Genaro Network

Roadmap towards multi-source data governance framework

Yellow Paper v1.0.1

摘要

Genaro 是一个以点对点加密存储及共享为基础的新型公有区块链平台。本平台旨在以链式共识 PoS (Proof of Stake)为基础，并有机融合可验证的存储数据完整性证明 SPoR (Sentinel Proof of Retrievability) 作为有益补充，实现高效的链上节点治理。其最终愿景是建立新型的分布式加密存储媒介，让每一个使用者拥有自己数据的使用权和分享权，为打造链上功能丰富的去中心化应用（DApp）及其生态环境提供稳定可靠的平台支撑。与其他公有链相比，Genaro 有如下方面的特点：首先，在 PoS 与 SPoR 的融合过程中，Genaro 将文件指纹(Sentinel)的使用方式进行了更适合分布式可用系统的改动，从而增强了其抵御重放攻击的能力。其次，在链式 PoS 共识设计上，Genaro 参考了 Casper (CFFG, CTFG)，Tendermint，Ouroboros 等著名 PoS 方法，分析了 PoS 的主要攻击方式并制定了相关的应对方案。第三，在治理结构上，Genaro 融合了存储数据完整性证明和 PoS 的优点来设计链上治理的方式，并通过链上治理对 PoS 的潜在问题进行了有效防御。此外，在公链的数据结构设计上，Genaro 也针对了存储部分的加密特点设计了 GSIOP 协议，用来对数据的使用层级进行设定。最后，针对增加的数据添加方式，Genaro 也增加了相关 VM 指令集。

目录

1 Genaro 的愿景

Genaro 是一个结合了点对点加密存储的公有链。其设计的初衷是为了让数据在加密后可以分享，赋予用户拥有着不同的数据权利，最终能让丰富的去中心化应用（DApp）建立在该公有链之上。截至目前，大多数 DApp 都是围绕着交易市场、赌博以及游戏等以交易为核心的。两个和数据相关的特性：（1）链上数据大小受限以及（2）链外数据不可直接使用，让 DApp 不能像现在的手机或网页 App 一样随处可见，导致现在区块链的功能比较单一。

Genaro 的设计目的是为了让数据拥有分布式存储的媒介，并且补齐其加密后的数据的功能性，即将加密数据处理的思想结合到区块链中。为了实现该目标，Genaro 的系统设计为三个部分：存储网络，公链，以及共识治理结构部分（图 1）。本黄皮书将围绕这三个点进行详细阐述。

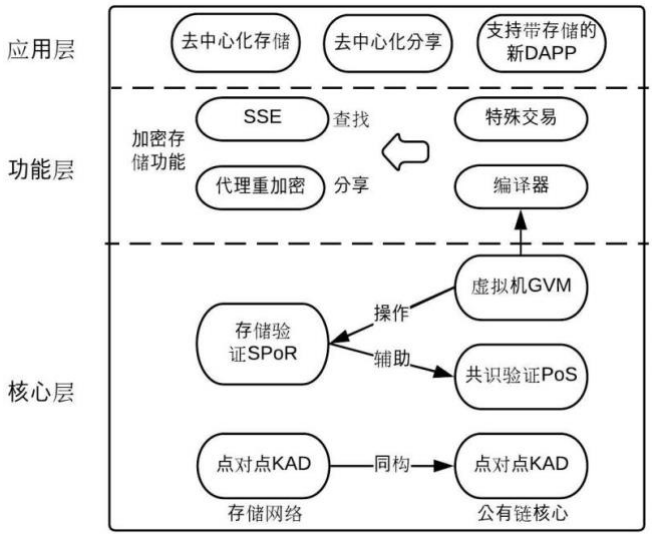

图 1. Genaro 系统架构

2 存储网络

2.1 DHT-分布式哈希表

P2P [1]，即点对点，其思想可以说是互联网哲学非常集中的体现：共同的参与，透明的开放，平等的分享。基于 P2P 技术的应用有很多，包括文件分享，即时通信，协同处理，流媒体通信等 [2]。通过这些应用的接触，分析和理解，P2P 的本质即是一种新的网络传播技术。这种新的传播技术打破了传统的架构，逐步地去中心化，扁平化，从而达到节点平等的未来趋势。

P2P 文件分享的应用(BTs/eMules 等)是 P2P 技术最集中的体现。Genaro 是以 P2P 文件分享网络作为入口，围绕一个文件网络系统，将其可操作性结合区块链的公式算法设计出新型扁平化，去中心化的云；同时保留了区块链公开，透明的特性。

P2P 文件分享网络的发展大致有以下几个阶段，包含 tracker 服务器的网络，无任何服务器的纯 DHT 网络， 混合型 P2P 网络。

分布式哈希表 DHT（Distributed Hash Table）是一种分布式存储方法。在 DHT 中，一类可由键值来唯一标示的信息按照某种约定/协议被分散地存储在多个节点上，可以有效地避免“中央集权式”的服务器（比如：tracker）的单一故障而带来的整个网络瘫痪。和中心节点服务器不同，DHT 网络中的各节点并不需要维护整个网络的信息，而是只在节点中存储其临近的后继节点信息，大幅减少了带宽的占用和资源的消耗。DHT 网络还在与关键字最接近的节点上备份冗余信息，避免了单一节点失效问题。

实现 DHT 的技术/算法有很多种，常用的有：Chord [3], Pastry [4], Kademlia [5]等。Genaro 使用的是 Kademlia 算法，因为 BT 及 BT 的衍生派（Mainline, Btspilits, Btcomet, uTorrent），eMule 及 eMule 各类 Mods （verycd, easy emules, xtreme） 等 P2P 文件分享软件都是基于该算法来实现 DHT 网

络。由于这些分享软件实现的协议并不相同，因此会有不兼容的问题。BT 采用 Python 的 Kademlia 实现称为 Khashmir。eMule 采用 C++的 Kademlia 实现称为 Kad。在众多的 P2P 协议中，出于链点对点库和存储网络点对点库同构的原因，Genaro 的实现基于 eMule 的 Kad，也就是实行 Kad 协议的 DHT 网络。在接下来的介绍中，我们会逐步讲解 Kademlia 算法本身的优点。

Kademlia 技术，通常又被称为第三代 P2P 技术，是一种 P2P 通用协议，适用于所有的分布式点对点计算机网络。Kademlia 定义了网络的结构，规划了节点之间的通讯以及具体的信息交互过程。在 Kademlia 中，网络节点之间使用 UDP 进行通信，通过一种分布式哈希表来存储数据，每个节点都会有一个自己的 ID，在用来标识节点本身的同时，也用以协助实现 Kademlia 算法和流程。

2.2 KAD 网络及节点

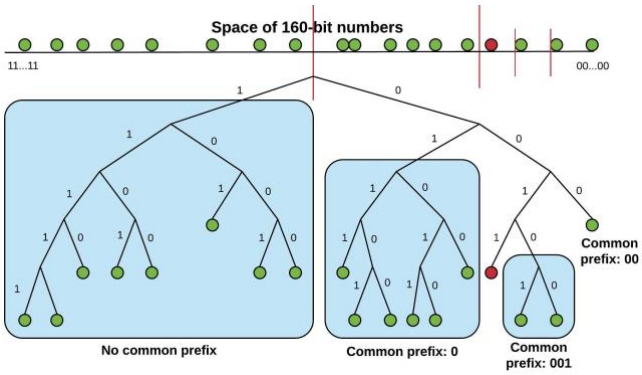

图 1. KAD 网络

KAD DHT 存储网络中的节点（node）包括如下特性：

- NodeId 在 KAD 中需要是 160 bits 或者 20 bytes;
- Contact 包含 NodeID (nodeId)，Address(string), UDP port number；
- Bucket [VaugeKConst]*Contact 用在 node 的 routing 中，一个 bucket 能包含的 k 个 node，所有的 node 在一个小时后会消失；
- VaugeKConst 统计上设为 20；
- Router 包含 Contact 和 KBucket，KBucket 是在 ID 的每一个 bit 中，都有一个 bucket。

