SNIPS：用于去中心化存储系统高效数据同步的简洁存储证明

拉钦·尼高德斯塔
万格大学

海因·梅林斯塔万格大学

arXiv:2304.04891v1 [cs.DC] 10 Apr 2023

摘要—去中心化存储系统中的数据同步对于确保足够的冗余以防止数据丢失至关重要。我们提出了*SNIPS*，这是首个用于同步存储节点的简洁存储证明算法。对等节点为其存储的数据块构造证明，并将其发送给验证者对等节点。验证者查询该证明以识别缺失数据块，随后请求这些数据块。该证明简洁，支持成员查询，并且每个数据块仅需少量比特。

我们在运行 Ethereum Swarm 的一个包含 1000 个对等节点的集群上评估了 SNIPS 算法。结果表明，与最先进的方法相比，SNIPS 将同步数据量减少了三个数量级。此外，生成和验证证明的开销与数据块数量呈线性关系，通常每个数据块仅需数十微秒。这些特性对我们的应用场景至关重要，因为我们设想频繁运行 SNIPS，以持续维持足够的冗余。

索引词—**数据同步、存储证明、去中心化存储系统、维护、冗余、数据维护、集合协调**

I. 引言

去中心化存储系统（例如Ethereum Swarm [1], Filecoin [2], 和星际文件系统（IPFS） [3] ）构建于大型点对点网络之上。网络中的对等节点协作汇集各自的存储容量，以提供一个统一的存储系统。Filecoin和Swarm中的对等节点会获得奖励，以激励它们向网络贡献存储空间和带宽。对等节点彼此并不信任；相反，它们依靠密码学原语来验证所负责存储的数据的完整性。完整性验证基于通过拆分文件获得的固定大小的数据块执行。数据块不可变，并通过其内容地址进行标识；内容地址是该数据块内容的密码学哈希。图 1 展示了将文件上传到去中心化存储系统的过程。

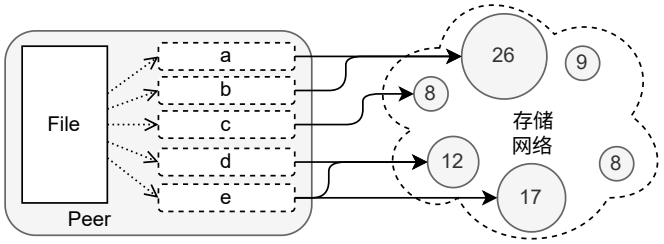


图 1：一个文件被拆分为多个数据块，并分发到大小不同的邻域（圆圈中的数字表示邻域大小）。

在上传期间，每个数据块都会根据其内容地址分发到一个邻域。邻域是由地址相似的对等节点组成的集群。这些相邻对等节点负责存储内容地址与对等节点地址相似的数据块。要下载文件，对等节点同样会尝试定位存储该文件数据块的邻域。如图 1所示，邻域的基数可能各不相同。因此，数据块邻域的大小决定了数据块的复制因子。由于重建文件需要访问其全部数据块，因此拥有足够的冗余至关重要。

近期关于 Ethereum Swarm 和 IPFS 网络动态的研究 [4], [5], [6], [7], [8] 表明，这些系统可能同时连接数万个对等节点。然而，研究还显示，这些系统的节点 churn 率很高，只有极少数对等节点能够连续在线 24 小时。因此，对等节点的邻域不断变化，可能会危及所存储文件的可用性。一项研究表明，数据丢失不可避免，最快可能在短短几天内发生 [4]。因此，存储系统必须维持足够的冗余，以降低数据丢失的可能性。而维持冗余则需要高效的数据同步方法。

当前系统中最先进的数据同步协议效率都很低。在 Swarm 中，该协议称为 Pullsync，它依赖对等节点交换各自存储的数据块标识符的长列表，以便另一对等节点请求缺失分块。为了优化 Pullsync，每个对等节点都会跟踪其邻居上一次的同步状态。然而，正如我们在第 II-C3 节所示，这种优化存在缺陷，可能导致不一致性。Filecoin [2]和 IPFS 使用一种称为 Bitswap [9] 的协议进行数据同步。与 Pullsync 一样，Bitswap 也会交换数据块标识符的长列表，以发现内容并进行同步。Bitswap 仍在积极研究中 [10], [11], [12], ，人们已提出多种优化方案，例如 GraphSync [13] 和 CAR mirror [14]。然而，这些方案仍依赖于交换数据块标识符的长列表。

此前关于点对点网络中数据同步的工作 [15], [16], [17] 提出了使用布隆过滤器 [18] 进行近似协调的协议。顾名思义，近似协调无法保证所有数据块都已同步。如果文件经过纠删码编码 [19]，只需该文件的一部分数据块即可重建文件。然而，在基于 Merkle 的存储系统中加入纠删码】【。

树并非易事，因为内容与元数据之间存在层级依赖关系 [20]。因此，我们关注所有数据块都必须同步的情况。

对等节点确保完全同步的一种节省空间的方法是使用存储证明（PoS）算法 [21]。PoS 算法让对等节点无需传输数据本身，就能相互证明自己存储了哪些数据。这类算法通常有三种不同的参与方：发起 PoS 查询的挑战者；通过生成证明来响应查询的证明者；以及验证证明的验证者。然而，验证通常只有完全失败或通过两种结果，不存在部分验证。即使某个对等节点拥有部分数据块，验证仍会失败，因为它没有全部数据块。此外，要同步数据，验证者不仅需要对证明进行部分验证，还需要确定缺失的是哪些数据块。采用基于 PoS 的朴素部分验证方法，为每个数据块生成一个证明，其通信开销将与直接交换数据块标识符相当。

A. 贡献

本文提出了 SNIPS，一种用于去中心化存储系统中高效数据同步的新型协议。SNIPS 使用一种新颖的类 PoS 构造来生成简洁的存储证明。这些证明很小，通常每个数据块仅需几比特，并且可以高效地生成和验证。我们的 PoS 构造是非交互式的，使验证者能够利用存储证明，而无需经过 PoS 算法中通常使用的前置 挑战 阶段。如果证明者和验证者拥有完全相同的数据块集合，那么双方都会立即确信对方的状态。此外，即使证明者和验证者拥有不相交的数据块集合，我们的构造也允许验证者对证明执行成员查询，以确定缺失了哪些数据块。对于某些不属于证明者证明的本地数据块，验证者可能会观察到假阳性。不过，我们的 SNIPS 协议可以协调这些假阳性。

使用SNIPS 协议的对等节点生成存储证明，并与其邻居交换这些证明。收到存储证明后，对等节点可以查询该证明，以识别其缺失的任何数据块，并向发送方请求缺失的数据块。通过准确性不断提高的证明迭代地消除假阳性。总之，SNIPS 证明提供两项功能；(1) 使接收方相信发送方拥有数据；(2) 接收方可以查询该证明，以确定其缺失哪些数据块。

我们在 Ethereum Swarm 中实现的 SNIPS 表明，它是一种可用于去中心化存储系统的实用协议。评估结果显示，与当前系统相比，传输的同步数据最多可减少三个数量级，从而显著节省带宽。此外，生成和验证证明通常每个数据块只需几十微秒。虽然我们的评估未与 Bitswap 及其衍生协议进行比较，但 SNIPS 的方法可与这些协议互补，我们预计 Bitswap 也能从这种方法中受益。

我们的贡献总结如下：

- 一种新的类 PoS 构造，用于创建适合部分验证并识别缺失数据块的存储证明。
- SNIPS：一种用于去中心化存储系统的数据同步协议，以及其在 Ethereum Swarm 中的实现。
- 对带宽使用量和同步开销的严谨模拟，以及在真实世界集群上对 SNIPS 的性能评估。

II. 预备知识

我们假设有一个内容寻址的去中心化存储系统 [3], [1], 构建于点对点网络之上，其中每个存储节点都与网络中的一部分对等节点相连。也就是说，对等节点被划分为若干小型邻域，这些邻域共同负责持久化地址空间中特定区域的数据块。

我们不信任存储对等节点，对等节点之间的网络可能会经历显著的节点 churn。我们依赖存储系统底层的网络在对等节点之间路由消息。

每个存储节点都可以使用密码学密钥对来签署和验证数字签名，并且我们假设密码学原语无法轻易被规避。令 $\langle m \rangle_p$ 表示一条由对等节点*p*签名的消息 m 。

对等节点的公钥还用于生成一个唯一地址，以便在覆盖网络中使用。该地址作为覆盖网络连通图的基准，使得对等节点更有可能与地址相似的其他对等节点建立连接。

在我们的算法中，我们假设所有对等节点都可以访问共享随机源。获取共享随机源的一种方法是使用公有区块链中未来区块的区块哈希 [22]。其他方法包括多方计算 [23], Shamir的秘密共享 [24], 可验证延迟函数 [25],或可验证随机函数 [26]。

我们定义以下指标来评估 SNIPS。

定义 *1*（相似度）：两个集合*A*和*B*之间的相似度为重叠系数 [27]; $|A \cap B|/\min(|A|, |B|)$。

定义 *2*（证明准确率）：设 M_i 为查询证明时识别出的缺失数据块集合。设 M_p 为对等节点在证明中缺失的数据块集合。因此，证明准确率为 $|M_i|/|M_p|$。

A. 存储证明模型

在我们的 PoS 模型中，只有证明者和验证者，也称为存储节点。我们类似 PoS 的构造使对等节点能够 (i) 高效同步数据块，以及 (ii) 与邻居验证其所存储数据块的完整性。我们假设对等节点拥有其存储的数据块的完整副本。

我们的构造无法向客户端提供存储保证，因为我们不依赖挑战阶段。我们假设客户端将其数据上传到存储系统。此外，去中心化存储系统中的数据块会被独立地存储在不同的对等节点上，而不考虑其在文件中的位置。因此，我们的

