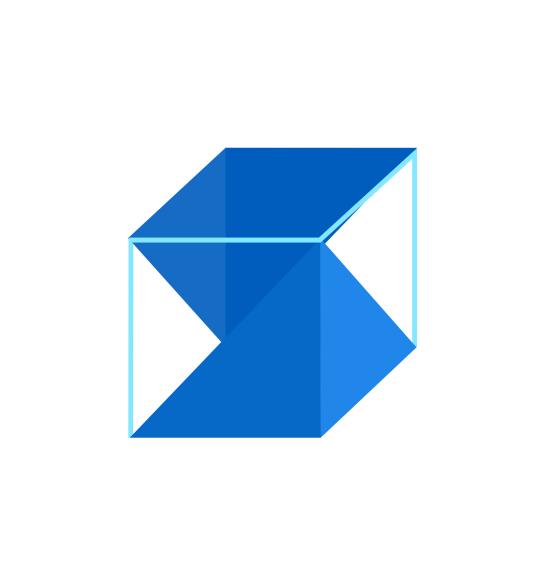
BAISS



BAISS： An AI Block-chain Project Provide Decentralized Platform Service  
IT IS NEW STANDER OF BLOCKCHAIN TECHNOLOGY

BAISS LAB, BAISS Foundation, BAISS Organization

November 12, 2020

V1.0

[https://github.com/BAISS/whitepaper](https://github.com/storj/whitepaper)

Copyright © 2020 BAISS Labs, BAISS Foundation, BAISS Organization

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 license (CC BY-SA 4.0).

All product names, logos, and brands used or cited in this document are properties of their respective own­ers. All company, product, and service names used herein are for identification purposes only. Use of these names, logos, and brands does not imply endorsement.

Contents

[0.1 Abstract 6](#bookmark41" \o "Current Document)

[0.2 Contributors 6](#bookmark45" \o "Current Document)

1. **[Introduction 7](#bookmark49" \o "Current Document)**
2. **[BAISS design constraints 9](#bookmark53" \o "Current Document)**
   1. [Security and privacy 9](#bookmark57" \o "Current Document)
   2. [Decentralization 9](#bookmark62" \o "Current Document)
   3. [Marketplace and economics 10](#bookmark67" \o "Current Document)
   4. [Amazon S3 compatibility 12](#bookmark72" \o "Current Document)
   5. [Durability, device failure, and churn 12](#bookmark88" \o "Current Document)
   6. [Latency 13](#bookmark93" \o "Current Document)
   7. [Bandwidth 14](#bookmark98" \o "Current Document)
   8. [Object size 15](#bookmark103" \o "Current Document)
   9. [Byzantine fault tolerance 15](#bookmark108" \o "Current Document)
   10. [Coordination avoidance 16](#bookmark113" \o "Current Document)
3. **platform 18**
   1. [platform overview 18](#bookmark119" \o "Current Document)
   2. [super node 19](#bookmark132" \o "Current Document)
   3. [Peer-to-peer communication and discovery 19](#bookmark137" \o "Current Document)
   4. [Redundancy 19](#bookmark142" \o "Current Document)
   5. [Metadata 23](#bookmark150" \o "Current Document)
   6. [Encryption 24](#bookmark155" \o "Current Document)
   7. [Audits and reputation 25](#bookmark160" \o "Current Document)
   8. [Data repair 25](#bookmark165" \o "Current Document)
   9. [Payments 26](#bookmark170" \o "Current Document)

**4 BC+AI=DDSSP ............................................................................ 27**

4.1 definitions

4.2 Peer classes

4.3 super node

4.4 node identity

4.5 peer to peer communication

4.6 node discovery

4.7 redundancy

4.8 structured file storage

4.9 metadata

4.10 node

4.11 encryption

4.12 authorization

4.13 audits

4.14 data repair

4.15 super node reputation

4.16 payments

4.17 bandwidth allocation

4.18 node reputation

4.19 Garbage collection

4.20 Mining Machine

4.21 Quality control and branding

**5 Road map ............................................................................ 27**

**6 Future work ............................................................................ 27**

**7 Reference ............................................................................ 27**

CONTENTS

0.1 Abstract

An AI Block-chain Project Provide Decentralized Platform Service represents a fundamental shift in the efficiency and ­economics of large-scale computing. Eliminating central control allows users to compute and share data without reliance on a third-party storage provider. Decentralization mitigates the risk of data failures and outages while simultaneously increasing the security and privacy of object computing. It also allows market forces to optimize for less expensive or no cost service at a greater rate than any single centralized provider afford. Although there are many ways to build such a system, there are some specific responsibilities any given implementation should address. Based on our experience with petabyte-scale platform service system, we introduce an AI Block-chain Project Provide Decentralized platform for considering these responsibilities and for building our distributed network. Additionally, we describe an initial concrete implementation for the entire platform.

Please address correspondence to [paper@BAISS.org.](mailto:paper@storj.io)

1. Introduction

The Internet is a monstrous decentralized and distributed network comprising billions of devices which are not constrained by a single company or person. A significant part of the data currently accessible through the Internet is a centralized system and is put away with a bunch of technology organizations that have the experience and funding to assemble huge server farms equipped for taking care of this huge measure of data. A couple of the difficulties looked by server farms are: data breaches, periods of unavailability on a grand scale, storage costs, and expanding and upgrading quickly enough to meet user demand for faster data and larger formats.

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

News coverage of data breaches over the past few years has shown us that the ­frequency of such breaches has been increasing by as much as a factor of 20 between 2005 and 2020 [[1]](#bookmark453" \o "Current Document). Decentralized service platform's process of protecting data makes data breaches more difficult than current methods used by data centers while, at the same time, costing less than current methods.

This model can address the rapidly expanding amount of data for which current solution struggle. With an anticipated 44 zettabytes of data exist by 2020 and a market that already grows to 92 billion USD in the same time frame [[2]](#bookmark455" \o "Current Document), we have identified several key market segments that decentralized computing service platform has the potential to ad­dress. As decentralized computing service capabilities evolve, it will be able to address a much wider range of use cases from basic object computing to content delivery networks (CDN).

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

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

We have organized the rest of this paper into seven additional chapters. Chapter [2](#bookmark51" \o "Current Document) dis­cusses the design space in which BAISS operates and the specific constraints on which our optimization efforts are based. Chapter [3](#bookmark116" \o "Current Document) covers our platform. Chapter [4](#bookmark173" \o "Current Document) proposes a new standard of block-chain DDSSP, while chapter [5](#bookmark299" \o "Current Document) explains what is our road map. Chapter [6](#bookmark347" \o "Current Document) covers future work. Finally, chapter [7](#bookmark356" \o "Current Document) covers references.

2. BAISS design constraints

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

1. Security and privacy

Any platform must ensure both the privacy and security of data stored regardless of whether it is centralized or decentralized. An AI Block-chain decentralized platforms must reduce any additional layer of complexity and risk link with the storage of data on unknown or mistrust node. Because decentralized service platforms cannot take many of the same shortcuts data center based approaches can (e.g. firewalls, DMZs, etc.), decentralized service platform must be designed from the scratch to support not only end-to- end encryption but also enhanced security and privacy at all levels of the system.

Different categories of data are also subject to different regulatory compliance. For exam­ple, the United States legislation for the Health Insurance Portability and Accountability Act (HIPAA) has special requirements for data center compatibility. European countries have to consider the General Data Protection Regulation (GDPR) regarding how single information must be protected and secured. Many users outside of the United States may feel they have significant geopolitical reasons to consider storing data in a way that limits the ability for US-based entities to impact their privacy [[3]](#bookmark458" \o "Current Document). There are many other regulations in other sectors regarding user's data privacy.

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

1. Decentralization

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

One of the main benefits for choose decentralization is to cut down costs for maintenance, and utilities. We believe that the number of smaller operators can be significantly underutilized . As building a decentralized platform, we have to focus a long tail of resources that are mostly unused that could provide affordable and geographic­ally distributed service. Conceivably, many small operators might have access to lower cost electricity than standard centralized centers. Some small operators environments are not enough to run a centralized system. We have found that in aggregate, enough small operator environments exist such that their combination over the internet consti­tutes significant opportunity and advantage for less-expensive and faster storage.

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

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

1. Marketplace and economics

Public cloud computing and storage, in particular, have proven to be an at­tractive business model for the large centralized cloud providers. Cloud computing is estimated to be a $196.4 billion dollar market in 2020, and is expected to reach $302.5 billion by 2021 [[4]](#bookmark460" \o "Current Document).

The public cloud computing and storage model has provided a compelling economic model to end users. Not only does it enable end users to scale on demand but also allows them to avoid the significant fixed costs of facilities, power, and data center personnel. Public cloud computing and storage has generally proven to be an economical, durable, and performant option for many end users when compared to on-premise solutions.

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

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

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

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

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

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

Additionally, the network must be able to account for ensuring efficient, timely billing and payment processes as well as regulatory compliance for tax and other reporting. To be as globally versatile as possible with payments, our network must be robust to accom­modate several types of transactions (such as cryptocurrency, bank payments, and other forms of barter).

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

1. Amazon S3 compatibility

At the time of this paper's publication, the most widely deployed public cloud is Amazon Web Services [[5]](#bookmark463" \o "Current Document). Amazon Web Services not only is the largest cloud services ecosystem but also has the benefit of first mover advantage. Amazon's first cloud services product was Amazon Simple Storage Service, or Amazon S3 for short. Public numbers are hard to come by but Amazon S3 is likely the most widely deployed cloud computing and storage protocol in existence. Most cloud computing and storage products provide some form of compatibility with the Amazon S3 application program interface (API) architecture.

Our objective is to aggressively compete in the wider cloud computing and storage industry and bring decentralized cloud computing and storage into the mainstream. Until a decentralized cloud computing and storage protocol becomes widely adopted, Amazon S3 compatibility creates a graceful transi­tion path from centralized providers by alleviating many switching costs for our users. To achieve this, the BAISS implementation allows applications previously built against Ama­zon S3 to work with BAISS with minimal friction or changes. S3 compatibility adds aggres­sive requirements for feature set, performance, and durability. At a bare minimum, this requires the methods described in Figure [2.1](#bookmark75" \o "Current Document) to be implemented.

1. // Bucket operations
2. CreateBucket(bucketName)
3. DeleteBucket(bucketName)
4. ListBuckets ()

5

1. // Object operations
2. GetObject(bucketName, objectPath, offset, length)
3. PutObject(bucketName, objectPath, data , metadata)
4. DeleteObject(bucketName, obj ectPath)
5. ListObjects(bucketName, prefix, startKey, limit, delimiter)

*Figure 2.1： Minimum S3 API*

1. Durability, device failure, and churn

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

For all devices, component failure is a guarantee. All hard drives fail after enough wear [[6]](#bookmark465" \o "Current Document) and servers providing network access to these hard drives will also eventually fail. Network links may die, power failures could cause havoc sporadically, and storage media become unreliable over time. Data must be stored with enough redundancy to recover from individual component failures. Perhaps more importantly, no data can be left in a single location indefinitely. In such an environment, redundancy, data mainte­nance, repair, and replacement of lost redundancy must be considered inevitable, and the system must account for these issues.

Furthermore, decentralized systems are susceptible to high churn rates where partic­ipants join the network and then leave for various reasons, well before their hardware has actually failed. For instance, Rhea *et al.* found that in many real world peer-to-peer systems, the median time a participant lasts in the network ranges from hours to mere minutes [[7]](#bookmark467" \o "Current Document). Maymounkov *et al.* found that the probability of a node staying connected to a decentralized network for an additional hour is an *increasing* function of uptime (Fig­ure 2.2 [[8]](#bookmark469" \o "Current Document)). In other words, nodes that have been online for a long time are less likely to contribute to overall node churn.

Churn could be caused by any number of factors. super node may go offline due to hardware or software failure, intermittent internet connectivity, power loss, complete disk failure, or software shutdown or removal. The more network churn that exists, the more redundancy is required to make up for the greater rate of node loss. The more redundancy that is required, the more bandwidth is needed for correct operation of the system. In fact, there is a tight relationship between network churn, additional redundancy, and bandwidth availability [[9]](#bookmark471" \o "Current Document). To keep background bandwidth usage and redundancy low, our network must have low network churn and a strong incentive to favor long-lived, stable node.