Kademlia 使用键值来确认在 KAD 网络上的节点和数据。KAD 的键值是不透明的，长度 160 个比特。参加进来的电脑每一个都会拥有一个键值，称为 NodeID，填充在 160 比特的键值空间中。由于 KAD 存储内容的时候是由 KV（key-value） 对来存储的，每一个在 KAD DHT 中的 data 也都是独立于 160 位键值中的 key 对应的空间的。

在开始时，一个 node 不了解任何其他的 nodes。当新的 nodes 注册后，找到这个节点的链接，然后将新的 NodeID 存起来。当存储满溢，contact 会被选择性去掉，然后在 bucket 内部被组织。从一个 node 找一个 NodeID 的方式，是从一个已知的路由表中，从一个 node 找到另一个最近的 node，直到找到 request node。

KAD 提供了其他 DHT 中无法自发提供的几个特性，包括：

- KAD 最小化了节点内 intro 信息的数量；
- 配置信息中包括网络中节点信息以及临节点信息，并在 key 的查找中通过副作用的关系自动传播；
- 在 KAD 中的节点是知道其他节点的，这个特性允许了通过更低延迟的路径进行路由询问；
- KAD 用了并行和异步的请求，可以避免失败节点的超时延迟；
- KAD 是对一些 DOS 攻击有抵抗性的。

Genaro 选择 KAD 的另一个原因是 Genaro 自身公链的点对点系统使用的也是 KAD，在进行账号管理的时候，可以通过同一个系统进行处理，对后期的实现操作来说相对容易。

2.3 异或矩阵以及距离计算

每个 KAD 节点都有 160 比特的 NodeID，每个数据的键值也是一个 160 位的标识符。想要确定 KV 对存在哪个节点中，KAD 是用了两个标识符间距离的概念了。在给定两个 160 位标识符，x 和 y 下，KAD 通过他们两个的“异或”来确定他们之间的距离，并表达为一个整数 $d(x,y) = x \oplus y$ 。XOR (异或) 获得到的是系统二叉树框架中对距离的定义。在一个完全的 160 位 ID 二叉树 ID 中，两个 ID 距离的大小是最小的包含两个节点的子树。当树不是一个完全二叉树的时候，距离 ID_x 最近的叶是与 x 共享最长公共前缀的叶。举例来说，在 0011 和 1001 之间的距离则是 $0011 \oplus 1001 = 1010$, 1010 通过整数表达则是 10，所以在这两个节点的距离则是 10。

2.4 节点状态 （ Contact ）

用 NND (Node Network Data) 来表示在 KAD 节点中的信息包含：

- IP Address
- UDP port
- Node ID

在 KAD 中，节点为了路由询问信息互相存下 contact 信息。在每个$0 < i < 160$, 每个节点会保留一个在距离与自己在2i 和 $2i + 1$之间节点的 NND 的清单。这个清单叫做 k-buckets。因此，节点会知道其子树中的一些节点。每一个 k-bucket 都会按照时间顺序存储：上一个收到却没有被最近看到的节点在头部，最近被看到的节点在尾部。当 KAD 节点从其他节点收到任何的信息(请求或者回复)，节点会对发送方的 NodeID 更新适合的 k-bucket。在更新 k-bucket 的时候，有以下三种情况：

- 如果发送的节点已经存在在接收方的 k-bucket，接收方将它放在列表的最尾。
- 如果节点没有在适合的 k-bucket，同时 bucket 有比 k 更少的键值对数组(entries)，那接收方会直接增添新发送方到列表的尾部。
- 如果合适的 k-bucket 已经满了，接收方通过 ping 一下 k-bucket 之前看到的 node 来决定接下来做什么。如果上一个最近看到的节点收到了失败的回复，它就会从 k-bucket 中被剔除出来，然后新的发送方会被安插在尾部。反过来说，如果最近看到的节点得到回复，就把它放在列表的最尾，新的发送 contact 就被丢弃。

KAD 的 k-bucket 有两个好处：

第一，k-bucket 实现了最近观察节点驱逐的 policy，除了留存节点永远不被从列表移除。P2P 系统的分析指出，一个节点越在上面，越有可能在下一个小时还在上面。通过保留老的存活 contact，k-bucket 最大化了留在线上节点的可能。

第二，k-bucket 提供了对某些 DoS 攻击的抵抗特性。一个恶意用户不能通过用大量新节点的方式刷每个节点的路由状态。KAD 节点只会在旧的节点离开他们的时候才会加入新的节点。每一个 KAD 节点也都有一个路由表。路由表则是一个二叉树，叶节点则是 k-buckets。节点的路由表包含了节点所有的 k-buckets，也就是当前节点的在不同子树中的邻居。

2.5 Kademlia 协议[15]

Kademlia 协议包含了四个 RPC (Remote Procedure Calls): PING, STORE, FIND_NODE 和 FIND_VALUE，其中：

- PING RPC 探测节点是否在线；
- STORE RPC 通知节点来存一个键值对 [key,value] 来保证之后的取回；
- FIND_NODE RPC 包含了 160 位的键位为一个 arg, FIND_NODE RPC 的接收方返回一个关于目标 id 最近 k 节点的 NND(contact)信息；

- FIND_VALUE RPC 功能类似 FIND_NODE，也是返回目标 id 对应的最近 k 节点。有一个例外：如果 RPC 接收方收到给定 key 的 STORE，这个会返回存储的值。

-

2.6 查找算法[15]

在 KAD 里面的节点查找过程是通过 KAD 根据给定的键值去定位 k 个最近节点的。KAD 在节点查找中选择采用的是递归算法。发起查找的一方首先从非空 k-bucket 中找到一个节点（或者，如果那个 bucket 有比α少的键值对数组，那他就只能通过键值得到α个最近节点）。发起方通过并行异步方式发送 FIND_NODE RPC 给选到的α个节点。α是一个系统的并发参数。在递归的阶段，发起方重新发送 find node 给之前发过 RPC 的节点。无法迅速回应的节点就会被移除，除非直到这些节点回复。如果一轮下来寻找节点没有找到任何比最近观察节点更近的节点，发起方会重新发送 find node 给那些没有被请求过的节点中寻找 k 个最近的。当发起者请求后得到 k 个最近观察节点的回复后，查找过程就会结束。每一个节点至少知道它每个子树中的一个节点，每一个节点都可以通过 NodeID 定位到其他节点。要存一个 KV 对，节点需要通过键值定位对应 k 个最近节点然后发送 STORE RPC。要找一个 KV 对，节点需要查找到对应键值最近的 k 个节点。但是，值（value）的查找会使用 FIND_VALUE 而不是 FIND_NODE， 而且这个过程当任何节点返回值的时候立刻停止。

2.7 缓存

在缓存的方面，一旦查找成功，发送请求的节点存储下它观察到 没有返回值的最近节点的键值 KV 对。因为拓扑的单向性，在未来查找相同键值的时候有可能在查找到最近节点前碰到缓存键值对数组。随着某个键值高频率的使用，系统可能会最终在很多节点中缓存。为了防止过度缓存，一个 KV 对在任何节点数据库的有效期是被设定为与节点数量反向指数关系的，这个节点数量是在当前节点和键值和拥有最近键值 ID 的节点之间的数量。

2.8 Bucket 刷新

KAD bucket 总体上通过节点之间的请求传输中保持刷新。在解决对于特定 ID 段没有查表的问题上，每一个节点刷新在上一个小时内没有被任何节点请求的 bucket。刷新 bucket 是通过先随机选择一个 ID 到 bucket 段然后使用在那个 ID 上使用 node lookup。