PoS 方案不会将数据块绑定到文件内的特定位置，因为我们的使用场景并不需要这样做。其他协议可以为客户端提供这些保证 [4], [21]。

B. 威胁模型

我们考虑内部攻击者（对等节点）可能对 SNIPS 发起的威胁。我们不考虑没有凭证的外部攻击者，因为存储对等节点会拒绝此类攻击者发送的消息。本文也不考虑对等节点串通；缓解串通攻击需要其他机制（见第 VII-B 节）。拒绝服务攻击不在本文讨论范围内，因为底层存储系统应当能够缓解此类攻击。我们重点关注旨在破坏存储对等节点所存储数据完整性的攻击形式：

重放攻击。我们考虑试图重放协议消息以触发非法行为的对等节点。

上传攻击。收到缺失数据块的请求后，攻击者可能会尝试上传不同的数据块。

污染攻击。恶意对等节点可能会尝试通过创建和分发无效数据块来污染存储系统。

不可抵赖攻击。 SNIPS 证明可以被视为对同步所包含数据块的承诺。我们考虑一种不可抵赖攻击，其中攻击者试图在分发证明后修改所包含的数据块。

此外，攻击者可能会尝试构造一个会导致虚假一致性的存储证明（见第III-D 节）。第 IV 节-E 讨论了这些威胁及其缓解措施。

C. Swarm

Swarm [1] 是一个全球去中心化存储和通信系统，将存储的数据分发到由对等节点组成的网络中。Swarm 的点对点覆盖网络基于 Kademlia [28]，拥有超过 2000 个活跃对等节点 [29]。

Swarm 中的每个对等节点都会向EVM 兼容的区块链（例如 Ethereum 或Gnosis）部署一个名为支票簿（checkbook）的智能合约。当对等节点为网络贡献资源时，该合约用于通过 BZZ 代币奖励对等节点。具体而言，对等节点付费下载数据块，并因交付数据块和转发消息而获得奖励。部署该合约后，每个对等节点都会生成一个唯一的对等节点地址，用于在网络中标识自身。该地址通过对以下内容的拼接结果进行哈希计算生成：对等节点的Ethereum公钥、网络标识符，以及紧接在部署对等节点支票簿合约的区块之后的区块的哈希。随后，该对等节点会被置于共享最长前缀的邻域中

以及对等节点的地址。

1) Swarm 中的数据存储： Swarm 将文件拆分为 4 KB 数据块。通过对每个数据块的内容进行加密哈希运算，为其生成唯一的 数据块标识符 ，也称为其内容地址。Swarm 创建一棵 128 叉默克尔树，其中文件的每个数据块都是一个叶节点。Merkle 树的内部节点和根节点也存储在 4 KB 数据块中，并包含其子节点的数据块标识符串联结果。

数据块标识符和对等节点地址共用同一个地址空间。当数据块上传到网络时，会根据其标识符 [30] 将其发送到距离最近的对等节点。该数据块随后由距离最近的对等节点的邻域中的每个对等节点进行复制。图 2 展示了在包含 1000 个对等节点的集群中测得的邻域大小分布。我们观察到 81 个大小各异的邻域（n），其大小为 $n \in [8, 26]$。由于邻域中的所有对等节点都会复制数据块，邻域大小的差异会导致数据块的复制因子也相应变化。

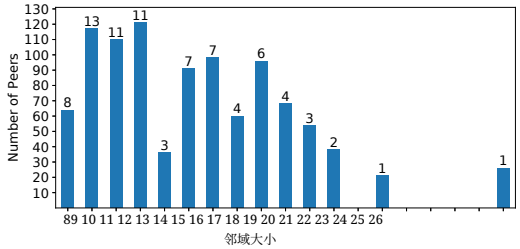


图 2：我们的集群中邻域大小的分布。

2) Swarm 的 *Pullsync* 协议： Swarm 的 Pull- sync 协议 [1], [30] 旨在同步相邻对等节点之间的数据块。尽管存在节点 churn，Pullsync 仍旨在确保邻域内对等节点之间最终达到一致。然而，Pullsync 不会清理或验证所存储数据的完整性。图 3 概述了该协议；以下是对其步骤的简化描述。

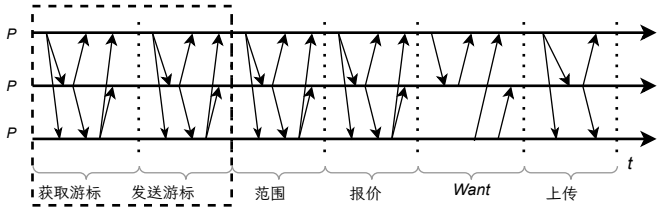


图 3：Swarm 的 Pullsync 协议概览。

邻域内的所有对等节点彼此运行Pullsync。每个对等节点都会监控网络中的拓扑变化，例如对等节点加入或离开，并在发生此类事件时触发Pullsync 协议。考虑响应拓扑变化的两个对等节点 P_1 和 P_2。

对等节点 P_1 向其相邻对等节点发送请求，以获取其邻域中存储的数据块数量（游标）。根据收到的游标， P_1 确定自己是否缺少任何数据块，以及哪个对等节点拥有它可能需要的数据块。

收到一系列数据块的请求后，对等节点 P_2会创建一个提供请求，其中包含该范围内的数据块标识符数组。当 P_1 收到提供请求后，会将其缺失的数据块标识符对应的位置设为 1，其余位置设为 0，从而填充位向量。随后，位向量会发送回 P_2，后者将获取缺失分块并将其传送给 P_1，从而完成同步过程。

3)Pullsync 中的不一致性： 通过检查图 3 中的前三个阶段，可以解释Pullsync 的不一致性。获取游标 和发送游标仅在两个对等节点首次同步时发送。此后，每个对等节点

将把另一方的游标保存在内存中，并将这些用于范围 阶段。

当对等节点删除数据块时，就会出现这个问题。删除数据块可能是由恶意行为或 Swarm Bee 客户端中的错误引起的。不过，删除也可能完全无害，例如由垃圾回收或激励层机制导致 [31], [1]。此外，Swarm Bee 客户端还向用户提供了用于删除数据块的 API。

当对等节点删除数据块时，除非被删除的是该节点最后添加的数据块，否则内部游标不会更新。因此，无论该对等节点运行多少次 Pullsync 协议，都不会同步已删除的数据块。这会导致邻域中存储的数据块出现不一致性，并最终可能造成数据块丢失。

D. 最小完美哈希函数

我们在第 III 节中描述的增强型 PoS 构造使用最小完美哈希函数（MPHF） [32], [33] 生成存储证明。MPHF 是从由 N个元素（键）组成的集合到整数 $[1, N]$ （索引值）的双射。集合中的每个键都映射到唯一一个值，且每个值都与唯一一个键对应。图 4 展示了这种映射。

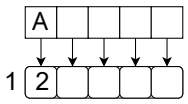


图 4：最小完美哈希函数双射映射。

最小完美哈希函数允许构建一个节省空间、无冲突的巨大数据集映射，最多包含 10^{12} 个键。虽然理论极限低至每个键 1.44 位，但 [34]，实际实现通常需要每个键几位。最小完美哈希函数 предназначендля静态数据集，不提供添加、更新或删除元素的操作。也就是说，键集合必须是静态的，并且预先已知。预期查找时间为 $O(1)$，并且可以在 $O(N)$ 时间内构建映射。

查询映射中属于原始集合的键会返回范围 $v \in [1, N]$ 内的相应值。然而，如果该键不属于原始集合，则可能返回范围 $v \in [0, N]$ 内的任意值。当返回值为 0时，该键不属于原始集合；但是，任何其他值 $v \in [1, N]$ 都可能是假阳性。我们为最小完美哈希函数定义一个查找函数：

$$\mathrm{Find}(elm) : \begin{cases} 0 & elm \text{ definitely not in the set} \\ v \in [1, N] & elm \text{ might be in the set} \end{cases}$$

III. 增强型存储证明构造

现在，我们可以介绍增强型非交互式 PoS 构造了。该构造生成的证明具有以下特性：(1) 它能使接收方确信发送方拥有这些数据；(2) 接收方可以查询证明，了解缺少哪些元素。该构造由对等节点的 证明者 和验证者 执行的两个算法组成。

A. 证明者算法

证明者使用算法 1 中的 CreateProof 函数，为其存储的数据块集合 $\mathcal{CP}$ 生成存储证明 。存储证明分两步构建。

1) 为每个存储的数据块创建持有证明： 证明者使用公式（1）为每个数据块 $a \in \mathcal{CP}$计算持有证明，如算法 1 第 7-10 行所示。我们将此证明称为数据块证明 [4]。

$$\mathrm{ChunkProof}_a : \quad cp_a = \mathrm{H}(nonce \mid\mid a) \tag{1}$$

数据块证明是将随机数（只使用一次的数字）与数据块的数据拼接后得到的密码学哈希。只要随机数不可预测，证明者就无法声称存储了实际上并未存储的数据块，因为生成证明时必须同时拥有数据块和随机数。因此，我们按照第 II 节中的讨论，从共享随机源中推导随机数。由此，随机数体现了数据块证明的新近性。我们在第 IV-D 节讨论证明生成的频率。作为一种优化，证明者可以针对有限范围 [起始，结束] 生成存储证明。不过，在我们的评估中，我们使用了全部数据块范围。

2) 将持有证明压缩为单个存储证明：证明者将数据块证明压缩为简洁的存储证明。我们通过构造一个最小完美哈希函数（第 11 行）来实现这一点，使每个数据块证明 *cp* 都映射到唯一的索引$idx \in [1, N]$，其中 N是数据块的数量。

最后，在第 12–14 行，我们建立从最小完美哈希函数索引到数据块标识符的反向映射。由于最小完美哈希函数是根据相同的数据块证明构造的，因此 Find 始终会返回有效映射。证明者可以利用此反向映射，确定验证者请求的缺失分块。

$$\begin{aligned} &\textit{index-id} : \textit{idx} \mapsto \textit{chunk.id} && \text{Reverse map} \\ &\textit{mphf} : \textit{cp} \mapsto \textit{idx} \end{aligned}$$

