访问控制对象的同步能力（共识数）：允许列表和拒绝列表的案例

DavideFrey ✉Inria、IRISA、CNRS、雷恩大学
MathieuGestin ✉Inria、IRISA、CNRS、雷恩大学
MichelRaynal ✉IRISA、Inria、CNRS、雷恩大学

arXiv:2302.06344v2 [cs.DC] 8 Aug 2023

摘要

本文借助 Herlihy 共识层次结构研究允许列表和拒绝列表对象的同步能力。文章将允许列表和拒绝列表规范化为分布式对象，并指出，尽管二者都可以视为更一般对象类型的特化形式，但它们本质上具有不同的同步能力。允许列表对象不要求参与进程之间进行同步，而拒绝列表对象则要求进程就特定的进程集合达成共识。随后，本文将这些结果应用于对使用允许列表和拒绝列表对象的匿名性保护系统进行更全面的分析。首先，介绍一种基于盲签名的电子投票方案。其次，使用拒绝列表和允许列表对象确定一种特定去中心化密钥管理系统的共识数。第三，介绍一种结合允许列表和拒绝列表对象的匿名货币转账协议。最后，利用本研究分析这些应用的属性，并突出说明它们在消息传递环境中能够实现的效率提升。

2012 ACM 学科分类 计算理论 → 分布式计算模型；安全与隐私 → 访问控制；安全与隐私 → 假名性、匿名性与不可追踪性

关键词与短语 访问控制、允许列表/拒绝列表、区块链、共识数、分布式对象、模块化、隐私、同步能力。

数字对象标识符 10.4230/LIPIcs.DISC.2023.39

资助 本研究部分由 PriCLeSS 项目和 SOTERIA H2020 项目资助。PriCLeSS 项目由法国国家研究署（ANR-10-LABX-07-01）旗下的 Labex CominLabs 卓越实验室资助。SOTERIA 获得欧盟地平线2020研究与创新计划的资助，资助协议编号为 No101018342。本文内容仅反映作者的观点。欧洲机构对本文所含信息的任何使用概不负责。

致谢 我们谨向匿名审稿人致谢，感谢他们提出富有见地的意见和建议，使本文得到了显著改进。

1 引言

区块链技术的兴起提高了公众和业界对分布式应用的兴趣，催生了众多将区块链应用于各种场景的项目。这些场景包括电子投票系统 [1], 命名服务 [2, 3], 身份管理系统 [4, 5], 供应链管理 [6], 或车载自组织网络 [7]。然而，将区块链当作能够解决众多分布式问题的瑞士军刀，恰恰凸显了人们对这些问题实际需求缺乏了解。由于规范不完善，这些应用的实现往往并非最优。

© Davide Frey、Mathieu Gestin 和 Michel Raynal；采用知识共享署名 4.0 国际许可协议授权。第37届国际分布式计算研讨会（DISC 2023）。编辑：Rotem Oshman；文章编号：39；页码：39:1–39:31 莱布尼茨国际信息学会议录 达格施图尔城堡——莱布尼茨信息学中心，达格施图尔出版社，德国

本文深入研究了分布式应用中广泛使用的一类问题，并提供了相关指南，帮助使用合理且足够的工具实现这些问题的解决方案。

与以往方法不同，该研究旨在了解系统进程之间实现特定分布式对象所需的同步程度。为实现这一目标，本文从 Herlihy 的共识数 [8]角度研究这类对象。该参数与共享内存分布式对象密切相关，在消息传递环境中则没有直接对应物。不过，在某些特定情况下，这些信息足以帮助我们更好地理解所分析的对象，从而提高消息传递实现的效率。例如，近期论文 [9, 10]表明，加密货币无需共识即可实现，因此也无需区块链。具体而言，Guerraoui 等人 [9] 表明，k资产转移的共识数为 k，其中 k 是可以从同一账户提取货币的进程数 [11]。类似地，Alpos 等人 [12]研究了 ERC20 代币智能合约的同步属性，并表明其共识数会随获准从同一账户发送代币的进程集合发生变化而改变。这两项研究考察了两种资产转移形式：经典形式，以及由 ERC20 代币实现的形式；后者允许进程动态授权其他进程。这些对象的共识数取决于特定且明确的进程。根据这项研究可以得出结论，共识算法只需在这些进程之间执行。因此，在这些特定情况下，对象共识数的信息可直接用于实现效率更高的消息传递应用。此外，尽管这项研究采用的是包含易崩溃进程的共享内存模型，其结果也可用于在消息传递环境中实现效率更高的拜占庭容错算法。本文旨在将这方面的认识拓展到更广泛的应用类别。