为了加入网络，节点 u 必须有 contact 到已经存在的节点 w, 节点 w 通常是每个网络上的 bootstrap 节点。u 将 w 插入在适当的 k-bucket。u 之后会通过自己的 Node ID 进行查找。最终，u 刷新了所有的 k-bucket，比现在临近点更远。在刷新中，u 填充自己的 k-bucket 的同时，也将自己添加到其它节点的 k-bucket。已经存在的节点，通过相似的使用其周围子树的完全信息，会知道新节点会存放哪个 KV 对。任何节点都要了解新节点，因此发送 STORE RPC 来传送相关 KV 对到新的节点。为了避免重复使用 STORE RPC 保存同样的内容到不同的节点，节点只会当他自己的 ID 比其他节点到键值更近才会传送 KV 对。

2.9 键值重新分发

为了保证 KV 对的持久性，节点必须定时重新分发键值。要不然，在两种情况下可能会导致查找有限键值失败：

首先，部分 k 节点先得到 KV 对，当 KV 对分发的时候他们离开了网络。

其次，新的节点可能用比现在分发键值的节点更近的 ID 加入网络。

在这两种情况下，拥有 KV 对的节点必须重新分发来保证它始终对 k 个最近到键值的节点全是可用的。为了补偿节点离开网络，KAD 每小时重新分发 KV 键值一次。KAD 提出了两个用来优化键值重新分发的机制，同时保证高效使用计算资源：

首先，当节点接收到给定 KV 对的 STORE RPC 时，它假定 RPC 也给其他 k-1 个最近节点发出了，然后接收方将不会在接下来的一小时内重新分发 KV 对。这个保证了只要重新分发的时间段没有完全同步进行，每一个节点将会对应给定 KV 对每小时重新分发一次。

其次，优化避免了在重新分发键值前节点查找。在 KAD 中，节点拥有在至少 k 节点的周围子树的完全信息。如果在重新分发 KV 对前，一个节点 u 在这 k 节点子树中刷新了所有 k-bucket，它将会自动地通过给定的键值找到 k 个最近的节点。

3 Genaro 公链

3.1 可搜索加密

在讲整个公链之前，我们需要介绍一个公链中使用到的技术，也就是可搜索加密（Searchable Encryption），以及其在公链的加密存储中起到的作用。首先，可搜索加密，顾名思义，就是基于密文进行搜索查询的方案，用密码学来保障用户的数据隐私信息。其次，可搜索加密拥有四点优势：可证明安全、控制搜索、隐藏查询以及查询独立，在去中心化存储领域，可搜索加密可以提供对于个人数据隐私的保障性。第三，可搜索加密可以做到十分简单和快速， 也不需要大量的预交互的情况下，可以做到实时性较高的操作。

在数据保护中，个人信息的隐私性是需要得到优先考虑的，其次要支持的就是动态数据的改动，也就是 Dapp 对于去中心化数据存储的修改。可搜索加密技术是 Genaro 的 I/O 流协议(GSIOP)中不可缺少的一环。

在可搜索加密的常用场景中，可搜索加密一般分为如下四个阶段：

(1) 数据加密阶段：数据拥有者在本地使用密钥对明文数据进行加密，然后将加密后的数据上传至数据存储托管单元。
(2) 生成检索陷门阶段：用户使用密钥和关键字生成对应的检索陷门，并将该陷门发送给数据存储托管单元，其中检索陷门能保密其包含的关键字内容。
(3) 密文检索阶段：根据收到的关键字检索陷门，数据存储托管单元对密文进行检索，并将满足检索条件的密文发送给用户，执行过程中数据存储托管服务器不能获得除检索结果之外的更多信息。
(4) 密文解密阶段：用户从数据存储托管单元获得返回的密文后，使用密钥解密出相关数据。

根据密钥类型不同，可搜索加密可进一步分为可搜索对称加密（SSE: Searchable Symmetric Encryption）和可搜索公钥加密（PEKS，Public-Key Encryption with Keyword Search）。Genaro 当前考虑的场景，是为数据拥有者自身提供加密数据搜索的功能服务，因此实现的方案是基于 SSE 的。在这里我们暂时不对 PEKS 进行展开讨论。SSE 方案由五个算法组成：

(1) $K = KeyGen(k)$：输入安全参数 k，输出随机产生的密钥 K。该操作通常在数据拥有者端本地执行。
(2) $(I, C) = Enc(K, D)$：输入密钥 K 和明文文件集 $D = (D1, D2, \dots, Dn)$，输出索引和密文文件集。该操作在数据拥有者端本地执行。
(3) $Tw = Trapdoor(K, W)$：输入密钥 K 和关键词 W，输出关键词对应的陷门。 该操作在数据拥有者端本地执行。

(4) $D(W) = Search(I, Tw)$：输入索引 I 和待搜索关键字的陷门 Tw，输出包含关键字 W 的文件的标识符集合。 Search 操作在 Genaro 中由密钥分发控件执行。

(5) $Di = Dec(K, Ci)$：输入密钥 K 和密文文件 Ci，输出解密后对应的明文文件 Di。该操作在数据拥有者端本地执行。

(6)

3.2 Genaro I/O 流协议(GSIOP)

在区块链的数据存储中，链上存储的尺寸限制以及链下数据无法自证的特性一直是 Dapp 设计的瓶颈。Genaro 自身有链外到链上的入口，而存储部分就是承担这部分的功能。所以数据的持有者可以通过在 Genaro 上初始化的办法，通过 Genaro 流协议在链下存储数据而达到链外数据标示的做法。相对于现有的预言机模式和原生的链上直接存储来说，可以在拥有更大存储空间的情况下对数据的所有权和私密性进行保障。而通过这样的标示，Genaro 流协议是为了通过加密的存储达到多方对文件改动的同时保证私密性。这也是 Genaro 一直认为 Dapp 存储的信息需要一定私密性（即不公开性）而设计的。GSIOP 是通过使用发挥 Genaro 存储的特性，将其中的加密存储的部分定制为一个可以广泛使用的协议，从而达到可以将数据流加密在 Genaro 存取的同时获得相关加密算法带来的特殊函数功能。在区块链实际使用中，链上数据的尺寸和链下数据的同步性是很难保证的，换句话来说，如果数据保留在链上，数据的大小一定是要考虑的问题，由于每一个区块能够保留的数据是有限的，增加数据大小会影响到块的大小。

链下数据的尺寸虽然不需要特殊考虑，但是对于数据的同步是需要特殊处理，也就是说，如果数据不加任何的确认机制，本身是无法直接上链的。

首先，链下的数据是否能够保障确认性，由于链下链上的身份的不唯一性，链下身份确认后的上链操作很难判断其是否是通过有公信力的部分来操作的；其次，链下的数据可以上传时可以被不同的节点多次操作处理，由于数据公开，其很难被分类挑出哪些部分是可用的。

举例说明，某恶意节点可以通过 A，B，C 账号对一个公开数据进行改动，而不会得到任何的惩罚，即便将 A，B，C 账号同时禁止改动，也会有新的账号 D 进行恶意操作，所以链下数据在上传之前的用户需要被有区别的划分出来。

现在的解决思路包括以闪电网络、雷电网络为主的链下解决方案，在数据链下到上链的时候，是通过类似于自己先验证再让链上节点验证的方式。链下链上通过一个类似于中心化的方式在提供。我们更希望将链下的数据通过一个协议来传，再通过同构的方式将这部分数据可以在链上同步。简单来说，就是给予链上链下一个数据的通道，用户只需要通过在 Genaro 上初始化数据，之后的赏罚就是通过链上治理解决。

GSIOP 就是通过加密的方式保证了链下存储可靠性的协议方案。首先通过加密算法将在数据初始化存储单元的时候对可更改者做了账号选择性，在存储账号中分为：数据拥有者，数据部分更改者以及数据监察者。

分别对应了三个在数据中不同的作用的用户。数据拥有者，拥有全改的密钥，拥有者拥有全改的操作，这部分数据在系统中被拥有者维护，所以可以做到数据与特定存储地址相关，并且数据拥有隐私的选择。数据部分更改者，这部分是数据拥有者选择可以更改相关部分数据的，数据可以被分段让不同的人改不同的部分，让数据更改者的地址与数据更改动作对应，这样的话能发现对数据改动出现问题的人。数据监察者，拥有查看数据并不更改数据的权利，在整个系统起到的作用就是对于相关坏地址上传的谬误数据进行排除。在初期，会有数量较多数据拥有者，规定修改者和监护者，在系统迭