算法 1 证明者：存储证明生成

```
1: 证明者的本地持久状态:
2:   CP                                          ▷证明者存储的数据块集合
3:   索引标识符                                   ▷反向映射：最小完美哈希函数索引到分块标识符
4: 函数 CreateProof(随机数，起始位置，结束位置)
5:   分块证明 ← { }
6:   cp-id ← ⟨ ⟩                                 ▷临时映射：分块证明到分块标识符
7:   对于每个 数据块 ∈ CP： 块标识 ∈ [ 开始、结束 ] 执行
8:     cp ← H(随机数 || 数据块)                    ▷数据块证明
9:     分块证明 ← 分块证明 ∪ cp
10:    cp-id[cp] ← 块标识
11:  最小完美哈希函数 ←新 最小完美哈希函数(分块证明)
12:  对于每个 cp ∈ 分块证明 执行                   ▷填充反向映射
13:    idx ←最小完美哈希函数.查找 (cp)             ▷ 证明中的 cp 索引
14:    索引-id[随机数][idx] ← cp-id[cp]            ▷保存 cp 的分块标识符
15:  返回 [ 最小完美哈希函数, 随机数, 开始, 结束 ]  ▷存储证明
```

B. 验证者算法

接下来，我们将说明验证者如何使用证明者提供的存储证明，通过算法 2 所示的查找缺失块函数，识别验证者可能缺失的数据块。

给定一个存储证明，验证者会为证明范围内的每个本地数据块 $\in \mathcal{CV}$ 计算数据块证明。随后，使用这些数据块证明通过查找函数（第 8 行）查询由证明者生成的最小完美哈希函数。因此，如果数据块证明存在于最小完美哈希函数证明中，查找函数将返回一个索引 $\in [1, N]$；否则，将返回索引 $= 0$。我们使用索引作为*mfc* 数组的索引。在第 10 行，验证者将*mfc*中的第索引个条目递减，以表明本地数据块 $\in$具有证明。验证者处理完所有数据块证明后，我们使用*mfc* 按如下方式识别缺失分块和冲突：

$$mfc[idx] \begin{cases} = 1 & \text{verifier } \mathbf{m}\text{issing chunk for } idx \\ = 0 & \text{verifier } \mathbf{f}\text{ound chunk locally} \\ < 0 & \text{verifier has } \mathbf{c}\text{ollision(s) for } idx \end{cases}$$

也就是说，*mfc* 中值仍等于 1 的条目必然缺失（第 15 行），我们将其添加到缺失 集合中。值等于0 的条目并未缺失，无需进一步处理。然而，对于值为负数的条目（第 17 行），验证者发现有两个或更多分块证明映射到了同一个索引值，即发生了冲突。我们将在第 III-C 节讨论这一问题。

必须认识到，验证者无法以数据块标识符或数据块证明的形式指定缺失数据块。相反，缺失集合包含索引值，证明者可以利用这些值在索引-标识符反向映射中查找缺失数据块的标识符。请回忆，证明者在算法 1 的第 12–14 行构建了索引-标识符。

算法 2 验证者：查找缺失分块

```
1: 验证者处的本地持久状态:
2:   CV                                          ▷验证者存储的数据块集合
3: 函数查找缺失块(证明)
4:   大小 ← | proof.mphf |                       ▷证明中的数据块数量
5:   mfc ← [1, ..., 1]^size                     ▷最初，所有数据块均缺失
6:   对于每个 数据块 ∈ CV : 块标识 ∈ [ 证明.start, 证明.end ] 执行
7:     cp ← H(证明.随机数 || 数据块)
8:     idx ← 证明.最小完美哈希函数.查找(cp)        ▷证明中的cp 索引
9:     若索引 ≠ 0 则                              ▷可能在该集合中
10:      mfc[idx] ← mfc[idx] − 1                 ▷发现数据块或冲突
11:  缺少 ← { }
12:  冲突 ← false
13:  对于 i ← [1, 大小] do
14:    如果 mfc[i] = 1 则
15:      缺失 ← 缺失 ∪ i                         ▷缺失的数据块
16:    否则如果 mfc[i] < 0 那么
17:      冲突 ← true                             ▷发现冲突
18:  返回 [ 缺失，冲突 ]                         ▷缺失分块
```

C. 冲突

本节说明验证证明时为何会发生冲突。回想一下，最小完美哈希函数的查找函数可能会产生假阳性：当一个不属于原始集合的数据块返回“可能在集合中”时，就可能发生这种情况。形式化地说：

定义 *3*（假阳性）：令 $\mathcal{CP}$ 为用于构建存储的 数据块证明 并令 $cp \notin \mathcal{CP}$。那么，如果 Find(cp) > 0，就会出现假阳性。

鉴于最小完美哈希函数可能产生假阳性，也可能发生冲突。当多个数据块映射到证明中的相同索引值。发生冲突意味着至少存在一个假阳性。形式化地：

定义 *4*（冲突）：令 $\mathcal{CP}$ 为用于构造存储证明的数据块集合，并令 $cp_a \in \mathcal{CP}$ 和$cp_b \notin \mathcal{CP}$，使得 $cp_a \neq cp_b$。则当以下条件成立时，存在冲突：

$$\text{Find}(cp_a) = \text{Find}(cp_b)$$

证明者和验证者分别独立地从 $\mathcal{CP}$ 和 $\mathcal{CV}$ 构建各自的数据块证明。由于 $\mathcal{CV}$ 可能不同于 $\mathcal{CP}$，且验证者使用来自 $\mathcal{CV}$ 的数据块证明来查询证明者的存储证明，如果两个或更多数据块证明映射到相同的索引值，验证者就可能观察到冲突（算法 2 第 8 行）。

我们模拟了一个单独的对等节点验证不同大小的证明，以确定出现假阳性的概率。正如我们从图5中看到的，证明中的数据块数量越多，观察到假阳性的概率就越高。给定一个包含 10^8 数据块的证明，观察到假阳性的概率为 99.78 %。尽管可能出现假阳性和冲突，但其影响是有限的，因为SNIPS 采用了反复生成证明的策略，直到发现所有缺失分块为止。我们在第 IV 节中描述了该策略，并在第VI-D节中对其进行了评估。事实上，我们推测，即使证明出现假阳性的概率为 100 %，SNIPS 仍然能够发现所有缺失分块。我们的理由是，即使存在假阳性，仍然有可能发现缺失分块——除非这些假阳性导致虚假一致性，这一点我们将在下文讨论。

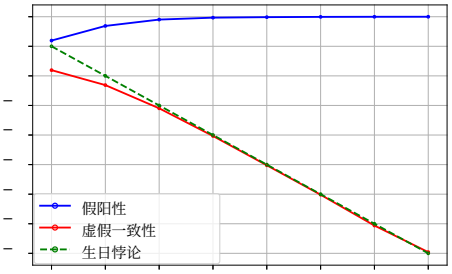


图 5：证明中数据块数量与假阳性和虚假一致性概率的关系。绿色虚线表示对虚假一致性的估计。

D. 虚假一致性

正如我们在第 III-C 节中所解释的，如果两个或更多分块证明映射到相同的索引值，验证者对等节点可能会观察到冲突。虽然此类冲突不太可能发生，但很容易检测并恢复。本节将讨论<style id='1'>虚假一致性——一个相关问题，只有在相似度低于 1 时才可能发生，即两个对等节点各自拥有对方没有的数据块。

定义 *5*（虚假一致性）：对等节点在查询证明后认为自己没有缺失分块。然而，至少有一个用于查询证明的数据块产生了假阳性，

且该假阳性的值恰好与最后一个剩余的预期值相匹配。

为解释虚假一致性，考虑两个对等节点 P_1 和 P_2之间的一次证明交换。对等节点 P_1 存储数据块 $\{a, b, c\}$，而对等节点 P_2 存储数据块 $\{a, b, d\}$，其中数据块 $c \neq d$。当对等节点 P_1 和 P_2 评估彼此的证明 以识别缺失分块时，可能会出现双方都得到假阳性的情况，从而产生两个不同的映射，如下所示：

$$P_1 : [a \mapsto 3, b \mapsto 1, d \mapsto 2]$$
$$P_2 : [a \mapsto 1, b \mapsto 2, c \mapsto 3]$$

在这种情况下，不存在冲突映射，也不存在映射到 0 的情况。也就是说， P_1 认为自己拥有存储证明 中的所有数据块；该证明由对等节点 P_2 发送，反之亦然。因此，两个对等节点都错误地认为彼此拥有相同的数据块集合。

图 5 展示了模拟单个对等节点验证不同大小的证明所得的结果，用于确定出现虚假一致性的概率。随着证明中数据块数量增加，出现虚假一致性的概率呈线性下降；当数据块数量较大时，该概率可忽略不计。例如，对于包含 1000 个数据块的证明（相当于对等节点仅存储 4 兆字节），观察到虚假一致性的概率低于 0.09%。

幸运的是，随着数据块数量增加，在实践中观察到虚假一致性的可能性很低。通过为每个证明使用不同的随机数，虚假一致性极有可能在下一次证明交换中得到解决。

我们可以使用生日悖论 [35], 对虚假一致性的概率进行建模，该概率在图 5 中以绿色线条表示。生日悖论是一种经典技术，用于确定集合中随机选择的两个或更多元素发生冲突的概率。在我们的案例中，我们令 N 表示证明中所有可能索引值的集合基数，并令 k表示交换证明的对等节点数量。该方程在方程（2）中给出如下：

$$P(c) = 1 - \frac{N!}{(N-k)!}\frac{1}{N^k} \tag{2}$$

由于只有2 个对等节点在交换证明，我们可以简化为表达式 $P(c)|_{k=2} = 1/N$。

IV. SNIPS 协议

本节介绍 SNIPS，这是一种用于去中心化存储系统的数据同步协议。SNIPS 允许存储对等节点定期检查缺失分块，带来的带宽开销可忽略不计，且无需事先进行挑战阶段。

SNIPS 提供了一种机制，使对等节点能够交换存储证明，并逐步提高同步准确性。与第 III 节中的 PoS 构造结合使用时，我们可以

在存储证明中为每个数据块使用更少的比特数。因此，SNIPS有助于在存储证明的大小与同步对等节点所需的迭代次数之间取得平衡。

我们首先概述 SNIPS 的三个阶段，如图 6 所示。虽然图 6 展示的不同阶段看起来像是同步进行的，但 SNIPS 完全以异步方式运行。每个对等节点都可以按照自己的节奏推进。

