Storj

一个点对点的云储存网络

Shawn Wilkinson (shawn@storj.io)

Tome Boshevski (tome@storj.io)

Josh Brandoff (josh.brandoff@gmail.com)

James Prestwich (james@storj.io)

Gordon Hall (gordonhall@openmailbox.org),

Patrick Gerbes (patrickgerbes@gmail.com)

Philip Hutchins (flipture@gmail.com)

Chris Pollard (cpollard1001@gmail.com)

With contributions from: Vitalik Buterin (v@buterin.com)

December 15, 2016

（ EthFans 译 ）

摘要

一个点对点的云存储网络，通过客户侧加密使用户无需倚靠第三方存储服务商就可以传输和分享数据。去除中心控制可以规避大多数传统数据失败和中断，同时可以大幅增加安全性，隐私性，以及数据自主性。一般来讲工业级存储系统难以通过点对点网络实现，因为数据可用性取决于受欢迎程度，而不是用途。我们提出一个搭配直接支付系统的挑战-回应验证系统作为解决方案。通过这个方法，我们可以定期检查数据健全性，同时也激励节点维护数据。我们进一步提出基于独立联合节点的一个寻址访问和性能的模型。

介绍

云存储几乎全部倚靠大型存储服务商作为值得信任的第三方来传输和存储数据。这个系统受累于固有的基于信任的商业模式。由于客户侧加密缺乏标准，传统的云面对各种各样的安全威胁，包括中间人攻击，病毒，应用缺陷导致的用户及公司数据的泄漏等等。此外，由于许多存储设备依赖于相同的底层架构，文件与系统即便是独立的，它们失效的风险却是高度相关的户侧加密来保证，与此同时数据可用性可以通过可读取性证明 PoR 来保障。底层架构失效以及安全漏洞的影响会大大减小。一个开放的数据存储市场可以通过激励现有设备和多方面竞争者的参与有效减低存储成本。网络中的数据会具有抗审查，抗篡改，抗未经授权读取，以及抗数据失效的特性。本文旨在对这样一个网络的具体实现方法以及与如此一个网络进行互动所需的一套工具进行描述。

2 设计

Storj 是一种创建分布式网络的协议，服务于节点间存储合约的形成以及执行。Storj 协议使得网络上的节点能够协商合约，传输数据，验证远程数据的完整性以及可用性，检索数据，并且向其它节点支付费用。每一个节点都是一个自治的机构，能够在无需人们强烈干预的情况下执行这些操作。本文档介绍了这些交互所需的许多基本工具，完整的协议文档可以在其它地方[1]获得。

2.1 文件即加密分片

一个分片是一个加密文件的一部分，存储在 Storj 网络上。分片技术在安全，隐私，性能以及可用性方面有大量优势。

文件在分片之前理应在客户端先完成加密，参考的实现是使用 AES256-CTR，但是近似的加密算法或者任何其它可取的系统可以被实现。加密文件保护了数据内容不被保管数据的存储提供者，或者说是农场主所查看。数据拥有者保留着加密密钥的完全控制权，因此也拥有数据访问的控制权。

数据拥有者可以分开保护“文件是如何分片的以及分片位于网络的哪个位置”信息的安全。随着网络中

分片集合的增长，如果不预先知道给定分片的位置信息，那么定位这些分片的集合将会变得指数级困难（看 6.3 节）。这也反映了文件的安全性与网络容量的平方是成正比的。

分片大小是一个可协商的合约参数。为了保护隐私，推荐分片大小标准化，是一个字节整数倍，例如 8 或者是 32 MB。小文件可以使用 0 或者随机数据进行填充。标准化的大小可以阻止旁路试图决定给定分片的内容，还可以隐藏网络中分片数据的流动。

对大型文件例如视频进行分片，并且跨节点分发分片，可以降低内容传递给任何给定节点带来的影响。带宽需求被更加均匀地分布在网络中。此外，终端用户可以享受到并行传输的优势，类似于 BitTorrent[2]或者其它点对点的网络。

由于节点通常依赖于独立的硬件和基础设施，数据故障是不相关的。这意味着创建冗余的分片镜像，或者对分片集合应用奇偶校验方案是确保可用性的一种极其有效的方法。可用性与存储数据的节点数量是成正比的。

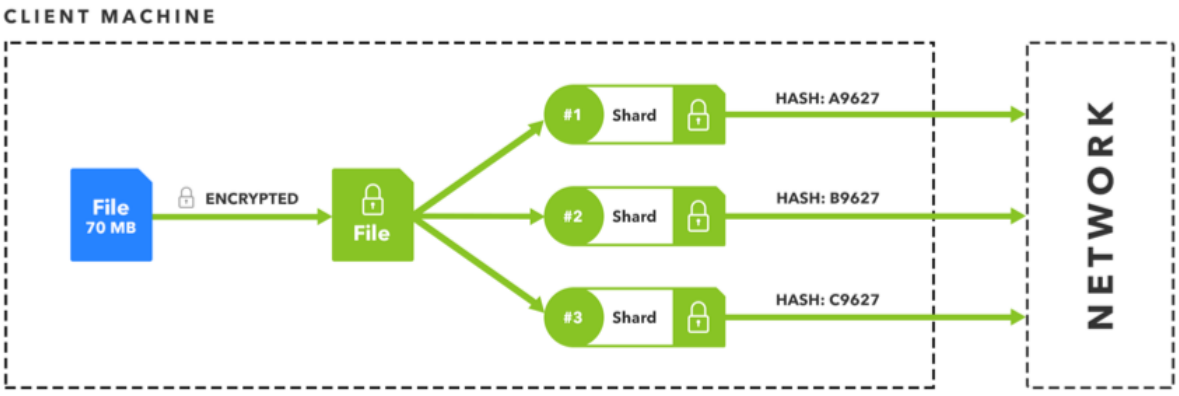

图 1：可视化分片过程

1. 文件被加密。

2. 加密过后的文件被分割成分片，或者多个文件结合形成一个分片。

3. 对每一个分片处理前执行审计（见 2.3 节）。

4. 分片可以传输到网络上。

2.2 Kademlia 与修改

Storj 建立在 Kademlia 之上，一个分布式哈希表（DHT）。值得注意的是，分片并不存储在哈希表中。相反，Kademlia 创建了一个高效消息路由以及其它所需特性的分布式网络。Storj 针对 Kademlia 核心功能（见附录 A）添加了多种消息类型以及改进，该哈希表可以用作存储数据的位置信息，或者是其它的用途。

2.2.1 签名验证

与 S/Kademlia 类似[4]，Storj 网络需要节点对消息签名。为了加入网络，一个节点首先必须创建一个 ECDSA 密钥对，（k_priv, k_pub）。Kademlia 节点 ID 对应于 ripemd160(sha256(k_pub))。 因此，Storj 网络中的每一个节点 ID 同时也是一个有效的比特币地址，节点可以使用该地址进行消费。节点签署所有的消息，并且在处理消息之前验证消息签名的有效性。这个修改使得可以在网络上实施长期的身份认证，并且提供了工作量证明，可以有效防止对 Kademlia 路由（见 5.2 节）的 Eclipse 攻击。未来该地址还有其它各种用途。

2.3 可检索性证明

可检索性证明确保一个特定数据片段存在于一个远程的主机上。理想的证明最小化消息大小，能够被快速计算，需要最小化的预处理，并且提供对文件可用性以及完整性的高置信度。为了向数据所有者提供数据完整性和可用性的知识，Storj 提供了一种标准格式，用于通过称为审计或心跳的挑战-响应交互来发布和验证可检索证明。