代后，一些稳定提供数据的数据拥有者会有特定的账号，也有可以被相信的修改者和监护者。

在第一个版本里，GSIOP 先对 uint 的数值进行改动，而对于这样的功能可以引申出第一类应用，也就是一个加密的侧链系统。数据拥有者在这个系统中就是持有 GNX 的用户，他可以通过锁定的方式来进行跨链的 GNX 使用，而 GNX 就是 Genaro 公链上的 Coin。可以通过将一个账户拥有在除 GNX 外的另一个链的账户信息的用户充当数据部分更改者的方式来接受关于另一个链使用 GNX 的信息并加密储存在储存网络中。最后另一个链的全节点充当监察者，他可以看到的确有信息通过这样的方式进来，并将中间的步骤储存。在最后用户要收回剩余 GNX 的时候，只需要调用所有相关的存储信息，就可以保证简单的跨链操作。

在之后的版本里，GSIOP 会提供 string 的调用改动操作，相关的功能引申应用也会在之后写到。

由于相关存储功能的加入，Genaro 在智能合约以及虚拟机部分的相应改动。Genaro 将以兼容现有 EVM 的方式，对现有的栈式虚拟机的操作码进行增加，以及在语法中增加相关的类型和指令。最后通过特殊交易来更改链中状态数据库中的状态，下面就是具体实现流程。

3.3 基于代理重加密的文件分享

Genaro 的目标之一就是让每一个使用 Genaro 的使用者都可以在加密的环境下进行分享。

现在能够做到这种加密环境下的分享主要有两类：一类是使用别人的公钥直接加密。另一种则是代理重加密（proxy re-encryption schemes）的办法。这两类加密算法都是可以实现我们对数据加密分享的特性的，相对来说，代理重加密是 Genaro 当前准备采纳的在公钥加密之外更加有可扩展性的一种做法。此外，Genaro 也在考虑及评估在将来使用基于属性加密的方法，实现加密数据面向具有不同属性的多用户群组分享[8]，达到更具细粒度的分享效果。

代理重加密是一个非对称的加密方案，允许一个代理，在不解密的前提下，将 A 的公共密钥下的密文转换为 Bob 的公钥下的密文。为了做到这一点，代理被赋予一个重新加密密钥$r_{A\to B}$。

代理重新加密的概念在 1998 年由 Blaze 等人被引入 [9]。他们的建议，通常被称为 BBS 方案，是双向的（从$r_{A\to B}$获得$r_{B\to A}$很容易）和多跳（重新加密过程是传递性的），但不抵抗共谋。Ateniese，Fu，Green 和 Hohenberger 提出[10]，基于双线性配对的新代理重新加密方案。特别是，他们提供了一个初步的基本方案（AFGH 方案），随后在整个论文中扩展以支持附加功能。由于 AFGH 方案拥有单向，单跳和抵抗共谋等优势，极具代表性，该方案即被 Genaro 所采用。

3.3.1 AFGH 方案

该方案基于 ElGamal 加密系统。令$\mathbb{G}_1$与$\mathbb{G}_2$为两组素数阶q,存在双线性映射$e:\mathbb{G}_1\times\mathbb{G}_2\to\mathbb{G}_2$；全局参数为$g\in\mathbb{G}_1$与$Z=e(g,g)\in\mathbb{G}_2$。

- 键值生成：用户 A 选择一个随机整数$a\in\mathbb{Z}_q$，并且产生其密钥与公钥：$s_A=a$与$p_A=g^a$。
- 重加密键值生成：用户 A 使用用户 B 的公钥和自己的私钥，计算出重加密密钥$r_{A\to B}=(P_B)^{1/s_A}=g^{b/a}\in\mathbb{G}_1$。
- 第一级加密：任何人都可以使用用户 A 的公钥p_A对发给它的信息进行加密。为了加密一个信息$m\in\mathbb{G}_2$，需要选择一个随机整数$k\in\mathbb{Z}_q$，计算出$c=\left(e(p_A,g^k),mZ^k\right)=(Z^{ak},mZ^k)\in\mathbb{G}_1\times\mathbb{G}_2$。
- 第二级加密：用户可以用以下公式计算出可以重加密的第二级密文$c=\left(p_A^k,mZ^k\right)=(g^{ak},mZ^k)\in\mathbb{G}_1\times\mathbb{G}_2$。

- 重加密：任何拥有重加密密钥$r_{A\to B}$可以将用户 A 的二级密文转换为用户 B 的一级密文，计算方式为：$c_B = \left(e(\alpha, r_{A\to B}), \beta\right) = (Z^{bk}, mZ^k) \in \mathbb{G}_1 \times \mathbb{G}_2$。
- 一级解密：用户 A 使用自己的密钥s_A将密文$c_A = (\alpha, \beta) \in \mathbb{G}_1 \times \mathbb{G}_2$转化为原始信息$m = \frac{\beta}{\frac{1}{\alpha^{s_A}}} = \frac{\beta}{\alpha^{1/a}}$。
- 二级解密：为了解密密文$c_A = (\alpha, \beta) \in \mathbb{G}_1 \times \mathbb{G}_2$，用户 A 的计算方式为：$m = \frac{\beta}{e(\alpha, g)^{\frac{1}{s_A}}} = \frac{\beta}{e(\alpha, g)^{1/a}}$。

该方案使用两个不同的密文空间：第一级密文用于不可委托的消息，而二级密文可以通过重加密转化为一级密文。AFGH 方案具有以下属性：

- 单向：重加密密钥$r_{A\to B}$不能用于导出相反$r_{B\to A}$。这个属性在非对称信任关系的环境中非常有用。
- 抗共谋：如果代理和用户 B 勾结，他们无法提取 Alice 的密钥。在大多数情况下，他们可以计算出弱秘密$g^{1/a}$，但是这个信息无法带来任何收益。
- 单跳：这个方案中重新加密是无法传递的。重新加密过程将密文从一个空间转换到另一个空间，所以此过程不能重复。

3.4 新的 VM opcodes

在 opcodes 的选择上，Genaro 使用在原有 ethereum VM 的基础上增加一些新的指令，这些指令是与存储相关的。对于每个指令，我们依旧沿用α来代表往栈上增添额外部分的数量以及δ来表示从栈上减去的部分。

表 1. Genaro 的 opcodes

Value	Mnemonic	δ	α	Description
0x5c	STORAGE_GAS	2	1	获取存储空间存在时长
				$\mu_s'[0] \equiv \{^{\sigma[\mu[0],\mu[1]]_{sg} if \sigma[u[0] mod 2^{160},\mu[1],\mu[2]]}_{0 otherwise}$
0x46	STORAGE_GAS_USED	2	1	获取存储分片大小 $\mu_s'[0] \equiv \{^{\sigma[\mu[0],\mu[1]]_{gu} if \sigma[u[0] mod 2^{160},\mu[1],\mu[2]]}_{0 otherwise}$
0x3f	STORAGE_GAS_PRICE	2	1	获取存储备份数 $\mu_s'[0] \equiv \{^{\sigma[\mu[0],\mu[1]]_{gp} if \sigma[u[0] mod 2^{160},\mu[1],\mu[2]]}_{0 otherwise}$
0x5d	SSIZE	2	1	获取存储空间大小 $\mu_s'[0] \equiv \{^{\sigma[\mu[0],\mu[1],]_{ss} if \sigma[u[0] mod 2^{160},\mu[1],\mu[2]]}_{0 otherwise}$
0x47	SENTINEL_HEFT	1	1	获取用户的 heft 属性 $\mu_{sh}'[0] \equiv \sigma[\mu[0]]_{sh}$
0x21	DATAVERSIONREAD	3	17	根据 dataVersion，file，address 获取侧链 dataVersion 对应版本状态
0x22	DATAVERSIONUPDATE	2	0	根据 fileID，address 设置开启 KDC 同步侧链状态到公链 $\mu_s'[0] \equiv \{^{1 if \sigma[u[0] mod 2^{160},\mu[1],\mu[2],]_{du} \neq \emptyset}_{0 otherwise}$