在证明 阶段，每个对等节点都会为其数据块构建存储证明。对等节点 P_1 构建证明 $S_{a,b,c}$，以表明其存储了数据块 $\{a, b, c\}$。同理， P_2构建$S_{b,c,d}$，而 P_3 构建 $S_{b,c}$。对等节点 P_1、 P_2和 P_3 将各自的存储证明发送给另外两个节点。

收到来自 P_2 的存储证明 $S_{b,\ c,\ d}$ 后， P_1 检测到缺少数据块 $\{d\}$，并在 选择 阶段通过R_d 消息向 P_2 请求该数据块。同样， P_2 检测到缺少数据块 $\{a\}$，并向 P_1 请求该数据块。以同样的方式，P_3 检测到缺少数据块 $\{a, d\}$，并分别向 P_1 和 P_2 请求这些数据块。最后，在 上传 阶段上传请求的数据块，恢复存储冗余。接下来，我们将更详细地解释各个阶段。

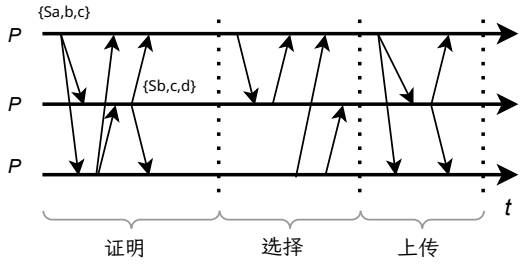


图 6：SNIPS 各阶段的时空示意图。

A. 证明阶段：存储证明生成

在证明阶段（如算法 3 所示），每个对等节点使用算法 1 中的 CreateProof 函数，为其存储的数据块集合构造存储证明。随后对该证明进行数字签名，以保持其完整性并将其与证明者的公钥关联起来。

证明阶段通过 NEWPROOF 消息以两种方式触发：由本地对等节点定期触发（见第 IV 节-D），或由另一个对等节点触发，以便快速从冲突中恢复（见算法4中的第 8 行）。

算法 3 证明阶段：存储证明生成

```
1: 收到 ⟨ NewProof | 随机数，开始，结束 ⟩ 执行
2:     证明 ← CreateProof(随机数、起始位置、结束位置)
3:     发送 ⟨ 证明 | 证明 ⟩ p                    由对等节点 ▷ 签署p
```

B. 选择阶段：请求缺失分块

在选择阶段（如算法4所示），存储节点预期会定期收到其他对等节点发送的 PROVE 消息。收到存储证明后，该节点会调用查找缺失块函数，以识别可能缺失的数据块。如果该节点检测到缺失分块，则可以向生成相应存储证明的对等节点发送 SELECT 消息来请求这些数据块。

如第 III-B 节所述，评估存储证明可能会导致冲突。冲突标志表示存在冲突。因此，为解决冲突，我们会触发一个新的证明阶段（第 8 行）。不过，我们会等到收到对等节点请求的数据块后再进行，这由 UPLOADDONE 消息指示。根据需要同步的对等节点邻域的存储大小

以及证明者和验证者之间的相似度，可能需要多次迭代才能解决所有冲突。

由于验证者只知道证明者缺失的分块索引，我们可能会从多个对等节点请求同一个数据块。因此，我们按顺序处理来自不同对等节点的证明，以避免重复上传数据块。然而，我们会并发地从同一个对等节点请求数据块。收到数据块后，我们会根据所有证明对其进行评估。

算法4 选择阶段：请求缺失分块

```
1: 收到 ⟨ 证明 | 证明 ⟩_p 执行                    来自对等节点p的已验证证明
2:     [ 缺失, 冲突 ] ← 查找缺失块(证明)
3:     如果缺失 = ∅ 则
4:         返回                                  ▷没有缺失内容
5:     回复 ⟨ 选择 | proof.nonce, 缺失 ⟩          ▷请求数据块
6:     如果冲突 则
7:         收到时 ⟨ 上传完成 ⟩ 执行
8:             回复 ⟨ 新证明: 证明 | proof.nonce, proof.start, proof.end ⟩
```

C. 上传阶段：发送缺失分块

算法 5 所示的上传阶段很简单。在此阶段，证明者通过上传缺失分块给验证者，以响应 SELECT 消息。

创建存储证明的证明者对等节点会存储与该存储证明关联的随机数的索引-标识符 反向映射。因此，为了便于上传，证明者会使用索引-标识符 映射。对于请求的缺失集合中的每个索引值，我们会在反向映射中查找对应的数据块标识符。然后，使用该数据块标识符在本地存储中查找数据块，$\mathcal{CP}$，并将该数据块发送给请求的验证者对等节点。最后，UPLOADDONE 消息向验证者对等节点表明上传已完成。

算法 5 上传阶段：发送缺失分块

```
1: 证明者的本地持久状态:
2: CP                                  ▷证明者存储的数据块集合
3: 索引-id                             ▷反向映射：最小完美哈希函数值到分块标识符
4: 收到 ⟨ SELECT | 随机数, 缺失 ⟩ 执行
5:     对于每个idx ∈ 缺失的执行
6:         数据块标识 ← 索引-标识[随机数][索引]
7:         数据块 ← {c : c ∈ CP ∧ c.标识 = 数据块标识}
8:         回复 ⟨ 上传 | 数据块 ⟩
9:     回复 ⟨ 上传完成 ⟩
```

D. 证明生成频率

我们简要探讨如何限制 SNIPS 的执行频率。例如，在去中心化存储系统中，我们设想 SNIPS 大约每天运行一次。

为此，我们可以配置协议执行，使其遵循公有区块链的区块生成频率。例如，每 k个区块，就从区块链提取一个新的随机数，以触发证明阶段。该随机数可以是区块链当前区块头的哈希。由于区块会定期生成，我们可以将协议执行限制在所需的频率。例如，以太坊的平均出块时间约为 12 秒 [36], [37]。因此，若要每天生成一次存储证明，我们可以每 7200 个区块触发一次执行（$24\text{h} \cdot 60\text{m} \cdot 60\text{s}/12\text{s}$）。

让对等节点针对同一个随机数按相同的时间间隔生成证明，可以摊销证明生成的成本。我们将把探索对等节点如何协调证明生成以节省成本留待未来工作。

E. 安全性分析与缓解策略

本节对 SNIPS进行了初步的安全分析，并提出了一些缓解措施。我们注意到，底层存储系统已经缓解了若干攻击向量。例如，本文不考虑遗漏和搭便车，因为底层存储系统应能缓解这些攻击。还需要开展更多工作，以了解 SNIPS 是否会为此类攻击暴露出额外的攻击向量，以及如何缓解这些攻击。下面我们将分析第 II-B 节中概述的存储节点完整性攻击。

重放攻击。 攻击者可以通过多次发送相同的Prove或 Select 消息来发动重放攻击。对等节点可能会处理这些重放消息，而不会影响存储节点的完整性。然而，由于重复工作和增加的网络流量，此类攻击会影响协议的性能。我们注意到，处理 NewProof和 Prove消息的成本高于处理 Select消息（见图 9a）。因此，我们也可以依赖存储系统对对等节点进行速率限制的机制。

一种替代的缓解策略是根据随机数和对等节点地址丢弃重放的消息。采用此策略后，对等节点无法针对相同的随机数重新传输消息，而必须等待下一次同步轮次。

上传攻击。 行为不端的对等节点可能会上传与选择阶段请求的数据块不同的数据块。为缓解这一问题，接收方对等节点首先检查上传的数据块是否属于其邻域。接着，该对等节点检查上传的数据块是否映射到先前从上传方对等节点收到的存储证明中的预期索引值。

例如，考虑图6 中的示例： P_2发送了一个 R_a 请求，随后从 P_1接收到了数据块 a 。因此， P_2 可以检查 a 是否在 P_1发送的原始存储证明 $S_{a,b,c}$ 中映射到 R_a 。如果映射不正确，则 P_2会拒绝该数据块。如果 P_1 发送了$S_{a,b,c}$已经拥有的其他数据块之一，那么 P_2 也会拒绝该数据块。

污染攻击。 行为不端的对等节点可能会尝试通过创建无效数据块来污染存储系统，例如并非由客户端上传的数据块。攻击对等节点可能会将无效数据块包含在存储证明中，并寄希望于其他对等节点请求这些数据块。SNIPS 无法阻止此类攻击；我们将其留给底层存储系统来缓解。一种简单的缓解策略是要求对等节点在每次上传时嵌入客户端的签名。

不可抵赖攻击。 当对等节点分发存储证明时，它会对证明中的数据块作出承诺。然而，攻击者可能会试图利用证明阶段发送的证明中的冲突，诱骗其他对等节点接受未经验证的数据块。

考虑一个包含两个对等节点 P_1 和 P_2的场景，其中 P_1 是攻击者。让 P_1 拥有数据块 $\{a, b\}$，而 P_2 拥有 $\{a\}$，但没有 b。在协议开始时， P_1 创建一个存储

证明 $S_{a,b}$ 并将其发送给 P_2。接下来，P_2 向 P_1发送请求 R_b，请求获取数据块 b。现在假设 P_1 能够找到一个特洛伊数据块 $b' \neq b$，使得 Find($cp_{b'}$) =Find(cp_b)。最后，P_1可以上传 b'，而 P_2 会将 b' 接受为有效数据块，以替代 b，尽管 b' 并未包含在 $S_{a,b}$中。

我们认为，对于规模通常的存储系统而言，这种攻击在实践中很难实施。首先，攻击者必须找到一个与 b 发生碰撞的候选数据块 b'。其次，b' 必须与 b 具有相同的数据块标识符前缀；否则，b' 将属于不同的邻域，而 P_2 将不会接受它。这需要对每个候选 b' 进行哈希运算，直到生成的数据块标识符具有相同的前缀。然后，攻击者必须按照第 III-A 节所述，计算 b' 的数据块证明，随后确定它是否与 b 发生碰撞。如果数据块证明未发生碰撞，攻击者就会转向下一个候选项 b'。