See section [7.3.3](#bookmark381" \o "Current Document) and Blake *et al.* [[9]](#bookmark471" \o "Current Document) for a discussion of how to repair bandwidth varies as a function of node churn.

1. Latency

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

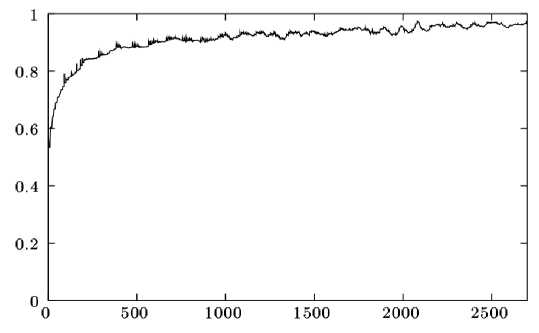


Figure 2.2： Probability of remaining online an additional hour as a function of uptime. The x axis represents minutes. The y axis shows the fraction of node that stayed online at least x minutes that also stayed online at least x + 60 minutes. Source: Maymounkov et al. [[8]](#bookmark469" \o "Current Document)

1. Bandwidth

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

In the United States and other countries, the method in which many residential in­ternet service providers (ISPs) operate presents two specific challenges for designers of a decentralized network protocol. The first challenge is the asymmetric internet connec­tions offered by many ISPs. Customers subscribe to internet service based on an adver­tised download speed, but the upload speed is potentially an order of magnitude or two slower. The second challenge is that bandwidth is sometimes "capped" by the ISP at a fixed amount of allowed traffic per month. For example, in many US markets, the ISP Comcast imposes a one terabyte per month bandwidth cap with stiff fines for customers who go over this limit [[10]](#bookmark473" \o "Current Document). An internet connection with a cap of 1 TB/month cannot av­erage more than 385 KB/s over the month without exceeding the monthly bandwidth allowance, even if the ISP advertises speeds of 10 MB/s or higher. Such caps impose sig­nificant limitations on the bandwidth available to the network at any given moment.

With device failure and churn guaranteed, any decentralized system will have a corre­sponding amount of repair traffic. As a result, it is important to account for the bandwidth required not only for data storage and retrieval but also for data maintenance and re­pair [[9]](#bookmark471" \o "Current Document). Designing a storage system that is careless with bandwidth usage would not only give undue preference to super node operators with access to unlimited high-speed bandwidth but also centralize the system to some degree. In order to keep the storage system as decentralized as possible and working in as many environments as possible, bandwidth usage must be aggressively minimized.

Please see section [7.1.1](#bookmark365" \o "Current Document) for a discussion on how bandwidth availability and repair traffic limit usable space.

1. Object size

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

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

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

1. Byzantine fault tolerance

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

We adopt the Byzantine, Altruistic, Rational (BAR) model [[11]](#bookmark475" \o "Current Document) to discuss participants in the network.

-*Byzantine* node may deviate arbitrarily from the suggested protocol for any reason. Some examples include nodes that are broken or nodes that are actively trying to sabotage the protocol. In general, a *Byzantine* node is a bad actor, or one that optimizes fora utility function that is independent of the one given for the suggested protocol.

-Inevitable hardware failures aside, *Altruistic* nodes are good actors and participate in a proposed protocol even if the rational choice is to deviate.

-*Rational* nodes are neutral actors and participate or deviate only when it is in their net best interest.

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

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

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

Note that creating a system that is robust in the face of Byzantine behavior does not require a Byzantine fault tolerant consensus protocol—we avoid Byzantine consensus. See sections [4.9,](#bookmark231" \o "Current Document) [6.2,](#bookmark352" \o "Current Document) and appendix [A](#bookmark385" \o "Current Document) for more details.

1. Coordination avoidance

A growing body of distributed database research shows that systems that avoid coordination wherever possible have far better throughput than systems where sub components are forced to coordinate to achieve correctness [[12](#bookmark477" \o "Current Document)[-19]](#bookmark490" \o "Current Document). We use Bailis *et al:s* informal def­inition that coordination is the requirement that concurrently executing operations syn­chronously communicate or otherwise stall in order to complete [[16]](#bookmark484" \o "Current Document). This observation happens at all scales and applies not only to distributed networks but also to concurrent threads of execution coordinating within the same computer. As soon as coordination is needed, actors in the system will need to wait for other actors, and waiting—due to coordination issues—can have a significant cost.

While many types of operations in a network may require coordination (e.g., opera­tions that require linearizabilit[y](#bookmark0" \o "Current Document)[[[1]](#footnote-0)](#bookmark0" \o "Current Document)[[15,](#bookmark482" \o "Current Document)[20,](#bookmark492" \o "Current Document)[21]](#bookmark494" \o "Current Document)), choosing strategies that avoid coordination (such as Highly Available Transactions [[15]](#bookmark482" \o "Current Document)) can offer performance gains of two to three orders of magnitude over wide area networks. In fact, by carefully avoiding coordination as much as possible, the Anna database [[17]](#bookmark486" \o "Current Document) is able to be 10 times faster than both Cas­sandra and Redis in their corresponding environments and 700 to 800 times faster than performance-focused in-memory databases such as Masstree or Intel's TBB [[22]](#bookmark496" \o "Current Document). Not all coordination can be avoided, but new platforms (such as Invariant Confluence [[16]](#bookmark484" \o "Current Document) or the CALM principle [[18,](#bookmark488" \o "Current Document) [19]](#bookmark490" \o "Current Document)) allow system architects to understand when coordination is required for consistency and correctness. As evidenced by Anna's performance successes, it is most efficient to avoid coordination where possible.

Systems that minimize coordination are much better at scaling from small to large workloads. Adding more resources to a coordination-avoidance system will directly increase throughput and performance. However, adding more resources to a coordination­dependent system (such as Bitcoin [[23]](#bookmark498" \o "Current Document) or even Raft [[24]](#bookmark500" \o "Current Document)) will not result in much additional throughput or overall performance.

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

3. platform

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

1. platform overview

All designs within our platform will do the following things：

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

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

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

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

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

The individual components are：

1. super node
2. Peer-to-peer communication and discovery
3. Redundancy
4. Meta data
5. Encryption
6. Audits and reputation
7. Data repair
8. Payments
9. super node

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

Because super nodes are selected via changing variables external to the protocol, node selection is an explicit, non-deterministic process in our platform. This means that we must keep track of which node were selected for each upload via a small amount of metadata； we can't select node for storing data implicitly or deterministically as in a system like Dynamo [[25]](#bookmark502" \o "Current Document). As with GFS [[26]](#bookmark504" \o "Current Document), HDFS [[27]](#bookmark506" \o "Current Document), or Lustre [[28]](#bookmark508" \o "Current Document), this decision implies the requirement of a metadata storage system to keep track of selected node (seesection [3.5)](#bookmark148" \o "Current Document).

1. Peer-to-peer communication and discovery

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

-provides peer reachability, even in the face of firewalls and NATs where possible. This may require techniques like STUN [[29]](#bookmark510" \o "Current Document), UPnP [[30]](#bookmark512" \o "Current Document), NAT-PMP [[31]](#bookmark514" \o "Current Document), etc.

-provides authentication as in S/Kademlia [[32]](#bookmark516" \o "Current Document), where each participant cryptograph­ically proves the identity of the peer with whom they are speaking to avoid man-in- the-middle attacks.

-provides complete privacy. In cases such as bandwidth measurement (see section [4.17)](#bookmark271" \o "Current Document), the client and super node must be able to communicate without any risk of eavesdroppers. The protocol should ensure that all communications are private by default.

Additionally, the platform requires a way to look up peer network addresses by a unique identifier so that, given a peer's unique identifier, any other peer can connect to it. This responsibility is similar to the internet's standard domain name system (DNS) [[33]](#bookmark518" \o "Current Document), which is a mapping of an identifier to an ephemeral connection address, but unlike DNS, there can be no centralized registration process. To achieve this, a network overlay, such as Chord [[34]](#bookmark520" \o "Current Document), Pastry [[35]](#bookmark522" \o "Current Document), or Kademlia [[8]](#bookmark469" \o "Current Document), can be built on top of our chosen peer-to-peer communication protocol. See Section [4.6](#bookmark203" \o "Current Document) for implementation details.

1. Redundancy

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

For example, suppose a certain desired level of durability requires a replication strategy that makes eight copies of the data. This yields an expansion factor of 8x, or 800%. This data then needs to be stored on the network, using bandwidth in the process. Thus, more replication results in more bandwidth usage for a fixed amount of data. As discussed in the protocol design constraints (section [2.7)](#bookmark96" \o "Current Document) and Blake *et al.* [[9]](#bookmark471" \o "Current Document), high bandwidth usage prevents scaling, so this is an undesirable strategy for ensuring a high degree of file dura­bility.

As an alternative to simple replication, *erasure codes* provide a much more efficient method to achieve redundancy. Erasure codes are well-established in use for both dis­tributed and peer-to-peer platform service system [[36](#bookmark524" \o "Current Document)[-42]](#bookmark536" \o "Current Document). Erasure codes are an encoding scheme for manipulating data durability without tying it to bandwidth usage, and have been found to improve repair traffic significantly over replication [[9]](#bookmark471" \o "Current Document). Importantly, they allow changes in durability without changes in expansion factor.

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

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

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

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

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

|  |  |  |  |
| --- | --- | --- | --- |
| *k* | *n* | Exp. factor | *P(D* | *p* = 10%) |
| 1 | 1 | 1 | 90.483741803595962% |
| 1 | 2 | 2 | 98.247690369357827% |
| 1 | 3 | 3 | 99.640050681691051% |
| 1 | 10 | 10 | 99.999988857452166% |
| 1 | 16 | 16 | 99.999999998036174% |

To see how these tables were calculated, we'll start with the simplifying assumption that *p* is the monthly node churn rate (that is, the fraction of node that will go offline in a month on average). Mathematically, time-dependent processes are modeled according to the Poisson distribution, where it is assumed that *λ* events are observed in the given unit of time. As a result, we model durability as the cumulative distribution function (CDF) of the Poisson distribution with mean *λ*=*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:

,

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

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

By being able to tweak the durability independently of the expansion factor, erasure coding allows very high durability to be achieved with surprisingly low expansion factors. Because of how limited bandwidth is as a resource, completely eliminating replication as a strategy and using erasure codes only for redundancy causes a drastic decrease in bandwidth footprint.

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

1. **Erasure codes' effect on streaming**

Erasure codes are used in many streaming contexts such as audio CDs and node com­munications [[38]](#bookmark528" \o "Current Document), so it's important to point out that using erasure coding in general does not make our streaming design requirement (required by Amazon S3 compatibility, see section [2.4)](#bookmark70" \o "Current Document) more challenging. Whatever erasure code is chosen for our platform, as with CDs, streaming can be added on top by encoding small portions at a time, instead of attempting to encode a file all at once. See section [4.8](#bookmark220" \o "Current Document) for more details.

1. **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 [[43]](#bookmark538" \o "Current Document). A long-tail response occurs in situations where a needed server has an unreasonably slow operation time due to a confluence of un­predictable factors. Long-tail responses are so-named due to their rare average rate of occurrence but highly variable nature, which is a probability density graph looks like a "long tail." In aggregate, long-tail responses are a big issue in distributed system design.

In Map Reduce, long-tail responses are called "stragglers." Map Reduce executes redun­dant requests called "backup tasks" to make sure that if specific stragglers take too long, the overall operation can still proceed without waiting. If the backup task mechanism is disabled in Map Reduce, basic operations can take 44% longer to complete, even though the backup task mechanism is causing duplicated work [[44]](#bookmark540" \o "Current Document).