我们的参考实现是使用默克尔树（Merkle trees[5]）和默克尔证明。在分片过程完成后，数据所有者生成一个包含 n 个随机挑战盐值（salt，密码学中使用到的随机数）s_0，s_1，...s_n-1 的集合，并且存储该集合。每一个挑战盐值均附加在数据 d 上，并且对所得的字符串进行哈希，形成一个前置叶子节点 p ：$p\_i = H(s\_i + d)$。前置叶子进一步进行哈希，所得数字摘要成为一棵默克尔树的叶子节点 l 集合，即 $l\_i = H(H(s\_i + d))$。向叶子节点集合中填充空白字符串的哈希值，直到其基

数是 2 的幂次方，以简化处理过程。

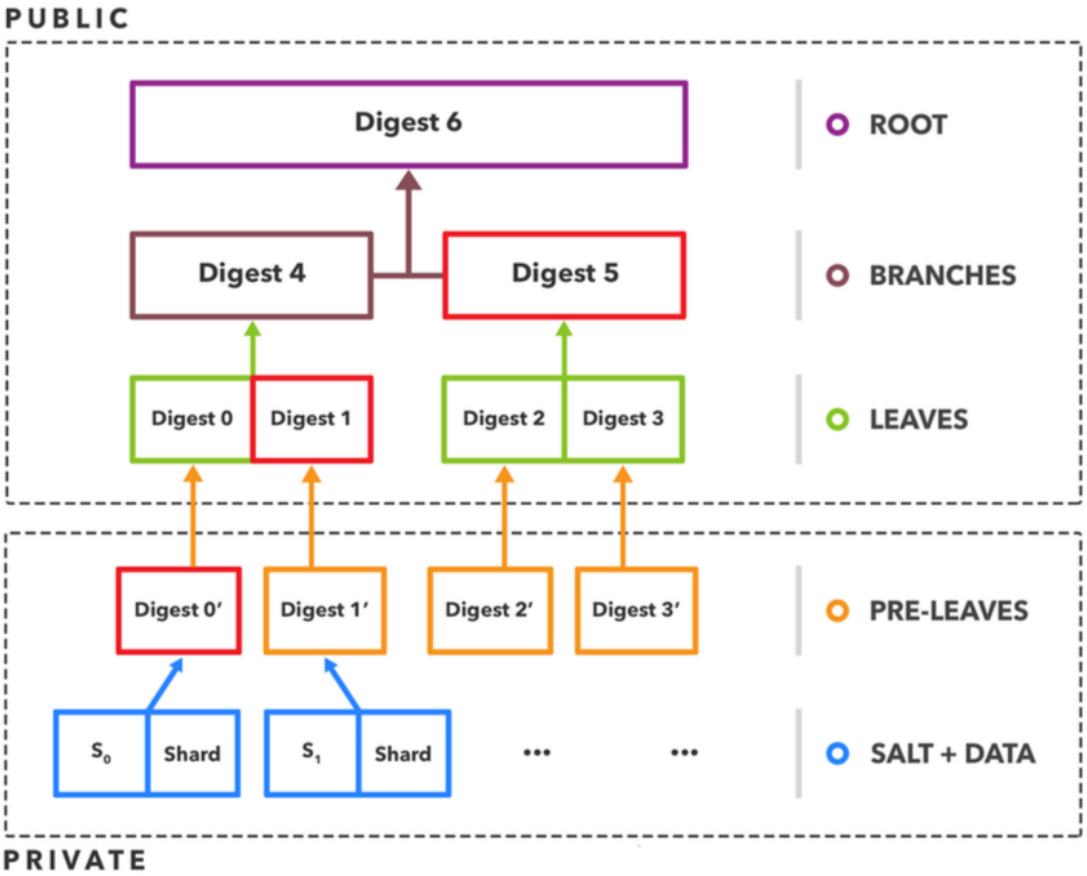

图 2：Storj 审计树，|/| = 4。红色轮廓线代表 s_0 的默克尔证明的元素。

数据所有者存储挑战盐值的集合，以及默克尔树根节点以及默克尔树深度，然后传输默克尔树的叶子节点到农场主。农场主将叶子节点与分片存储在一起。数据所有者周期性地从存储的集合中选择一个挑战，并向农场主发起该挑战。可以根据任意合理的模式来选择挑战，但挑战不应该被重复使用。农场主使用该挑战与数据来生成前置叶子节点。所得前置叶子节点，与叶子节点集合一起，被用来生成一个默克尔证明，然后将该证明返回给数据所有者。

Storj 默克尔证明总是恰好由 $\log_2(|/|) + 1$ 个哈希值组成，因此即使针对一个大型的默克尔树，这也是一个紧凑的传输。数据所有者使用存储的默克尔树根节点以及树深度来验证接收到的默克尔证明，过程是：(1)证明的长度等于树深度；（2）证明提供的哈希值可以重新生成存储的根哈希值。该方案不允许错误的负面或者错误的正面结果，因为哈希函数要求每个位保持不变以产生相同的输出。

2.3.1 部分审计

默克尔树审计方案对于数据所有者来说需要大量的计算开销，因为整个分片必须被哈希多次以生成前置叶子。该方案的一个扩展时使用数据的子集来执行部分审计，从而减少计算开销。这也具有降低农场主资源 I/O 负担的优点。

此扩展方案依赖于两个附加的可选参数：一个分片内字节索引 x 的集合与一个以字节为单位的片段长度，b。数据所有者存储一个 3 元组（s，x，b）的集合。为了生成前置叶子 i，数据所有者将 s_i 附加到位于 x_i 处的 b_i 个字节前面。在审计过程中，验证者将(s，x，b)_i 传输给农场主，农场主使用该信息生成一个前置叶子。默克尔证明和验证过程与上述相同。