给定 l 位的前缀长度和 n 个数据块的证明，找到一个同时具有相同数据块标识符前缀且与 b 发生碰撞的数据块 b' 的概率为 $1/(2^l n)$。因此，给定 16 位前缀和 100 万个数据块的证明，找到特洛伊数据块的概率为 $1.53 \cdot 10^{-11}$。

为进一步缓解不可抵赖攻击，例如针对规模较小的存储系统，我们可以在存储证明中加入“证明校验和”。证明校验和是存储证明中各数据块证明的哈希，即 H($cp_1 \parallel cp_2 \parallel \dots$)。

我们简要指出，攻击者也可能试图利用虚假一致性的可能性。然而，这种攻击至少会与不可抵赖攻击一样昂贵。

V. 实现

我们将 SNIPS 实现为 Swarm Bee v1.9.0 中的一个软件包。该实现包含约 800 行 Go 代码，以及约 1900 行用于基准测试、测试和指标收集的代码。我们的实现基于第 IV 节所述的协议。对于最小完美哈希函数，我们采用了 [38] 中的实现，并使用了推荐的扩展因子 $\gamma = 2$。

本节将介绍协议的实现细节。我们首先介绍证明者和验证者使用的协议消息与数据结构。接着，我们将说明证明者和验证者的实现方式，包括一些优化。最后，我们将介绍评估框架的若干方面。

A. 协议消息

我们使用 protocolbuffers [39] 定义了 SNIPS 的四种消息。首先，证明消息包含一个 8 字节的随机数、两个 32 字节的起始和结束字段、一个 4 字节的条目（用于记录证明的长度），以及一个长度可变的条目，用于存放 MPHF 证明本身。此外，证明还经过签名，因此会额外增加 97 字节的 Swarm 专用签名。该签名使对等节点能够验证证明的创建者是否位于同一邻域。

Select 消息仅包含一个 8 字节的随机数和一个位向量。位向量使我们能够在同一条消息中高效地选择多个数据块。

NewProof 消息仅包含一个 8 字节的随机数。这与第 IV 节所述的协议有所不同；该协议中的 NewProof 消息还包含证明的起始位置和结束位置。这是因为在我们的实现中，证明者会跟踪起始位置和结束位置，因此无需将它们包含在消息中。最后，UploadDone 消息仅用作信号，不包含任何数据。

B. 证明者

在某个时刻，网络中的每个对等节点都会充当证明者。因此，对等节点一启动，就会开始监听添加到区块链中的新区块。

启动证明生成的另一个触发条件是对等节点收到 NewProof 消息。在这种情况下，对等节点会检查 *id-proof* 缓存中是否已存在相同随机数对应的证明。如果存在，对等节点就会从 *id-proof* 缓存中获取该证明，而不是重新生成。因此，通过缓存数据块证明，我们可以分摊生成数据块证明的成本。如果对等节点针对相同随机数在同一间隔上发起证明（例如每 7200 个区块一次），这种做法尤其有用。

C. 验证者

验证者使用与证明者相同的 *id-proof* 缓存。此外，验证者使用位向量表示 Select 消息中的缺失数据块，而不是使用索引列表。我们发现，随着缺失数据块数量增加，使用位向量的优势也会随之增大。位向量带来的额外开销，使其与仅发送单个缺失数据块的索引相当。

D. 区块链作为共享随机性源

为获得共享随机性源，我们实现了一个小型探测器，用于监听以太坊区块链上的新区块。收到新区块后，探测器会将其区块编号与下一个预期区块编号进行比较，以确定该区块是否为区间中的下一个区块。如果区块编号大于或等于下一个预期区块编号，探测器就会向区块链查询下一个预期区块编号对应的区块哈希。因此，即使我们查询区块链获取新区块的速度较慢，一旦追赶上进度，我们也会像其他对等节点一样使用相同的区块哈希作为随机数。

E. 评估框架

我们简要介绍为评估 SNIPS 和 Pullsync 的性能而实现的评估框架。为收集指标，我们使用 Prometheus [40] 监控系统。我们为 SNIPS 和 Pullsync 注册了值得关注的指标，例如发送的消息数、消息中的字节数、处理消息所需的时间等。此外，由于协议在终止前不会等待数据块完成同步，我们使用 Prometheus 监控对等节点的活动，以确定同步何时完成。

在 SNIPS 和 Pullsync 中，通信都局限于邻域内。为确保我们只收集邻域内对等节点的指标，且不受外部干扰，我们在 Swarm Bee 客户端中添加了一个 API 端点，用于查询某个对等节点的邻域。

VI. 评估

本节介绍我们对 SNIPS 的评估。我们在集群上使用真实的 Swarm 部署，从传输数据量、带宽节省、同步时间、计算时间以及每个数据块的带宽需求等方面测量了 SNIPS 的性能。我们将测量结果与 Swarm 的 Pullsync 协议进行了比较。我们评估了包含数据块丢失 (CL)和添加新数据块 (CA)场景下的完整同步过程。

我们的结果表明，与 Pullsync 相比，SNIPS 使用的同步数据最多可减少三个数量级。SNIPS 的计算开销表明，它是一种实用的协议，每个数据块只需数十微秒即可生成并验证证明。我们还模拟了 SNIPS 在最具挑战性的同步场景中的性能，此时对等节点之间的相似度 为 0。模拟结果表明，SNIPS 在严苛条件下仍能表现良好，并且始终能够高效地同步对等节点。我们调整了数据块数量，使邻域的总存储量从 1 兆字节到 1000 兆字节不等，部分实验中则高达 10 GB。

A. 实验设置

实验在一个由 30 台运行 Ubuntu 18.04.4 LTS 的物理机器组成的集群上进行。每台机器配备 32 GB 内存、Intel Xeon E-2136 3.30 GHz CPU、1.5 TB SSD 硬盘和 10 Gbit/s 网卡。我们使用该集群运行一个包含 1000 个 Swarm 对等节点的大型网络。借助 Kubernetes [41] 和 Helm [42], , 我们将 Swarm 对等节点部署在 28 台机器上，其中一台用于托管私有以太坊网络，另一台用于管理实验执行。

我们的Swarm 网络分布如图2所示。共有 81 个邻域，其规模介于8到26 个对等节点之间。由于 SNIPS 的同步操作仅限于单个邻域内，我们使用包含 8,17 和26个对等节点的邻域对该协议进行了评估。因此，对于整个系统的带宽节省，一个粗略的费米估算是 $81\times$ 单个邻域的节省量。

B. 一致性 *Pullsync*

由于 Pullsync 的不一致性，比较 SNIPS 和 Pullsync 的性能颇具挑战。此外，SNIPS 的阶段少于 Pullsync，且 SNIPS 的发起者希望上传数据块，而 Pullsync 的发起者希望下载数据块。为此，我们的持续时间测量从协议启动时开始，直到所有对等节点完全同步为止。然而，这一测量并非没有注意事项，因为 Pullsync 会将新的数据块传播给其邻居，从而缩短数据块分发时间，同时还可能因重复上传数据块而浪费带宽。虽然 SNIPS 可以轻松实现类似的优化，但我们的首要目标是减少传输数据量。

对于数据传输测量，我们能够将数据同步流量与分块上传流量区分开来。不过，为了评估分块丢失场景，我们修改了 Pullsync，使其不会导致不一致性。添加新数据块时，我们将 SNIPS 与原版 Pullsync 进行比较。

C. SNIPS 与 *Pullsync* 的真实场景比较

我们首先通过在集群中相同条件下测量邻域的<style id='1'>传输的数据 和<style id='3'>同步时间 ，将SNIPS与 Pullsync 进行比较。对于每种配置，我们将实验重复了 10 次，并将<styleid='5'>相似度固定为 1，这意味着 $|A \cap B| = \min(|A|$<style id='9'>$, |B|)$。

在 CL 场景中，对等节点最初存储了 1 兆字节至 1000 兆字节的数据块数据。随后，我们将数据块丢失量从 0% 调整到 100%。图 7 (a, b, c) 分别展示了同步存储大小为 8,17 和 26 的邻域时传输的元数据。与 Pullsync（虚线）相比，SNIPS（实线）传输的元数据少 2–3 个数量级。对于所有邻域存储大小，结果均一致。

图 7 (e, f, g) 分别展示了同步大小为 8,17 和 26 的邻域所需的时间。同步时间包括下载同步元数据和上传数据块所需的时间。对于小存储大小（1、10 兆字节）和 8 个对等节点的邻域，SNIPS 始终比 Pullsync 更快。对于 17 个和 26 个对等节点的邻域以及较大的存储大小（100、1000 兆字节），当数据块丢失率超过 30% 和 50% 时，Pullsync 的速度略快。尽管传输的元数据更多，但在这些情况下 Pullsync 仍然更快，这是因为它采用了 Gossip 优化，详见第 VI-B 节。我们的实验是在通过未饱和的 10 Gbit/s 链路连接的集群节点上进行的。我们推测，在对等节点通过带宽较低的链路连接的互联网环境中，Gossip 优化的效果会较弱。

在 CA 场景中，我们向一个对等节点添加 1 兆字节至 1000 兆字节的新数据块数据，并观察同步行为。同步大小为 8的邻域所传输的元数据、17 和 26 的情况如图 7d 所示。正如预期，对于 Pullsync，元数据量与上传的新数据块数据呈线性关系。然而，SNIPS 呈现次线性关系，且上传量越大，这一趋势越明显。总体而言，SNIPS 传输的数据比 Pullsync 少 1 至 1.5 个数量级。