By using erasure codes, we are in a position to create Map Reduce-like backup tasks for storage [[39,](#bookmark530" \o "Current Document)[40]](#bookmark532" \o "Current Document). For uploads, a file can be encoded to a higher *(k,n*) ratio than necessary for desired durability guarantees. During an upload, after enough pieces have uploaded to gain required redundancy, the remaining additional uploads can be canceled. This cancellation allows the upload to continue as fast as the fastest node in a set, instead of waiting for the slowest node.

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

The outcome is that every request is satisfiable by the fastest node 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 node of a random sub population turns what could be a potential liability (highly variable performance from individual actors) into a great source of strength fora distributed storage network, while still providing great load balancing characteristics.

This ability to over-encode a file greatly assists dynamic load balancing of popular con­tent on the network. See section [6.1](#bookmark348" \o "Current Document) for a discussion on how we plan to address load balancing very active files.

1. Metadata

Once we split an object up with erasure codes and select super node on which to store the new pieces, we now need to keep track of which super node we selected. We allow users to choose storage based on geographic location, performance characteristics, available space, and other features. Therefore, instead of implicit node selection such as a scheme using consistent hashing like Dynamo [[25]](#bookmark502" \o "Current Document), we must use an explicit node selection scheme such as directory-based lookups [[45]](#bookmark542" \o "Current Document). Additionally, to maintain Amazon S3 compatibility, the user must be able to choose an arbitrary key, often treated like a path, to identify this mapping of data pieces to node. These features imply the necessity of a metadata storage system.

Amazon S3 compatibility once again imposes some tight requirements. We should support: hierarchical objects (paths with prefixes), per-object key/value storage, arbitrarily large files, arbitrarily large amounts of files, and so forth. Objects should be able to be stored and retrieved by arbitrary key; in addition, deterministic iteration over those keys will be required to allow for paginated listing.

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

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

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

An additional platform focus is enabling this component-metadata storage—to be interchangeable. Specifically, we expect the platform to incorporate multiple implemen­tations of metadata storage that users will be allowed to choose between. This greatly assists with our design goal of coordination avoidance between users (see section [2.10)](#bookmark111" \o "Current Document).

Aside from scale requirements, to implement Amazon S3 compatibility, the desired API is straightforward and simple： *Put* (store metadata at a given path), *Get* (retrieve meta­data at a given a path), *List* (paginated, deterministic listing of existing paths), and *Delete* (remove a path). See Figure [2.1](#bookmark75" \o "Current Document) for more details.

1. Encryption

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

Encryption should use a pluggable mechanism that allows users to choose their de­sired encryption scheme. It should also store metadata about that encryption scheme to allow users to recover their data using the appropriate decryption mechanism in cases where their encryption choices are changed or upgraded.

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

Because each file should be encrypted differently with different keys and potentially different algorithms, the metadata about that encryption must be stored somewhere in a manner that is secure and reliable. This metadata, along with other metadata about the file, including its path, will be stored in the previously discussed metadata storage system, encrypted by a deterministic, hierarchical encryption scheme. A hierarchical encryption scheme based on BIP32 [[46]](#bookmark544" \o "Current Document) will allow sub trees to be shared without sharing their parents and will allow some files to be shared without sharing other files. See section [4.11](#bookmark240" \o "Current Document) for a discussion of our path-based hierarchical deterministic encryption scheme.

1. Audits and reputation

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

Many platform service systems use probabilistic per-file audits, called *proofs of retrievability,* as a way of determining when and where to repair files [[47,](#bookmark546" \o "Current Document)[48]](#bookmark548" \o "Current Document). We are extending the probabilistic nature of common per-file proofs of retrievability to range across all possible files stored by a specific node. Audits, in this case, are probabilistic challenges that confirm, with a high degree of certainty and a low amount of overhead, that a super node is well- behaved, keeping the data it claims, and not susceptible to hardware failure or malintent. Audits function as “spot checks" [[49]](#bookmark550" \o "Current Document) to help calculate the future usefulness of a given super node.

In our storage system, audits are simply a mechanism used to determine a node's degree of stability. Failed audits will result in a super node being marked as bad, which will result in redistributing data to a new node and avoiding that node altogether in the future. super node uptime and overall health are the primary metrics used to determine which files need repair.

As is the case with proofs of retrievability [[47,](#bookmark546" \o "Current Document) [48]](#bookmark548" \o "Current Document), this auditing mechanism does not audit all bytes in all files. This can leave room for false positives, where the verifier believes the super node retains the intact data when it has actually been modified or partially deleted. Fortunately, the probability of a false positive on an individual partial audit is easily calculable (see section [7.2)](#bookmark367" \o "Current Document). When applied iteratively to a super node as a whole, detection of missing or altered data becomes certain within a known and modifiable error threshold.

A reputation system is needed to persist the history of audit outcomes for given node identities. Our overall platform has flexible requirements on the use of such a system, but see section [4.15](#bookmark261" \o "Current Document) for a discussion of our initial approach.

1. Data repair

Data loss is an ever-present risk in any distributed storage system. While there are many potential causes for file loss, super node churn (super node joining and leaving the network) is the largest leading risk by a significant degree compared to other causes. As discussed in section [2.5,](#bookmark86" \o "Current Document) network session time in many real world systems range from hours to mere minutes [[7]](#bookmark467" \o "Current Document). While there are many other ways data might get lost, such as corruption, malicious behavior, bad hardware, software error, or user initiated space reclamation, these issues are less serious than full node churn. We expect node churn to be the dominant cause of data loss in our network.

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

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

1. Payments

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

For our platform to achieve low latency and high throughput, we must not have transactional dependencies on a blockchain (see section [2.10)](#bookmark111" \o "Current Document). This means that an ade­quately performant storage system cannot afford to wait for blockchain operations. When operations should be measured in milliseconds, waiting for a cluster of node to proba­bilistically come to an agreement on a shared global ledger is a non-starter.

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

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

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

4. BC+AI=DDSSP

We believe the Dynamic Distributed Searching Storage Protocol (DDSSP), we've developed to be relatively fundamental given our design constraints. However, within the DDSSP still remains some freedom in choosing how to implement each component.

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

As with our previous version [[37]](#bookmark526" \o "Current Document), we will publish changes to this concrete architecture through our BAISS Improvement Proposal process [[50]](#bookmark552" \o "Current Document).

1. Definitions

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

1. **Actors**

**Business Client** A user or application that will develop AIDapp or train data from the network.

**Mining Machine** A cohesive collection of network services and responsibilities. It includes any application or service that implements *liblink* and wants to store and/or retrieve data. It is not expected to remain online like the other two function and is relatively lightweight. It performs encryption, erasure encoding, and coordinates with the other behalf of the customer/client.

**liblink** A library which provides all necessary functions to interact with *super node* and directly. This library will be available in a number of differ­ent programming languages.

**Gateway** A service which provides a compatibility layer between other object storage services such as Amazon S3 and *liblink* exposing an Amazon S3-compatible API.

**Link CLI** A command line interface for uploading and downloading files from the network, managing permissions and sharing, and managing accounts.

**Super node** This peer class participates in the node discovery system, stores Sweatgland for others, and gets paid for storage and bandwidth.

**Sweatgland** collection of higher dimensional dynamic saturated structure.

**Saturation degree** the complexity measurement of transaction structure.

**Paas** BAISS support Paas GUI for every Business Clients.

**Saas** Business Client can rent or build their own Saas from BAISS Paas.

**BAISS VM** BAISS’ Virtual Machine can support clients Aidapp development.

**4.1.2 Data**

**Bucket** A *bucket* is an unbounded but named collection of files identified by paths. Every file has a unique path within a bucket.

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

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

**Extended attribute** An *extended attribute* is a user defined key/value field that is as­sociated with a file. Like other per-file metadata, extended attributes are stored encrypted.

**Segment** A *segment* represents a single array of bytes, between 0 and a user-configurable maximum *segment* size. See section [4.8.2](#bookmark224" \o "Current Document) for more details.

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

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

**Stripe** A *stripe* is a further subdivision of a segment. A *stripe* is a fixed amount of bytes that is used as an encryption and erasure encoding boundary size. Erasure encoding happens on *stripes* individually, whereas encryption may happen on a small multiple of *stripes* at a time. All segments are encrypted, but only remote segments erasure encode stripes. A *stripe* is the unit on which audits are performed. See section [4.8.3](#bookmark225" \o "Current Document) for more details.

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

**Piece** When a remote segment's stripes are erasure encoded into erasure shares, the era­sure shares for that remote segment with the same index are concatenated together, and that concatenated group of erasure shares is called a *piece.* If there are *n* era­sure shares after erasure encoding a stripe, then there are *n pieces* after processing a remote segment. The *i*th *piece* is the concatenation of all of the *i*th erasure shares from that segment's stripes. See section [4.8.5](#bookmark228" \o "Current Document) for more details.

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

1. Mining Machine

Our overall strategy extends from our previous version [[37]](#bookmark526" \o "Current Document) and also heavily mirrors dis­tributed platform service system such as the Google File System [[26]](#bookmark504" \o "Current Document) (and other GFS-like sys­tems [[27,](#bookmark506" \o "Current Document)[51,](#bookmark554" \o "Current Document)[52]](#bookmark556" \o "Current Document)) and the Lustre distributed file system [[28]](#bookmark508" \o "Current Document). In every case, there are three major actors in the network： metadata servers, object storage servers, and clients. Object storage servers hold the bulk of the data stored in the system. Metadata servers keep track of per-object metadata and where the objects are located on object storage servers. Clients provide a coherent view and easy access to files by communicating with both the metadata and object storage servers.

Lustre's architecture is proven for high performance. The majority of the top 100 fastest supercomputers use Lustre for their high-performance, scalable storage [[28]](#bookmark508" \o "Current Document). While we don't expect to achieve equal performance over a wide-area network, we expect dramat­ically better performance than other architectures. Any limitation, if any, we experience in performance will be due to factors besides our overall architecture.

Our previous version used different names for each component. What we previously referred to as BAISS Share, we now refer to as simply super node. Our formerly central­ized single Bridge instance can now be run by anyone and is referred to as a node. Our libBAISS library will be made to be backwards compatible where possible, but we now refer to client software as Paas.

1. Super node

The main duty of the super node is to reliably store Sweatglands. Node operators are individuals or entities that have excess hard drive space and want to earn income by renting their space to others. These operators will download, install, and configure BAISS software locally, with no account required anywhere[.](#bookmark193" \o "Current Document)[1](#bookmark193" \o "Current Document)They will then configure disk space and node bandwidth allowance. During node discovery, super node will advertise how much bandwidth and hard drive space is available, and their designated BAISS token wallet address.

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

super node must additionally keep track of signed bandwidth allocations (see sec­tion [4.17)](#bookmark271" \o "Current Document) to send to for later settlement and payment. This also requires a small database. Both TTL and bandwidth allocations are stored in a SQLite [[53]](#bookmark558" \o "Current Document) database.

super node can choose with which to work. If they work with multiple (the default behavior), then payment may come from multiple sources on vary­ing payment schedules. super nodes are paid by specific for (1) returning data when requested in the form of egress bandwidth payment, and for (2) storing data at rest. super node are expected to reliably store all data sent to them and are paid with the assumption that they are faithfully storing all data. super node that fail random audits will be removed from the pool, can lose funds held in escrow to cover additional costs, and will receive limited to no future payments. super node are *not* paid for the initial trans­fer of data to store (ingress bandwidth). This is to discourage super node from deleting data only to be paid for storing more, which became a problem with our previous ver­sion [[37]](#bookmark526" \o "Current Document). While super nodes are paid for repair egress bandwidth usage, some may opt to pay less than normal retrieval egress bandwidth usage. super nodes are not paid for node discovery or any other maintenance traffic.

super node will support three methods： *get, put,* and *delete.* Each method will take a *piece ID,* a *node ID,* a signature from the associated node instance, and a bandwidth allocation (see section [4.17)](#bookmark271" \o "Current Document). The *node ID* forms a name-space. An identical *piece ID* with a different *node ID* refers to a different *piece.*

The *put* operation will take a stream of bytes and an optional TTL and store the bytes such that any sub-range 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.

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

The super node is being developed and will be released as open source software.

