in the near future is low. However, at some point the probabilistic availability will fall below safety thresholds. At that point the data owner must initiate a retrieve and rebuild process.

Because node uptimes are known via the audit process, tolerance levels may be optimized based on the characteristics of the nodes involved. Many strategies may be implemented to handle this process.

Erasure coding is desirable because it drastically decreases the probability of losing access to a file. It also decreases the on-disk overhead required to achieve a given level of availability for a file. Rather than being limited by the least available shard, erasure coding schemes are limited by the least-available $n + 1$ nodes (see Section 6.1).

4.5 Encryption

TODO

4.6 Structured file storage

TODO

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`.

Clients encrypt `paths` before they ever leave the client computer.

File A `file` is a collection of `stream`s. Every `file` has exactly one default `stream` and may have 0 or more named `stream`s. Multiple `stream`s allow flexible support of extended attributes, alternate data streams, resource forks, and other slightly more esoteric filesystem features.

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

Stream A `stream` is an ordered collection of 0 or more `segment`s. `segment`s have a fixed maximum size, and so the more bytes the `stream` represents through `segment`s, the more `segment`s there are.

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.

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. Encryption and erasure encoding happen on `stripe`s individually. Encryption happens on all `segment`s, but erasure encoding only happens on `remote segment`s.

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 share`s from that `segment`'s `stripe`s.

Piece Storage Node A node in the network that is responsible for storing `piece`s. These are operated by `farmer`s.

Farmer A person or group that is responsible for running and maintaining `piece storage node`s.

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

4.6.1 Files as Streams

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."[8] Being able to support arbitrarily named sets of keys/values dramatically improves compatibility with other storage platforms.

Every `file` will have at least one `stream` (the default `stream`) and many files may never have another `stream`.

4.6.2 Streams as Segments

Because `stream`s are used for data (the default `stream`) and metadata (extended attributes, etc.), `stream`s should be designed both for small data and large data. If a `stream` only has very little data, it will have one small `segment`. If that `segment` is smaller than the metadata it would require to store across the network, the `segment` will be an `inline segment` and the data will be stored directly inline with the metadata.

For larger `stream`s, past a certain size the data will be broken into multiple large `remote segment`s. 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 `segment`s 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 `segment`s 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[9] or other peer-to-peer networks.

4.6.3 Segments as Stripes

In many situations it's important to be able to access just a portion of some 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 operation. In these cases it's useful to be able to decode and decrypt only parts of a file.

A `stripe` is no more than a couple of kilobytes, and encrypting and 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.

`stripe`s should be encrypted client-side before being erasure encoded. The reference implementation uses AES256-GCM by default but XSalsa20+Poly1305 is also provided. This protects the content of the data from the `farmer` housing the data. The data owner retains complete control over the encryption key, and thus over access to the data.

It's important to use authenticated encryption to defend against data corruption (willful or negligent), and with a monotonically increasing nonce to defeat reordering attacks. The nonce should be monotonically increasing across `segment`s throughout the `stream`. If `stripe` i is encrypted with nonce j, `stripe` $i + 1$ should be encrypted with nonce $j + 1$. Each `segment` should get a new encryption key whenever the content in the `segment` changes to avoid nonce reuse.

4.6.4 Stripes as Erasure Shares

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

For example, let's say a `stripe` is broken into 40 `erasure share`s ($n = 40$), where any 20 ($k = 20$) are needed to reconstruct the `stripe`. Each of the 40 `erasure share`s will be $\frac{1}{20}$th the size of the original `stripe`.

All n `erasure share`s have a well defined index associated with them. The ith share will always be the same, given the same input parameters.

Because peers generally rely on separate hardware and infrastructure, data failure is not correlated. This implies that erasure codes are an extremely effective method of securing availability. Availability is proportional to the number of nodes storing the data.

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

4.6.5 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 shares separately, we pack all of the `erasure share`s together into a few `piece`s. In an (n, k) 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 an $\frac{n}{k}$th of a `stripe`, each `piece` is an $\frac{n}{k}$th of a `segment`, and only k `piece`s are needed to recover the full `segment`.

4.6.6 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. As the set of `segment`s in the network grows, it becomes exponentially more difficult to locate any given `piece` set without prior knowledge of their locations (see Section 6.3). This implies that security of the `remote segment` is proportional to the square of the size of the network.

4.7 Network state

TODO

4.8 Authorization

TODO

4.9 Farmer Reputation

TODO

4.10 Payments

TODO

4.11 Payer Reputation

TODO

4.12 Repair and Maintenance

TODO

4.13 Proofs of Retrievability

TODO Proofs of retrievability guarantee the existence of a certain piece of data on a remote host. The ideal proof minimizes message size, can be calculated quickly, requires minimal pre-processing, and provides a high degree of confidence that the file is available and intact. To provide knowledge of data integrity and availability to the data owner, Storj provides a standard format for issuing and verifying proofs of retrievability via a challenge-response interaction called an audit or heartbeat.

Our reference implementation uses Merkle trees [10] and Merkle proofs. After the sharding process the data owner generates a set of n random challenge salts $s_0, s_1, ...s_{n-1}$ and stores the set of salts s. The challenge salts are each prepended to the data d, and the resulting string is hashed to form a pre-leaf p as such: $p_i = H(s_i + d)$. Salts are prepended, rather than appended, in order to defeat length extension attacks. Pre-leaves are hashed again, and the resulting digests become the set of leaves l of a standard Merkle tree such that $l_i = H(H(s_i + d))$. The leaf set is filled with hashes of a blank string until its cardinality is a power of two, to simplify the proof process.