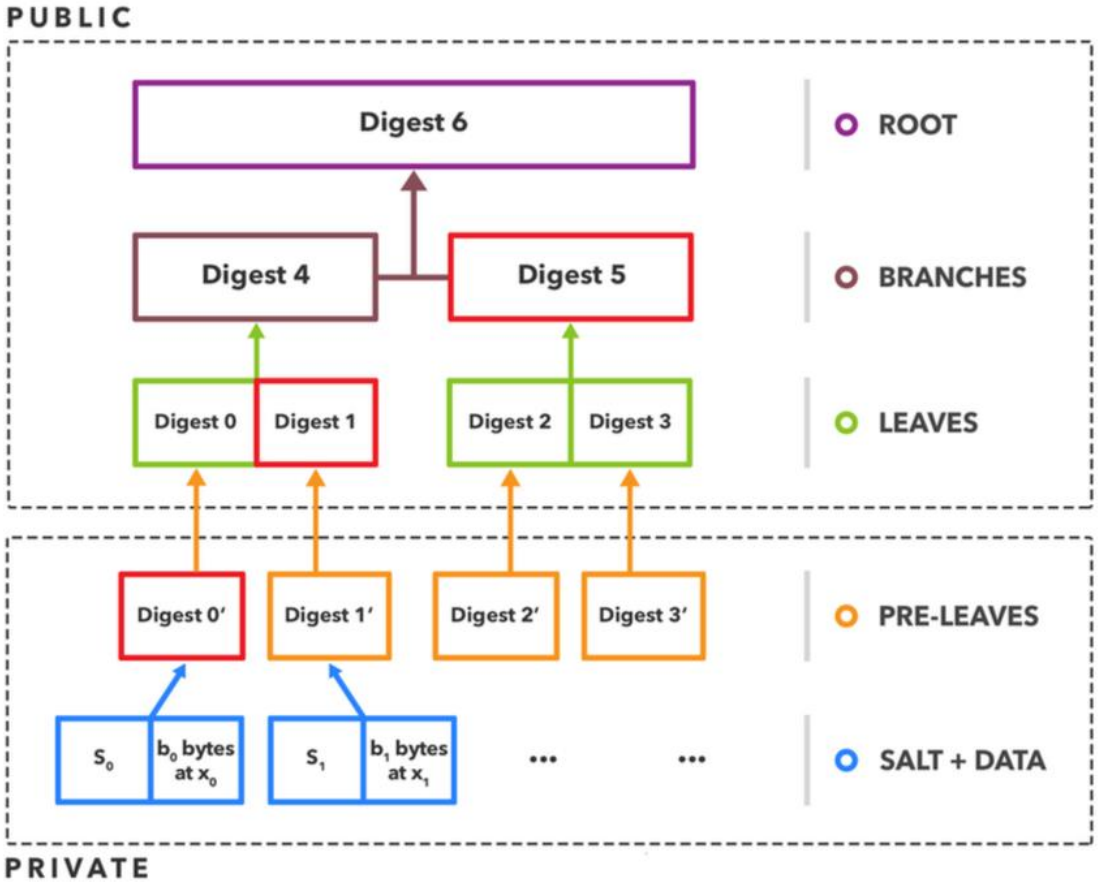

图 3：Storj 审计树，|I|=4 以及部分审计，红色轮廓线代表 s_0 的默克尔证明的元素

部分审计仅仅提供了农场主保留完整文件的可能性保证。它允许错误的正面结果，即验证者相信农场主保留着完整分片，但实际上该分片已经被修改或者部分删除。一个独立部分审计上出现错误的正面结果的概率是很容易被计算出的（见 6.4 节）。

因此数据所有者可以对“分片任然完整且可用”有一个已知的置信度。实践中，这会变得更加复杂，因为农场主可能会采取高明的策略来试图击败部分审计。幸运的是，在迭代审计过程下，这是一个有界的问题，几个连续误报的概率变得非常低，即使是在文件的小部分已经被删除的情况下。

此外，部分审计可以很轻松地与完整审计混合，而无需重新构建默克尔树或者修改验证过程。可以设想出许多混合部分审计与完整审计的策略，每种审计策略随着时间的推移可以提供不同水平的置信度。

该方案的进一步扩展使用一个确定性种子而不是一个字节索引集合。该种子用于生成文件中许多非连续的字节。要求许多非连续随机字节将会给那些尝试实施审计逃避策略而无需额外处理或者 I/O 负担的恶意农场主带来额外的阻力。

2.3.2 其它可检索性证明方案

检查了其它的审计方案，但被普遍认为是不可行的。例如，Shacham 和 Waters 曾提出过一个比默克尔树方案具有更多优势的紧凑证明[6]。该构建方案允许数据所有者以最小化的数据存储来构建一个无尽的流式挑战。它还允许公开对挑战回应进行验证。

然而，初期实现表明，Shacham-Waters 方案所需的客户端预处理至少需要一个比基于哈希的方法高一个量级的计算时间，这对大多数应用程序来说太慢了。

可检索性证明是一个正在进行的研究领域，未来也许可以发现其他实用的方案。随着可检索性证明方案的发现与实现，方案的选择成为一个可以协商的合约参数。这将会允许每一个数据所有者和节点实现各种各样的方案，然后针对特定的目的选择一个最优的方案。

2.3.3 发起审计

为了发起审计，Storj 扩展了 Kademlia 消息集合，添加了一种新类型：AUDIT（Kademlia 消息类型的全部扩展，见附录 A）。这些消息由数据所有者发送，包含数据的哈希值以及一个挑战。农场主必须回应一个默克尔证明，正如上述所描述的。一旦接收并且验证了默克尔证明的有效性，数据所有者必须根据事先同意的条款向农场主支付一定的费用。

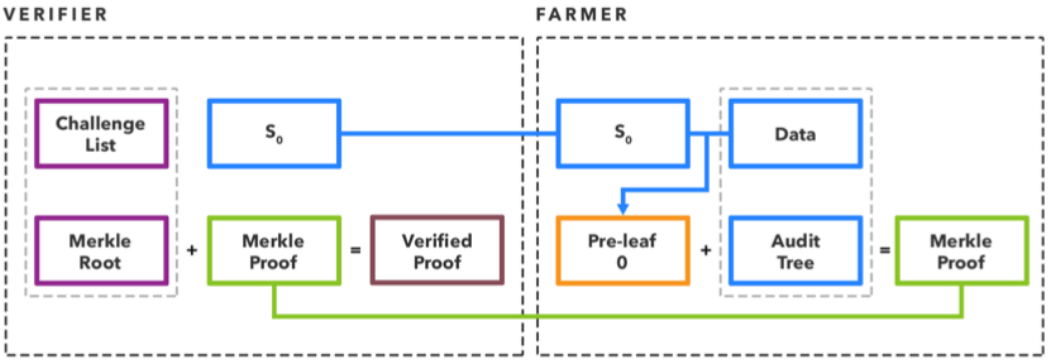

图 4：发起与验证 Storj 审计

2.4 合约与协商

数据存储通过标准的合约形式来进行协商。合约是一个版本化的数据结构，描述了数据所有者与农场主的关系。合约应该包含节点所需的所有必要信息，以让节点随着时间推移可以形成关系，转移数据，创建和响应审计，以及任意的支付。这包括分片哈希，分片大小，审计策略，以及支付信息。Storj 实现了一个发布/订阅系统以连接有兴趣形成一个合约的实体（见 2.6 节）。

每一个实体应该存储一份签过名的合约拷贝。合同仅为数据所有者与农场主的利益而存在，因为没有其它的节点能够验证关系的条款或状态。将来，合约信息也许会被存储在分布式哈希表中，或者是一个外部的总账，比如区块链，这可以允许关系条款的一些外部验证。

合约系统扩展了 Kademlia，添加了四种新的消息类型：OFFER，CONSIGN，MIRROR，和 RETRIEVE。

为了协商一个合约，一个节点创建一个 OFFER 消息，将它发送给一个预期的伙伴。通过 2.6 节中描述的发布/订阅系统来寻找一个预期的伙伴。这条 OFFER 消息包含了一个完整构建的合约，其中描述了想要的关系。这两个节点重复性地交换签过名的 OFFER 消息。对于 OFFER 循环中的每一条新消息，节点要么选择中止协商，回复一条新签署的还价消息，要么通过联合签署来接受合约。一旦两个实体全都签署了 OFFER 消息，则他们各自在本地存储该消息，使用数据的哈希来作为索引该消息的键。

一旦达成协议，数据所有者发送一条 CONSIGN 消息给农场主。该消息包含审计树的叶子节点。农场主必须回应一个 PUSH 的令牌，来授权数据所有者通过 HTTP 传输来上传数据（见 2.10 节）。这个令牌是一个随机数，并且可以委托给一个第三方实体。

RETRIEVE 消息表示从一个农场主处检索一个分片。这些消息几乎与 CONSIGN 消息是一样的，但是不包含审计树的叶子节点。农场主使用一个 PULL 令牌来回应一个有效的 RETRIEVE 消息，PULL 令牌授权数据所有者通过一个单独的 HTTP 连接来下载数据。