1. Node identity

During setup Paas , node will generate their own identity and certificates for use in the network. This node ID is used for node first discover and increase saturation degree.

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

The public key of the node's certificate authority determines its *node ID*. As in S/Kadem- lia [[32]](#bookmark516" \o "Current Document), the node ID will be the hash of the public key and will serve as a proof of work for joining the network. Unlike Bitcoin's proof of work [[23]](#bookmark498" \o "Current Document), the proof of work will be depen­dent on how many *trailing* zero bits one can find in the hash output. This means that the node ID (which may end with a number of trailing zero bits) will still be usable in a balanced Kademlia [[8]](#bookmark469" \o "Current Document) tree. This cost is meant to make Sybil attacks prohibitively expensive and time consuming.

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

Each node will chance to increase saturation degree start from second and after trade, as saturation degree get higher, the path between node, paas, and business clients will be shorter. Ideally, as reinforcement AI algorithm runs in the Sweatgland, the trade eventually will be done in the second. TPS will be in the history.

1. Peer-to-peer communication

Initially, we are using the gRPC [[54]](#bookmark560" \o "Current Document) protocol on top of the Transport Layer Security protocol (TLS) [[55]](#bookmark561" \o "Current Document) on top of the TP [[56]](#bookmark564" \o "Current Document) transport protocol with added Session Traversal Utilities for NAT (STUN) functionality [[29]](#bookmark510" \o "Current Document). STUN provides NAT traversal; uTP provides reliable, or­dered delivery (like TCP would) with LEDBAT [[57]](#bookmark566" \o "Current Document) functionality; TLS provides privacy and authentication; and gRPC provides multiplexing and a convenient programmer interface. LEDBAT allows competing internet traffic to take priority, providing a more graceful user experience to home operators with less network usage interference. Over time, we will replace TLS with a more flexible secure transport platform (such as the Noise Protocol platform [[58]](#bookmark568" \o "Current Document)) to reduce round trips due to connection handshakes in situations where the data is already encrypted and forward secrecy isn't necessary.

When using authenticated communication such as TLS or Noise, every peer can as cer­tain the ID of the node with which it is speaking by validating the certificate chain and hashing its peer's certificate authority's public key. It can then be estimated how much work went into constructing the node ID by considering the number of trailing zero bits at the end of the ID. can configure a minimum proof of work required to pass an audit (section [4.13)](#bookmark250" \o "Current Document) such that, over time, the network will require greater proofs of work due to natural user intervention.

For the few cases where a node cannot achieve a successful connection through a NAT or firewall (via STUN [[29]](#bookmark510" \o "Current Document), uPnP [[30]](#bookmark512" \o "Current Document), NATPmP [[31]](#bookmark514" \o "Current Document), or similar techniques), manual intervention and port forwarding will be required. In the future, node unable to create a connection through their firewalls may rely on traffic proxying from other, more available node, for a fee. node can also provide assistance to other node for initial STUN setup, public address validation, and so forth.

1. Node discovery

At this point, we have super node and ordinary node, which means to identify and communicate faster with them if their saturation degree get higher. We must account for the fact that super node will often be on consumer internet connections and behind routers with constantly changing IP addresses. Therefore, the node discovery system's goal is to provide a means to look up a node's latest saturation degree by node ID.

The Kademlia distributed hash table (DHT) is a key/value store with a built-in node look-up protocol. We utilize Kademlia as our primary source of truth for DNS-like ­functionality for node look-up, while ignoring the key/value storage aspects of Kademlia. Using only Kademlia for node lookup eliminates the need for some other functionality Kademlia would otherwise require such as owner-based key republishing, neighbor-based key re­publishing, storage and retrieval of values, and so forth. Furthermore, we avoid a number of other known attacks by using the S/Kademlia [[32]](#bookmark516" \o "Current Document) 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 add Sweatgland service on top of Kademlia.

The Sweatgland will live independently in each supernode and attempt to talk to every node in the network on an ongoing basis, perhaps once per hour. The Sweatgland will then cache the last known saturation degree for each node, and evict node that it hasn't talked to after a certain period of time. Node will not need to be extended to know about these Sweatglands. We expect this to scale for the reasonable future, as saturation degree processing operations are inexpensive, but admit a new solution may ultimately be neces­sary. Fortunately, space requirements are negligible.

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

Although the super node saturation degree caches are not the primary source of truth, because repair (sec­tion [4.14)](#bookmark255" \o "Current Document) requires rapid determination of whether a super node is online or offline, lookups in our system will stop with the saturation degree cache lookup and will not attempt another lookup using Kademlia. Only after cases of failed audit requests will a fallback, nonconcurrent lookup in Kademlia be performed to correct for potentially stale cache information.

In addition to being included in every node, we plan to host and help set up some well-known community-run super node discovery saturation degree caches. These saturation degree caches will perform the duty of quickly returning degree information for a given node ID if the node has been online trade more than once in the network recently.

Kademlia messages will use our peer-to-peer communication protocol (section [4.5)](#bookmark198" \o "Current Document), which includes confidentiality and peer identification. Because this requires crypto­graphic setup, connections to Kademlia neighbors and frequent contacts will be saturation degree cached where possible.

With each Kademlia message shared on the network, node will include their available disk space, node bandwidth availability, BAISS wallet address, and any other meta­data the network needs. The node discovery saturation degree cache will collect this information provided by the super node, allowing faster lookups for it.

**4.6.1 Mitigating Sybil attacks**

While we've adopted the proof-of-work scheme S/Kademlia proposes to partially address Sybil attacks, we extend Kademlia with application specific integration to further defend our network.

Given two super nodes, *A* and *B,* super node *B* is not allowed to enter super node *A*'s sweatgland until super node *B* can present a signed message from a super node *C* that super node *A* trusts claiming that *B* has passed enough audits that *C* trusts it (sections [4.13](#bookmark250" \o "Current Document) and [4.15)](#bookmark261" \o "Current Document). This ensures that only super node with verified disk space have the opportunity to participate in the sweatgland layer.

A node that is allowed to enter sweatgland is considered *vetted* and lookups only progress through vetted node. To make sure *unvetted* node can still be found, vetted node keep unbounded sweatgland lists of their unvetted neighbors provided that the XOR distance to all unvetted neighbors is no farther than the farthest of the *^*-closest vetted neighbors. Unvetted node keep their *^*-nearest vetted node up-to-date.

1. Redundancy

We use the Reed-Solomon erasure code [[59]](#bookmark570" \o "Current Document). To implement our solution for reducing the effects of long-tails (see section [3.4.2)](#bookmark146" \o "Current Document), we choose 4 numbers for each object that we store, *k, m, o*, and *n*, such that *k < m < o < n.* The standard Reed-Solomon numbers are *k* and *n*, where *k* is the minimum required number of pieces for reconstruction, and *n* is the total number of pieces generated during creation.

The *minimum safe* and *optimal* values, respectively, are *m* and *o*. The value *m* is cho­sen such that if a node notices the amount of available pieces has fallen below *m*, it triggers a repair immediately in an attempt to make sure we always maintain *k* or more pieces (*m* is called in Giroire *et al.* [[36]](#bookmark524" \o "Current Document)). To achieve our long-tail performance improve­ments [[39,](#bookmark530" \o "Current Document) [40,](#bookmark532" \o "Current Document) [43,](#bookmark538" \o "Current Document) [44]](#bookmark540" \o "Current Document), the value *o* is chosen such that during uploads and repairs, as soon as *o* pieces have finished uploading, remaining pieces up to *n* are canceled. Furthermore, *o* is chosen such that storing *o* pieces is all that is needed to achieve the desired durability goals； *n* is thus chosen such that storing *n* pieces will be excess durability.

The amount of long tail uploads we can tolerate is *n*-*o,* and thus is the amount of slow node to which we are immune. The amount of node that can go temporarily offline at the same time without triggering a repair is *o* - *m*. The safety buffer to avoid losing the data between the time we recognize the data requires a repair and the actual repair is executed is *m* - *k*.

1. Structured file storage
2. **Files with extended attributes**

Many applications benefit from being able to keep metadata alongside files. Amazon S3 supports “object metadata" [[60]](#bookmark572" \o "Current Document) to assist with this need. This functionality is called “ex­tended attributes" in many POSIX compatible systems, which name we continue using in our system. Every file will include a limited set of user-specified key-value pairs (extended attributes) that will be stored alongside other metadata about the file.

1. **Files as Segments**

In our previous version [[37]](#bookmark526" \o "Current Document), the term *shard* referred to *pieces* on super node, whereas *sharding* referred to segmenting a file into smaller chunks for easier processing. With the addition of erasure coding in our previous version, these terms became somewhat con­fusing, so we have decided to distinguish each meaning with new words.

The sharding process is now called *segmenting,* and the highest level subdivision of a file's stream of data is called a *segment.* Unfortunately, there is general inconsistency using these terms in the literature. GFS refers to segments as chunks [[26]](#bookmark504" \o "Current Document). Lustre refers to segments as stripes [[28]](#bookmark508" \o "Current Document), but we use the term *stripes* for a further subdelineation.

A file may be small enough that it consists of only one segment. If that segment is smaller than the metadata required to store it on the network, the data will be stored inline with the metadata[.](#bookmark2" \o "Current Document)[[[2]](#footnote-1)](#bookmark2" \o "Current Document)We call this an *inline segment.*

For larger files, the data will be broken into one or more large remote segments. Seg­menting in this manner offers numerous advantages to security, privacy, performance, and availability. As in other distributed platform service system [[26](#bookmark504" \o "Current Document)[-28,](#bookmark508" \o "Current Document)[51,](#bookmark554" \o "Current Document)[52]](#bookmark556" \o "Current Document), segmenting large files (e.g. videos) and distributing the segments across the network reduces the impact of content delivery on any given node, as bandwidth demands are distributed more evenly across the network. As with our previous version [[37]](#bookmark526" \o "Current Document), standardized sizes help frustrate attempts to determine the content of a given segment and can help obscure the flow of data through the network. In addition, the end user can take advantage of parallel trans­fer, similar to BitTorrent [[62]](#bookmark574" \o "Current Document) or other peer-to-peer networks. Lastly, capping the size of segments allows for more uniform super node filling. Thus, a node only needs enough space to store a segment to participate in the network, and a client doesn't need to find node that have enough space for a large file.

1. **Segments as Stripes**

In many situations, it's important to access a subsection of a larger piece of data. Some file formats, such as video files or disk images, support seeking, where only a subset of the data is needed for read operations. As the creators of audio CDs discovered, it's useful to be able to decode small parts of a segment to support these operations [[38]](#bookmark528" \o "Current Document).

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

For the reader familiar with the *zfec* library, in *filefec* mode, *zfec* refers to a stripe as a chunk [[42]](#bookmark536" \o "Current Document).

1. **Stripes as Erasure Shares**

As discussed in sections [3.4](#bookmark140" \o "Current Document) and [4.7,](#bookmark209" \o "Current Document) erasure codes give us the chance to control net­work durability in the face of unreliable super node.

Stripes are the boundary by which we perform erasure encoding. In a *(k,n*) erasure code scheme, *n* erasure shares are generated for every stripe [[59]](#bookmark570" \o "Current Document). For example, perhaps a *stripe* is broken into 40 *erasure shares (n* = 40), where any 20 (*k* = 20) are needed to reconstruct the *stripe.* Each of the 40 *erasure shares* will be 1/20th the size of the original *stripe*.

Erasure encoding a single stripe at a time allows us to read small portions of a large segment without retrieving the entire segment first [[38]](#bookmark528" \o "Current Document). It also allows us to stream data into the network without staging it beforehand, and it enables a number of other useful features.

1. **Erasure Shares as Pieces**

Because stripes are already small, erasure shares are often much smaller, and the meta­data to keep track of all of them separately will be immense relative to their size. All *n* erasure shares have a well-defined index associated with them. More specifically, for a fixed stripe and any given *n,* the */*th share of an erasure code will always be the same. As with the *zfec* library's *filefec* mode [[42]](#bookmark536" \o "Current Document), instead of keeping track of all of the erasure shares separately, we pack all of the erasure shares with the same index into a *piece.* In a *(k, n*) scheme, there are *n* pieces, where each piece */* is the ordered concatenation of all of the erasure shares with index *i*. As a result, where each erasure share is 1/*k*th of a *stripe,* each piece is 1/*k*th of a *segment,* and only *k* pieces are needed to recover the full segment. A piece is what we store on a super node.