The data owner stores the set of challenges, the Merkle root and the depth of the Merkle tree, then transmits the Merkle trees leaves to the farmer. The farmer stores the leaves along with the shard. Periodically, the data owner selects a challenge from the stored set, and transmits it to the farmer. Challenges may be selected according to any reasonable pattern, but should not be reused. The farmer uses the challenge and the data to generate the pre-leaf. The pre-leaf, along with the set of leaves, is used to generate a Merkle proof, which is

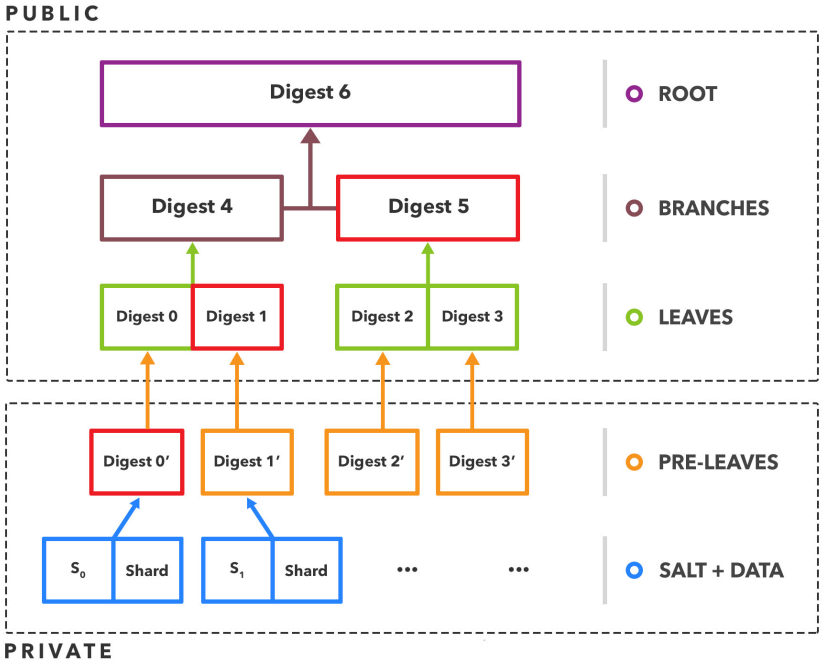

Figure 1: Storj Audit Tree with $|l| = 4$
Red outlines indicate the elements of a Merkle proof for s_0

sent back to the data owner.

The Storj Merkle proof always consists of exactly $log_2(|l|) + 1$ hashes, and thus is a compact transmission, even for large trees. The data owner uses the stored Merkle root and tree depth to verify the proof by verifying that its length is equal to the tree depth and the hashes provided recreate the stored root. This scheme does not allow false negatives or false positives, as the hash function requires each bit to remain intact to produce the same output.

4.13.1 Partial Audits

TODO The Merkle tree audit scheme requires significant computational overhead for the data owner, as the entire shard must be hashed many times to generate pre-leaves. An extension of this scheme utilizes subsets of the data to perform partial audits, reducing computational overhead. This also has the advantage of significantly reducing I/O burden on farmer resources.

This extension relies on two additional selectable parameters: a set of byte indices x within the shard and a set of section lengths in bytes, b. The data

owner stores a set of 3-tuples (s, x, b). To generate pre-leaf i, the data owner prepends s_i to the b_i bytes found at x_i. During the audit process, the verifier transmits $(s, x, b)_i$, which the farmer uses to generate a pre-leaf. The Merkle proof is generated and verified as normal.

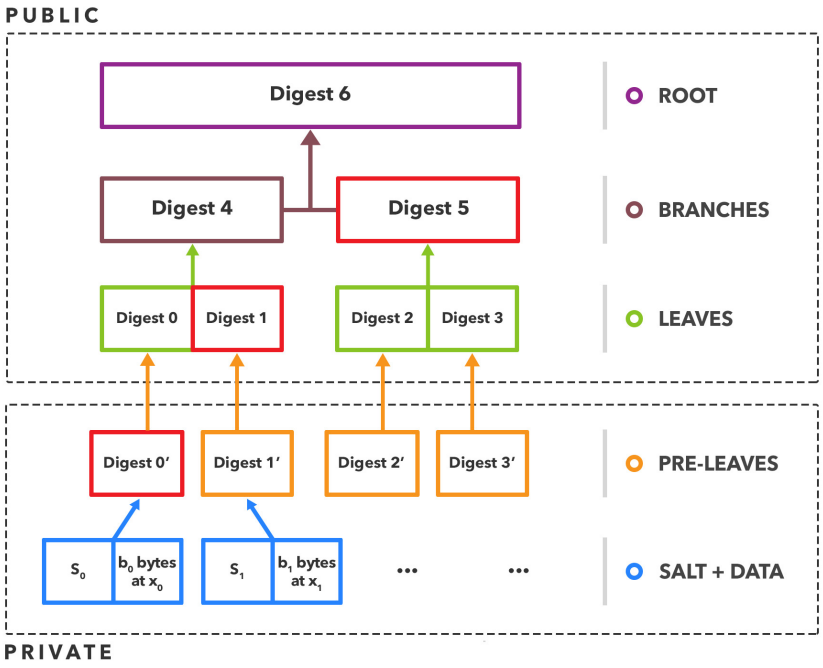

Figure 2: Storj Audit Tree with $|l| = 4$ and Partial Audits
Red outlines indicate the elements of a Merkle proof for s_0

Partial audits provide only probabilistic assurance that the farmer retains the entire file. They allow for false positive results, where the verifier believes the farmer retains the intact shard, when it has actually been modified or partially deleted. The probability of a false positive on an individual partial audit is easily calculable (see Section 6.4)

Thus the data owner can have a known confidence level that a shard is still intact and available. In practice, this is more complex, as farmers may implement intelligent strategies to attempt to defeat partial audits. Fortunately, this is a bounded problem in the case of iterative audits. The probability of several consecutive false positives becomes very low, even when small portions of the file have been deleted.