MIRROR 消息指示农场主从另一个农场主那里检索数据。这允许数据所有者创建冗余的分片拷贝而无需耗费大量额外的带宽或时间。经过一个成功的 OFFER/CONSIGN 处理过程后，数据所有者也许初始化了一个另外的 OFFER 循环，该循环过程涉及相同的数据，和不同的农场主。数据收发者向原始的农场主发起一条 RETRIEVE 消息，然后向镜像农场主发送一条包含检索令牌的 MIRROR 消息，而不是直接向镜像农场主发送 CONSIGN 消息。MIRROR 消息中的令牌授权镜像农场主从原始农场主那里检索数据。应该通过一条 AUDIT 消息来立即验证 MIRROR 过程成功与否。

2.5 支付

Storj 是支付无关的。协议和合约都不需要一个特定的支付系统。目前的实现假定使用 Storjcoin，但是许多其它的支付类型可以被实现，包括比特币，以太币，自动清算所转移（ACH transfer），或者是实物的物理转移。

参考实现将会使用 Storjcoin 微支付通道，目前正处于开发过程中[8]。微支付通道允许直接配对付款来审计，从而最大限度地减少了农场主与数据所有者之间所需的信任。然而，由于数据存储是廉价的，审计金额是及其微小的，通常每一次审计花费在 0.000001 美元之下。

Storjcoin 允许比其他候选货币更加精细的付款，从而最大限度地减少各方之间的信任。此外，微支付通道机制要求通道中流通的总价值在整个通道的生命周期中托管到第三方（或者说区块链）。这就降低了货币流通速度，暗示了价值波动会严重影响微支付通道的经济激励。一个独立代币的使用创造了与外界变化一定的绝缘性，Storjcoin 的大量供应最大程度地减少了市场上代币托管带来的影响。

全新的支付策略必须包含一种货币，一个存储价格，一个检索价格，以及一个付款接收方。强烈建议全新的支付策略考虑数据所有者如何证明已经付款，以及农场主如何在没有人为干预的情况下验证收款。微支付网络，如闪电网络[9]，以及其它支付策略的实现细节留待感兴趣的人去探索。

2.6 Quasar

Storj 实现了一个点对点发布/订阅系统，被称为 Quasar[10][11]。Quasar 利用布隆过滤器提供了基于主题的发布/订阅系统[12]。主题描述了广播节点想要的合约参数范围，包括合约大小和带宽承诺。主题列表在协议层是标准化的，并且易于扩展[13]。这通过确保消息到达感兴趣的节点（见 2.4 节）来促进合同提供与协商的过程。

为了实施 Quasar，Storj 扩展了 Kademlia，添加了三种新的消息类型：SUBSCRIBE，UPDATE，和 PUBLISH。这些消息促进了过滤器的创建与传播。每一个节点以深度 K = 3 的布隆过滤器形式维护着它订阅的主题信息，以及它的邻居订阅的主题信息。级别 1 的过滤器代表一跳的节点订阅。

SUBSCRIBE 从邻居处请求过滤器列表。为了构建一个初始化的过滤器列表，节点给它们三个最近的邻居发送 SUBSCRIBE 消息。节点使用它们目前的过滤器列表回应一个 SUBSCRIBE 请求。发送请求的节点将收到的过滤器列表添加到它自己的衰减布隆过滤器中。

UPDATE 将本地过滤器改变推送给三个最近的邻居。当一个节点订阅一个新主题的时候，它必须发送一个 SUBSCRIBE 请求来学习它邻居的状态，然后发送一个 UPDATE 请求来通知邻居其新的订阅。通过 SUBSCRIBE/UPDATE 循环来交换过滤器列表，节点更好地了解到什么是它们邻居想要的以及通过邻居可以达到的节点。

PUBLISH 会在网络中广播一条消息。在 Storj 中，这些通常是部分构建合约的公告（见 2.4 节）。

PUBLISH 消息被发送到一个节点的最近三个邻居，并且包含一个主题参数，其摘要将会与过滤器列表中的每一个过滤器进行比较。如果主题摘要在 K = 0 的过滤器处被发现，节点将会处理该消息。如果主题摘要在过滤器列表中其它任意的过滤器中被发现，节点将会前向转发该消息到它的邻居们。如果没有发现匹配的过滤器，该消息将会被转发给在节点路由表中一个随机挑选的节点。

为了防止消息的重复发送，节点在转发 PUBLISH 消息时，会将它们自己的节点 ID 加入到该消息中，表示该节点已经接收到该消息。当转发消息时，节点忽略掉在消息列表中出现的 ID 对应的节点。这以最小限度的通信开销，阻止了消息的冗余传播。

PUBLISH 消息也包括一个存活时间（TTL）参数，以跳数来衡量。节点将不会传播跳数已经超过存活时间的消息。为了防止垃圾攻击，节点也将会拒绝传播存活时间超过该节点将会给出新消息的存活时间的消息。

由于基于订阅过滤器的路由过程，PUBLISH 消息通过网络随机传播，直到它们找到一个过滤列表中包含订阅或者错误的正确结果（布隆过滤器的特点，即匹配了过滤器，也不一定真正存在）的节点。一旦订阅由过滤器指示，消息就会快速向订阅者移动。如果收到了一个误报，消息将会再次进行随机路由。

显而易见，数据所有者需承担重大的责任，以维护 Storj 网络上的数据可用性和完整性。由于节点不能被信任，而且，像挑战集（challenge set）这样的隐藏信息不能安全地外包给不可信任的个体，数据所有者有责任去协商合同，预处理分片，发布和验证审计，提供付款，通过分片收集管理文件状态，管理文件密钥等等。这些功能中有许多都需要较长的正常运行时间和大量的基础架构，尤其是对于一组现用文件来说。用户运行的应用程序，如文件同步，是无法有效地在网络上管理文件的。

为了从大量的客户应用中用简单的方式访问网络，Storj 实现了一种精简客户端（thin-client）模型，将信任委托给一个专用服务器来管理数据所有权。这与比特币和其他加密货币生态系统中发现的 SPV（译者按：Special Purpose Vehical, 特殊目的实体）钱包的概念相似。数据所有者的负担可以通过各种方式跨客户端和服务器进行拆分，通过改变委托的信任量，服务器还可以提供各种其他有价值的服务。我们把这种专用服务器称为 Bridge，它已经被开发出来并作为免费软件发布。任何个人或组织都可以运行他们自己的 Bridge 服务器来使网络接入更便利。

2.7 冗余方案

云对象存储通常拥有或租用服务器来存储客户文件。它们使用 RAID 方案或多数据中心方法来保护文件免受物理或网络故障的影响。 由于 Storj 对象存在于不可信任对等体的分布式网络中，所以农民不应该依赖于采用与传统云存储公司相同的数据丢失安全措施。事实上，农民可以随时简单的关掉节点。因此，强烈建议数据所有者实施冗余方案以确保其文件的安全性。因为该协议只处理各个分片的合约，所以可以使用许多冗余方案。以下介绍三个方案。

2.7.1 简单镜像

最简单的解决方案是跨几个节点镜像分片。镜像通过确保每个分片存在的多个副本来防止硬件故障。该方案分片的可用性为 $P = 1-\prod_{0}^{n} a_n$ ，其中 an 为存储分片 n 的节点的正常运行时间。因为需要组合所有分片，文件的可用性等于最少的可用分片的可用性。在丢弃合约的情况下，可以检索该分片的冗余副本，并在网络上找到新的位置。 这是参考实现的当前行为。

2.7.2 M 的 K 擦除编码

Storj 将很快实现客户端 Reed-Solomon 擦除编码[14]。擦除编码算法将文件分解成 k 个分片，并以编程方式创建 m 个奇偶校验分片，总共给出 k + m = n 个分片。这些 n 个分片中的任何 k 个分片可用于重建文件或任何丢失的分片。那么文件的可用性就是 $P = 1-\prod_{0}^{m} a_m$ 通过 m + 1 个最少可用节点的集合。在丢失单个分片的情况下，可以检索文件，重建丢失的分片，然后为丢失的分片协商一个新的合约。