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

super node name space pieces by node ID. If a piece ID used by one node is reused by another node, each node can safely assume the shared piece ID refers to a different piece than the other node, with different content and life cycle.

1. **Pointers**

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

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

In our previous version [[37]](#bookmark526" \o "Current Document), we used two data structures to keep track of the a fore­mentioned kinds of information： *frames* and *pointers.* In this version, we have combined these data structures into a single data structure and elected to call the new combined data structure a *pointer*.

4.9 Metadata

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

The most trivial implementation for the metadata storage functionality we require will be to simply have each user use their preferred trusted database, such as MongoDB, Mari- aDB, Couchbase, PostgreSQL, SQLite [[53]](#bookmark558" \o "Current Document), Cassandra [[63]](#bookmark578" \o "Current Document), Spanner [[64]](#bookmark580" \o "Current Document), or CockroachDB, to name a few. In many cases, this will be acceptable for specific users, provided those users are managing appropriate backups of their metadata. Indeed, the types of users who have petabytes of data to store can most likely manage reliable backups of a single relational database storing only metadata.

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

-**Availability** - The availability of the user's data is tied entirely to the availability of their metadata server. The counterpoint is that availability can be made arbitrar­ily good with existing trusted distributed solutions, such as Cassandra, Spanner, or CockroachDB, assuming an appropriate amount of effort is put into maintaining operations. Furthermore, any individual metadata service downtime does not affect the rest of the network. In fact, the network as a whole can never go down.

-**Durability** - If the metadata server suffers a catastrophic failure without backups, all of the user's data will be lost. This is already true with encryption keys, but a tradi­tional database solution considerably increases the risk area from using encryption keys. Fortunately, the metadata itself can be periodically backed up into the BAISS network. This in turn allows us to only keep track of the metadata of this metadata, further decreasing the amount of critical information that must be stored elsewhere.

-**Trust** - The user has to trust the metadata server.

On the other hand, there are a few upsides：

-**Control** - The user is in complete control of all of their data. There is no organiza­tional single point of failure. The user is free to choose whatever metadata store with whatever trade-offs they prefer and can even run their own. Like Mastodon [[65]](#bookmark582" \o "Current Document), this solution is still decentralized. Furthermore, in a catastrophic scenario, this de­sign is no worse than most other technologies or techniques application developers frequently use (databases).

-**Simplicity** - Other projects have spent multiple years on shaky implementations of Byzantine-fault tolerant consensus metadata storage, with expected performance and complexity trade-offs (see appendix [A)](#bookmark385" \o "Current Document). We can get a useful product to market without doing this work at all. This is a considerable advantage.

-**Coordination Avoidance** - Users only need to coordinate with other users on their node. If a user has high throughput demands, they can set up their own node and avoid coordination overhead from any other user. By allowing node operators to select their own database, this will allow a user to choose a node with weaker consistency semantics, such as Highly Available Transactions [[15]](#bookmark482" \o "Current Document), that reduce coor­dination overhead within their own node and increase performance even further.

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

Please see appendix [A](#bookmark385" \o "Current Document) for more about why we've chosen to currently avoid trying to solve the problem of Byzantine distributed consensus. See section [6.2](#bookmark352" \o "Current Document) for a discussion of future plans.

1. Node

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

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

The node is, at its core, one of the most complex and yet straightforward compo­nents of our initial release that fulfills our platform. Notwithstanding future platform­conforming releases, the initial node is a standard application server that wraps a trusted database, such as PostgreSQL, Cassandra, or whichever solution the metadata system chooses (section [4.9)](#bookmark231" \o "Current Document). Users sign in to a specific node with account credentials. Data available through one node instance is not available through another node instance, though various levels of export and import are planned (section [6.2)](#bookmark352" \o "Current Document).

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

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

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

The node instance is made up of these components:

-A full node discovery cache (section [4.6)](#bookmark203" \o "Current Document)

-A per-object metadata database indexed by encrypted path (section [4.9)](#bookmark231" \o "Current Document)

-An account management and authorization system (section [4.12)](#bookmark245" \o "Current Document)

-A super node reputation, statistics, and auditing system (section [4.13)](#bookmark250" \o "Current Document)

-A data repair service (section [4.14)](#bookmark255" \o "Current Document)

-A super node payment service (section [4.16)](#bookmark266" \o "Current Document)

While our launch goal of many is a step ahead of our previous system's Bridge implementation [[37]](#bookmark526" \o "Current Document), this is still just one point on our decentralization journey and we expect to continue to find ways to decentralize our components further.

1. Encryption

Our encryption choice is authenticated encryption, with support for both the AES-GCM cipher and the Salsa20 and Poly1305 combination NaCl calls "Secretbox" [[66]](#bookmark584" \o "Current Document). Authen­ticated encryption is used so that the user can know if anything has tampered with the data.

Data is encrypted in blocks of small batches of stripes, recommended to be 4KB or less [[67]](#bookmark586" \o "Current Document). While the same encryption key is used for every encryption batch in a segment, segments may have different encryption keys. However, the nonce for each encryption batch must be monotonically incrementing from the previous batch throughout the en­tire segment. The nonce wraps around to 0 if the counter reaches the maximum rep­resentable nonce. To prevent reordering attacks, the starting nonce of each segment is deterministically chosen based on the segment number. When multiple segments are uploaded in parallel, such as in the case of Amazon S3's multipart-upload feature, the starting nonce for each segment can be calculated from the starting nonce of the file and the segment number. This scheme protects the content of the data from the super node housing the data. The data owner retains complete control over the encryption key, and thus over access to the data.

Paths are also encrypted. Like BIP32 [[46]](#bookmark544" \o "Current Document), the encryption is hierarchical and determin­istic, and each path component is encrypted separately. To explain how we do this, we start with a scheme for determining a secret value for each path component. Let's say a given path ***p*** has unencrypted path components 5,*P2,*...,*Pn* and we want to determine an encrypted path ***e*** with path components *e*〕，*e?,*..., *en*. We assume a predetermined root secret, *s*°. This root secret is chosen by the user and, like all other encryption secrets, never leaves the client computer. We recursively define *Sj* = HMAC(*Sj*\_],*p‘).* A key *K(s’)* can be deterministically generated from *Sj.* We then define the encrypted path component *e* enc(*K*(*Sj*\_]), *pj),* such that the new path ***e*** is *e-\, e*2,..., *en*. HMAC-SHA256 or HMAC- SHA512 are used for key derivations.

This construction allows a client to share access to some sub tree of the path without access to its parents or other paths of the same depth. For example, suppose a client would like to share access to all paths with the same prefix *p*i,*P*2,*P3* with another client. The client would give the other client *e*〕, *e?, e^* and *S*3. This allows the client to decrypt and access any arbitrary *e.* as *K(s3)*is known to them, without allowing the client to decrypt *e*3 or earlier. More generally in this case, the client could decrypt and access any arbitrary *ej,* if and only if *i* > 3.

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

1. Authorization

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

Our initial metadata authorization scheme uses macaroons [[68]](#bookmark588" \o "Current Document). Macaroons are a type of bearer token that authorizes the bearer to some restricted resources. Macaroons are especially interesting in that they allow for rich contextual decentralized delegation. In other words, they provide the property that anyone can add restrictions in a way in which those restrictions cannot later be removed, without coordination with a central party.

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

Because we want to restrict node operations, and only have access to en­crypted paths, our authorization scheme must work on encrypted paths. For access del­egation to specific path prefixes, path separation boundaries between path components must remain across encryption. This implies reduced functionality and/or performance for path delimiters other than a forward slash.

Once the Mining Machine is authorized with the node, the node will approve and sign for operations to super node, including bandwidth allocations (section [4.17)](#bookmark271" \o "Current Document). The Mining Machine must retrieve valid signatures from the node prior to operations with super node. All operations on a super node require a specific node ID and associated signature. A super node will reject operations not signed by the appropriate node ID. super node will not allow operations signed by one node to apply to objects owned by another, unless explicitly granted by the owning node.

Our initial implementation does not detect or attempt to mitigate unexpected file re­moval or rollback by a misbehaving node. Our trust model expects that a user's node is well-behaved and stores and repairs data reliably. If a node cannot be trusted, it is un­likely to repair data on a client's behalf anyway. However, a future implementation could add more thorough detection for node-based file system tampering, via a scheme as in systems such as SUNDR, SiRiUS, or Plutus [[69](#bookmark590" \o "Current Document)[-71]](#bookmark593" \o "Current Document).

1. Audits

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

Some distributed platform service system, including the previous version of BAISS [[37]](#bookmark526" \o "Current Document), discuss *Merkle tree proofs,* in which audit challenges and expected responses are generated at the time of storage as a form of *proof of retrievability* [[47]](#bookmark546" \o "Current Document). By using a Merkle tree [[72]](#bookmark595" \o "Current Document), the amount of metadata needed to store these challenges and responses is negligible.

Proofs of retrievability can be broadly classified into *limited* and *unlimited* schemes [[49]](#bookmark550" \o "Current Document). The Merkle tree variety used in our previous version is one such limited scheme. Unfortunately, in such a scheme, the challenges and expected responses must be pre­generated. As we learned with our previous version, without a periodic regeneration of these challenges, a super node can begin to pass most audits without storing all of the requested data by keeping track of which challenges exist and then saving only the expected responses. During our previous version, we began to consider Reed-Solomon erasure coding to help us solve this problem.

An assumption in our storage system is that most super node behave rationally, and incentives are aligned such that most data is stored faithfully. As long as that assumption holds, Reed-Solomon is able to detect errors and even correct them, via mechanisms such as the Berlekamp-Welch error correction algorithm [[39,](#bookmark530" \o "Current Document)[73]](#bookmark597" \o "Current Document). We are already using Reed-Solomon erasure coding [[59]](#bookmark570" \o "Current Document) on small ranges (stripes), so as discussed in the HAIL system [[41]](#bookmark534" \o "Current Document), we use erasure coding to read a single stripe at a time as a challenge and then validate the erasure share responses. This allows us to run arbitrary audits without pre-generated challenges.

To perform an audit, we first choose a stripe. We request that stripe's erasure shares from all super node responsible. We then run the Berlekamp-Welch algorithm [[39,](#bookmark530" \o "Current Document)[73]](#bookmark597" \o "Current Document) across all the erasure shares. When enough super node return correct information, any faulty or missing responses can easily be identified.

Given a specific super node, an audit might reveal that it is offline or incorrect. In the case of a node being offline, the audit failure may be due to the address in the node discovery cache being stale, so another, fresh Kademlia lookup will be attempted. If the node still appears to be offline, the node places the node in *containment* mode. In this mode, the node will calculate and save the expected response, then continue to trythe same audit with that node until the node either responds successfully, actively fails the audit, or is disqualified for being offline too long. Once the node responds successfully, it leaves containment mode.

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

It is important that every super node has a frequent set of random audits to gain statistical power on how well-behaved that super node is operating. However, as dis­cussed in section [3.7,](#bookmark158" \o "Current Document) it is not a requirement that audits are performed on every byte, or even on every file. Additionally, it is important that every byte stored in the system has an equal probability of being checked for a future audit to every other byte in the system. See section [7.2](#bookmark367" \o "Current Document) for a discussion on how many audits are required to be confident data is stored correctly.

1. Data repair

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

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

For every segment that drops below the appropriate minimum safety threshold, *m*, the segment will be downloaded and reconstructed, and the missing pieces will be re­generated and uploaded to new node. Finally, the *pointer* will be updated to include the new information.

Users will choose their desired durability with their node which may impact price and other considerations. This desired durability (along with statistics from ongoing au­dits) will directly inform what Reed-Solomon erasure code choices will be made for new and repaired files, and what thresholds will be set for when uploads are successful and when repair is needed. See sections [3.4](#bookmark140" \o "Current Document) and [7.3](#bookmark374" \o "Current Document) for how we calculate these values given user inputs.