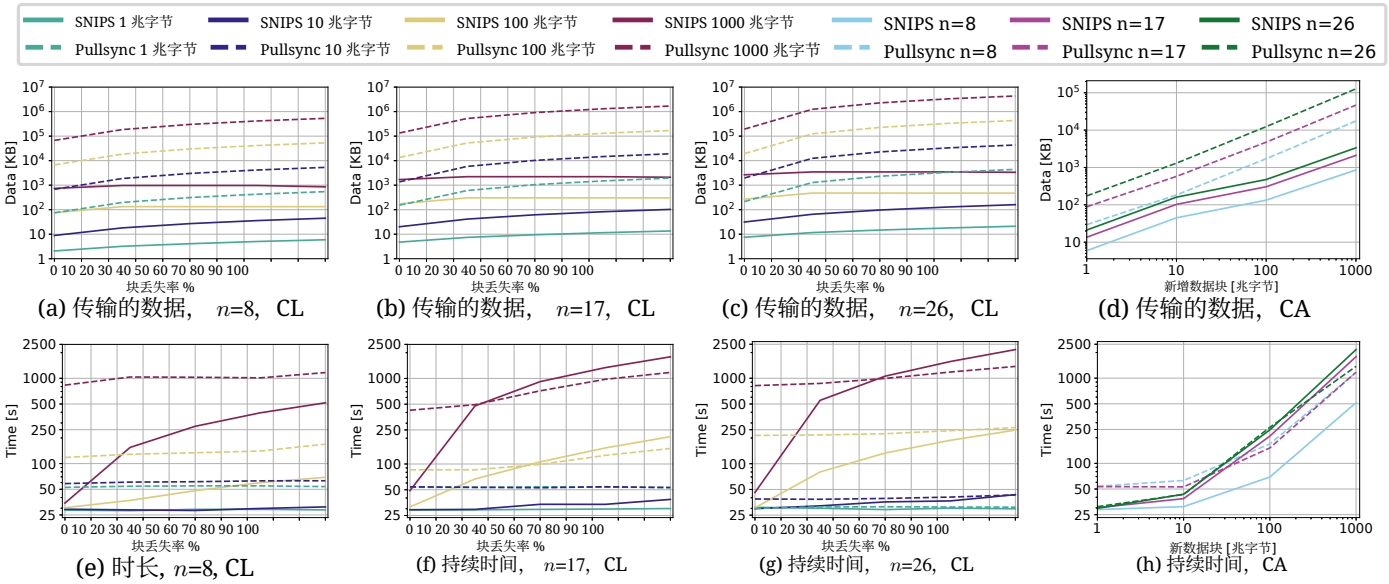
图 7h 展示了同步大小为 8、17 和 26 的邻域所需的时间。对于 10 兆字节的新数据块，SNIPS 始终比 Pullsync 更快。然而，由于 gossip 优化，当新数据块达到 100 兆字节或更多时，Pullsync 略占优势。

我们在表 I 中以平均带宽节省和加速比的形式总结了相同的结果。

D. 相似度程度不同的 *SNIPS*

回顾第 II 节中定义的指标：相似度、 $|A\cap B|/\min(|A|, |B|)$ 和 证明准确率、 $|M_i|/|M_p|$。现在，我们模拟相似度为 < 1 时 SNIPS 的性能，这意味着两个对等节点都拥有对方没有的数据块。正如第 III-C 节所述，这会使同步过程更加复杂，因为使用不属于原始证明的数据块证明进行查询可能会导致冲突。

在所有模拟中，我们为对等节点初始化相同数量的数据块。根据相似度设置，对等节点完成同步后存储的数据块总量将在相似度为 1（数据块完全重叠）时的初始数量与相似度为 0（数据块完全不相交）时的初始数量之间变化。



(a) 传输的数据， n=8，CL (b) 传输的数据， n=17，CL (c) 传输的数据， n=26，CL (d) 传输的数据，CA

(e) 时长, n=8, CL (f) 持续时间， n=17，CL (g) 持续时间， n=26，CL (h) 持续时间，CA

图 7：同步对等节点邻域时 SNIPS 与 Pullsync的比较。我们将邻域存储大小设置为1、10、100和1000兆字节。对等节点邻域大小为 $n = \{8, 17, 26\}$。图(a、b、c) 展示了随着数据块丢失率 (CL)变化而变化的传输的数据。图(e、f、g) 展示了随着数据块丢失率 (CL) 变化而变化的时长。图 (d)展示了随着新增数据块数据量 (CA) 变化而变化的传输的数据。图(h) 展示了随着新增数据块数据量(CA)变化而变化的时长。时长包括传输同步元数据和上传数据块所需的时间。

表 I：平均带宽 h 节省量（KB）以及使用 SNIPS 相较于 Pullsync 的加速效果（ms）。RSE 是相对标准误差 r.

对等节点	Size	CL：带宽节省 平均值（KB）	CL：带宽节省 平均值（%）	CL：带宽节省 相对标准误（%）	CA：带宽节省 平均值（KB）	CA：带宽节省 平均值（%）	CA：带宽节省 相对标准误（%）	CL：加速 平均值（ms）	CL：加速 平均值（%）	CL：加速 相对标准误（%）	CA：加速比 平均值（ms）	CA：加速比 平均值（%）	CA：加速比 相对标准误（%）
8	1 MB	308	98.7	0.4	23	79.8	0.6	25400	47.0	1.2	25069	46.6	1.4
	10 兆字节	2981	99.1	0.1	139	75.5	0.4	31821	52.0	1.5	31859	50.6	2.2
	100 兆字节	29849	99.6	0.1	1582	92.2	0.1	89170	64.6	1.9	99949	59.1	2.1
	1000 兆字节	298957	99.7	0.0	16630	95.1	0.3	741126	73.0	1.5	651543	55.8	1.5
17	1 MB	1038	99.1	0.4	75	84.7	0.2	24056	44.9	1.2	23150	43.4	1.0
	10 兆字节	10166	99.4	0.1	474	82.2	0.1	20513	38.4	2.3	14385	27.2	3.7
	100 兆字节	91024	99.7	0.0	4449	93.6	0.1	-3833	-3.5	1.4	-56847	-37.5	0.6
	1000 兆字节	909988	99.8	0.0	44463	95.5	0.0	-156825	-20.7	1.5	-614137	-52.3	1.2
26	1 MB	2293	99.4	0.3	152	87.8	0.6	955	3.1	1.3	1354	4.4	1.6
	10 兆字节	22619	99.6	0.1	1142	87.7	0.2	4451	11.1	2.5	-307	-0.7	3.2
	100 兆字节	226274	99.8	0.1	11907	96.2	0.1	96969	41.7	1.2	17451	6.6	1.3
	1000 兆字节	2265603	99.9	0.0	123972	97.4	0.1	-29329	-2.8	1.6	-788973	-57.3	1.8

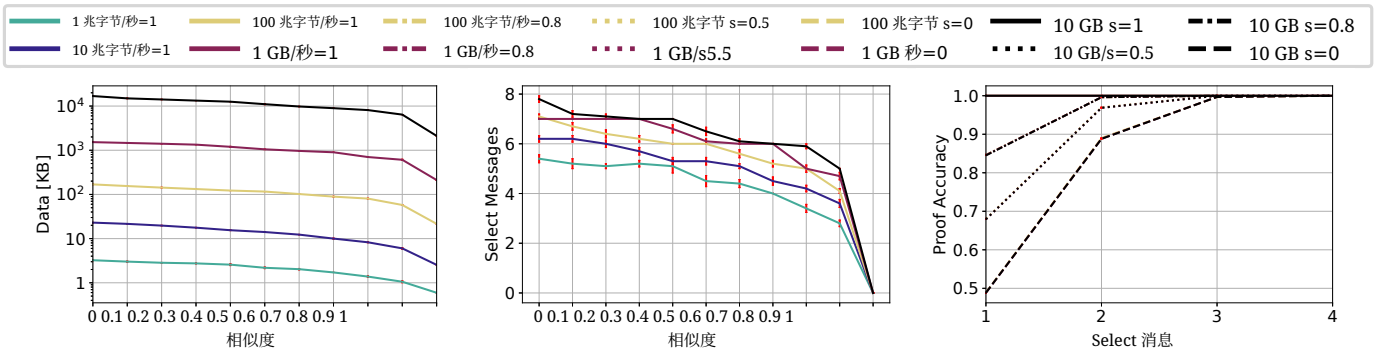
我们首先模拟传输的同步元数据。正如预期，图 8a 中相似度为 1 时每个对等节点传输的元数据，与图 7 中分块丢失率为 0% 时传输的元数据非常接近。对于所有存储大小，当相似度从完全重叠（1）降至完全不相交（0）时，传输的元数据增幅略低于一个数量级。传输元数据的增长源于对等节点需要依次执行更多轮证明和选择阶段，才能完全同步。每轮选择阶段结束后，对等节点之间的同步程度都会提高，随后生成的证明也会更大、更准确。

图 8b 展示了两个对等节点完全同步所需的 Select消息数量。由于两个对等节点最初拥有相同数量的数据块，当相似度为 1 时，对等节点已经完成同步，无需 Select消息。数量Select消息的数量随着相似度的降低而增加。

最后，我们评估了证明的准确率，如图8c所示。有趣的是，数据块数量不会影响证明准确率。当相似度为 1 时，证明准确率始终为 1，因为通过一条 Select 消息即可识别所有缺失分块。此外，证明准确率会随着 Select消息数量的增加而提高。这是因为每个证明包含的数据块都比前一个更多，从而使对等节点能够识别出更多缺失分块。完全同步所需的 Select消息数量最多为 4。

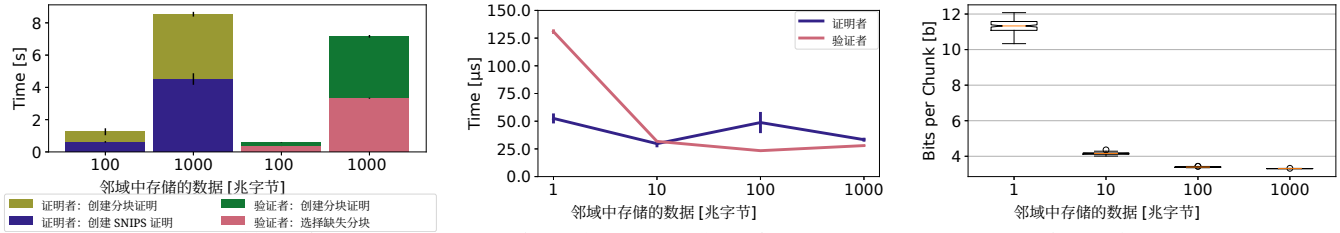
E. 计算开销

我们通过提取运行实验所得日志中的时间，测量了从 SNIPS 证明中创建和选择缺失分块（验证）所花费的时间。



(a) 传输的数据　(b) Select消息　(c) 证明准确率

图 8：SNIPS 在不同相似度 $s = [0, 1]$ 下的性能。标签表示同步过程开始前每个对等节点的初始存储。(a) 两个对等节点的同步数据。(b) 两个对等节点完成同步所需的 Select消息数量。(c) 证明准确率与迭代 Select 消息数量的关系。竖线表示标准误差。