为了防止文件丢失，数据所有者应设置分片丢失的容忍级别。请考虑 40 位擦除编码方案。数据所有者可能会容忍在 40 份之内损失 5 个分片，因为知道在不久的将来 16 次更难以进入的机会很低。然而，在某些时候，概率可用性将低于安全阈值。此时，数据所有者必须启动检索

和重建程序。

由于通过审计过程已知节点正常运行时间，因此可以根据所涉及的节点的特性来优化容忍水平。可以实施许多策略来处理这个过程。

擦除编码是期望的，因为它大大降低了失去访问文件的可能性。 它还降低了为文件实现给定级别的可用性所需的磁盘空间。擦除编码方案受不是受到最少的可用分片的限制，而是受最少可用的 n + 1 个节点限制（见第 6.1 节）。

2.8 KFS

为了促进农民在磁盘存储，Storj 实现了一个名为 KFS 的本地文件存储[15]。农户最初直接使用文件系统来存储分片。后来，农户使用了一个单一的 LevelDB 实例。这两种方法都没有扩大规模。例如，LevelDB 压缩处理时间与存储大小线性关系，并在进行时锁定读取和写入。这显着影响了存储超过 100GB 的节点的性能和可用性。KFS 是一组 LevelDB 实例中的抽象层，旨在解决扩展问题。

2.8.1 理由

LevelDB 是一个键值存储。它具有许多理想的品质，包括长期的支持，可移植性以及按字典排序的键驱动的高性能读取。虽然 LevelDB 压缩通常是一个理想的功能，但它严重限制了扩展。它在下端系统上的影响更大，也可能会根据使用的磁盘类型而有所不同。压缩还会在此期间阻止读取和写入，从而致使 Storj 节点有效地脱机直到进程完成。

然而，由于 LevelDB 实例便于创建，打开和关闭，压缩成本可以通过管理一组受大小限制的 LevelDB 实例来限制。实例可以进行临时初始化，并根据需要进行打开和关闭。水平扩展多个 LevelDB 实例具有许多可扩展性的优点。主要是减轻压缩对操作的影响。因为压缩在每个实例

上单独运行，而不是横跨整个数据集，所以压缩引起的问题是最小化的。尽管整个分片集中的压缩将采用大致相同的计算量（压缩与数据线性关系），但它现在每个实例都分别出现。也就是说，压缩被分解成 256 个更小的进程独立运行。

使用 KFS 压缩只锁定个别存储器，留下许多其他的存储器可读写。操作和压缩均匀地分布在数百个存储器中，这意味着通过压缩阻止操作的机会很小。而之前的压缩将会将整个分片集阻塞几秒钟（或者在低端硬件上更长时间），它现在只能在较短的时间段内阻止分片集的小部分。

2.8.2 S-存储器和路由

而不是单个大型实例，KFS 将分片存储在 B 大小限制的 LevelDB 实例中，称为 S-存储器。集合 S-存储器 L0，L1，LB-1 形成 BTable.S-存储器具有固定的最大尺寸（以字节为单位）S.因此，KFS 存储的最大尺寸为 S * B 字节。Storj 目前使用 S = 32 GB，B = 256，总容量为 8 TB.KFS 要求有一个参考标识符，可以是 R≥log2（B）的任意 R 字节密钥。Storj 节点将使用其 160 字节节点 ID。按照以下方法将进入的分片分到 S-存储器：

1.令 g = [log2（B）]。

2.让 h 成为 R 的第一个 g 位。

3.让 i 成为分片哈希的第一个 g 位。

4.令 n = h⊕i。

5.将分片存放在 Ln 中。

这种排序算法是快速，确定性的，并且仅使用容易获得的信息。其中 B 是 2 的幂，它还可以在所有 S-存储器上均匀分配分片，如下面的图 5 所示。

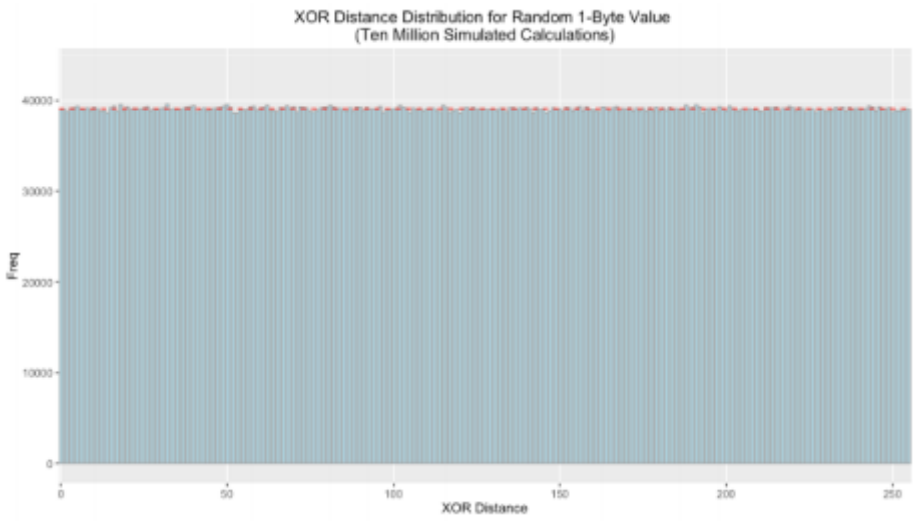

图 5：XOR 随机字节的距离

达到 S 字节的 S-存储器不能存储更多的分片。农民可以通过使用合约中的数据哈希字段计算 n 来确定合约谈判中存储器 Ln 是否已满，并且应该拒绝导致 S-存储器超载的合约。当 KFS 实例中的所有存储器都已满时，可以进行第二个实例。鉴于农民 KFS 实例的 8 TiB 上限比大多数可用驱动器大，所以对大多数硬件来说，这是不可能的。

2.8.3 分片哈希的关键

如前所述，LevelDB 按照字典顺序排列项目。KFS 利用这个优势优化读和写的效率。数据以 C 字节（或更少）的块存储。默认情况下 C = 128 KB。这些块由完整内容的哈希后跟空格和数字索引写入。这确保了键/值很小，并且读取和写入 S-存储器是按顺序的。它还允许有效的数据流进和流出 S 存储器。

由于按字典顺序排列数字字符串的特殊性，索引子字符串应以基数 10 表示，并且具有常数的字符数。索引字符串的长度 l 应由 l = [log10S / C]定义。对于默认参数 l = 6，因此 i = 3753 的块将具有索引数'003753'。这确保分片的块连续存储。

为了保持顺序，从而最大化读/写性能，特定分片的块的键应该是生成如下的字符串：

1.确定块索引数，并将其编码为字符串。

2.使用'0'预处理索引字符串，直到达到长度 l。

3.将 H（数据）编码为十六进制字符串。

4.将一个空格追加到散列。

5.将修改后的块索引附加到散列。

2.8.4 性能优势

在初始测试中，KFS 在各种文件大小的读取，写入和取消链接方面胜过香草 LevelDB。 KFS 在文件大小，操作和存储设备类型的几乎所有组合中显示较低的均值和较低的差异。 它特别擅长取消链接和写入大文件，将方差减少几个数量级。 这些测试的完整方法及其结果可以在别处找到[16]。