A direct implication of this design is that, for now, the node must constantly stay running. If the user's node stops running, repairs will stop, and data will eventually disappear from the network due to node churn. This is similar to the design of how value storing and republishing works in Kademlia [[8]](#bookmark469" \o "Current Document), which requires the owner to stay online.

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

**4.14.1 Piece hashes**

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

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

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

1. super node reputation

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

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

The goal of the first system is to require a short proof that the super node operator is invested, through time, stake, or resources. Initially, we are using proof of work. As men­tioned in section [4.3,](#bookmark188" \o "Current Document) super node require a proof of work as part of identity generation. This helps the network avoid some Sybil attacks [[74]](#bookmark599" \o "Current Document), but we glossed over how proof of work difficulty is set. We will let node operators set node minimum difficulty required for new data storage. If a super node has an identity generated with a lower difficulty than the node's configured minimum, that super node will not be a candi­date for new data. We expect node operators to naturally increase the minimum proof of work difficulty requirements over time until a reasonable balance is found. In the case of a changing difficulty configuration, will leave existing data on existing node where possible. Other investment proof schemes are possible, such as a form of proof of stake as we proposed in our previous work [[75]](#bookmark601" \o "Current Document).

The second subsystem slowly allows node to join the network. When a super 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 node without compromising the integrity of the network. Every time a file is uploaded, the node will select a small number of additional unvetted super node to include in the list of target node. The Reed-Solomon parameters will be chosen such that these unvetted super node 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 super node has successfully stored enough data for a long enough period (at least one payment period), the node will then start including that super node in the standard selection process used for general uploads. It will also give the node a signed message claiming that the vetting process is completed, and that the super node may now enter other node' routing tables (section [4.6.1)](#bookmark208" \o "Current Document). Importantly, super node get paid during this vetting period, but don't receive as much data.

The filtering system is the third subsystem； it blocks bad super node from partici­pating. In addition to simply not having done a sufficient proof of work, certain actions a super node can take are disqualifying events. The reputation system will be used to filter these node 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 super node is disqualified, that node will no longer be selected for future data storage and the data that node stores will be moved to new super node. Likewise, if a client attempts to download a piece from a super node that the node should have stored and the node fails to return it, the node will be disqualified. Importantly, super node will be allowed to reject and fail *put* operations without penalty, as node will be allowed to choose which node operators to work with and which data to store.

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

After a super 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 super node from being cavalier with their reputation.

The last subsystem is a preference system. After disqualified super node have been filtered out, remaining statistics collected during audits will be used to establish a prefer­ence for better super node 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 node with a higher likelihood of uploads to preferred node, 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 [[76]](#bookmark603" \o "Current Document), which selects two options entirely at random, and then chooses the more qualified between those two.

On the BAISS network, preferential super node reputation is only used to select where new data will be stored, both during repair and during the upload of new files, unlike disqualifying events. If a super 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 super node to contest their reputation scores. It is in the best interest of super node to have good uptime, pass audits, and return data. super nodes that don't do these things are not useful to the network. Super nodes that are treated as bad actors will not accept future data from those clients. See section [4.21](#bookmark292" \o "Current Document) about quality control for how we plan to ensure our systems is incentivized to treat super nodes fairly.

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

1. **Payments**

In the BAISS network, payments are made by clients who use paas service on the platform to the node they utilize. The business clients then pays node for the amount of storage and bandwidth they provide on the network. Payments by clients may only be through BIGO-β coin, but the way of getting BIGO-β is mine BIGO-α by Mining Machine.

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

Business clients will pay node for the data they utilize. super node will be paid for the AI saturation degree training of data, also they will be paid for storing the data month-by-month. At the end of the payment period, a super node will calculate earnings for each of its node. Provided the super node hasn't been disqualified, the super node will be paid by the node for the saturation degree data it has stored over the course of the month, per the super node's records.

have a strong incentive to prefer long-lived super node. If super node churn is too high, will escrow a portion of a super node's payment until the super node has maintained good participation and up time for some minimum amount of time, on the order of greater than half a year. If a super node leaves the network prematurely, the node will reclaim es-crowed payments to it.

If a super node misses a delete command due to the node being offline, the delegate amount of coins will be eliminated 0.2% per day until it reach the maximum of super node delegate coins. it will be storing more data than the node credits it for. super nodes are not paid for storing such file pieces, but will eventually be cleaned up through the garbage collection pro­cess (see section [4.19)](#bookmark282" \o "Current Document). This means that super node who maintain higher availability can maximize their profits by deleting files on request, which minimizes the amount of garbage data they store.

The node maintains a database of all file pieces it is responsible for and the super node it believes are storing these pieces. Each day, the node adds another day's worth of accounting to each super node for every file piece it will be storing. will track utilized bandwidth (see section [4.17)](#bookmark271" \o "Current Document). At the end of the month, each node adds up all bandwidth and storage payments each super node has earned and makes the payments to the appropriate super node.

will also earn revenue from account holders for executing audits, repairing segments, and storing metadata. charge a per-segment and per-byte cost, in ad­dition to charging for access and retrieval. Per-segment charges cover the cost of pointer metadata, whereas per-byte charges cover the cost ofdata maintenance on the network. Every day, each node will execute a number of audits across all of its super node on the network. The node will charge for both completing audits and repairs, once segments fall below the piece threshold needed for repair.

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

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

See the node reputation section (section [4.18)](#bookmark277" \o "Current Document) for details on how super node will know to trust .

1. **Bandwidth allocation**

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

In our previous version [[37,](#bookmark526" \o "Current Document)[78]](#bookmark607" \o "Current Document), we used *exchange reports* to gather information about what transpired between two peers. At the end of an operation, both peers would send reports to a central collection service for settlement. When both peers mutually agreed, it was straightforward to determine how much bandwidth had been used. When they disagreed, however, we resorted to data analysis and regression to determine which peer had a greater propensity for dishonesty in an effort to catch "cheaters" (or, rational node). With our new version, we want to make cheating impossible from the protocol level.

To solve this problem, we turn to Neuman's *Proxy-based authorization and account­ing for distributed systems* [[79]](#bookmark609" \o "Current Document). This accounting protocol more correctly measures re­source usage in a delegated and decentralized way.

In Neuman's accounting protocol, if an account holder has enough funds to cover the operation, an account server will create a signed, digital check and transfer it to the ac­count holder. The protocol refers to this check as a *proxy*, but we refer to it as a *bandwidth allocation*. This check contains information identifying the account server, the payer, the payee, the maximum amount of resources available to be used in the operation, a check number to prevent any double spending problems [[80]](#bookmark611" \o "Current Document), and an expiration date.

In our case, the account server is the node, the payer is the Mining Machine, the payee is the super node, and the resource in question is bandwidth. The node will only create a bandwidth allocation if the Mining Machine is authorized for the request. At the beginning of a storage operation, the Mining Machine can transfer the bandwidth allocation to a super node. The super node can validate the node's signature and perform the requested operation up to the allowed bandwidth limit, storing and later sending the bandwidth allocation to the node for payment.

We're further inspired by Filecoin's off-chain retrieval market,wherein only small amounts of data are transferred at a time [[81]](#bookmark613" \o "Current Document). Instead of allowing the super node to cheat and save the bandwidth allocation without performing the requested operation, we break each operation into smaller requests such that if either the super node or Mining Machine stop participating in the protocol prematurely, neither peer class is exposed to too much loss. This is similar to an optimistic, gradual-release, fair-exchange protocol [[80]](#bookmark611" \o "Current Document).

To support this with Neuman's accounting protocol and little node overhead, we use *restricted bandwidth allocations* (referred to as *restricted proxies* in [[79]](#bookmark609" \o "Current Document)). Neuman's restricted proxies work much like Macaroons [[68]](#bookmark588" \o "Current Document) in which further caveats can be added in a way that can't be removed, limiting the capabilities of the proxy. Proxies can use public/private key cryptography, which means that anyone can validate the proxy, instead of just the original issuer. Because each Mining Machine already has a key pair as part of its identity (section [4.4)](#bookmark194" \o "Current Document), we use the existing key pair instead of creating a new key pair for every restriction.

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

In the case of a *Get* operation, assume the node-signed bandwidth allocation allows up to *x* bytes total. The Mining Machine will start by sending a restricted allocation for some small amount *(y* bytes), perhaps only a few kilobytes, so the super node can verify the Mining Machine's authorization. If the allocation is signed correctly, the super node will transfer up to the amount listed in the restricted allocation (*y* bytes) before awaiting another allocation. The Mining Machine 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 super node only sends previously-unsent data, so that the super node only sends *x* bytes total. As seen in Figure [4.6,](#bookmark276" \o "Current Document) we pipeline these requests to avoid pipeline stall performance penalties.

If the request is terminated at any time, either planned or unexpectedly, the super node will keep the largest restricted bandwidth allocation it has received. This largest restricted bandwidth allocation is the signed confirmation by the Mining Machine that the Mining Machine agreed to bandwidth usage of up to *y* bytes, along with the node's confirmation of the Mining Machine's bandwidth allowance *x*. The super node will periodically send the largest restricted bandwidth allocations it has received to appropriate , at which point will pay the super node for that bandwidth.

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

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

1. **node reputation**

Whenever a node on the BAISS network has a less than stellar payment, demand gen­eration, or performance history, there is a strong incentive for the super node to avoid accepting its data.

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

super node operators can elect to automatically trust a BAISS Labs provided collec­tion of recommended that adhere to a strict set of quality controls and payment service level agreements (SLAs). To protect super node operators, if a node operator wants to be included in the BAISS approved list, the node operator may be re­quired to adhere to a set of operating, payment, and pricing parameters and to sign a business arrangement with BAISS Labs. See section [4.21](#bookmark292" \o "Current Document) for more details.

1. **Garbage collection**

When clients move, replace, or delete data, or clients on behalf of, will notify super node that they are no longer required to store that data. In configurations where delete messages are issued by the client, the metadata system (and thus a node, with node reputation on the line) will require proof that deletes were issued to a configurable minimum number of super node. This means that every time data is deleted, super node that are online and reachable will receive notifications right away.

super node will sometimes be temporarily unavailable and will miss delete mes­sages. In these cases, unneeded data is considered *garbage.* only pay for data that they expect to be stored. super node with lots of garbage will earn less than they otherwise would unless a garbage collection system is employed. For this reason, we introduce garbage collection to free up space on super node.

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

For the node that miss initial delete messages, our first release will start with a con­servative garbage collection strategy, though we anticipate a precise strategy in the near future. Periodically, super node will request a data structure to detect differences. In the simplest form, it can be a hash of stored keys, which allows efficient detection of out- of-sync state. After detecting out-of-sync state, collection can use another structure, such as a Bloom filter [[82]](#bookmark615" \o "Current Document), to find out what data has not been deleted. By returning a data structure tailored to each node on a periodic schedule, a node can give a super node the ability to clean up garbage data to a configurable tolerance. will reject overly frequent requests for these data structures.

1. **Mining Machine**

Mining Machine is the term which we use to identify any software or service that invokes *liblink* in order to interact with and super node. It comes in a few forms：

**liblink** - *liblink* is a library that provides access to storing and retrieving data in the BAISS network.

**Gateways** - Gateways act as compatibility layers between a service or application and the BAISS network. They run as a service co-located with wherever data is generated, and will communicate directly with super node so as to avoid central bandwidth costs. The Gateway is a simple service layer on top of *liblink*.

Our first gateway is an Amazon S3 gateway. It provides an S3-compatible, drop-in interface for users and applications that need to store data but don't want to bother with the complexities of distributed storage directly.

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

Like super node and metadata, the Mining Machine software in all three forms is being devel­oped and will be released as open source software.