In addition, partial audits can be easily mixed with full audits without re-structuring the Merkle tree or modifying the proof verification process. Many

audit strategies that mix full and partial verification can be envisioned, each of which provides different levels of confidence over time.

A further extension of this scheme could use a deterministic seed instead of a set of byte indexes. This seed would be used to generate indexes of many non-consecutive bytes in the file. Requiring many non-consecutive random bytes would provide additional resistance against malicious farmers attempting to implement audit evasion strategies without significant extra overhead from processing or I/O.

4.13.2 Other Proof-of-Retrievability Schemes

TODO Other audit schemes were examined, but deemed generally infeasible. For example, Shacham and Waters proposed a compact proof [11] with several advantages over Merkle-tree schemes. This construction allows for an endless stream of challenges to be generated by the data owner with minimal stored information. It also allows for public verifiability of challenge responses.

However, initial implementations indicate that the client-side pre-processing required for the Shacham-Waters scheme requires at least one order of magnitude more computation time than hash-based methods, rendering it too slow for most applications.

Proof of retrievability is an area of ongoing research, and other practical schemes may be discovered in the future. As proof of retrievability schemes are discovered and implemented, the choice of scheme may become a negotiable contract parameter. This would allow each data owner and node to implement a wide variety of schemes, and select the most advantageous scheme for a given purpose.

4.13.3 Issuing Audits

TODO To issue audits, Storj extends the Kademlia message set with a new type: AUDIT (for a full list of Kademlia extensions, see Appendix A). These messages are sent from data owners to farmers and contain the hash of the data and a challenge. The farmer must respond with a Merkle proof as described above. Upon receipt and validation of the Merkle proof, the data owner must issue payment to the farmer according to agreed-upon terms.

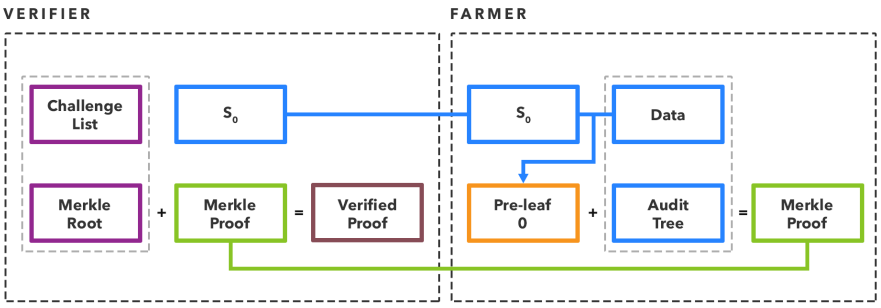

Figure 3: Issuing and Verifying Storj Audits

4.14 Contracts and Negotiation

TODO Data storage is negotiated via a standard contract format [12]. The contract is a versioned data structure that describes the relationship between data owner and farmer. Contracts should contain all information necessary for each node to form a relationship, transfer the data, create and respond to audits over time, and arbitrate payments. This includes shard hash, shard size, audit strategy, and payment information. Storj implements a publish/subscribe system to connect parties interested in forming a contract (see Section 2.6).

Each party should store a signed copy of the contract. Contracts exist solely for the benefit of the data owner and farmer, as no other node can verify the terms or state of the relationship. In the future, contract information may be stored in the DHT, or in an external ledger like a Blockchain, which may allow some outside verification of relationship terms.

The contracting system extends Kademlia with four new message types: OFFER, CONSIGN, MIRROR, and RETRIEVE.

To negotiate a contract, a node creates an OFFER message and sends it to a prospective partner. Prospective partners are found via the publish/subscribe system described in Section 2.6. The OFFER message contains a fully-constructed contract that describes the desired relationship. The two nodes repeatedly swap signed OFFER messages. For each new message in the OFFER loop, the node either chooses to terminate negotiations, respond with a new signed counter-offer, or accept the contract by countersigning it. Once an OFFER is signed by both parties, they each store it locally, keyed by the hash of the data.

Once an agreement is reached, the data owner sends a CONSIGN message

to the farmer. The message contains the leaves of the audit-tree. The farmer must respond with a PUSH token that authorizes the data owner to upload the data via HTTP transfer (see Section 2.10). This token is a random number, and can be delegated to a third party.

RETRIEVE messages signify the intent to retrieve a shard from a farmer. These messages are nearly identical to CONSIGN messages, but do not contain the audit-tree leaves. The farmer responds to a valid RETRIEVE message with a PULL token that authorizes download of the data via a separate HTTP transfer.

MIRROR messages instruct a farmer to retrieve data from another farmer. This allows data owners to create redundant copies of a shard without expending significant additional bandwidth or time. After a successful OFFER/CONSIGN process, the data owner may initiate a separate OFFER loop for the same data with another farmer. Instead of issuing a CONSIGN message to the mirroring farmer, the data owner instead issues a RETRIEVE message to the original farmer, and then includes the retrieval token in a MIRROR message to the mirroring farmer. This authorizes the mirroring farmer to retrieve the data from the original farmer. The success of the MIRROR process should be verified immediately via an AUDIT message.

4.15 Payment

TODO Storj is payment agnostic. Neither the protocol nor the contract requires a specific payment system. The current implementation assumes Storjcoin, but many other payment types could be implemented, including BTC, Ether, ACH transfer, or physical transfer of live goats.

The reference implementation will use Storjcoin micropayment channels, which are currently under development [13]. Micropayment channels allow for pairing of payment directly to audit, thus minimizing the amount of trust necessary between farmers and data owners. However, because data storage is inexpensive, audit payments are incredibly small, often below $0.000001 per audit.