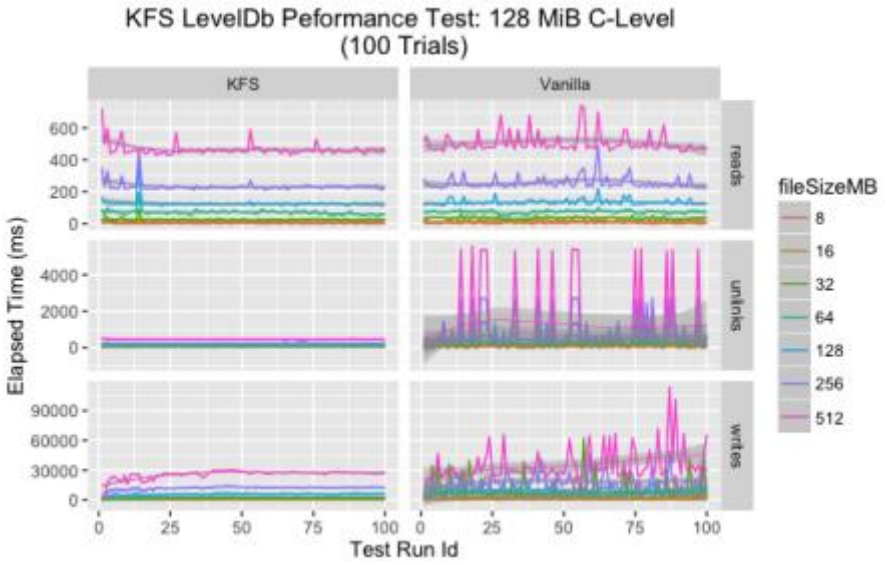

图 6：KFS 和 LevelDB 的相对性能

2.9 NAT 穿越和反向 HTTP 通道

由于存在 NAT 和其他不利的网络条件，并非所有设备都可以公开访问。为了使非公共节点能

够参与到网络中，Storj 实现了一个反向通道系统。

为了方便这个系统，Storj 用三种附加的消息类型扩展了 Kademlia：探测，发现通道和打开通道。通道系统还利用 2.6 节中详细介绍的发布/订阅系统。

探测消息允许节点确定它是否是可公共寻址的。消息被发送到公共可寻址节点，通常是已知的网络种子。接收节点发出单独的 PING 消息。接收节点然后响应于具有 PING 结果的探测消息。加入网络的节点应立即发送一个探测到任何已知节点。

对他们初始探测收到否定响应的节点应向任何已知节点发出发现通道的请求。该节点必须通过发布/订阅系统先前发布通道通知的三个联系人进行响应。通道提供者必须是可公开寻址的。

一旦非公共节点收到通道提供商的列表，它会向通道提供商发出打开通道的请求。如果它们有能力，提供者必须为该节点提供一个隧道。要打开连接，提供者把通道信息以肯定的反应发回。然后，通道节点打开与供应商的长期连接，并且为了反应通道地址更新它自己的合约信息。

TCP 插座通过自定义反向隧通道库 Diglet [17]运行通道。Diglet 为通用反向通道提供了一个简单灵活的界面。它可以通过命令行和程序访问。

2.10 数据传输

数据通过 HTTP 传输[18]。农民暴露可能会上传或下载分片的端点。客户的请求通过先前的 CONSIGN 和 RETRIEVE 消息提供的代币进行身份验证。这种传输机制对协议不是至关重要的，并且将来可能会实现许多替代方案。

3.网络接入

显而易见，数据所有者需承担重大的责任，以维护 Storj 网络上的数据可用性和完整性。由于节点不能被信任，而且，像挑战集（challenge set）这样的隐藏信息不能安全地外包给不可信

任的个体，数据所有者有责任去协商合同，预处理分片，发布和验证审计，提供付款，通过分片收集管理文件状态，管理文件密钥等等。这些功能中有许多都需要较长的正常运行时间和大量的基础架构，尤其是对于一组现用文件来说。用户运行的应用程序，如文件同步，是无法有效地在网络上管理文件的。

为了从大量的客户应用中用简单的方式访问网络，Storj 实现了一种精简客户端（thin-client）模型，将信任委托给一个专用服务器来管理数据所有权。这与比特币和其他加密货币生态系统中发现的 SPV（译者按：Special Purpose Vehical, 特殊目的实体）钱包的概念相似。数据所有者的负担可以通过各种方式跨客户端和服务器进行拆分，通过改变委托的信任量，服务器还可以提供各种其他有价值的服务。我们把这种专用服务器称为 Bridge，它已经被开发出来并作为免费软件发布。任何个人或组织都可以运行他们自己的 Bridge 服务器来使网络接入更便利。

3.1 Bridge

我们对该模型的参照实例包括一个 Bridge 服务器和一个客户端库。Bridge 提供一个对象储蓄，也就是说，Bridge 的主要功能就是向应用程序开发人员公开一个 API（译者按：Application Programming Interface, 应用程序编程接口）。开发者应该能够通过简单的客户端使用 Bridge，而不需要非常了解网络、审核程序或者加密货币。Bridge 的 API 是一个可以简化开发过程的抽象层。这使开发者能够创建许多使用 Storj 的程序，使 Storj 网络覆盖许多用户。

在目前的实践过程中，Bridge 负责合同协商、发布审核和验证、支付和文件状态，客户端负责加密、预处理和文件密钥管理。Bridge 通过具象状态传输接口（RESTful API（译者按：

Representational State Transfer，具象状态传输））开放了对这些服务的访问。这样，客户即使对 Storj 协议和网络完全无经验，也可以利用该网络。*此外，因为专用服务器具有可信的较高正常运行时间，客户可以被纳入不可信的用户空间应用中去。*

Bridge 旨在仅仅储存元数据，它不会缓存加密分片，而且除了公共储蓄桶（public buckets）以外并不持有密钥。关于文件，Bridge 能够与第三方分享的知识仅是诸如访问模式一类的元数据。该系统保护客户的隐私，并使客户对数据访问有完全的控制，同时将保持网络中文件可用性的责任委托给 Bridge。

我们可以设想，让 Bridge 升级，以达到不同级别的委托信任。一个 Bridge 的客户端可能希望保留对发布和验证审核，或者管理分片指针（译者按：指向该分片所在的节点信息）的控制，或者客户端可以选择两个或以上不相关的 Bridge 来管理其审计，以最小化它对任一 Bridge 服务器的信任。从长远来看，数据所有者的任何功能都可以通过委托给两方或多方进行拆分。

3.2 Bridge API and Client

Bridge API 和客户端

完整的 Bridge API 文件超出了本白皮书的范畴，但你可以在其他地方找到它[19]。第一个完整的客户端实现是在 JavaScript 中，C 语言、Python 和 Java 的实现正在进行中。

因为文件不能够简单地被贴到 API 端点上，所以 Bridge API 和客户端的结构和现有的对象存

储不同。客户端实现了通过简单而熟悉的界面来隐藏在 Storj 网络中管理文件的复杂性。复杂的网络操作尽可能地被抽象出简单的函数调用。

上传过程的简单介绍如下：

1. 客户端收集数据并进行预处理。
2. 客户端通知 Bridge 等待上传的数据。
3. 与网络节点协商合约。
4. 返还合约节点的 IP 地址和授权代币给客户。
5. 客户端使用 IP 地址和代币联系挖矿节点，并上传数据。
6. 客户端将审核信息传送给 Bridge，并委托信任。
7. 以证明数据被正确传送。
8. Bridge 承担起发布审核，支付文件储存者和管理文件状态的责任。
9. Bridge 通过 API 向客户端开放文件元数据。