事实上，资产转移，无论是加密货币还是非同质化代币，都不是区块链生态系统中的唯一应用。具体而言，如前所述，电子投票 [1], 命名 [2, 3], 或身份管理 [4,5] 等多种应用会将区块链用作实现某种形式访问控制的工具。这通常通过实现两种通用对象来达成：允许列表和拒绝列表。允许列表提供一种选择加入机制。一组管理者可以维护一份授权方名单，即允许列表。要访问资源，某一方（用户）必须证明允许列表中存在一个与其身份相关联的元素。拒绝列表则提供一种选择退出机制。在这种情况下，管理者会维护一份已撤销元素名单，即拒绝列表。要访问资源，某一方（用户）必须证明拒绝列表中没有添加与其对应的元素。换言之，允许列表和拒绝列表分别支持对元素列表进行集合成员关系证明和集合非成员证明。

允许列表和拒绝列表对象执行的证明通常需要提供隐私保障。例如，Sovrin 隐私保护的去中心化身份管理系统（DIMS） [4] 为每个可验证凭证关联一个允许列表[1]，其中包含可以使用该可验证凭证的设备标识符。当设备使用凭证与验证者交互时，需要证明与其关联的标识符属于允许列表。该证明必须以零知识方式完成，否则验证者将获知设备的身份，进而可能将其作为用户的伪标识符。因此，允许列表和拒绝列表对象分别支持集合成员关系的零知识证明或集合非成员资格的零知识证明。

[1] 实际上，这是结合了允许列表和拒绝列表的一种变体，我们将在附录 A 中讨论。

尽管相似，允许列表和拒绝列表对象在证明机制的处理方式上存在显著差异。对于允许列表而言，即使管理者确实授予了某个进程访问资源的权限，禁止该进程访问资源也不会产生安全风险。因此，用户先被允许访问资源、随后被拒绝、之后又再次被允许访问资源的瞬态时期不会造成问题。相反，对于拒绝列表而言，在访问资源被拒绝后又重新获得访问权限会带来严重的安全问题。因此，拒绝列表对象被定义为具有额外的抗闪烁性质，以禁止这些瞬态时期。该性质是允许列表对象与拒绝列表对象之间的主要区别，也是它们具有不同共识数的原因。

现有系统 [1, 2, 3, 4, 5] 在区块链基础设施之上实现允许列表和拒绝列表对象，因此修改其内容需要网络级共识。如前所述，本文从共识数 [11]的角度研究这一差异。研究表明：(i) 允许列表对象的共识数为 1，这意味着允许列表可以在不依赖共识的情况下实现；(ii) DenyList 对象的共识数则等于能够对其执行证明操作的进程数，且只有这些进程需要同步。因此，这两种数据结构都可以在不依赖区块链所提供的网络级共识的情况下实现，从而为基于这些数据结构的应用实现更高效的实现方案铺平了道路。

总而言之，本文提出了以下三项贡献。

1. 本文将允许列表和拒绝列表形式化定义为分布式对象，并对其进行研究（第 3 节）。2. 本文分析了这些对象的共识数：研究表明，允许列表不需要进程间同步（第 5 节），而拒绝列表则需要其集合非成员证明的所有验证者进行同步（第 6 节）。3. 本文运用这些理论结果，为其最优实现提供了思路，具体包括 DIMS、电子投票系统以及匿名资产转移协议的实现（附录 B 和 C）。

据我们所知，本文首次从分布式算法的角度研究允许列表和拒绝列表。因此，我们相信，研究结果能够为确定近期使用这些列表的分布式对象的共识数提供有力工具，并帮助更高效地实现此类对象。