Storjcoin allows much more granular payments than other candidate currencies, thereby minimizing trust between parties. In addition, the mechanics of micropayment channels require the total value of the channel to be escrowed for the life of the channel. This decreases currency velocity, and implies that value fluctuations severely impact the economic incentives of micropayment channels. The use of a separate token creates a certain amount of insulation from outside

volatility, and Storjcoin's large supply minimizes the impact of token escrow on the market.

New payment strategies must include a currency, a price for the storage, a price for retrieval, and a payment destination. It is strongly advised that new payment strategies consider how data owners prove payment, and farmers verify receipt without human interaction. Micropayment networks, like the Lightning Network [14], solve many of these problems, and are thus an ideal candidate for future payment strategies. Implementation details of other payment strategies are left as an exercise for interested parties.

5 Product details

TODO 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, and hidden information like challenge sets cannot be safely outsourced to an untrusted peer, data owners are responsible for negotiating contracts, pre-processing shards, issuing and verifying audits, providing payments, managing file state via the collection of shards, managing file encryption keys, 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. This is similar to the SPV wallet concept found in Bitcoin and other cryptocurrency ecosystems. The burdens of the data owner can be split across the client and the server in a variety of ways. By varying the amount of trust delegated, the server could also provide a wide variety of other valuable services. This sort of dedicated server, called Bridge, has been developed and released as Free Software. Any individual or organization can run their own Bridge server to facilitate network access.

5.1 Farmer

TODO

5.2 Bridge

TODO Our reference implementation of this model consists of a Bridge server, and a client library. Bridge provides an object store, which is to say, the primary function of Bridge is to expose an API to application developers. Developers should be able to use the Bridge via a simple client without requiring knowledge of the network, audit procedures, or cryptocurrencies. The Bridge API is an abstraction layer that streamlines the development process. This enables developers to create many applications that use the Storj network, allowing the network to reach many users.

In the current implementation, Bridge assumes responsibility for contract negotiation, audit issuance and verification, payments, and file state, while the client is responsible for encryption, pre-processing, and file key management. The Bridge exposes access to these services through a RESTful API. In this way, the client can be completely naive of the Storj protocol and network while still taking advantage of the network. In addition, because the dedicated server can be relied on to have high uptime, the client can be integrated into unreliable user-space applications.

Bridge is designed to store only metadata. It does not cache encrypted shards and, with the exception of public buckets, does not hold encryption keys. The only knowledge of the file that Bridge 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 Bridge.

It is possible to envision Bridge upgrades that allow for different levels of delegated trust. A Bridge client may want to retain control over issuing and validating audits, or managing pointers to shards. Or a client may choose to authorize two or more unrelated Bridges to manage its audits in order to minimize the trust it places in either Bridge server. In the long run, any function of the data owner can be split across two or more parties by delegating trust.

5.3 Client Library

TODO Full documentation of the Bridge API is outside the scope of this whitepaper, but is available elsewhere [15]. The first complete client implementation is in JavaScript. Implementations in C, Python, and Java are in progress.

Because files cannot simply be POSTed to API endpoints, the structures of the Bridge API and client are different from existing object stores. Clients are implemented to hide the complexity of managing files on the storj network through simple and familiar interfaces. As much as possible, complex network operations are abstracted away behind simple function calls.

A brief summary of the upload process follows:

1. The client gathers and pre-processes data.
2. The client notifies Bridge of data awaiting upload.
3. Bridge negotiates contracts with network nodes.
4. Bridge returns the IP addresses of contracted nodes, and authorization tokens to the client.
5. The client uses the IP addresses and tokens to contact the farming nodes and upload the data.
6. The client transfers the audit information to the Bridge, delegating trust.
7. Bridge immediately issues an audit and verifies the response, to prove data was transferred correctly.
8. Bridge assumes responsibility for issuing audits, paying farmers, and managing file state.
9. Bridge exposes file metadata to the client via the API.

The download process is similar.

1. The client requests a file by an identifier.
2. Bridge validates the request and provides a list of farmer IP addresses and tokens.
3. The client uses the addresses and tokens to retrieve the file
4. The file is reassembled and decrypted client-side.

The JavaScript library accepts file, handles pre-processing, and manages connections as directed by Bridge. It also makes decrypted downloads available to applications as files, or as streams. A sample CLI using the library is available as Free Software at https://github.com/storj/core-cli. It has been tested with a wide variety of file sizes, and is capable of reliably streaming 1080p video from the Storj network.

5.3.1 Application Development Tools

TODO The primary function of Bridge and the Bridge API is to serve applications. To this end clients and tools in a wide variety of languages are under development.

Storj.js[16] seeks to provide a standard in-browser interface for downloading files from Storj. Though in early stages, it can already communicate with Bridge, retrieve file pointers and tokens, retrieve shards from farmers, reassemble shards, and append the completed file to the DOM. This allows web developers to easily reference Storj objects from within a page, and rely on them being delivered properly to the end user. This could be used to provide any service from in-browser document editing to photo storage.

Key and file management tools for web backends are in early planning stages, including Storj plugins for standard backend tools like content management systems. These tools should help content-driven application developers work with files on the Storj network. Standardizing these tools around permissioning files by user could help create data portability between services as discussed in section 4.2.

Bridges to other protocols and workflows are also planned. The Storj CLI lends itself to shell scripting automation. Similar tools for FTP, FUSE, and common tools for interacting with files will be developed in the future.

5.4 Bridge as an Authorization Mechanism

TODO Bridge can be used to manage authorization for private files stored on the network. Because Bridge manages the state of each contract under its care, it is a logical provider of these services. It can manage a variety of authorization-related services to enable sharing and collaboration.

5.4.1 Identity and Permissioning

TODO The Bridge API uses public-key cryptography to verify clients. Rather than the Bridge server issuing an API key to each user, users register public keys with the Bridge. API requests are signed, and the Bridge verifies that the signature matches a registered public key. Bridge organizes file metadata into buckets to facilitate management. Buckets can be permissioned individually by registering a set of public keys to the Bucket.