下载步骤也类似。

1. 客户端通过标识符请求文件。
2. Bridge 验证请求，并提供文件储存者的 IP 地址列表和代币。

3. 客户端使用这些地址和代币来接收文件。

4. 文件在客户端被重新组合并解密。

该 JavaScript 库接收文件，进行预处理，并根据 Bridge 指示管理连接。它还让解密下载可用于文件或者流格式的应用程序。一个使用该库的 CLI（按：Command-line Interface，命令行界面）示例在 https://github.com/storj/core-cli 上可以免费获取。它经过了多种大小的文件测试，能够可信地从 Storj 网络中传输 1080p 视频。

3.2.1 程序开发工具

Bridge 和 Bridge API 的首要功能是服务于应用程序。为此，正在开发各种不同语言的客户端和工具。

Storj.js[20]力求提供一个标准的浏览器界面，用于从 Storj 下载文件。虽然在早期阶段，它已经可以与 Bridge 交流，检索文件指针和代币，从储存者处检索分片，重组分片，并将完成的文件附加到 DOM（Document Object Model，文件对象模型）。这样，网络开发者可以轻松地从同一页面内引用 Storj 对象，并依靠它们来无恙地传递给终端用户。这可以应用于提供任何从浏览器文档编辑到图像储存等各种服务。

网络后端的密钥和文件管理工具还处于早期计划阶段，包括用于标准化后端工具的 Storj 补丁，如内容管理系统。这些工具可以帮助以内容驱动的程序开发者使用 Storj 网络上的文件。用户标准化这些文件许可工具可以帮助创建服务器之间的数据移动性，详见第 4.2 节的讨论。

对于其他协议和工作流程的 Bridge 也在计划中。Storj 的 CLI 适用于程序化脚本（Shell Scripting）的自动化。未来还会开发类似的工具用于 FTP（File Transfer Protocol，文件传输协议），FUSE（Filesystem in Userspace，用户空间文件系统）和用于文件交互的通用工具。

3.3 Bridge 作为授权机制

Bridge 可以被用作管理网络上私人文件的授权。因为 Bridge 管理着每一份合约的状态，它是这些服务在逻辑上的提供者。它可以管理各种授权服务来实现分享和协作。

3.3.1 身份和许可

Bridge API 用公钥加密来验证用户。用户跟 Bridge 注册公钥，而不是 Bridge 服务器发送 API 密钥给每个用户。所有 API 请求都要签名，随后 Bridge 验证该签名和注册的公钥匹配。Bridge 将文件的元数据组织到桶（bucket）中以便于管理。可以通过将一系列公钥注册到 bucket 上使单独获得许可。

程序开发者可以使用它轻松地委托权限给应用、服务器或者其他开发者。例如，一个文件同步服务的开发者可以为每名用户创建一对该服务的密钥，并将每名用户划分到仅用该对用户密钥才能访问的桶中。每个桶的使用都被追踪，因此超过其配额的用户可以通过编程撤销其写入权限。这就提供了一个用户许可以及各种组织工具间的逻辑分离。

3.3.2 密钥迁移

由于分片加密密钥都保存在生成它们的设备上，数据的迁移性是个问题。在 Bridge 的参照实例和客户端方便了客户端之间文件加密密钥的安全传输。客户端生成一个保密性强的种子文件，默认情况下是随机生成的十二个单词组成的句子。要加密给定的文件，客户端会根据种子文件，桶的 ID 和文件 ID 生成一个特定的关键字。

用户只需一次将种子导入每个新设备，就可以保持设备永久同步。这也有利于备份，因为用户只需要保存种子文件而不是每个新生成的文件密钥。

3.3.3 公开文档

和其他对象储蓄一样，Bridge 允许开发者通过公共储蓄桶创建和传播公开文档。Bridge 服务器让开发者可以上传加密密钥，然后允许匿名用户检索文件密钥和一套文件指针。公共储蓄桶有助于向网页或者面向公众的应用发送内容。

一个不需要通过 Bridge 来共享和检索公共文件的系统也可以被创建出来。人们可以在任何平台上公开发布指针和密钥，客户可以直接向文件储存者直接支付下载费用。实际上，这将会和有激励的 BT 下载很类似。服务于平台的指针功能与追踪器类似，可以简化下载。目前我们尚不清楚这个系统是否比现有的 BT 下载有明显的优势。

3.3.4 文件共享

将来，Bridge 可以在应用程序或用户之间启动特定文件的共享。由于所有文件都存在在共享

网络上，这是一个标准化和身份管理的问题。

Bridge 还可以使用第三方身份来源，比如 PGP（按：Pretty Good Privacy）密钥服务器或者 Keybase[21]，以实现安全地个人对个人的文件共享。分层密钥策略（由 LastPass[20]使用）也可以允许共享单个文件。其他的加密方案如代理重新加密也看起来很有前景。对于一个简单的例子来说：如果文件密钥被强加密并托管给一个 Bridge，文件可以共享到任何能通过 Keybase 认证的社交媒体句柄。Bridge 可以向相应的客户端同时发送单个加密的文件密钥和转置密钥，从而可以在不把文件开放给 Bridge 的情况下访问或者以任何方式修改文件。

这些密钥管理方案的详细描述不在本文范围之内，本文只需让读者注意到它们的存在，许多有用的策略可以并行实现，并且意识到专用 Bridge 可以以不同方式促进它们的实现。

3.4 Bridge 作为一个网络信息库

如前所述，数据所有者应负责合约协商和文件状态管理。在有对于网络上个人的足够信息时，合约选择成为维护文件状态的强大工具。一个 Bridge 可以与多个文件储蓄者有多个有效合同，因此可以访问这些储蓄者的信息。Bridge 可以使用这些信息来智能地将分片分布在一组储蓄者中，以达到特定的性能目标。

例如，通过执行合约，一个 Bridge 节点收集有关文件储蓄者通信延迟、审核成功率、审核响应延迟和可用性的数据，通过额外的一点工作，Bridge 还可以收集节点的可用带宽的信息。通过收集关于储蓄者的大量数据，Bridge 节点可以智能地选择一组储蓄者，共同提供一定质量的服务保障。

换句话说，Bridge 可以利用其关于网络上个体的了解来根据客户要求定制服务，可以快速组装一套合约以满足任何服务需求，而不仅仅是一套有限的服务层。这使得客户可以确定文件的延迟、下载带宽和位置，并对实现目标有信心。例如，一个在线视频应用可以指定对高带宽的需求，而存档储蓄只需要高的可用性。放在一个足够大的网络中看，任何需求都能够被满足。

安全的分布式计算是一个未解决的问题，因此每个 Bridge 服务器都使用自己累积的网络知识。Bridge 可以基于其对文件储蓄者的性能和可靠性等知识，来提供一定的服务质量，这在一个单纯的分布式网络中是无法实现的。

3.5 Bridge 作为服务

在委托信用成本并不是很高的情况下，客户可以使用第三方 Bridge，因为 Bridge 并不储存数据而且无法获得密钥，所以这仍是对传统的数据中心模型的一个巨大改进。很多 Bridge 服务器提供的服务，如许可和智能合约，都能带来相当大的网络效应。随着数据量的增加，数据集将呈指数增长，表明 Bridge 在共享基础架构和信息方面有着很强的经济激励。

使用对象储蓄的应用程序将委托大量的信任给储蓄供应者，供应者可以选择运营公共 Bridge 作为服务。应用开发者可以将信任委托给 Bridge，就像传统的对象储蓄一样，但程度稍小。在未来的更新中，我们会允许客户和 Bridge 直接有各种程度的责任分配（随之带来不同的信任水平）。这将把程序开发者的巨大运营负担转嫁到服务供应者身上。这也让开发者可以使用标准化的支付机制来支付储蓄费用，如信用卡，而不需要管理一个加密钱包。Storj Labs Inc. 目前提供这项服务。