3.5 新指令

添加新 opcodes 之后，Genaro 增加了新的指令到原有 solidity 语言中，并保持兼容的同时增加对数据的操作可能。

值类型-增加部分：

存储地址 d_storage: 保存 32 字节的值，对应存储中的空间地址。存储地址类型也有成员，作为所有存储部分的基础。

存储地址成员：

- read 和 update

调用方式：

```
1. d_storage ds = 0x1234567890123456789012345678901 2;
```

```
//读特定空间的特定版本的数据
uint256 result = ds.read.version(0x03);
uint256 newresult = result + 3;
assert(newresult < expectation);
return;
//开启数据定时更新功能
assert(ds.update);
return;
```

存储地址可查询的内容，由于在智能合约中，可以通过以下作为判断点来停止合约执行：

```
d_storage ds = 0x123456789012345678901234567890123456789012;
uint32 bucketID = 0x123456789012345678901234567890123456789012;
mapping(d_storage => uint32 []) public file;
file[ds].push(bucketID);
//查看是否大于 3 个月
sgas(ds,bucketID)< 3 month;
//查看是否备份数大于 6
require (ds.slice >6);assert(ds.update);
//查看是否分片大小大于 1GB
require(ds.sused > 1 GB);
// 查看空间是否是 3GB
require(ds.ssize == 3 GB);
// 检查一个地址所拥有的文件指纹总数是否大于 10000
Check_sentinel (user_address)> 10000;
```

3.6 新特殊交易

增加特殊交易是为了适应公链的押注以及存储验证属性的，在系统中分别通过增加以下特殊交易：

(1) 押注交易（用户发起的交易），其作用为支持 PoS 中押注的动作。用户使用自己账户中的余额进行押注，押注后，押注的金额将被扣除变为用户的 stake 属性。

(2) 存储权重同步交易（存储网络发起的交易），其作用为将存储结果同步在公链作为辅助。存储网络指定的特殊账户发起 heft 值的同步交易，将 heft 同步至链上。

(3) 用户购买存储空间的交易，其主要作用为支持用户在链上直接支付存储容量对应价格。在交易中根据用户提供的空间参数确定购买金额，成功扣除金额后表示交易成功。在交易过程中可将一些过期历史存储空间的记录进行清理，在购买存储空间后，可根据购买额度绑定赠送一些流量。

(4) 用户购买流量的交易，流量在存储中代表着上传下载流量。用户购买的流量将累加在总流量之上，比如购买了 1G 空间，购买 10G 流量就是可以下载 10 次。

4 共识治理结构

在介绍 Genaro 的共识之前，首先普及一下共识常识。共识基本上就是给分布式的处理器确定一个特定值的办法，共识简单的来说就是对一个特定值来投票。给定一系列处理器，每一个拥有一个初始值：

- 所有的非错误处理最终达成一个值；
- 所有处理决定在这个值上做事情；
- 被确定的值需要被某些处理器提出。

这三个特性分别称为确定性(Termination)，同一性(Agreement)，可证明性(Validity)，任何的算法拥有这三个特性即可以说解决了共识的问题。[16]

确定性和同一性是相对来说很容易自己解释的，我们只是不需要从有错节点获取任何操作，即假设他们本身就是有错的。对于可证明性，我们需要去除那些无论初始值是什么都给出错误选择的节点。这种算法肯定是满足确定性和同一性的，但是空洞无用。

4.1 FLP 不可能特性

FLP (Fischer, Lynch and Patterson)不可能性 [11]证明实际上涉及一种稍弱的共识形式：对于终止，仅仅是一些非错误的过程决定就足够了。需要注意的是，有一个强大的进程来一直决定价值是不够的，

因为这个过程可能会失败, 另一个将不得不采取它的地方, 甚至微弱的终止来满足要求。

FLP 不可能特性的结论是：在非同步情况下的错误检测，在检测无边界的处理器处理它的问题然后返回特定值是不可能的，即无法判断这个处理器是宕机了还是长时间没有回复。在研究中证明了没有异步算法可以在保障宕机错误的时候能够亦保证确定性，而且在没有宕机的情况下也是无法保障的。

FLP 不可能特性证明了，在没有任何验证确定“0-1”值的异步算法能够保证当宕机错误发生时能立刻终结当前进程，而且在没有类似错误发生的时候依旧是对的。因为在异步的情况下，异步分布式系统就会将系统中的状态收敛到一个“双值状态”，在这种状态下，任何的 0 输入或者 1 输入都会破坏其中的平衡，但是在最后都会回到双值状态。这样，系统就被证明是无限而且非终结的方式运行的，所以系统会走入一个有限循环，这个循环的结束就是完全走出这样的状态，也就是说需要在之间加入一个可以导向状态的一个工作。Fault-Tolerant 的共识是可以通过 vote 出来，但是我们依旧很难证明他们可以终结了，所以如果需要的是“0-1”之间结果为 1 可能性的保障。我们可以通过一个流程证明，但是如果需要逻辑上完全完美的保障还是达不到的。在 Genaro 共识的设计中，考虑到没有一种情况在不添加额外信息的条件下能够破坏双值状态，我们需要在链式共识中设计分叉处理办法的时候是一定需要通过额外信息的添加的，Genaro 在这里添加的是与存储相关的信息。

4.2 CAP 定理

Genaro 在区块链共识的设计中，也就是一个特定的分布式计算机数据系统中，必须遵循 CAP 定理。CAP 定理又被称作布鲁尔定理，定理中证明了对于分布式电脑系统不可能同时保证的三个部分：一致性(Consistency)，可用性(Availability) 和分区容错性 (Partition tolerance) [12]。

这个一致性不是主要用于 ACID 数据库一致性，具体表现特征是每次的读都获得最新的写的结果或者是一个错误。可用性是指在系统中每次通过请求都会获得回复，非错误，但是并不需要要求这个是最新的写结果。而分区容错性在里面就是当在网络中有一些节点之间的信息尽管在丢失的情况下，系统依旧进行工作。在设计共识的时候，CAP 定理就需要被提前考虑到，也就是说，满足一致性的情况下，区块的终结(Finality)就可以被同步执行，而满足可用性的情况，只能做到一个异步执行的终结系统，在这个系统中，FLP 不可能特性就需要被强调。即在 CAP 中，容错性得到保障后，选择同步（类 BFT 共识）或者异步（Nakamoto 链式共识）。

4.2.1 系统容错

作为在 CAP 中不能同时保障里面，唯一缺少不了的部分，我们需要有针对性地将它单独拿出来进行讨论。在 Genaro 的设计中，我们是要优先保证链的可用性，就是达到公链的多节点高并发的标准，在这种情况下，就需要放弃一致性的特点，也就是要保留系统容错性。一般降低容错的办法是通过特殊数据或者状态检查上，也就是说通过一种特殊广播也就是特殊交易；以及另外一种 timeout 或者特定的协议来做容错。

在分布式存储系统中，当系统存在故障导致信息丢失或者重复且不存在恶意节点（即无错误信息）时，其共识问题被称为 CFT (Crash Fault Tolerance)。在存在恶意节点时，则称为 BFT (Byzantine Fault Tolerance)。为了解决分布式系统内常见的拜占庭容错性的问题，Genaro 借鉴了业界最经典的 PBFT (Practical Byzantine Fault Tolerance) 算法，由 Miguel Castro 和 Barbara Liskov 在 1999 年提出[13]。该共识的核心就是设计一套异步的方法来满足一致性和拜占庭容错。那么我们在设计中在拜占庭容错的兼容性上需要对 PBFT 进行理解和嫁接。在之前对 CAP 定理的阐述中，已经说明链式共识区块链也是一种 AP 类型的分布式系统，也就是说出了在一致性上不同，在异步情况下的拜占