2 预备知识

2.1 计算模型

模型

令 Π 为由 N 个异步、顺序执行且易崩溃的进程组成的集合 $p_1, \cdots p_N$。顺序执行意味着每个进程每次只调用其自身算法中的一个操作。我们假设本地处理时间为瞬时，但系统是异步的。这意味着非本地操作可能需要有限但任意长的时间，而且不同进程的时钟相对速度未知。最后，进程可能崩溃：任意数量的进程都可能过早且永久地停止执行。发生崩溃的进程称为故障进程；否则称为正确进程。该系统采用实名制：每个进程由一个唯一的正整数标识，且所有其他进程都知道此标识。

通信

进程通过类型为 T 的共享对象进行通信。共享对象上的每个操作都关联两个事件：一次调用 和一次响应。对象类型 T 由一个元组 $(Q,\ Q_0,\ O,\ R,\ \Delta)$ 定义，其中 Q 是状态集合，$Q_0 \subseteq Q$ 是初始状态集合，O 是进程可用于访问此对象的操作集合，R 是这些操作的响应集合，而 $\Delta \subseteq \Pi \times Q \times O \times R \times Q$ 是定义进程如何访问和修改对象的转移函数。

历史与线性一致性

一个历史 [8] 是算法执行过程中调用和响应构成的序列。历史中没有匹配响应的调用，H，称为待处理调用。顺序历史是指第一个事件为调用，且每个调用（最后一个调用除外）都紧接着其对应的响应。子历史是历史中事件构成的子序列。历史 H 的进程子历史 $H|p_i$ 是 H 中所有关联进程为 p_i的事件构成的子序列。给定对象 x，我们也可以类似地定义对象子历史 $H|x$。如果历史 H 和 H' 满足$H|p_i = H'|p_i,\ \ \forall i \in \{1,\ \cdots\ \ N\}$，则二者等价。

本文将共享对象 x的规范定义为所有允许的子历史 $H|x$构成的集合。如果该集合中的所有历史都是顺序历史，我们就称其为顺序规范。合法历史是指满足以下条件的历史 H：对于该历史中的所有对象 x_i，$_H|x_i$ 都属于 x_i的规范。历史 H 的完成历史 H 通过为 H 中的所有待处理调用补上相应的匹配响应而得到。历史H 在操作上诱导出一个非自反偏序 $<_H$；也就是说，若操作 op_0 的响应先于操作 op_1的调用，则有 $op_0 <_H op_1$。如果$<_H$ 是一个全序，则该历史是顺序历史。正确进程执行的算法若总能在有限步数后终止¯，则称为无等待。若 H的某个完成历史 H 与某个合法顺序历史 S 等价，且 $<_H \subseteq <_S$，则历史 H 是可线性化的。

共识数

类型为 T 的对象（记作 cons(T)）的共识数，是在由 n 个进程组成的系统中，能够通过原子读写寄存器和类型为 T 的对象无等待实现共识对象的最大 n。如果类型为 T 的对象能够在任意数量进程组成的系统中无等待实现共识对象，我们就称该对象的共识数为 ∞。Herlihy [11] 证明了以下著名定理。

▶**定理 1.** 设 X 和 Y 是两种原子对象类型，且 $cons(X) = m$，$cons(Y) = n$，并且 $m < n$。在超过 m 个进程组成的系统中，无法通过类型为 X 的对象和读写寄存器无等待实现类型为 Y 的对象。

我们将使用原子快照对象和共识对象来确定一组 k 进程中拒绝列表对象和允许列表对象的共识数。单写多读（SWMR） [13] 原子快照对象是一个大小固定的数组，支持两种操作：快照和更新。Snapshot() 操作允许进程在一次原子操作中读取整个数组。Update(v, i) 操作允许进程将值 v 写入数组的第 i 个位置。Afek 等人展示了如何使用读写寄存器 [13], 无等待地实现 SWMR 快照对象，即该对象类型的共识数为 1。本文假设所使用的所有原子快照对象都是 SWMR 的。共识对象为进程提供一个一次性操作 $propose()$。当一个

进程 p_i 调用 $propose(v)$ 时，它提议 v。该调用返回一个已决定的值，并满足以下三个性质。

有效性：如果某个正确进程决定值 v，则该值 v 曾由某个进程提议；一致性：任何两个正确进程都不会作出不同的决定；以及终止性：每个正确进程最终都会作出决定。 k-共识对象是至多由 k 个进程访问的共识对象。