(a) 创建和选择所需时间　(b) 每块创建和选择所需时间　(c) 每块开销

图9：空间和时间开销。(a) 从 SNIPS 证明中创建和选择缺失分块所花费的时间。(b) 从 SNIPS证明中创建和选择缺失分块时每个数据块所花费的时间。(c)不同邻域存储大小（每块比特数）下 SNIPS 证明的每个数据块开销。竖线表示误差线。

表 II：从 SNIPS 证明中创建并选择缺失分块所花费的时间。

操作	证明者 100 兆字节 1000 兆字节	验证者 100 兆字节 1000 兆字节
分块证明	622 毫秒 4517 毫秒	359 ms 3327 ms
创建	628 毫秒 4017 毫秒	
选择		240 毫秒 3838 毫秒

在第 VI-C 节中，我们在集群上进行了测试。由于创建数据块证明是两种操作的共同部分，因此我们单独统计了创建数据块证明所花费的时间。结果如图 9a 和表 II 所示。

结果表明，验证者生成数据块证明所花的时间略少于证明者。此外，选择缺失分块所花的时间略少于生成证明所花的时间。

相比之下，我们发现，在 Pullsync 的 Offer 阶段，向一个对等节点发送包含大量数据块标识符的列表，平均需要 73.2 秒（对应 100 MB 数据块）和 728.6 秒（对应 1000 MB 数据块）。Offer 阶段的部分操作是并行执行的，因此还需要进一步研究，才能确定顺序操作所花的确切时间。

接下来，我们测量了耗时与证明大小之间的相关性。图 9 b 显示，这两种操作所需的时间都只有几十微秒。随着证明大小的增加，每个数据块的耗时似乎趋于稳定。对于 1 兆字节的证明，由于签署证明需要固定开销，因此每个数据块的耗时略高。对于更大的证明，这项开销会分摊到更多数据块上，从而降低每个数据块的耗时。

F. 存储开销

我们从第 VI-C 节所述实验中提取了 SNIPS 证明的大小。由于证明大小与数据块数量成正比，我们以每块比特数来衡量存储开销。结果如图 9c 所示。对于包含 1000 MB 数据块的证明，开销接近每块 3.3 比特。与第 VI-E 节一样，对于 1 MB 数据块，开销主要由固定开销构成。在这种情况下，1 MB 数据块的每块比特数较高，是由数字签名、随机数和 [起始，结束] 范围导致的。不过，对于更大的存储大小，这些开销会被摊薄。

VII. 相关工作

我们将本文的工作与两类主要相关工作进行比较：PoS 构造和数据同步协议。

A. 布隆过滤器变体与集合协调

布隆过滤器 [18] 是一种用于动态集合的概率型数据结构，支持与我们的类 PoS

构造类似的成员查询。布隆过滤器使用 k 个哈希函数，将每个元素映射到位数组中的 k 个位。查询布隆过滤器时，结果会是 k 个数组位置。如果 k 个数组位置中有任意一个为 0，那么该元素肯定不在集合中。否则，我们认为该元素可能在集合中，因为成员查询存在假阳性。增加每个元素所用的位数可以降低布隆过滤器的假阳性率。

布隆过滤器与我们的 PoS 式构造之间一个关键区别在于，我们的成员查询始终会返回一个索引值 $[0, N]$，可用于识别缺失元素。正如先前的研究所指出的 [43], ，除了对全集中的所有可能元素逐一进行成员查询外，如何根据布隆过滤器的数组位置识别缺失元素并不明显。

可逆布隆过滤器（IBF） [43] 是对标准布隆过滤器的扩展，当集合足够小时，它允许查询提取集合中的元素。IBF 在每个单元格中嵌入一个异或字段，该字段根据单元格中的所有键计算得出。两个 IBF 可以尝试利用异或字段识别其对称差中的元素。

[17] 提出了使用 IBF 进行集合协调的方法。他们的结果表明，只有当对称差小于 30% 时，该方法才能以较高概率识别缺失元素。相比之下，我们已经证明，即使对称差达到 100%（最大值），SNIPS 也始终能够识别缺失元素。

布谷鸟过滤器 [44]是布隆过滤器的一种替代方案，支持删除和近似成员测试。将元素通过两个不同的哈希函数进行哈希，再将哈希值映射到至少一个由此产生的候选桶中，即可将元素插入布谷鸟过滤器。

近期工作 [45] 提出了使用标记布谷鸟过滤器（MCF）进行多方集合协调的*MCFSyn* 。每个对等节点生成一个 MCF 向量，并将其发送给中心化参与方。随后，中心化参与方创建一个总体MCF 向量，并将其分发给各个对等节点。然后，对等节点可以通过将其 MCF 向量与总体MCF 向量进行比较来识别缺失元素。该协议要求在协议持续期间对等节点集合保持固定。在该协议中，假阳性（将缺失元素识别为集合中的元素）和假阴性（将集合中的元素识别为缺失元素）均有可能发生，从而导致过多的通信和不准确的协调。相比之下，SNIPS 是一种去中心化协议，不要求对等节点集合固定，也不会出现假阴性。

B. 存储证明算法和累加器

PoS 算法 [21], [46], [47]允许对等节点证明自己持有一段数据，而无需透露数据本身。PoS 协议涉及三个参与方：挑战者、证明者和验证者。 [48], 所总结的那样，PoS 算法的特性、安全性和性能差异很大。

一个相关的特性是公开可验证性，首次在 [21]中引入。借助此特性，任何人都可以担任验证者并验证该证明。具备此特性的算法要求原始数据所有者生成并持久化存储一些元数据，供其他对等节点创建存储挑战并验证证明。这些元数据会带来额外开销，并且需要为每个应可公开验证的数据块生成。如果这些元数据丢失，原始数据所有者必须重新生成它们。我们的类 PoS构造允许任何拥有原始数据集的人在无需额外元数据的情况下验证证明。

我们简要提及，某些 PoS方案 [49], [50]可以通过对每个副本采用不同的编码，或依赖时序假设，来检测串通的对等节点。然而，这些依赖方式并不适用于去中心化存储系统。

累加器是一种表示元素集合的密码学构造，并允许在不披露元素本身的情况下生成成员证明 [51]。通用累加器 [52]允许证明者生成成员证明和非成员证明。其他变体包括基于双线性映射 [53]和 Merkle树 [54]的累加器。与我们的类 PoS 构造不同，累加器可以对成员查询给出确定性答案。然而，对于每个数据块，它们都需要不同的见证，而该见证必须不断更新，可能会使系统不堪重负。

C. rsync

一种常用的数据同步工具是*rsync*。rsync算法 [55]旨在减少在两个对等节点之间传输文件所需的数据量。Rsync通过仅传输目标位置尚不存在的块，并使用滚动校验和检测文件内的更改来实现这一点。然而，rsync 不适用于去中心化存储系统，因为它一次只能同步一个对等节点。

VIII. 结论

本文介绍了 SNIPS，这是一种用于去中心化存储系统中数据同步的新型协议。高效的数据同步能够频繁运行并响应网络节点 churn，这对于维持存储系统中足够的冗余级别至关重要。SNIPS 采用类似 PoS 的构造来生成支持成员查询的存储证明。对等节点与相邻对等节点交换存储证明，并查询这些证明以识别缺失分块，随后请求这些分块。每个数据块的证明通常只需几个比特。

我们的贡献包括在真实世界的集群上进行严格实验，以表明SNIPS 是一种实用的协议。此外，我们还通过模拟最坏情况场景下的性能，证明了 SNIPS 的正确性。

我们将 SNIPS 与 Ethereum Swarm 中用于数据同步的最先进协议 Pullsync 进行比较。我们的结果表明，Pullsync 容易受到不一致性的影响，不提供存储保证，并且使用的同步数据量最多比 SNIPS 多三个数量级。