庭容错的部分是类似的，下面就简单的对 PBFT 进行说明。

PBFT 假设一个异步分布式的系统模型，并且假设节点失效独立发生。PBFT 通过加密技术来防止欺骗攻击和重放攻击，并且检测受损信息。信息中包含了公钥签名，消息验证编码以及通过无碰撞哈希函数生成的消息摘要。所有节点都知道其他节点的公钥以进行签名验证。

PBFT 算法是一种状态机复制算法：即服务作为状态机进行建模，并且可以在分布式网络中不同节点进行复制。每个状态机的副本包括服务状态及服务的操作。副本在一系列被称作“视图(view)”的部署过程中运作。在一个视图中，一个副本作为主节点，其他副本作为备份节点。令所有副本的集合用 $\mathbb{R}$ 表示，f为有可能失效的副本个数，且$\mathbb{R} = 3f + 1$。当主节点失效时，视图将被更换。PBFT 算法通过以下步骤实现：

(1) 用户向主节点发送请求调用服务操作，并且用时间戳 t 来保证客户端请求只会执行一次；

(2) 主节点将请求广播给备份节点。主节点在收到用户请求后，将触发一个三阶段协议：预准备、准备和执行。预准备和准备阶段用于将同一视图中的请求进行排序，无论主节点是否失效。准备及执行阶段用于确保在不同师徒之间的请求是完全排序的。

(3) 副本执行请求并且向用户发送回执$<reply, v, t, c, i, r>$，其中包括了当前的视图编号 v，v是视图编号，t是时间戳，i是副本的编号，r是请求执行的结果。使得用户能够通过跟踪视图编号来推算出当前主节点编号。用户可以通过点对点消息向当前它认为的主节点发送请求。

(4) 用户需要等待$f + 1$个不同节点回馈相同的结果，并且将该结果作为操作的过程。反馈信息需要有相同的时间戳 t 和执行结果 r。这是由于失效的副本节点不超过f个，因此需要$f + 1$个副本的一致性响应。当用户没有在有限时间内收到回复时，该请求将向所有的副本节点广播；若该请求已经在副本处理过了，则副本向用户重新发送结果；如果主节点未将该请求向网络进行广播，则最终将被足够的副本节点认为主节点失效，并且导致视图变更。

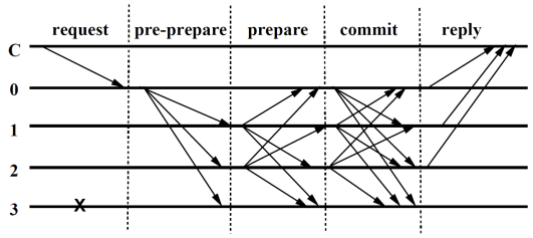

图 2. PBFT 算法流程

PBFT 的算法弹性是最优的，即在失效节点不超过f的情况下可以同时保证安全性和活性。

安全性是指副本复制服务满足线性一致：就像中心化系统一样执行原子化操作。安全性规定了失效副本的上限，然而并不限制失效用户的数量。这是由于失效客户点的行为将会被非失效副本一致对待。对于失效用户可能带来的破坏，例如失效用户向分享文件系统中写入的垃圾数据，PBFT 算法通过访问控制审核用户并且拒绝用户发起无权执行的操作。用户的访问权限可以被服务改变，且改权限撤销操作可以被所有用户一致观察到。

PBFT 并不依赖同步性以确保安全性，因此同步性必须用以保证活性，否则算法可以被用于在异步系统中实现共识。在本算法中，所有的用户最终将受到对其请求的回应，只要失效副本的数量不超过f，delay(t)，即 t 时刻发出的消息到它被目标最终接收的时间不会无限增长。即我们在设计中，需要对时间进行控制。

由于 PBFT 本身在公有链中会在相互通信中对节点的总量有所限制，而其异步 BFT 的思想就是时间控制与节点控制，在对 Genaro 设计异步的 BFT 兼容特性中提供了设计思路。也就是说，节点

同步数量需要和采样的时间进行一个相关的测量和控制。

4.3 点对点加密存储（SPoR）

Genaro 的最终目的是要有一个可以加密分享的公有链，那么首先我们需要保证的是加密分享的文件本身是可以被证明存储完整性的，而不是存储后无法找回。基于这样的选择，我们需要一个可靠有效的办法去验证文件存储完整性，及其能被完整取回的相关证明。Genaro 在这里选择的是具有代表性的 SPoR 算法[6]，该算法提供了一套完备的，可证明安全的，文件存储完整性验证的理论体系。我们可以使用这套存储完整性验证算法为之后即将介绍的 Genaro Chain 样式共识算法提供重要的辅助信息，达到有机融合，优势互补。在这里我们重点强调 Sentinel，也就是说通过存储获得的信息是整个流程中的关注点，相对于 PoW 以及其他 PoW 替代共识机制，这种设计会消耗更低，而在本质上这是做了个验证哈希组合的共识。这部分辅助信息在之后我们做链上治理中提供了对应的权重，也就是说我们的治理并不是简单押注和线下的治理方式。我们是通过存储完整性验证算法，去审计链上存储节点的贡献，并结合押注的线上治理方式，来逐步优化并维持系统的稳定性。这种方式的优势在于其不会受链下的一些不确定因素对系统的稳定性产生干扰。SPoR（Sentinel Proof of Retrievability），是传统 PoR 的一种算法，通过设立特定的文件指纹（Sentinel）来侦测数据可验证性。文件指纹是一个随机值的区块，并且通过加密使其与文件区块无法区分(indistinguishable)。SPoR 协议结构包括如下三部分[6]：

- 建立阶段：验证节点 V 对文件 F 进行加密，并将文件指纹植入文件 F 的随机位置，其中指纹的检查值是随机构建的。令$\tilde{F}$为植入指纹后的文件。
- 验证阶段：为了验证存储节点保有文件 F，节点 V 选择$\tilde{F}$中部分指纹的位置，并且要求存储节点返回相应的指纹值。
- 安全：因为文件 F 已经被加密，且文件指纹值是随机的，存储节点无法区分指纹和原文件的某一部分。因此我们实现了如下的特性：如果存储节点删除或修改了文件实体的一部分，有很大的概率其指纹的相应部分也会被修改。当验证节点查询和验证了足够的指纹后，它可以检测到存储节点是否修改或者删除了文件实体。

假设一个文件包含 b 个区块：$F[1],\dots,F[b]$。编码函数需要如下四个步骤：

(1) 纠错：我们将文件 F 切个成含 k 个区块的部分。对于每个部分，我们采用(n,k,d)纠错代码 C，使得每个部分扩展成为 n 个区块，并产生一个新文件$F' = F'[1],\dots,F'[b]$，包含$b' = bn/k$个区块。

(2) 加密：我们将对称密钥密码E应用于F'，得到新文件F''。由于需要在存储节点删除或损坏文件区块时可以恢复文件F，因此需要数据区块独立加密，即密码E可以独立地在文本区块上运行。

(3) 创造文件指纹：令$f:\{0,1\}^j \times \{0,1\}^* \to \{0,1\}^l$为一个简单的单向函数，则可以通过$a_\omega = f(k,\omega)$计算出一组文件指纹$\{a_\omega\}_{\omega=1}^{s}$。将以上文件指纹应用于$F''$，得到文件$F'''$。

(4) 置换：令$g:\{0,1\}^j \times \{1,\dots,b'+s\}^* \to \{1,b'+s\}$为一个伪随机置换。我们将g应用于文件$F'''$，得到输出文件$\tilde{F}$。

举例说明：选择大小为 128 的区块，且 AES 块的大小为 128 位，并产生足够大小的标记以防止暴力的指纹猜测攻击。让我们考虑使用通用的（$255,223,32$)-Reed-Solomon 码，借助于标准的“条带化”技术，我们可以获得GF^{218}上的$(255,223,32)$代码。因此一个文件由$n = 255$个区块组成。