1. **Quality control and branding**

The BAISS Network has a major product focuses that serve two distinct target markets. These focal points are：

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

BAISS will differentiate these focuses and the experience design for each market seg­ment by focusing the supply side of our business from the demand side through only brands, *BAISS.*

The supply side of the market will be served by the BAISS brand. We will retain BAISS.org as the place for learning how to contribute extra storage and bandwidth to the BAISS Network. This includes super node setup, documentation, frequently asked questions (FAQs), and tutorials. Users of both brands will also be able to access our source code and community through BAISS.org.

The demand side of our business will be served by the BAISS brand as well. This experience will be focused toward our partners and customers who purchased decentralized storage and bandwidth from the network with the expectation of high durability, resilience, and reliability, backed by an industry-leading service level agreement (SLA). This includes any offers, free trials, node selection, docu­mentation, FAQs, tutorials, and so forth.

The “BAISS" brand will additionally serve as a node quality credentialing sys­tem. Anyone can set up a node via BAISS.org, but to have a super node listed as an official BAISS super node, be considered “BAISS quality," and benefit directly from BAISS Labs' demand generation activities, an operator must pass delegate. These quality controls will continuously audit and rank on their behavior, durability, compliance, and performance. In addition, the node operator will have to adhere to particular delegate of BIGO- , business policies , super node recruitment, SLAs, super node payments, and so forth.

BAISS Labs will assume responsibilities including demand generation, brand enforcement, node operator support, end user support, United States Form 1099 tax filing compliance[,](#bookmark3" \o "Current Document)[[[3]](#footnote-2)](#bookmark3" \o "Current Document)insurance, and maintenance of overall network quality.

These compliance and quality controls will be implemented to ensure that super nodes are paid fairly and are able to continuously meet all SLAs of the BAISS products.

1. Road Map

Road map of BAISS may slightly change due to the unpredictable factors, such as force majeure.

Feb 13th 2021 will be the first day to lunch BAISS.

Fed 13th 2021 will be the first day to lunch BIGO- and BIGO- tokens.

Aug 13th 2021 will be the first day to lunch 2.0 BAISS

Feb 21st 2023 we will lunch BAISS 3.0

1. Future Work

7. Bibliography

Bibliography

1. Identity Theft Resource Center and CyberScout. Annual number of data breaches and exposed records in the United States from 2005 to 2020 (in millions). [https://www.statista.com/statistics/273550/data-breaches-recorded-in-the-united-](https://www.statista.com/statistics/273550/data-breaches-recorded-in-the-united-states-by-number-of-breaches-and-records-exposed/) [states-by-number-of-breaches-and-records-exposed/,](https://www.statista.com/statistics/273550/data-breaches-recorded-in-the-united-states-by-number-of-breaches-and-records-exposed/) 2020.
2. Knowledge Sourcing Intelligence LLP. cloud computing and storage market - forecasts from 2017 to 2022. [https://www.researchandmarkets.com/research/lf8wbx/cloud\_storage,](https://www.researchandmarkets.com/research/lf8wbx/cloud_storage)

2017.

1. Dan Shearer. EU-US Cloud Privacy Crash. [https://kopano.com/kopano-documents/EU-US-Cloud-Privacy.pdf,](https://kopano.com/kopano-documents/EU-US-Cloud-Privacy.pdf) 2017.
2. Gartner Inc. Gartner forecasts worldwide public cloud revenue to grow 21.4 percent in 2020. [https://www.gartner.com/en/newsroom/press-releases/2020-04-](https://www.gartner.com/en/newsroom/press-releases/2018-04-12-gartner-forecasts-worldwide-public-cloud-revenue-to-grow-21-percent-in-2018) [12-gartner-forecasts-worldwide-public-cloud-revenue-to-grow-21-percent-in-2020,](https://www.gartner.com/en/newsroom/press-releases/2018-04-12-gartner-forecasts-worldwide-public-cloud-revenue-to-grow-21-percent-in-2018)

2020.