2.2 数论预备知识

密码学承诺

一种密码学承诺 是一种密码学方案，使证明者能够在隐藏值 v 的同时对其作出承诺。承诺方案是一种两阶段协议。首先，证明者使用函数$Commit$计算一个称为承诺的绑定值 C。$Commit$ 的输入为值 v 和一个随机数 r。证明者将这个具有隐藏性和绑定性的值 C 发送给验证者。在第二阶段，证明者向验证者揭示已承诺的值 v 和随机性 r 。随后，验证者可以验证先前收到的承诺 C 是否对应于所传输的值 v 和 r。这一承诺协议是零知识证明（ZKP）协议的核心。

集合运算的零知识证明

零知识证明（ZKP）系统是一种密码协议，允许证明者向验证者证明关于值 x 的某个布尔命题，同时不泄露有关 x 的任何信息。零知识证明系统针对复杂度类 $\mathcal{NP}$ 中的特定语言 $\mathcal{L}$ 进行初始化。证明机制以 $\mathcal{L}$ 为输入，并输出证明 π。知道 $\mathcal{L}$ 和 π 后，任何验证者都可以验证证明者知道一个值 $x \in \mathcal{L}$[2]。然而，验证者无法获知用于生成证明的值 x 。下文假设存在用于集合成员关系和集合非成员关系的高效非交互式零知识证明系统（例如，可以使用基于 [14] 的构造）。

3 允许列表和拒绝列表对象：定义

分布式允许列表和拒绝列表对象类型，是允许一组管理者控制对资源的访问的对象类型。此处使用“资源”一词来描述用户希望实现的目标，该目标受访问控制策略保护。如果用户成功证明自己有权访问资源，则会获准访问该资源。首先，我们介绍允许列表对象类型。然后，我们讨论 DenyList 对象类型。

允许列表对象类型是两种最常见的访问控制机制之一。要访问资源，进程 $p \in \Pi_V$ 需要证明自己知道某个元素 v，而该元素此前已由进程 $p_M \in \Pi_M$授权，其中 $\Pi_M \subseteq \Pi$ 是管理者集合，$\Pi_V \subseteq \Pi$ 是获准执行证明操作的进程集合。我们将 Π_V中的进程称为验证者。集合 Π_V 和 Π_M 是预先定义且静态的。它们是该对象的参数。根据对象的用途，这些子集可以很小，也可以包含 Π中的所有进程。

[2] 记号 $x \in \mathcal{L}$ 表示： x 是由语言 $\mathcal{L}$表达的问题实例的解

进程 $p \in \Pi_V$ 通过调用 PROVE(v) 操作，证明 v 此前已获授权。如果某个管理者此前调用过 APPEND(v) 操作，则称该操作有效。直观而言，我们可以将 APPEND(v) 操作的调用视为授权某个进程访问资源的行为。另一方面，由证明者进程 $p \in \Pi_V$执行的 PROVE(v) 操作，会向 Π_V 中的其他进程证明其已获授权。然而，这一证明本身还不够。证明的验证者必须能够验证，确实调用过有效的 PROVE 操作。为此，允许列表对象类型还配备了 READ() 操作。 Π 中的任何进程都可以调用此操作；它会返回所有已调用的有效 PROVE 操作，以及调用这些操作的进程身份。READ 操作返回的列表可以是 PROVE 操作列表的任意排列。 Π 中的所有进程都可以调用 READ 操作。[3]