Application developers can use this to easily delegate permissions to applications, servers, or other developers. For instance, the developer of a file syncing service could create a keypair for each user of that service, and divide each user into a separate Bucket accessible only by that users keypair. Usage of each Bucket is tracked separately, so users who have exceeded their allotment could have write permissions revoked programmatically. This provides a logical separation of user permissions, as well as a variety of organizational tools.

5.4.2 Key Migration

TODO Because shard encryption keys are stored on the device that generated them, data portability is an issue. The reference implementation of Bridge and the client facilitate the transfer of file encryption keys between clients in a safe way. Clients generate a cryptographically strong seed, by default a randomly generated twelve word phrase. To encrypt a given file, the client generates a key deterministically based on the seed, Bucket ID and File ID.

The user can import the seed one time to each new device, which permanantly keeps the devices syncronized. This also facilitates backup since users only have to store the seed, not every newly generated file key.

5.4.3 Public Files

TODO Bridge, like other object stores, allows developers to create and disseminate public files via public Buckets. The Bridge server allows the developer to upload the encryption key, and then allows anonymous users to retrieve the file key and the set of file pointers. Public Buckets are useful for content delivery to webpages, or to public-facing applications.

A system to share and retrieve public files without need of a Bridge could also be created. Pointers and keys could be posted publicly on any platform, and clients could be required to pay farmers directly for downloads. In practice this would be very similar to an incentivized torrent. Platforms serving pointers function similarly to trackers facilitating torrents. It is unclear whether this system would have significant advantages over existing torrent networks.

5.4.4 File Sharing

TODO In the future, the Bridge could enable sharing of specific files between applications or users. Because all files co-exist on a shared network, this is a

problem of standardization and identity management.

Bridge could also use a third-party source of identity, like a PGP keyserver or Keybase[17], to enable secure person-to-person file sharing. A tiered keying strategy (as used by LastPass[16]) could also allow for the sharing of individual files. Other cryptographic schemes like proxy re-encryption seem promising. For a simplified example: if file keys are strongly encrypted and escrowed with a Bridge, files could be shared to any social media handle that could be authenticated via Keybase. Bridge could send the corresponding client a single encrypted file key along with a transposition key, thus enabling access to a file without exposing the file to Bridge, or modifying the file in any way.

A thorough description of these key management schemes is outside the scope of this paper. It is enough to note that they exist, that many useful strategies can be implemented in parallel, and that a dedicated Bridge can facilitate them in many useful ways.

5.5 Bridge as a Network Information Repository

TODO As noted earlier, data owners are responsible for negotiating contracts and managing file state. With enough information about peers on the network, contract selection becomes a powerful tool for maintaining file state. A Bridge will have many active contracts with many farmers, and will therefore have access to information about those farmers. A Bridge could use this information to intelligently distribute shards across a set of farmers in order to achieve specific performance goals.

For instance, via the execution of a contract, a Bridge node gathers data about the farmers communication latency, audit success rate, audit response latency, and availability. With minimal additional effort, the Bridge could also gather information about the nodes available bandwidth. By gathering a large pool of reliable data about farmers, a Bridge node can intelligently select a set of farmers that collectively provides a probabilistic guarantee of a certain quality of service.

In other words, the Bridge can leverage its knowledge about peers on the network to tailor the service to the clients requirements. Rather than a limited set of service tiers, a Bridge could assemble a package of contracts on the fly to meet any service requirement. This allows the client to determine the optimal latency, bandwidth, or location of a file, and have confidence that its goals will be met. For instance, a streaming video application may specify a need for high bandwidth, while archival storage needs only high availability. In a sufficiently

large network, any need could be met.

Secure distributed computation is an unsolved problem and, as such, each Bridge server uses its accumulated knowledge of the network. The Bridge is able to provide a probabilistic quality of service based on its knowledge the performance and reliability of farmers that a distributed network alone cannot provide.

5.6 Bridge as a Service

TODO In cases where the cost of delegating trust is not excessively high, clients may use third-party Bridges. Because Bridges do not store data and have no access to keys, this is still a large improvement on the traditional data-center model. Many of the features Bridge servers provide, like permissioning and intelligent contracting, leverage considerable network effects. Data sets grow exponentially more useful as they increse in size, indicating that there are strong economic incentives to share infrastructure and information in a Bridge.

Applications using object stores delegate significant amounts of trust to the storage providers. Providers may choose to operate public Bridges as a service. Application developers then delegate trust to the Bridge, 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 clients and Bridges. 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.

5.7 S3 gateway

TODO

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 Fast Byzantine Consensus

TODO

6.2 Distributed Repair

TODO

6.3 Federated Bridges

TODO Bridge nodes could cooperate to share data about the network in a mutually beneficial federation. This would allow each Bridge to improve the quality of service that it provides by improving the quality of information available.

Bridges could also, with the consent of users, cooperate to share file metadata and pointers among themselves. This would allow a user to access their file from any Bridge, rather than being dependent on a single Bridge. A tiered set of fallback Bridges storing the same access information is a desirable feature, as it hedges against downtime from a solo Bridge. Some solvable permissioning issues may exist, but there is no reason to believe a standard format and algorithm for syncing state across Bridges may not be developed.

6.4 Data Portability

TODO By encouraging use of data format and access standards, Storj aims to allow portability of data between applications. Unlike a traditional model, where control of data is tied to the service used to access the data, data access may be tied to individual users because Storj forms a common underlying layer. User data can be tied to persistent cryptographic identities, and authenticated without exposing data to third parties. Siloing data in applications is a harmful relic of traditional models. Building cross-compatibility into the future of data storage greatly improves user privacy and user experience.