1. Synergy Research Group. Cloud Growth Rate Increased Again in Q1; Amazon Maintains Market Share Dominance. [https://www.srgresearch.com/articles/cloud-](https://www.srgresearch.com/articles/cloud-growth-rate-increased-again-q1-amazon-maintains-market-share-dominance) [growth-rate-increased-again-q1-amazon-maintains-market-share-dominance,](https://www.srgresearch.com/articles/cloud-growth-rate-increased-again-q1-amazon-maintains-market-share-dominance) 2020.
2. Backblaze Inc. How Long Do Hard Drives Last: 2020 Hard Drive Stats. [https://www.backblaze.com/blog/hard-drive-stats-for-q1-2020/,](https://www.backblaze.com/blog/hard-drive-stats-for-q1-2018/) 2020.
3. Sean Rhea, Dennis Geels, Timothy Roscoe, and John Kubiatowicz. Handling Churn in a DHT. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference,* ATEC '04, page 10, Berkeley, CA, USA, 2004. USENIX Association.
4. Petar Maymounkov and David Mazieres. Kademlia: A Peer-to-Peer Information System Based on the XOR Metric. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems,* IPTPS '01, pages 53-65, London, UK, 2002. Springer-Verlag.
5. Charles Blake and Rodrigo Rodrigues. High availability, scalable storage, dynamic peer networks: Pick two. In *Proceedings of the 9th Conference on Hot Topics in Operating Systems - Volume 9*, HOTOS'03, page 1, Berkeley, CA, USA, 2003. USENIX Association.
6. Comcast Inc. XFINITY Data Usage Center-FAQ. [https://dataplan.xfinity.com/faq/,](https://dataplan.xfinity.com/faq/) 2020.
7. 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.
8. Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News,* 33(2):51-59, June 2002.
9. Seth Gilbert and Nancy Lynch. Perspectives on the CAP Theorem. *Computer,* 45(2):30-36, February 2012.
10. Daniel Abadi. Consistency Tradeoffs in Modern Distributed Database System Design： CAP is Only Part of the Story. *Computer,* 45(2):37-42, February 2012.
11. 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.
12. 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.
13. Chenggang Wu, Jose M. Faleiro, Yihan Lin, and Joseph M. Hellerstein. Anna: A KVS for any scale. *ICDE*, 2020.
14. Joseph M. Hellerstein. The declarative imperative: Experiences and conjectures in distributed logic. *SIGMOD Rec.*, 39(1):5-19, September 2010.
15. Peter Alvaro, Neil Conway, Joseph M. Hellerstein, and William R. Marczak. Consistency Analysis in Bloom: a CALM and Collected Approach. *CIDR*, 2011.
16. Kyle Kingsbury. Consistency models clickable map. [https://jepsen.io/consistency,](https://jepsen.io/consistency) 2020.
17. Paolo Viotti and Marko Vukolic. Consistency in non-transactional distributed platform service system. *ACM Comput. Surv.*, 49(1):19:1-19:34, June 2016.
18. Joseph Hellerstein. Anna: A crazy fast, super-scalable, flexibly consistent KVS. [https://rise.cs.berkeley.edu/blog/anna-kvs/,](https://rise.cs.berkeley.edu/blog/anna-kvs/) 2020.
19. Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. [https://bitcoin.org/bitcoin.pdf,](https://bitcoin.org/bitcoin.pdf) 2008.
20. 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.
21. 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.
22. Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles,* SOSP '03, pages 29-43, New York, NY, USA, 2003. ACM.
23. Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. In *Proceedings of the 2010 IEEE 26th Symposium on Mass platform service system and Technologies (MSST),* MSST '10, pages 1-10, Washington, DC, USA, 2010. IEEE Computer Society.
24. Lustre. Introduction to Lustre Architecture. [http://wiki.lustre.org/imagesZ6/64/LustreArchitecture-v4.pdf,](http://wiki.lustre.org/images/6/64/LustreArchitecture-v4.pdf) 2017.
25. J. Rosenberg, R. Mahy, P. Matthews, and D. Wing. Session Traversal Utilities for NAT (STUN). RFC 5389, RFC Editor, October 2008. [http://www.rfc-editor.org/rfc/rfc5389.txt.](http://www.rfc-editor.org/rfc/rfc5389.txt)
26. ISO. ISO/IEC 29341-1:2011： Information technology — UPnP Device Architecture, 2011.
27. S. Cheshire and M. Krochmal. NAT Port Mapping Protocol (NAT-PMP). RFC 6886, RFC Editor, April 2013. [http://www.rfc-editor.org/rfc/rfc6886.txt.](http://www.rfc-editor.org/rfc/rfc6886.txt)
28. Ingmar Baumgart and Sebastian Mies. S/Kademlia: A practicable approach towards secure key-based routing. In *ICPADS*, pages 1-8. IEEE Computer Society, 2007.
29. P. Mockapetris. Domain names - implementation and specification. STD 13, RFC Editor, November 1987. [http://www.rfc-editor.org/rfc/rfc1035.txt.](http://www.rfc-editor.org/rfc/rfc1035.txt)
30. Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '01, pages 149-160, New York, NY, USA, 2001. ACM.
31. Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, Middleware '01, pages 329-350, London, UK, 2001. Springer-Verlag.
32. Frederic Giroire, Julian Monteiro, and Stephane Perennes. Peer-to-peer platform service system: A practical guideline to be lazy. *IEEE Global Communications Conference (GlobeCom),* 12 2010.
33. Shawn Wilkinson, Tome Boshevski, Josh Brandoff, James Prestwich, Gordon Hall, Patrick Gerbes, Philip Hutchins, and Chris Pollard. BAISS: A peer-to-peer cloud computing and storage network v2.0. [https://BAISS.io/BAISSv2.pdf,](https://storj.io/storjv2.pdf) 2016.
34. Vijay K. Bhargava, Stephen B. Wicker, IEEE Communications Society., and IEEE Information Theory Society. *Reed-Solomon codes and their applications / edited by Stephen B. Wicker, Vijay K. Bhargava ; IEEE Communications Society and IEEE Information Theory Society, co-sponsors*. IEEE Press Piscataway, NJ, 1994.
35. Jeff Wendling and JT Olds. Introduction to Reed-Solomon. [https://innovation.vivint.com/introduction-to-reed-solomon-bc264d0794f8,](https://innovation.vivint.com/introduction-to-reed-solomon-bc264d0794f8) 2017.
36. Netanel Raviv, Yuval Cassuto, Rami Cohen, and Moshe Schwartz. Erasure correction of scalar codes in the presence of stragglers. *CoRR,* abs/1802.02265, 2020.
37. Kevin D. Bowers, Ari Juels, and Alina Oprea. HAIL: A High-availability and Integrity Layer for cloud computing and storage . In *Proceedings of the 16th ACM Conference on Computer and Communications Security,* CCS '09, pages 187-198, New York, NY, USA, 2009.ACM.
38. Zooko Wilcox. zfec： filefec.py's encode\_file. [https://github.com/tahoe-lafs/zfec/commit/2594d395923dd945cd62,](https://github.com/tahoe-lafs/zfec/commit/2594d395923dd945cd62) 2007.
39. Jeffrey Dean and Luiz Andre Barroso. The Tail at Scale. *Communications of the ACM*, 56:74-80, 2013.
40. Jeffrey Dean and Sanjay Ghemawat. MapReduce： Simplified Data Processing on Large Clusters. *Commun. ACM,* 51(1):107-113, January 2008.
41. J. Paiva and L. Rodrigues. Policies for Efficient Data Replication in P2P Systems. In *2013 International Conference on Parallel and Distributed Systems,* pages 404-411, Dec 2013.
42. Peter Wuille. BIP32: Hierarchical Deterministic Wallets. [https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki,](https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki) 2012.
43. Ari Juels and Burton S. Kaliski, Jr. PORs： Proofs of Retrievability for Large Files. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 584-597, New York, NY, USA, 2007. ACM.
44. 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.
45. Kevin D. Bowers, Ari Juels, and Alina Oprea. Proofs of retrievability: Theory and implementation. In *Proceedings of the 2009 ACM Workshop on Cloud Computing Security*, CCSW '09, pages 43-54, New York, NY, USA, 2009. ACM.
46. Shawn Wilkinson. SIP01: SIP Purpose and Guidelines, (2016). [https://github.com/BAISS/sips/blob/master/sip-0001.md.](https://github.com/storj/sips/blob/master/sip-0001.md)
47. Michael Ovsiannikov, Silvius Rus, Damian Reeves, Paul Sutter, Sriram Rao, and Jim Kelly. The Quantcast File System. *Proc. VLDB Endow.,* 6(11):1092-1101, August 2013.
48. Salman Niazi, Mahmoud Ismail, Seif Haridi, Jim Dowling, Steffen Grohsschmiedt, and Mikael Ronstrom. HopsFS: Scaling Hierarchical File System Metadata Using newSQL Databases. In *Proceedings of the 15th Usenix Conference on File and Storage Technologies,* FAST'17, pages 89-103, Berkeley, CA, USA, 2017. USENIX Association.
49. D. Richard Hipp et al. SQLite. [https://www.sqlite.org/,](https://www.sqlite.org/) 2000.
50. Google Inc. What is gRPC? [https://grpc.io/docs/guides/index.html,](https://grpc.io/docs/guides/index.html) Accessed 2020.
51. T. Dierks and 巳 Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246, RFC Editor, August 2008. [http://www.rfc-editor.org/rfc/rfc5246.txt.](http://www.rfc-editor.org/rfc/rfc5246.txt)
52. Arvid Norberg. uTorrent transport protocol. [http://www.bittorrent.org/beps/bep\_0029.html,](http://www.bittorrent.org/beps/bep_0029.html) 2009.
53. S. Shalunov, G. Hazel, J. Iyengar, and M. Kuehlewind. Low Extra Delay Background Transport (LEDBAT). RFC 6817, RFC Editor, December 2012. [http://www.rfc-editor.org/rfc/rfc6817.txt.](http://www.rfc-editor.org/rfc/rfc6817.txt)
54. Trevor Perrin. The Noise Protocol platform. [https://noiseprotocol.org/noise.pdf,](https://noiseprotocol.org/noise.pdf) 2020.
55. 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.
56. Amazon Inc. Amazon Simple Storage Service - Object Metadata. [https://docs.aws.amazon.com/AmazonS3/latest/dev/UsingMetadata.html,](https://docs.aws.amazon.com/AmazonS3/latest/dev/UsingMetadata.html) Accessed 2020.
57. Tao Ma. ext4: Add inline data support. [https://lwn.net/Articles/468678/,](https://lwn.net/Articles/468678/) 2011.
58. Bram Cohen. The BitTorrent Protocol Specification. [http://www.bittorrent.org/beps/bep\_0003.html,](http://www.bittorrent.org/beps/bep_0003.html) 2008.
59. Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35-40, April 2010.
60. James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Dale Woodford, Yasushi Saito, Christopher Taylor, Michal Szymaniak, and Ruth Wang. Spanner: Google's globally-distributed database. In *OSDI,* 2012.
61. Eugen Rochko and others. Mastodon: Your self-hosted, globally interconnected microblogging community. [https://github.com/tootsuite/mastodon,](https://github.com/tootsuite/mastodon) 2016.
62. Daniel J. Bernstein. Cryptography in NaCl. [https://cr.yp.to/highspeed/naclcrypto-20090310.pdf,](https://cr.yp.to/highspeed/naclcrypto-20090310.pdf) 2009.
63. Daniel J. Bernstein. NaCl: Validation and verification. [https://nacl.cr.yp.to/valid.html,](https://nacl.cr.yp.to/valid.html) 2016.
64. Arnar Birgisson, Joe Gibbs Polit乙 Ulfar Erlingsson, Ankur Taly, Michael Vrable, and Mark Lentczner. Macaroons: Cookies with contextual caveats for decentralized authorization in the cloud. In *Network and Distributed System Security Symposium*, 2014.
65. Jinyuan Li, Maxwell Krohn, David Mazieres, and Dennis Shasha. Secure Untrusted Data Repository (SUNDR). In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6,* OSDI'04, pages 9-9, Berkeley, CA, USA, 2004. USENIX Association.
66. Eu-Jin Goh, Hovav Shacham, Nagendra Modadugu, and Dan Boneh. SiRiUS： Securing Remote Untrusted Storage. In *NDSS,* volume 3, pages 131-145, 2003.
67. Mahesh Kallahalla, Erik Riedel, Ram Swaminathan, Qian Wang, and Kevin Fu. Plutus： Scalable Secure File Sharing on Untrusted Storage. In *Proceedings of the 2Nd USENIX Conference on File and Storage Technologies,* FAST '03, pages 29-42, Berkeley, CA, USA, 2003. USENIX Association.
68. Ralph C. Merkle. A digital signature based on a conventional encryption function. In Carl Pomerance, editor, *Advances in Cryptology — CRYPTO '87,* pages 369-378, Berlin, Heidelberg, 1988. Springer.
69. Lloyd R. Welch and Elwyn R. Berlekamp. Error correction for algebraic block codes. US Patent US4633470A, 1986.
70. John R. Douceur. The Sybil Attack. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems,* IPTPS '01, pages 251-260, London, UK, 2002. Springer-Verlag.
71. Shawn Wilkinson and James Prestwich. SIP02： Bounding Sybil Attacks with Identity Cost, (2016). [https://github.com/BAISS/sips/blob/master/sip-0002.md.](https://github.com/Storj/sips/blob/master/sip-0002.md)
72. Michael Mitzenmacher. The Power of Two Choices in Randomized Load Balancing. *IEEE Trans. Parallel Distrib. Syst.*, 12(10)：1094-1104, October 2001.
73. Fabian Vogelsteller and Vitalik Buterin. ERC-20 Token Standard, (2015). [https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md.](https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md)
74. Braydon Fuller. SIP09： Bandwidth Reputation and Accounting, (2017). [https://github.com/BAISS/sips/blob/master/sip-0009.md.](https://github.com/Storj/sips/blob/master/sip-0009.md)
75. B. C. Neuman. Proxy-based authorization and accounting for distributed systems. In *The 13th International Conference on Distributed Computing Systems,* pages 283-291, May 1993.
76. Bruce Schneier. *Applied Cryptography (2nd Ed.): Protocols, Algorithms, and Source Code in C.* John Wiley & Sons, Inc., New York, NY, USA, 1995.
77. Protocol Labs. Filecoin: A decentralized storage network. [https://filecoin.io/filecoin.pdf,](https://filecoin.io/filecoin.pdf) 2017.
78. Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422-426, July 1970.
79. Asit P. Basu, David W. Gaylor, and James J. Chen. Estimating the probability of occurrence of tumor for a rare cancer with zero occurrence in a sample. *Regulatory Toxicology and Pharmacology*, 23(2):139 - 144, 1996.
80. Harold Jeffreys. An invariant form for the prior probability in estimation problems. *Proceedings of the Royal Society of London. Series A, Mathematical and Physical Sciences,* 186(1007):453-461,1946.
81. Jim Gray. Notes on data base operating systems. In *Operating Systems, An Advanced Course,* pages 393-481, London, UK, 1978. Springer-Verlag.
82. 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.
83. Kyle Kingsbury. Strong consistency models. [https://aphyr.com/posts/313-strong-consistency-models,](https://aphyr.com/posts/313-strong-consistency-models) 2014.
84. Justin Sheehy. There is no now. *Queue,* 13(3):20:20-20:27, March 2015.
85. Amazon Inc. Amazon Simple Storage Service - Data Consistency Model. [https://docs.aws.amazon.com/AmazonS3/latest/dev/](https://docs.aws.amazon.com/AmazonS3/latest/dev/Introduction.html%23ConsistencyModel) [Introduction.html#ConsistencyModel,](https://docs.aws.amazon.com/AmazonS3/latest/dev/Introduction.html%23ConsistencyModel) Accessed 2020.
86. 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.
87. 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.
88. Leslie Lamport. The part-time parliament website note. [https://www.microsoft.com/en-us/research/publication/part-time-parliament/,](https://www.microsoft.com/en-us/research/publication/part-time-parliament/) Accessed 2020.
89. Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133-169, May 1998.
90. Leslie Lamport. Paxos made simple. [https://www.microsoft.com/en-us/research/publication/paxos-made-simple/,](https://www.microsoft.com/en-us/research/publication/paxos-made-simple/) 2001.
91. 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.
92. Robbert Van Renesse and Deniz Altinbuken. Paxos made moderately complex. *ACM Comput. Surv.*, 47(3):42:1-42:36, February 2015.
93. Barbara Liskov and James Cowling. Viewstamped replication revisited. Technical Report MIT-CSAIL-TR-2012-021, MIT, July 2012.
94. Robbert van Renesse and Fred B. Schneider. Chain replication for supporting high throughput and availability. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6*, OSDI'04, page 7, Berkeley, CA, USA, 2004. USENIX Association.
95. Flavio Paiva Junqueira, Benjamin C. Reed, and Marco Serafini. Zab： High-performance broadcast for primary-backup systems. *IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN),* pages 245-256, 2011.
96. lulian 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.
97. H. Howard, D. Malkhi, and A. Spiegelman. Flexible Paxos： Quorum intersection revisited. *ArXiv e-prints*, August 2016.
98. 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.
99. 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.
100. 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.
101. 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.
102. Jean-Philippe Martin and Lorenzo Alvisi. Fast byzantine consensus. *IEEE Trans. Dependable Secur. Comput.,* 3(3):202-215, July 2006.
103. 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.
104. Ramakrishna Kotla. Zyzzyva： Speculative byzantine fault tolerance. *ACM Transactions on Computer Systems (TOCS)*, 27, Issue 4, Article No. 7, December 2009.
105. P. L. Aublin, S. B. Mokhtar, and V. Quema. RBFT： Redundant Byzantine Fault Tolerance. In *2013 IEEE 33rd International Conference on Distributed Computing Systems*, pages 297-306, July 2013.
106. Christopher N. Copeland and Hongxia Zhong. Tangaroa： a Byzantine Fault Tolerant Raft, 2014.
107. Jae Kwon. Tendermint： Consensus without mining. [https://tendermint.com/docs/tendermint.pdf,](https://tendermint.com/docs/tendermint.pdf) 2014.
108. Pierre-Louis Aublin, Rachid Guerraoui, Nikola Knezevic, Vivien Quema, and Marko Vukolic. The Next 700 BFT Protocols. *ACM Trans. Comput. Syst.,* 32(4):12:1-12:45, January 2015.
109. Leemon Baird. The Swirlds hashgraph consensus algorithm: Fair, fast, byzantine fault tolerance, 2016.
110. 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.](https://eprint.iacr.org/2016/199)
111. 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.
112. Vitalik Buterin and Virgil Griffith. Casper the friendly finality gadget. *CoRR,* abs/1710.09437, 2017.
113. Serguei Popov. The Tangle. [https://iota.org/IOTA\_Whitepaper.pdf,](https://iota.org/IOTA_Whitepaper.pdf) 2020.
114. Team Rocket. Snowflake to Avalanche: A Novel Metastable Consensus Protocol Family for Cryptocurrencies. [https://ipfs.io/ipfs/QmUy4jh5mGNZvLkjies1RWM4YuvJh5o2FYopNPVYwrRVGV,](https://ipfs.io/ipfs/QmUy4jh5mGNZvLkjies1RWM4YuvJh5o2FYopNPVYwrRVGV) 2020.
115. Pierre Chevalier, Bart+omiej Kaminski, Fraser Hutchison, Qi Ma, and Spandan Sharma. Protocol for Asynchronous, Reliable, Secure and Efficient Consensus (PARSEC). [http://docs.maidsafe.net/Whitepapers/pdf/PARSEC.pdf,](http://docs.maidsafe.net/Whitepapers/pdf/PARSEC.pdf) 2020.
116. James Mickens. The saddest moment. *;login: logout*, May 2013.

[https://scholar.harvard.edu/files/mickens/files/thesaddestmoment.pdf.](https://scholar.harvard.edu/files/mickens/files/thesaddestmoment.pdf)

1. Linearizable operations are atomic operations on a specific object where the order of operations is equivalent to the order given original “wall clock” time. [↑](#footnote-ref-0)
2. [↑](#footnote-ref-1)
3. US Form 1099 is required by law for any payments to an individual in a given year exceeding a total of $600. [↑](#footnote-ref-2)