可以为允许列表对象添加可选的匿名性属性，以支持保护隐私的实现。该属性可确保其他进程无法获知由 PROVE(v) 操作证明的值 v 。

允许列表对象类型被正式定义为一种顺序对象，其中每次调用后都会立即得到响应。因此，操作序列定义了一个全序，每个操作都可以通过其在序列中的位置来标识。

▶**定义 2**。 允许列表对象类型支持三种操作： 追加、PROVE 和 READ。这些操作看起来像是按某个序列 Seq 执行的，其中：

- 终止性。 正确进程调用的 PROVE、追加或读取操作始终会返回。追加 有效性。 进程 p 对 APPEND(x) 的调用在以下情况下有效**：** $p \in \Pi_M \subseteq \Pi$；**并且**$x \in \mathcal{S}$，其中$\mathcal{S}$ 是一个预定义集合。否则，该操作无效。PROVE 有效性。 **如果**进程 p 对 op =PROVE(x) 的调用有效，**则**： $p \in \Pi_V \subseteq \Pi$；**并且**一个有效的 APPEND($x$) 操作在 Seq 中出现在 op 之前。否则，该调用无效。*Progress*。 **如果**调用了有效的 APPEND(x) 操作，**则**Seq 中存在某个位置，使得任何进程 $p \in \Pi_V$ 在此位置之后调用的 PROVE(x) 操作都将有效。
- 读取 有效性。 进程 $p \in \Pi_V$ 对 op =READ() 的调用返回 Seq 中出现在 op 之前的有效 PROVE 调用列表，以及调用每个操作的进程名称。可选 —— 匿名性。 假设进程 p 调用了 PROVE(v) 操作。如果进程 p' 调用 READ() 操作，则 p' 无法获知值 v ，除非 p 泄露了其他信息。[4]

[3] 通常，允许列表对象在消息传递环境中实现。在这种情况下，读取操作是隐式的。每个进程都知道分布式对象的一个本地状态，并且可以随时检查该状态。在共享内存环境中，我们需要显式定义这一读取操作。[4] 匿名性属性只保护值 v。所考虑的系统是实名制的，因此进程的身份已为人所知。不过， v 的匿名性可以隐藏其他信息，例如向系统中的某个进程发送请求的客户端身份。第 7 节将讨论这些示例。因此，匿名性属性并不违背 READ 有效性属性，因为该属性只会泄露进程身份。

允许列表对象以仅追加的方式定义。该定义使其能够用于构建本文探讨的所有使用场景。然而，某些使用场景可能需要带有额外删除操作的拒绝列表。这一变体将在附录 A 中研究。

DenyList 对象类型可以非正式地描述为一种访问策略：与允许列表对象类型相反，所有用户最初都获准访问资源。管理者的职责是撤销此授权。管理者通过调用 APPEND(v) 操作来撤销某个用户的权限。用户使用 PROVE(v) 操作证明自己未被撤销。只有当管理者此前撤销了值 v 时，PROVE(v) 调用才无效。

Π 中的所有进程都可以通过调用 READ() 操作来验证 PROVE 操作的有效性。此操作与允许列表的读取操作类似。它会返回有效 PROVE 调用的列表，以及调用该操作的进程名称。

DenyList 与允许列表对象类型之间有一个显著差异。对于允许列表，如果用户在获得授权后无法立即访问资源，那么恶意行为就不会对系统造成损害——系统状态与先前状态等同。然而，对于 DenyList，未被纳入考量的撤销可能会使恶意用户访问资源并损害系统。换言之，在 DenyList 情况下，访问资源时必须考虑当前可用的、最新的撤销列表。

为此，DenyList 对象类型被定义为具有一项额外性质。抗闪烁性质确保，如果某个 APPEND 操作被某次 PROVE 操作考虑在内，那么后续每次 PROVE 操作都会将其考虑在内。抗闪烁性质与进展属性共同确保撤销机制尽可能即时。DenyList 对象被正式定义为一个顺序对象，其中每次调用后都会立即得到响应。因此，操作序列定义了一个全序，每个操作都可以通过其在序列中的位置来标识。

▶ **定义 3.** 该 DenyList 对象类型支持三种操作：追加、PROVE 和 READ。这些操作看起来就像按序列 Seq 中的顺序执行，其中：

- 终止性。 由正确进程调用的 PROVE、追加或读取操作总会返回。追加 有效性。 进程 p 对 APPEND(x) 的调用有效 **当且仅当**:$p \in \Pi_M \subseteq \Pi$; **且**$x \in \mathcal{S}$，其中 $\mathcal{S}$ 是预定义集合。
- 否则，该操作无效。PROVE 有效性。 **如果**正确进程 p 对 op =PROVE(x) 的调用无效，**那么**:$p \notin \Pi_V \subseteq \Pi$; **或者**有效的 APPEND($x$) 在 Seq 中出现在 op_P 之前。否则，该操作有效。PROVE 防闪烁。 **如果**正确进程 $p \in \Pi_V$ 对操作 op =PROVE(x) 的调用无效，**那么**Seq 中出现在 op 之后的任何 PROVE(x) 操作都无效。[5]