Applications implementing these standards would be broadly compatible. When access is tied to users rather than services, privacy and control are preserved. A user may grant access to a service that backs up their hard drive, which places those files in Storj. The user could separately grant access to a photo-sharing service, which could then access any photos in the backup. The user gains seamless portability of data across many applications, and application

developers gain access to a large pool of existing users.

Permissioning in this system may be managed by a service like a Bridge, tied to a web of trust identity via services like Keybase, or handled by a distributed self-sovereign identity system. Smart contract systems, e.g. Ethereum [18] contracts, seem like a sensible long-term choice, as they can provide file permissions based on arbitrary code execution. Some problems may exist with respect to management of the private information required for identity and permissioning systems, but sufficient solutions likely exist.

While this system represents a significant step up in both usability and value, there are unmitigable security issues. Unfortunately, as in any cryptographic system, it is impossible to revoke access to data. Applications may cache data or forward it to third parties. Users, by definition, trust application developers to handle their data responsibly. To mitigate these risks, Storj Labs intends to provide incentives to developers to build free and open-source software. No application can be completely secure, but auditable code is the best defense of users privacy and security.

The potential advantages in terms of user experience and privacy are great, but more research is needed. Many open questions exist with respect to permissioning mechanisms. At worst a unified backend powering interoperable applications provides equivalent security to current data-center based models. Storj hopes to collaborate with other forward-thinking data-driven projects to create and advocate for these open standards.

6.5 Reputation Systems

TODO Storj, like many distributed networks, would profit immensely from a distributed reputation system. A reliable means of determining reputation on a distributed system is an unsolved problem. Several approaches have been detailed, and some implemented in practice but none have achieved consensus among researchers or engineers. A brief review of several of these approaches follows.

One inherent downside of distributing information across a network is the additional latency required for decisionmaking. It is difficult to say whether any distributed reputation system can accurately assess the bandwidth, latency, or availability of peers on a distributed network in a manner suitable to object storage, especially as market demand for these shifts over time. Nevertheless, a reliable distributed reputation would be an extremely useful tool for interacting with and understanding the network.

6.5.1 Eigentrust and Eigentrust++

TODO Eigentrust [19] attempts to generalize the ledger approach to generate global trust values in a distributed system using a transitive-trust model. Nodes keep and exchange trust vectors. For networks with a large majority of trustworthy peers, the value of each local trust vector converges to a shared global trust vector as nodes learn more about the network via information exchange.

Eigentrust++ [20] identifies several attack vectors and modifies Eigentrust to improve performance and reliability in the presence of malicious nodes. Eigentrust++ is currently implemented in NEM [21]. Secure global convergence to a shared trust value for each node is a key feature for any distributed reputation system.

6.5.2 TrustDavis

TODO TrustDavis [22] implements reputation as insurance. Nodes provide references for other nodes in the form of insurance contracts. Nodes seeking a prospective partner for an economic transaction also seek insurance contracts protecting them from the actions of that partner. Reputation in this system may be thought of as a graph, with vertices representing nodes, and directed edges representing the monetary value that a node is willing to stake on behalf of another. Nodes that are distant in this graph may still transact by purchasing a set of insurance contracts that traverses these edges. TrustDavis in practice thus encounters the same routing problem found on other distributed systems like the Lightning Network.

Denominating trust in terms of monetary value is attractive for an economic network like Storj, but the mechanics of insurance contracts in a system like this represent an extremely difficult problem. Notably, because failures to deliver payment propagate backwards through the insurance route, the financial burden always falls on the node that trusted an untrustworthy node, rather than the untrustworthy nodes.

6.5.3 Identity Maintenance Costs

TODO Storj is exploring a reputation system that leverages public Blockchains to solve a narrow set of identity problems [23]. This system requires nodes to spend money directly to maintain reputation. Nodes invest in their identity over time by making small standardized payments to their own Storj network node

ID. Because the ID is a Bitcoin address to which the node holds the private key, these funds are fully recoupable, except for the miners fees. In this system, nodes prefer to interact with nodes that have a long history of regular transactions. Over time these indicate monetary investment in an identity equal to the sum of the miners fees paid.

The payment required to participate in this system should be significantly less than the expected return of operating a network node. If set correctly, this recurring monetary payment for an identity bounds the size and duration of Sybil attacks without affecting cooperative nodes. Legitimate nodes would easily recoup their identity expense, while Sybil operators would find their expenses outstripping their returns. Unfortunately, this approach solves a relatively small subset of identity issues on the network, and it is difficult to see how it could be extended to other problem sets.

6.6 OFFER Loop Strategies

TODO Many negotiation strategies can exist and interact via the OFFER loop. Full exploration of negotiation strategies is beyond the scope of this paper, but a few interesting areas are immediately apparent. Simple examples include price floors and ceilings, but complex models could be built to base strategies on market trends and the subjective value of a shard. Negotiation strategies executed by autonomous agents are an area of (fascinating) ongoing research. Storj will be one of the first large-scale machine-driven marketplaces. As such, improving negotiation efficiency is critical to the long-term efficiency of the market.

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 Spartacus

TODO Spartacus attacks, or identity hijacking, are possible on Kademlia. Any node may assume the identity of another node and receive some fraction of messages intended for that node by simply copying its Node ID. This allows for targeted attacks against specific nodes and data. This is addressed by implementing Node IDs as ECDSA public key hashes and requiring messages be signed. A Spartacus attacker in this system would be unable to generate the corresponding private key, and thus unable to sign messages and participate in the network.

A.2 Sybil

Sybil attacks involve the creation of large amounts of nodes in an attempt to disrupt network operation by hijacking or dropping messages. Kademlia, because it relies on message redundancy and a concrete distance metric, is reasonably resistant to Sybil attacks. A nodes 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 until a large portion of the network consists of colluding Sybil nodes.