4 未来研究的领域

Storj 是一项正在进行的工作，许多功能计划被设计在未来的版本。分布式系统功能在规模方面相对较少例子，许多研究领域仍然开放。

4.1 联合的 Bridges

Bridges 接节点可以合作以共享在互利的联盟中关于网络的数据。 这将允许每个 bridge 通过提高可用信息的质量来提高其提供的服务质量。

bridge 也可以在用户的同意下，合作共享文件元数据和指针。这将允许用户从任何 bridge 访问其文件，而不是依赖于单个 bridge。存储相同访问信息的分层的后备 bridge 是一个理想的特征，因为它解决了单独的 bridge 的停机时间。可能存在一些可解决的许可问题，但是没有理由相信不能开发跨桥的同步状态的标准格式和算法。

4.2 数据可移植性

通过鼓励使用数据格式和访问标准，Storj 旨在允许应用程序之间的数据可移植性。与传统的模式不同，数据的控制与用于访问数据的服务相关联，因为 Storj 形成了一个共同的底层，因此数据访问可能会与个人用户相关联。用户数据可以与持久加密身份相关联，并且在不暴露数据给第三方的情况下进行身份验证。应用中的遍布全球的数据是传统模式的有害遗物。构建数据存储未来的交叉兼容性大大提高了用户的隐私和用户体验。

实现这些标准的应用程序将是广泛兼容的。当访问与用户而不是服务相关联时，隐私和控制将被保留。用户可以授予对备份其硬盘驱动器的服务的访问权限，这些服务将这些文件放置在 Storj 中。用户可以单独授权访问照片共享服务，然后可以访问备份中的任何照片。 用户在许多应用程序中获得数据的无缝可移植性，应用程序开发人员可以访问大量现有用户。

该系统的许可可以由像 bridge 这样的服务来管理，通过诸如 Keybase 之类的服务与信任身份网络相

连，或者由分布式的自治身份系统处理。智能合约系统，例如以太坊[22]合约似乎是一个明智的长期选择，因为他们可以提供基于任意代码执行的文件权限。在管理身份和许可系统所需的私人信息方面可能存在一些问题，但可能存在充分的解决方案。

虽然这个系统在可用性和价值方面都有重大的进步，但是存在着难以解决的安全问题。不幸的是，像在任何加密系统中一样，撤销对数据的访问是不可能的。应用程序可以缓存数据或将其转发给第三方。根据定义，用户信任应用程序开发人员会负责地处理其数据。为了减轻这些风险，Storj 实验室打算为开发人员提供免费和开源软件的激励。没有应用程序可以完全安全，但可审计的代码是用户隐私和安全的最佳防御。

在用户体验和隐私方面的潜在优势是巨大的，但需要更多的研究。在许可机制方面存在许多公开的问题。最糟糕的是，统一的后端激励可互操作应用程序为当前基于数据中心的模型提供了同等的安全性.storj 希望与其他前瞻性数据驱动项目协作，以创建和倡导这些开放标准。

4.3 声誉系统

像许多分布式网络一样，Storj 将从分布式的信誉系统中获益匪浅。在分布式系统上确定信誉的可靠手段是一个未解决的问题。有几种方法已经详细说明，有些方法已经在实践中得到实施，但没有一个在研究人员或工程师之间达成共识。以下简要回顾几种方法。

通过网络分发信息的一个固有的缺点是为了决策额外的潜在延迟。很难说分布式信誉系统是否能够以适合对象存储的方式准确地评估分布式网络上的带宽，延迟或节点的可用性，特别是市场需求随着时间的推移而变化。然而，可靠的分布式声誉将是与网络交互和理解网络非常有用的工具。

4.3.1 IPFS Bitswap

IPFS 提出了 BitSwap 分类帐的概念[23]。BitSwap 分类帐是对过去与其他节点的交互的简单的本地计帐。由于 IPFS 和 BitSwap 协议主要涉及 Kademliastyle DHT 中对象的分发，BitSwap 分类帐计算发送的字节数和接收到的字节数。这些分类帐不是试图达成关于系统声誉状态的全球共识，而是仅处理一

对一的关系，而不考虑延迟，带宽，可用性或其他服务质量因素。节点的大部分行为留给实施者，并对潜在的交换策略进行了一些讨论。这意味着 BitSwap 分类帐规模很大，且非常通用。

4.3.2 Eigentrust 和 Eigentrust ++

Eigentrust [24]试图推广分类帐方法，在使用传递信任模型的分布式系统中生成全球信任值。节点保持和交换信任向量。对于具有大多数可信赖点的网络，每个本地信任向量的值收敛到共享的全球信任向量，因为节点通过信息交换更多地了解网络。

Eigentrust ++ [25]识别几个攻击向量，并修改 Eigentrust，以提高恶意节点存在时的性能和可靠性。Eigentrust ++目前在 NEM 中实施[26]。 将每个节点的全球融合保证为共享信任值是任何分布式信誉系统的核心功能。

4.3.3 TrustDavis

TrustDavis [27]实行保险声誉。节点以保险合约形式为其他节点提供参考。寻求经济交易的潜在合作伙伴的节点还寻求保护合约，保护他们免受该合作伙伴的伤害。该系统中的声望可以被认为是一个图，顶点代表节点，而有向边界代表一个节点愿意代表另一个的货币价值。在此图中远处的节点仍然可以通过购买穿过这些边缘的一组保险合约进行交易。在实践中，TrustDavis 遇到了与闪电网络等其他分布式系统相同的路由问题。

在货币价值方面的信任对于像 Storj 这样的经济网络来说是有吸引力的，但是像这样的系统中的保险合约的机制是一个非常困难的问题。值得注意的是，由于交付付款的失败通过保险路由向后传播，财务负担总是落在信任不可信节点的节点上，而不是不可信赖的节点。

4.3.4 身份维护成本

Storj 是正在开发一个利用公共区块链解决一系列身份问题的声誉系统[28]。该系统要求节点直接花钱来保持声誉。节点通过对自己的 Storj 网络节点 ID 进行小规模标准支付来投资他们的身份。由于 ID 是

节点持有私钥的比特币地址，所以这些资金除了矿工费用外都是完全可以偿还的。在这个系统中，节点选择与具有长期交易历史的节点进行交互。这些时间表明货币投资的标识等于所支付的矿工费用的总和。

参与该系统所需的支付应远远低于运行网络节点的预期回报。如果设置正确，这种身份的经常性货币支付限制了 Sybil 攻击的大小和持续时间，而不会影响协作节点。合法的节点将很容易地收回他们的身份费用，而 Sybil 运营商会发现他们的费用超过他们的回报。不幸的是，这种方法解决了网络上身份问题的一小部分，很难看到如何扩展到其他问题集。

4.4 OFFER 循环策略

许多谈判策略可以存在并通过 OFFER 循环进行交互。谈判策略的全面探讨超出了本文的范围，但有一些有趣的领域是立竿见影的。简单的例子包括价格的下限和上限，但可以建立基于市场趋势和分片的主观价值作为策略的复杂模型。自治机构执行的谈判战略是（迷人）正在进行的研究领域。 Storj 将是第一个大型机器驱动市场之一。 因此，提高谈判效率对于市场的长期效率至关重要。

5.攻击

如同每一个其他的分布式系统，存在一系列的攻击项量。许多这些攻击风险共同存在于所有的分布式系统之中。有些是存储系统特有的，适用于所有的分布式存储系统。