[5] 允许列表对象类型与 DenyList 对象类型之间唯一的区别是这项抗闪烁性质。正如第 5 节和第 6 节所示，允许列表对象的共识数为 1，而 DenyList 对象的共识数为 $k = |\Pi_V|$。因此，共识数上的这种差异完全是由抗闪烁性质造成的。一个开放问题是，这项性质的某种变体是否能够将任何共识数为 1 的对象转变为共识数为 k 的对象。

进程	操作	初始状态	Res- 应答	最终状态	条件
$p_i \in \Pi_M$	追加(y)	(列出值 $= \{x \in S\}$, 证明 $=(\{(p_j \in \Pi,\ \widehat{S} \subseteq S,\ \mathsf{P} \in P_{\mathcal{L}_{\widehat{S}}})\})$	True	(列出值 $\cup\{y\}$, 证明)	$y \in S$
p_i	追加(y)	(列出值 $= \{x \in S\}$, 证明 $=(\{(p_j \in \Pi,\ \widehat{S} \subseteq S,\ \mathsf{P} \in P_{\mathcal{L}_{\widehat{S}}})\})$	假	(列出值, 证明)	$p_i \notin \Pi_M \vee y \notin S$
$p_i \in \Pi_V$	PROVE(y)	(列出值 $= \{x \in S\}$, 证明 $=(\{(p_j \in \Pi,\ \widehat{S} \subseteq S,\ \mathsf{P} \in P_{\mathcal{L}_{\widehat{S}}})\})$	$(\mathcal{A},\ \mathsf{P})$	(列出值, 证明 $\cup\{(p_i, A,\ \mathsf{P})\}$)	$\forall y \in L_{\mathcal{A}} \wedge A \subseteq$ 列出值 $\wedge \forall \mathsf{P} \in \mathcal{P}_{\mathcal{L}_{\mathcal{A}}} \wedge \mathrm{C}(y, \widehat{S}) = 1$
p_i	PROVE(y)	(列出值 $= \{x \in S\}$, 证明 $=(\{(p_j \in \Pi,\ \widehat{S} \subseteq S,\ \mathsf{P} \in P_{\mathcal{L}_{\widehat{S}}})\})$	False	(列出值, 证明)	$\forall y \notin L_{\mathcal{A}} \vee A \not\subseteq$ 列出值 $\vee \forall \mathsf{P} \notin \mathcal{P}_{\mathcal{L}_{\mathcal{A}}} \vee \forall p_i \notin \Pi_V$ $\vee \mathrm{C}(y, \widehat{S}) = 0$
$p_i \in \Pi$	READ()	(列出值 $= \{x \in S\}$, 证明 $=(\{(p_j \in \Pi,\ \widehat{S} \subseteq S,\ \mathsf{P} \in P_{\mathcal{L}_{\widehat{S}}})\})$	证明	(列出值, 证明)	

Table 1 PROOF-LIST 对象的转移函数 Δ

象。

- READ 有效性。 进程 $p \in \Pi_V$ 对 op =READ() 的调用会返回 Seq 中出现在 op 之前的有效 PROVE 调用列表，以及调用每个操作的进程名称。
- 可选——匿名性。假设进程 p 调用 PROVE(v) 操作。如果进程 p' 调用 READ() 操作，那么 p' 无法获知该值 v ，除非 p 泄露了额外信息。

4 证明列表对象规范

第 5 节和第 6 节提出使用共识数这一概念，分析允许列表和拒绝列表对象类型的同步能力。这两种对象有许多相似之处。事实上，唯一的区别在于用户执行的证明类型以及它们的非闪烁属性。因此，本节定义了 PROOF-LIST 对象类型的形式化规约。这是一种新的通用对象，可实例化以描述允许列表或拒绝列表对象类型。