A.2.1 Google

The Google attack, or nation-state attack, is a hypothetical variant of the Sybil attack carried out by an entity with extreme resources. Google attacks are hard to address, as it is difficult to predict the actions of an organization with orders of magnitude more resources than the sum of the resources of network participants. The only reliable defence against a Google attack is to create a network whose resources are on the same order of magnitude as the attackers. At that scale, any attack against the network would represent an unsustainable

commitment of resources for such an organization.

A.2.2 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 he reaches a certain threshold he pulls the strings on each puppet to execute a hostage attack with the data involved, or simply drops each node from the network. Again, the best defence against this attack is to create a network of sufficient scale that this attack is ineffective. In the meantime, this can be partially addressed 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.3 Eclipse

TODO mention S/Kademlia

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.

A.3.1 Tunnel Eclipse

TODO Because tunneled connections rely on the tunnel provider, it is trivial for a tunnel provider to eclipse nodes for which it provides tunneled connections. This attack cannot affect publicly addressable nodes, so it can be trivially defeated with proper configuration. This attack can be mitigated by encrypting messages intended for tunneled nodes, thus removing the malicious tunnel provider's ability to inspect and censor incoming messages. Like a typical eclipse attack, any node that suspects it is the victim of a tunnel eclipse can easily generate a new Node ID, and find a new tunnel.

A.4 Hostage Bytes

The hostage byte attack is a storage-specific attack where malicious farmers 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 several nodes (see Section 2.7). As long as the client keeps the bounds of its erasure encoding a secret, the malicious farmer 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.

A.5 Cheating Owner

TODO A data owner may attempt to avoid paying a farmer for data storage by refusing to verify a correct audit. In response the farmer may drop the data-owners shard. This attack primarily poses a problem for any future distributed reputation system, as it is difficult for outside observers to verify the claims of either party. There is no known practical publicly verifiable proof of storage, and no known scheme for independently verifying that a privately verifiable audit was issued or answered as claimed. This indicates that a cheating client attack is a large unsolved problem for any reputation system.

A.6 Faithless Farmer

TODO While the farming software is built to require authentication via signature and token before serving download requests, it is reasonable to imagine a modification of the farming software that will provide shards to any paying requestor. In a network dominated by faithless farmers, any third-party can aggregate and inspect arbitrary shards present on the network.

However, even should faithless farmers dominate the network, data privacy is not significantly compromised. Because the location of the shards that comprise a given file is held solely by the data owner, it is prohibitively difficult to locate a target file without compromising the owner (see Section 6.3). Storj is not designed to protect against compromised data owners. In addition, should a third-party gather all shards, strong client-side encryption protects the contents of the file from inspection. The pointers and the encryption key may be secured separately. In the current implementation of Bridge, the pointers and the keys are held by the Bridge and the client, respectively.

A.7 Defeated Audit Attacks

TODO A typical Merkle proof verification does not require the verifier to know the depth of the tree. Instead the verifier is expected to have the data being validated. In the Storj audit tree, if the depth is unknown to the verifier the farmer may attack the verification process by sending a Merkle proof for any hash in the tree. This proof still generates the Merkle root, and is thus a valid proof of some node. But, because the verifier does not hold the data used to generate the tree, it has no way to verify that the proof is for the specific leaf that corresponds to the challenge. The verifier must store some information about the bottom of the tree, such as the depth of the tree, the set of leaves nodes, or the set of pre-leaves. Of these, the depth is most compact, and thus preferable.

Using the pre-leaf as an intermediary defeats another attack, where the farmer simply guesses which leaf corresponds to the current challenge. While this attack is unlikely to succeed, its trivially defeated by forcing the farmer to provide the pre-leaf. The farmer cannot know the pre-leaf before the challenge is issued. Requiring transmission of the pre-leaf also allows the data owner to proceed through the challenge set linearly instead of being forced to select randomly. This is desireable because it allows the data owner to maintain less state information per tree.

B Selected Calculations

The following are several interesting calculations related to the operation of the network.

B.1 Failure of k-of-n Erasure Coding

The chance of failure of k-of-n erasure coding, assuming probability p every shard stays online, is calculated as a binomial distribution:

$$\Pr_{failure}(n;k,p) = \sum_{i=0}^{k-1} p^i (1-p)^{n-i} \binom{n}{i}$$

n	k	p	$\Pr_{failure}\, n,k,p$
18	6	0.5	4.812e-02
18	6	0.75	3.424e-05
18	6	0.9	5.266e-10
18	6	0.98	6.391e-19
36	12	0.5	1.440e-02
36	12	0.75	2.615e-08
36	12	0.9	1.977e-17
36	12	0.98	1.628e-34

Code:

```
def fac(n): return 1 if n==0 else n * fac(n-1)
def choose(n,k): return fac(n) / fac(k) / fac(n-k)
def bin(n,k,p): return choose(n,k) * p ** k * (1-p) ** (n-k)
def prob_fail(n,k,p): return sum([bin(n,i,p) for i in range(0,k)])
```

Therefore, with well-chosen k and n, in addition to recovery methods described above, the statistical chance of shard or file loss is quite small.

B.2 Difficulty of Eclipsing a Target Node

The probability of eclipsing a targeted node in the a network with k nodes in h hashes is modeled by a similar binomial distribution:

$$\Pr_{success}(h,k) = \sum_{i=3}^{h-1} k^{-i} (1-\frac{1}{k})^{h-i} \binom{h}{i}$$

Code:

```
def fac(k): return 1 if k==0 else k * fac(k-1)
def choose(h,k): return fac(h) / fac(k) / fac(h-k)
def bin(i,h,k): return choose(h,i) * k ** -i * (1-(1.0/k)) ** (h-i)
def prob_succ(h,k): return sum([bin(i,h,k) for i in range(3,h)])
```

h	i	$\Pr_{success}\, h, i$
100	100	7.937e-02
100	500	1.120e-03
100	900	2.046e-04
500	100	8.766e-01
500	500	8.012e-02
500	900	1.888e-02
900	100	9.939e-01
900	500	2.693e-01
900	900	8.020e-02

B.3 Beach Size

As the number of shards on the network grows, it becomes progressively more difficult to locate a given file without prior knowledge of the locations of its shards. This implies that even should all farmers become faithless, file privacy is largely preserved.