让我们考虑具有$b = 2^{27}$块的文件 F，即 2GB 的文件。这个文件刚刚扩展错误编码为 14%，大小为$b' = 153,477,870$。假设我们添加了$s = 1,000,000$个指纹。因此，要存储的块的总数为$b' + s = 154,477,870$，即文件$\tilde{F}$中的文件总数。总文件扩展了大约 15%。

考虑$\epsilon = 0.005$，即攻击者已经破坏了数据块与$\tilde{F}$未使用指纹的1/2%。则$C = b' / n = 601,874$且$\mu = n\epsilon(b' + s)/(b' - \epsilon(b' + s)) \approx 1.29$（μ是每个部分损坏区块的平均数的上限）。通过[6]中定理 1，可得$Ce^{(d/2-\mu)}(d/2\mu)^{-d/2} \approx 601,874 \times e^{14.71}(12.41)^{-16} \approx 4.7 \times 10^{-6}$，即攻击者将文件呈现为无法检索的文件的概率。

假设$q = 1,000$，即验证者在每个挑战中查询 1000 个哨兵。由于哨兵的总数是$s = 1,000,000$，验证者可以在整个文件的生命周期内进行 1000 次挑战（每天一次，持续约三年的挑战）。发现损坏文件的可能性是至少每个挑战$1 - (1 - \epsilon/4)^q \approx 71.3\%$。当然，这并不是绝对的大，但对于大多数情况下可能就足够了，因为检测文件损坏是一个累积过程。举例来说，只有 12 个挑战检测失败概率小于百万分之一。

4.3.1 更好的取样机制

Genaro 的创新点在于其根据公有链的特性，对之前的 SPoR 算法进行了一些改动。在公链的运行中，本身会有伪随机的 hash 产生，而且在存储过程中，如果每一次检验的指纹是数量较小的话，矿工在进行攻击的时候成本相对会低一点。结合了这两个特性，Genaro 设计了一个符合公链存储的取样机制。在之前的 SPoR 中，我们使用了特定的文件指纹，也就是 sentinel 来保障文件存储回溯的有效性，在新的版本中，我们将使用更好的取样机制代替之前的做法，具体参考了[7]。在下文中，我们以文件指纹来替代英文 sentinel，粒度是指文件指纹的尺寸大小。具体有：

- 将原有单一的文件指纹变为随机多选择型，也就是将之前的每个文件指纹拥有更细的粒度，这样在选择矿工时更容易给长期做贡献的矿工。

  举例说明：在文件指纹粒度较大为 6 个的时候，矿工只要超过 1/6 的贡献能力就能够拿到一个单位的文件指纹，要想获得下一个需要贡献再加上 1/6，当粒度为 12 个的时候，之前贡献 1/6 的矿工虽然拿到 2 个文件指纹，但是想要多拿到一个文件指纹只需要增加 1/12 的贡献即可，所以粒度大的时候，对于想要做多贡献的好矿工是一种激励。

- 每次选择文件指纹的时候为多次选择统一校验，以随机选择指纹组的方式通过进行可回溯的检验。相对于单个指纹的检验，矿工（即在存储系统中负责接收存储的部分），相对容易预测出验证节点给出的验证问题，从而进行预先给出特定指纹而删除特定文件的重放攻击。为了技术上降低重放攻击的可能性，我们采取了随机选择指纹组校验的方式。这样可以通过随机组类的方式改进单一校验从而大大降低重放攻击的几率，并且通过更细粒度增加了攻击难度。

  举例说明，在原有的情况下，在单一校验时，矿工如果拥有 4 号文件指纹，那么他只需要将 4 号文件指纹提出来作为验证信号的回执就可以持续进行重放攻击，而不考虑是否将 4 号文件保存在自己手上，唯一能够确保的只是通过镜像产生的文件没有攻击。否则如果镜像方同时使用，那么将会产生该文件无法取回的情况；那在指纹组校验时，如果矿工拥有 4，5，6，7，8，9 号单个文件的 6 个指纹，一次需要 5，7，9 的指纹组和一次需要 4，7 的指纹组在验证信号给出所需要的

回执是截然不同的。在这种情况下做重放攻击的话，矿工需要生成 63 组，也就是说需要 3 次验证都失败就可以断定矿工并没有做贡献并排出此矿工，所以在攻击者难度上是大大增加了。而实际上我们开发上使用的是比案例中更细的粒度，可以做到增加难度。

- 随机数由区块生成的哈希的具体字节选择，这样可以保证无法预测出每次检验的文件指纹。在区块链中生成一个随机数相对来说选择有限，我们选择了每次区块演进生成的哈希的部分字节作为选择的依据，而这些字段相对来说预测的概率比较小。也就是说如果某个矿工想在特定块作恶，它首先需要在系统中通过押注以及贡献达到特定的标准，同时只是为了打重放攻击以及与此同时与他一起备份的矿工同时做，这种概率比较小，也需要很大的成本才能达到。

新的文件指纹组取样机制相比较于之前的取样机制，拥有更好的防治被矿工重放攻击可回溯证明的特性，以及相较于粒度较大的文件指纹拥有更好的激励好矿工的机制。

4.4 Chain 样式 PoS

在介绍链式 PoS 之前，简单对比一下其他主流的链式共识。现在最主流的就是链式 PoW，又称作中本聪式，最有代表性的就是比特币，以及以太币。现有的 PoW 已经将 CPU/GPU 类提到了 5000GH 和内存 I/O 类的工作量提到了 3000TH。Monero 以及 bytecoin 使用的 CryptoNight 被 ASIC 破解了，导致抗 ASIC 的算法越来越有限，比如 blake2b。随着 ASIC 的内存配置越来越高，相关 PoW 的 ASIC 化会越来越严重，作为一个新的公链如果延续使用 PoW，就会很轻易被 51%攻击。而且，在能源消耗上，是通过消耗能源的方式对整个系统进行成本控制，这个在 Genaro 设计初期就认为是不可持续的。基于这些考虑，Genaro 公链在设计最初就使用了链式 PoS 的设计思路。

在设计链样式共识 PoS (Proof of Stake)的细节上，需要考虑到分叉情况的选择问题，或者以很快的方式将分叉的部分很快排除。也就是说针对 PoS 设计中，我们需要考虑两个 PoS 的问题：没有抵押问题 (Nothing at Stake) 和长程攻击问题（Long Term Attack）[14]。

4.4.1 没有抵押问题

这个问题存在于 PoS 系统设计的原理在于，相对于 PoW 持续增高的工作成本，PoS 只需要通过抵押就可以进行共识参与共识。抵押中很容易做到可质押通证多的用户可以进行多个分支进行证明的步骤，而这种情况也是对与证明者来说收益最高的选择。

举例说明，在同时多个分叉的时候，攻击者同时对多个分支进行认证的收益是最大的，因为无论哪一个是最长链，他都会因此获得收益，而对于整个稳定系统的话，就是因为 FLP 不可能特性一直陷入双值状态的平衡。

我们在之前提到，系统中需要有提供状态导向的一个工作，Genaro 的这个工作是通过额外添加存储信息部分，这部分会在存储混合共识部分讲到。

4.4.2 长程攻击问题

这个攻击方式是通过 CAP 定理中我们保留可用性方式来进行攻击的。攻击者需要做的，是通过这种系统设计的属性进行攻击，由于保留了可用性，在这种情况下，一致性没有得到保障，也就是攻击者可以通过在很远的区块分叉这样的操作来进行攻击。在整个分布式系统中，系统也是认为很久区块前的回复刚刚得到回复，所以我们的新办法需要保证通过特殊交易或者状态检查来判定。Genaro 在这

里使用的是后者，这部分会在存储混合共识部分讲到。

4.4.3 存储叠加混合共识

出于需要解决上述两个链式 PoS 的问题的需要，Genaro 通过存储部分与公链结合的方式，通过优势互补的方式来做到混合共识。

首先，需要简单的重提一下存储部分相关组件，文件指纹组。文件随机指纹对，即 sentinel random pair, 我们通过该文件随机指纹对和 PoR 算法来保证上传到分布式存储里面的文件完整性，即其是可以通过定时随机抽查的方法被找回的。