PROOF-LIST 对象类型是一种分布式对象类型，其状态是一对数组（列出值，证明）。第一个数组 列出值 表示已授权/已撤销元素的列表。它是集合 $\mathcal{S}$ 中对象构成的数组，其中 $\mathcal{S}$ 是潜在元素全集。第二个数组 证明是关于列出值数组的断言列表。给定一组管理者 $\Pi_M \subseteq \Pi$ 和一组验证者 $\Pi_V \subseteq \Pi$，PROOF-LIST 对象支持三种操作。首先，APPEND(v) 操作会将值 $v \in \mathcal{S}$ 追加到列出值数组中。管理者集合中的任何进程都可以调用此操作。其次，PROVE(v) 操作会将关于元素 $v \in \mathcal{S}$ 的有效证明追加到证明数组中，该证明是相对于列出值数组而言的。任何进程 $p \in \Pi_V$ 都可以调用此操作。第三，READ() 操作会返回证明数组。

集合 Π_V 和 Π_M 是 Π 的静态预定义子集。对其组成没有任何限制。这些集合的选择仅取决于允许列表或拒绝列表的使用方式。根据使用方式的不同，它们既可以只包含 Π 中的一小部分进程，也可以包含系统中的全部进程集合。

为了表示进程 p生成的证明，我们使用一个依赖于集合 $\mathcal{A}$ 的复杂度类 $\mathcal{NP}$ 的抽象语言 $\mathcal{L}_{\mathcal{A}}$ 。在第 5 节和第 6 节中，我们将分别为允许列表和拒绝列表对象指定这种语言。其思路是， p 针对值 $v \in \mathcal{S}$ 生成证明 π 。进程 p 调用 PROVE 只有在添加到 证明 数组中的证明 π 有效时才有效。当 $v \in \mathcal{L}_{\mathcal{A}}$ 时，证明 π 有效——也就是说， v 是由 $\mathcal{L}_{\mathcal{A}}$ 表示的问题实例的一个解，其中 $\mathcal{L}_{\mathcal{A}}$ 是一个复杂度类的语言

$\mathcal{NP}$ [6]，它依赖于列出值数组的子集$\mathcal{A}$ （$\mathcal{A} \subseteq \mathcal{S}$）。我们用$\mathcal{PL}_{\mathcal{A}}$ 表示相对于语言$\mathcal{L}_{\mathcal{A}}$的有效证明集合。$\mathcal{PL}_{\mathcal{A}}$ 可以表示零知识证明，也可以表示显式证明。

如果证明 π 有效，则 PROVE 操作返回 ($\mathcal{A}$, Acc.$Prove(v, \mathcal{A})$)，其中 Acc.$Prove(v, \mathcal{A})$ 是该操作生成的证明，而 $\mathcal{A}$ 是应用该证明的 列出值的子集。否则，PROVE 操作返回“False”。此外，证明数组还会存储调用 PROVE 操作的进程名称。

形式上，证明列表对象类型由元组 (Q, Q_0, O, R, Δ) 定义，其中：$\widehat{}$有效状态的集合为 Q =(列出值 $=\{x \in$S$\}$，证明 $=\{(p \in \Pi$, S $\subseteq$S, P $\in$P$^{\mathcal{L}\widehat{s}})\}$)，其中列出值是 S 的子集，而证明是元组的集合。证明中的每个元组都包含与其适用的集合相关联的证明，以及发出该证明的进程标识符；有效初始状态的集合为 $Q_0 = (\emptyset, \emptyset)$，即列出值数组和证明数组均为空的状态；可能的操作集合为 $O =$ {追加(x)，PROVE(y)，READ()}，其中 $x, y \in$ S；可能的响应集合为 $R = \Big\{$True、False、($\widehat{}$S $\subseteq$S, P $\in$P$^{\mathcal{L}\widehat{s}}$)、{($p \in \Pi$, $\widehat{}$S$' \subseteq$S, P$' \in$P$^{\mathcal{L}\widehat{s}}$)}$\Big\}$，其中 True 是追加操作成功时的响应，($\widehat{}$S, P) 是$\widehat{}$PROVE 操作成功时的响应，{($p$, S′, P′)} 是读取操作的响应，而 False 是操作失败时的响应；并且转移函数为 Δ。证明列表对象类型支持 5 种可能的转移。我们在表1中定义了 Δ 的 5 种可能转移。