The probability of locating a targeted file consisting of k shards by n random draws from a network containing N shards is modeled as a hypergeometric distribution with $K = k$:

$$Pr_{Success}(N, k, n) = \frac{\binom{N-k}{n-k}}{\binom{N}{n}}$$

N	k	n	$\Pr_{success}\, N, k, n$
100	10	10	5.777e-14
100	10	50	5.934e-04
100	10	90	3.305e-01
100	50	50	9.912e-30
100	50	90	5.493e-04
500	50	200	1.961e-22
500	50	400	7.361e-06
900	10	200	2.457e-07
900	10	400	2.823e-04
900	10	800	3.060e-01
900	50	200	1.072e-35
900	50	400	4.023e-19
900	50	800	2.320e-03

Code:

```
def fac(k): return 1 if k==0 else k * fac(k-1)
def choose(h,k): return fac(h) / fac(k) / fac(h-k)
def hyp(N,k,n): return choose(N-k,n-k) / float(choose(N,n))
def prob_success(N,k,n): return hyp(N,k,n)
```

B.4 Partial Audit Confidence Levels

Farmers attempting to game the system may rely on data owners to issue partial audits. Partial audits allow false positives, where the data appears intact, but in fact has been modified. Data owners may account for this by ascribing confidence values to each partial audit, based on the likelihood of a false positive. Partial audit results then update prior confidence of availability. Data owners may adjust audit parameters to provide desired confidence levels.

The probability of a false positive on a parital audit of n bytes of an N byte shard, with K bytes modified adversarially by the farmer is a hypergeometric distribution with $k = 0$:

$$Pr_{falsepositive}(N, K, n) = \frac{\binom{N-K}{n}}{\binom{N}{n}}$$

N	K	n	$\Pr_{falsepositive} N, K, n$
8192	512	512	1.466e-15
8192	1024	512	1.867e-31
8192	2048	512	3.989e-67
8192	3072	512	1.228e-109
8192	4096	512	2.952e-162

Code:

```
def fac(k): return 1 if k==0 else k * fac(k-1)
def choose(h,k): return fac(h) / fac(k) / fac(h-k)
def hyp(N,K,n): return float(choose(N-K, n) / choose(N,n)
def prob_false_pos(N,K,n): return hyp(N,K,n)
```

As demonstrated, the chance of false positives on even small partial audits becomes vanishingly small. Farmers failing audits risk losing payouts from current contracts, as well as potential future contracts as a result of failed audits. Dropping 10% of a shard virtually guarantees a loss greater than 10% of the contract value. Thus it stands to reason that partially deleting shards to increase perceived storage capcity is not a viable economic strategy.

C Reed-Solomon

TODO

D Kademlia

TODO

E S/Kademlia

TODO

F Macaroons

TODO

References

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

[2] Backblaze Inc. How long do hard drives last: 2018 hard drive stats, 2018.

[3] 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.

[4] 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.

[5] G. Hall. Diglet server, (2016). `http://diglet.me`.

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

[7] J. S. Plank. A tutorial on reed-solomon coding for fault-tolerance in raid-like systems, (1996). `http://web.eecs.utk.edu/~plank/plank/papers/CS-96-332.pdf`.

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

[9] B. Cohen. Incentives build robustness in bittorrent, (2003). `http://www.bittorrent.org/bittorrentecon.pdf`.

[10] R.C. Merkle. Protocols for public key cryptosystems, (April 1980). `http://www.merkle.com/papers/Protocols.pdf`.

[11] H. Shacham, B. Waters. Compact proofs of retrievability, (2008). `https://cseweb.ucsd.edu/~hovav/dist/verstore.pdf`.

[12] G. Hall. Storj core class: Contract, (2016). `http://storj.github.io/core/Contract.html`.

[13] F. Barkhau. Trustless micropayment channels, (2016). `https://github.com/F483/counterparty-documentation/blob/micropayments/Developers/micropayments.md`.

[14] J. Poon, T. Dryja. The bitcoin lightning network: Scalable off-chain instant payments, (2016). https://lightning.network/lightning-network-paper.pdf.

[15] G. Hall. Storj core, (2016). http://storj.github.io/core/.

[16] LogMeIn Inc. Lastpass password manager. https://www.lastpass.com.

[17] Keybase Inc. Keybase. https://www.keybase.io.

[18] V. Buterin *et al.* A next-generation smart contract and decentralized application platform, (2014). https://github.com/ethereum/wiki/wiki/White-Paper.

[19] S. Kamvar, M. Schlosser, H. Garcia-Molina. The eigentrust algorithm for reputation management in p2p networks, (2003). http://ilpubs.stanford.edu:8090/562/1/2002-56.pdf.

[20] X. Fan, L. Liu, M. Li, Z. Su. Eigentrust++: Attack resilient trust management, (2012). https://pdfs.semanticscholar.org/c5a6/c05d833179073e3517be6c7e7ab0c9d59b29.pdf.

[21] M. Takemiya *et al.* Nem technical reference, (2015). https://www.nem.io/NEM_techRef.pdf.

[22] D. DeFigueiredo, E. Barr. Trustdavis: A non-exploitable online reputation system, (2005). http://earlbarr.com/publications/trustdavis.pdf.

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