SNIPS 对其他去中心化存储系统的潜在影响似乎值得期待，但仍有待全面探索。我们认为，类 PoS 构造还有其他用途，但将其留待未来工作探讨。

参考文献

[1] V. Tr´on, 《Swarm 之书——v1.0 预发布版，2020 年 11 月 7 日》，访问日期：2022-09-10。 [在线]。获取地址：https://www.ethswarm.org/The-Book-of-Swarm.pdf [2] P. Labs, 《Filecoin：一种去中心化存储网络》，2017 年 7 月，技术报告。访问日期：2023-02-20。 [在线]。获取地址：https://filecoin.io/filecoin.pdf [3] J. Benet, 《IPFS——内容寻址、版本化的 P2P 文件系统》，arXiv 预印本 arXiv:1407.3561，2014 年。[4] R. Nygaard、H. Meling 和 J. I. Olsen, 《去中心化存储系统中的经济高效数据维护》，载于第 38 届 ACM/SIGAPP 应用计算研讨会论文集，美国纽约州纽约市，2023 [ ] 10 1145 3555776 3577728 。在线。获取地址：https://doi.org/。/。[5] S. Henningsen、M. Florian、S. Rust 和 B. Scheuermann, 《绘制星际文件系统》，载于 2020 年 IFIP 网络会议（Networking）。IEEE，2020 年，第 289–297 页。[6] D. Trautwein、A. Raman、G. Tyson、I. Castro、W. Scott、M. Schubotz、B. Gipp 和 Y. Psaras, 《IPFS 的设计与评估：去中心化网络的存储层》，载于 ACM SIGCOMM 2022 会议论文集，2022 年，第 739–752 页。[7] S. Henningsen、S. Rust、M. Florian 和 B. Scheuermann, 《爬取 IPFS 网络》，载于 2020 年 IFIP 网络会议（Networking）。IEEE，2020 年，第 679–680 页。[8] E. Daniel 和 F. Tschorsch, 《被动测量 IPFS 节点 churn 和网络规模》，arXiv 预印本 arXiv:2205.14927，2022 年。[9] D. Dias、J. Johnson 和 J. Benet, 《Bitswap——协议规范》，访问日期：2023-02-20。[在线]。获取地址：https://github.com/ipfs/specs/blob/main/BITSWAP.md [10] A. De la Rocha、D. Dias 和 Y. Psaras, 《利用 Bitswap 加速内容路由：IPFS 和 Filecoin 中的多路径文件传输协议》，2021 年。[11] P. Labs, 《项目：超越 Bitswap》，访问日期：2023-02-20。 [在线]。获取地址：https://github.com/protocol/beyond-bitswap [12] Z. Lakhani, 《穿越多重宇宙的笔记：IPFS ↔ car pool ↔ car mirror》，访问日期：2023-02-20。[在线]。获取地址：https://talk.fission.codes/t/notes-through-the-multiverse-ipfs-car-pool-car-mirror/364/ [13] P. Labs, 《Graphsync 线协议的实现》，访问日期：2023-02-20。 [在线]。获取地址：https://github.com/ipfs/go-graphsync [14] B. Zelenka, 《Car mirror v0.1.0》，访问日期：2023-02-20。 [在线]。获取地址：https://github.com/fission-codes/spec/blob/main/car-pool/car-mirror/SPEC.md [15] S. Agarwal 和 A. Trachtenberg, 《估算远程集合之间的差异数量》，载于 2006 年 IEEE 信息论研讨会（ITW'06）埃斯特角城。IEEE，2006 年，第 217–221 页。[16] J. W. Byers、J. Considine、M. Mitzenmacher 和 S. Rost, 《在自适应覆盖网络中进行知情内容交付》，IEEE/ACM 网络汇刊，第 12 卷，第 5 期，第 767–780 页，2004 年。[17] D. Eppstein、M. T. Goodrich、F. Uyeda 和 G. Varghese, 《差异何在？无需先验上下文的高效集合协调》，ACM SIGCOMM 计算机通信评论，第 41 卷，第 4 期，第 218–229 页，2011 年。[18] B. H. Bloom, 《允许错误的哈希编码中的空间/时间权衡》，ACM 通讯，第 13 卷，第 7 期，第 422–426 页，1970 年。[19] M. O. Rabin, 《面向安全、负载均衡和容错的信息高效分散》，ACM 杂志（JACM），第 36 卷，第 2 期，第 335–348 页，1989 年。[20] R. Nygaard、V. Estrada-Gali˜nanes 和 H. Meling, 《Snarl：用于提升文件可用性和存储利用率的纠缠 Merkle 树》，载于第 22 届国际中间件会议论文集，Middleware '21 系列。美国纽约州纽约市：计算机协会，2021 年，第 236–247 页。 [在线]。获取地址：https://doi.org/ 10 1145 3464298 3493397 。/。G. Ateniese、R. Burns、R. Curtmola、J. Herring、L. Kissner、Z. Peterson 和 D. Song, 《不可信存储中的可证明数据持有》，载于第 14 届 ACM 计算机与通信安全会议论文集，2007 年，第 598–609 页。[22] A. Reutov, 《预测以太坊智能合约中的随机数》，访问日期：2022-12-05。[在线]。获取地址：https://blog.positive.com/predicting-random-numbers-in-ethereum-smart-contracts-e5358c6b8620 [21]

[23] R. Canetti、U.Feige、O. Goldreich 和 M. Naor, 《自适应安全的多方计算》，载于《第二十八届 ACM 计算理论年会论文集》，1996 年，第 639–648 页。 [24] E.Syta、P. Jovanovic、E.K. Kogias、N.Gailly、L. Gasser、I. Khoffi、M.J.Fischer 和 B. Ford, 《可扩展的抗偏置分布式随机性》，载于《2017 IEEE安全与隐私研讨会（SP）》。Ieee，2017 年，第 444–460页。[25] D. Boneh、J. Bonneau、B. B¨unz和 B.Fisch, 《可验证延迟函数》，《密码学电子印本文档库》，论文 2018/601，2018 年，https://eprint.iacr.org/2018/601。 [在线]。可用地址：https://eprint.iacr.org/ 2018 601 / S. Micali、M. Rabin和 S.Vadhan, 《可验证随机函数》，载于《第四十届计算机科学基础年会》（目录号 99CB37039），1999 年，第 120–130 页。[26] [27] M.Vijaymeena 和 K. Kavitha, 《文本挖掘中的相似度度量综述》，《机器学习与应用：国际期刊》，第3卷，第 2 期，第 19–28页，2016 年。[28] P. Maymounkov 和D. Mazieres, 《Kademlia：基于XOR 度量的对等信息系统》，载于《对等系统国际研讨会》。施普林格，2002 年，第 53–65 页。[29] J. Gulja˘s, 《网络统计 ——Swarm 扫描》，访问日期：2022-12-06。[在线]。可用地址：https://swarmscan.resenje.org [30] S. team, 《面向自主管理数字社会的存储与通信基础设施》，访问日期：2022-09-10。 [在线]。可用地址：https://www.ethswarm.org/swarm-whitepaper.pdf [31] S. Foundation, 《Swarm网络存储激励机制》，访问日期：2022-12-07。[在线]。可用地址：https://medium.com/ethereum-swarm/the-mechanics-of-swarm-networks-storage-incentives-3bf68bf64ceb [32] E. Esposito、T.M.Graf和 S. Vigna, 《Recsplit：通过递归分割实现最小完美哈希》，载于《2020 年第二十二届算法工程与实验研讨会论文集（ALENEX）》。SIAM，2020年，第 175–185 页。 [33] A. Limasset、G. Rizk、R.Chikhi和P.Peterlongo, 《面向海量键集合的快速且可扩展的最小完美哈希》，arXiv 预印本 arXiv:1702.03154，2017 年。 [34] D. Belazzougui、F. C.Botelho和M. Dietzfelbinger, 《哈希、置换与压缩》，载于《欧洲算法研讨会》。施普林格，2009 年，第682–693 页。 [35] I.J.Good, 《概率与证据权衡》，1950年。[36] 《以太坊平均区块时间图表》，访问日期：2022-09-19。 [在线]。可用地址：https://etherscan.io/chart/blocktime [37] 《以太坊2.0 阶段 0——信标链》，访问日期：2022- 09-19。 [在线]。可用地址：https://github.com/ethereum/consensus-specs/blob/v0.11.1/specs/phase0/beacon-chain.md#time-parameters [38] S.Herle, 《面向大型键集的快速可扩展最小完美哈希》，访问日期：2022-12-07。 [在线]。可用地址：https://github.com/opencoff/go-bbhash [39] Google, 《ProtocolBuffers》，2022年，访问日期：2022-05-24。 [在线]。可用地址：https://developers.google.com/protocol-buffers [40] Prometheus, 《Prometheus——监控系统与时间序列数据库》。[在线]。可用地址：https://prometheus.io/ [41] Kubernetes, 《生产级容器编排》。 [在线]。可用地址：https://kubernetes.io/ [42] Helm, 《Kubernetes 的包管理器》。[在线]。可用地址：https://helm.sh/ [43] D.Eppstein 和 M.T.Goodrich, 《通过牛顿恒等式和可逆布隆过滤器识别往返数据流中的掉队者》，《IEEE 知识与数据工程汇刊》，第 23 卷，第 2 期，第297–306 页，2010年。[44] B. Fan 、 D. G. Andersen 、 M.Kaminsky 和 M.D.Mitzenmacher, 《布谷鸟过滤器：实际性能优于布隆过滤器》，载于《第十届 ACM 新兴网络实验与技术国际会议论文集》，2014年，第 75–88页。 [45] L. Luo、D. Guo、Y. Zhao、O. Rottenstreich、R.T.Ma和 X. Luo, 《MCFSyn：一种采用标记布谷鸟过滤器的多方集合协调协议》，《IEEE并行与分布式系统汇刊》，第 32 卷，第11 期，第 2705–2718 页，2021 年。 [46] A. Juels和B. S.Kaliski Jr, 《Pors：大型文件的可检索性证明》，载于《第十四届ACM 计算机与通信安全会议论文集》，2007 年，第 584–597 页。[47] H. Shacham 和 B. Waters, 《紧凑型可检索性证明》，载于《密码学与信息安全理论及应用国际会议》。施普林格，2008 年，第 90–107 页。

[48] A. Yang、J. Xu、J. Weng、J. Zhou 和 D. S. Wong，“云存储中具有数据动态性的轻量级且保护隐私的可委托存储证明”，*IEEE* 云计算汇刊，第 9 卷，第 1 期，第 212–225 页，2018 年。[49] I. Damg˚ard、C. Ganesh 和 C. Orlandi，“无需时序假设的副本存储证明”，载于密码学进展——*CRYPTO 2019*。尚：施普林格，2019 年，第 355–380 页。[50] R. Curtmola、O. Khan、R. Burns 和 G. Ateniese，“多副本可证明数据持有：多副本可证明数据持有”，载于*2008*年第*28*届国际分布式计算系统会议。美国纽约州纽约：IEEE，2008 年，第 411–420 页。[51] J. Benaloh 和 M. d. Mare，“单向累加器：数字签名的去中心化替代方案”，载于密码技术理论与应用研讨会。施普林格，1993 年，第 274–285 页。[52] J. Li、N. Li 和 R. Xue，“具有高效非成员证明的通用累加器”，载于应用密码学与网络安全国际会议。施普林格，2007 年，第 253–269 页。[53] L. Nguyen，“基于双线性配对的累加器及其应用”，载于*RSA*会议密码学家专题。施普林格，2005 年，第 275–292 页。[54] F. Baldimtsi、J. Camenisch、M. Dubovitskaya、A. Lysyanskaya、L. Reyzin、K. Samelin 和 S. Yakoubov，“应用于保护匿名性的撤销的累加器”，载于*2017*年*IEEE*欧洲安全与隐私研讨会（*EuroS&P*）。IEEE，2017 年，第 301–315 页。[55] A. Tridgell，“高效排序与同步算法”，博士学位论文，澳大利亚国立大学，1999 年。