举例来讲，就是上传文件的时候，文件通过增添指纹记录之后存储，通过冗余存储相关办法增加可靠性，然后定时会有检查存储方的相关操作，具体就是通过区块哈希得到的伪随机数来检查抽查指纹组记录中的随机指纹对，如果返回值是正确的，就是验证通过，错误或者超时操作都不会通过。将之前的做法混合到我们链中，可以看到我们如何解决之前提到的两个问题。

针对没有抵押问题，我们就是将总检验指纹记录组的次数提供到区块链共识操作中，由于检验的总记录是在链上同步的，所以在多个分支上面，只有在主链上才会同步到最新的也就是相对值较大的指纹检验记录总数。这部分额外信息就是之前提到推动双值状态平衡的信息部分，拥有这部分信息就可以对错误分支押注的攻击者进行额外惩罚。

针对长程攻击，和上一个解决办法一样，由于引入了指纹校验记录，在相对久的区块分支的部分可以通过总数判定，即便一开始分支可以演进，前进到验证点之后，由于指纹总数少于主分支而停止。

在解决这两个相关问题之后，一个新的问题随之产生，也就是说，如果长程攻击从上一个的指纹验证点开始进行攻击，只要通过快速出块的方式，会不会产生一种新的攻击方式。Genaro 将会介绍共识的治理结构，通过链上治理的方式来解决该问题。

4.5 数据治理结构设计

治理结构作为共识中不可缺少的承上启下的一环，也是其主要区别于互联网的一环。如果在一致性的网络中，系统中的验证可以通过在规定时间内得到证明的话，那么在可用性的网络中，系统中的每一个节点就需要遵循特定数据治理结构进行广播才能维持整个系统的容错特性。

在 Genaro 公链的数据治理结构中，通过在存储网络中获得的额外信息我们可以设计如下的治理方式：

- 每一个存储节点需要通过押注才可以进入系统贡献存储单元；
- 存储后通过存储贡献量和押注量组合的综合权重进行排名；
- 排名高的进入 101 个节点在公链中进行出块动作，101 个节点的选择我们是基于对于成功 PoS 案例中的学习，相对于 EOS 中 DPoS 的 21 个节点，101 节点拥有对新加入节点的包容性的同时增加新增节点洪水攻击的难度；
- 每一轮委员会出块都会使用轮流出块的方式，排名高的节点轮次会多。现有的出块操作，分为轮流出块以及随机数出块。在 ouroboros 中，对随机数进行了加密的沟通，所以按照一个特定的随机数进行出块，因为判定了 PoS 中都不一定是好的节点。Genaro 通过存储数据能被找回的额外信息来通过链上治理找出好节点然后轮询。通过 Genaro 的办法，可以更快的进行出块，因为省掉了随机数沟通的操作；
- 大节点可以选择小节点为自己贡献存储。

在这种治理结构下:

(1) 每一个存储节点想要开始在系统中做贡献需要押注来进入系统，在设计中，需要存储节点相

对稳定，这种对于准入机制加入门槛的机制可以做到在选择初级节点的时候已经得到了一部分的筛选，因为当数据需要被储存的时候，他需要的储存空间和映射备份的空间都需要相对健壮的存储节点才能保证之后的可被取回。

(2) 由于委员会成员是通过存储贡献和押注量的综合权重进行排名的，在链上系统的节点需要同时被存储网络和公有链两部分认可才能够被委员会接受。也就是说只靠质押就进场的方式不是完全行的通的，存储网络内的贡献排名保证了稳定对系统做贡献的节点可以长期留在委员会，从而避免只有质押的节点长期留在委员会。

(3) 大节点可以通过让小节点贡献存储的方式，用自己的小部分收益鼓励更多的小节点参与到存储网络的存储建设中；同时大节点也可以通过这种方式提升自己的排名。这样的参与机制可以让中型节点和大型节点有机会博弈。虽然对大节点，中节点，小节点有不同的分工，但是在总体的机制设计上是有公平的部分，中型节点可以和大节点有竞争，小节点可以出让自己的存储部分获得额外的区块奖励。

5 总结及展望

Genaro 网络目前的技术优势包括：（1）文件指纹组的设计，（2）独创的混合共识机制中将存储共识 SPoR 和链式 PoS 的结合的处理方式，（3）独特的链上数据治理方式，（4）加密算法带来的特定功能上 GSIOP 的使用特性，以及（5）将存储信息深入到 VM 栈的指令级别以及满足存储的特殊交易，(6)通过时空结合的办法，每次检测可被取回的时候打上时间标签保证之后只有在文件取回的时候一次性检查，减少多次询问矿工的次数，增大系统的网络稳定性。在未来，Genaro 会考虑更多的加密存储特性，包括用去冗余加密保证单个加密文件不多余处理来增加矿工收益，以及加密文件流存储使得用户可以进行边下载边看等流媒体操作。在共识方面，Genaro 将提供更加细粒度的共识改进，从而保障中小节点的参与度，增加新的 VM 的指令集来扩大增加更多的应用场景。

参考资料

[1] Schollmeier, R. (2002). A definition of peer-to-peer networking for the classification of peer-to-peer architectures and applications, *Proceedings of the First International Conference on Peer-to-Peer Computing*, IEEE.

[2] Bandara, H. M. N. D. and Jayasumana, A. P. (2012). Collaborative Applications over Peer-to-Peer Systems – Challenges and Solutions. Peer-to-Peer Networking and Applications. arXiv:1207.0790.

[3] Stoica, I., Morris, R., Karger, D., Kaashoek, M. F. and Balakrishnan, H. (2001). Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review*. 31(4), 149.

[4] Rowstron, A. and Druschel, P. (2001). Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, Heidelberg, Germany, 329–350.

[5] Maymounkov, P. and Mazieres, D. (2002). Kademlia: a peer-to-peer information system based on the XOR metric, *International Workshop on Peer-to-Peer Systems*, 53-65.

[6] Juels, A., and Kaliski, B. (2007). PORs: Proofs of retrievability for large files. *Proceedings of CCS* 2007, 584–97, ACM Press.

[7] Naor, M. and Rothblum, G. N. (2009). The Complexity of Online Memory Checking. *Journal of the ACM* (JACM), 56(1).

[8] Yu, S., Wang, C., Ren, K. and Lou, W. (2010). Achieving secure, scalable, and fine-grained data access control in cloud computing. INFOCOM, 2010 Proceedings IEEE, 1-9.

[9] Blaze, M., Bleumer, G. and Strauss, M. (1998). Divertible protocolsand atomic proxy cryptography. Advances in Cryptology—EUROCRYPT'98, 127–144.

[10] Ateniese, G., Fu, K., Green, M. and Hohenberger, S. (2005). Improved proxy re-encryption schemes with applications to secure distributed storage. In Proceedings of the 12th Annual Network and Distributed System Security Symposium, 29-44.

[11] Michael J.Fischer, Nancy A.Lynch and Michael S.Paterson, (1985) Impossibility of Distributed Consensus with One Faulty Process, Journal off Association for Computing Machinery.

[12] Lynch, N. and Gilbert, S. (2002). Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services, ACM SIGACT News, 33(2), 51-59.

[13] Castro, M. and Liskov, B. (1999). Practical Byzantine Fault Tolerance, In the Proceedings of the Third Symposium on Operating Systems Design and Implementation, New Orleans, USA.

[14] Bentov, I., Gabizon, A. and Mizrahi, A. (2016). Crytocurrencies without Proof of Work. International Conference on Financial Cryptography and Data Security.142-157

[15] Maymounkov, Mazieres, (2002) Kademlia: A Peer-to-peer information System Based on the XOR Metric. IPTPS '01 Revised Papers from the First International Workshop on Peer-to-Peer Systems. 53-65

[16] A.D.Kshemkalyani, M.Singal, Distributed Computing: Principles, Algorithms, and Systems, ISBN:9780521189842, paperback edition, Cambridge University Press, March 2011. 756 pages