Δ 函数的第一个状态转换模拟了一次有效的追加调用：管理者集合 Π_M 中的某个进程将值 $y \in \mathcal{S}$ 添加到列出值 数组中。Δ 函数的第二个状态转换表示一次失败的追加调用。调用此函数的进程 p_i 未获授权修改列出值 数组，即 $p_i \notin \Pi_M$；或者它尝试追加的值无效，即 $y \notin \mathcal{S}$。Δ 函数的第三个状态转换描述了一次有效的 PROVE 操作，其中一个有效证明被添加到证明 数组中。函数 C 将用于表达拒绝列表实现的抗闪烁性质。它是一个布尔函数，输出为 0 或 1。Δ 函数的第四个状态转换表示一次无效的 PROVE 调用：证明无效，或者出具该证明所针对的集合不是列出值 数组的子集。最后，第五个状态转换表示一次读取操作。它返回证明 数组，且不修改对象的状态。

语言$\mathcal{L}_{\mathcal{A}}$ 并不直接依赖于列出值 数组。因此，PROVE 操作的有效性取决于集合$\mathcal{A}$的选择。

5 允许列表对象的共识数

本节基于 PROOF-LIST 对象，给出一个允许列表对象规范。随后，我们使用该规范来分析该对象类型的共识数。

[6] 本文中，$\mathcal{L}_{\mathcal{A}}$ 可以是以下语言之一：值 v 属于$\mathcal{A}$ （允许列表），或者值 v 不属于$\mathcal{A}$ （拒绝列表）。

我们将允许列表对象定义为一个 PROOF-LIST 对象，并给出其规范，其中 C(y，$\mathcal{S}$) $= 1$ 且

$$\forall y \in \mathcal{S}, y \in \mathcal{L}_{\mathcal{A}} \Leftrightarrow (\mathcal{A} \subseteq \mathcal{S} \wedge y \in \mathcal{A}). \tag{1}$$

换句话说，y 属于集合$\mathcal{A}$。利用 Δ 函数的第三个转换，我们可以看出$\mathcal{A}$ 也应该是列出值数组的子集。因此，该规范支持对列出值进行集合成员关系证明。为此类语言定义的 PROOF-LIST 对象遵循允许列表的规范。为了支持这一论述，我们提供了该对象的一种实现。

为了实现允许列表对象，算法1使用两个原子快照对象。第一个表示列出值 数组，第二个表示证明 数组。这些对象都是由 N 个条目组成的数组。此外，我们使用一个名为“Proof”的函数，该函数以集合$\mathcal{S}$ 和元素 y 为输入，输出一个证明，证明 $y \in$ 列出值。该函数作为黑盒使用，可以输出显式证明——显式证明可以是元组(y,$\mathcal{A}$)，其中$\mathcal{A} \subseteq$ 列出值——也可以输出零知识证明。

共享变量 PROVE(v) 是 AS-LV $\leftarrow N$ 维原子快照对象，初始值为 $\{\emptyset\}^N$；7：如果 $p \notin \Pi_V$，则 AS-PROOF $\leftarrow N$ 维原子快照对象，初始值为 $\{\emptyset\}^N$；操作 APPEND(v)：8：返回 false；9：A $\leftarrow$ AS-LV.Snapshot()；1：如果 ($v \in$ S) $\wedge$ ($p \in \Pi_M$)，则 10：如果 $v \in$ A，则 2：local-values $\leftarrow$ AS-LV.Snapshot()[p]；3：AS-LV.Update(local-values $\cup$ v, p)；4：返回 true；11：$\pi_{\text{set-memb}} \leftarrow$ Proof($v \in$ A)；12：proofs $\leftarrow$ AS-PROOF.Snapshot()[p]；13：AS-PROOF.Update(proofs $\cup$(p, A $\pi_{\text{set-memb}}$), p)；5：否则返回 false；14：返回 (A $\pi_{\text{set-memb}}$)；操作 READ()：15：否则返回 false。6：返回 AS-PROOF.Snapshot()。

算法1 使用原子快照对象实现允许列表对象

▶**定理4.** 算法*1*以无等待方式实现允许列表对象。

证明。 固定算法 E 的一次执行，该算法如算法1所示。每次调用都是由有限次本地操作和原子快照访问组成的序列。由于原子快照原语可以在读写共享内存模型中以无等待方式实现，因此每个正确进程都能在有限个自身步骤内完成每次调用。

令 H 为执行 E的历史。我们定义 H，即 H的已完成历史。H 中的任意调用都可以在 H中完成。我们给出 H的已完成历史 H：

- 任何未执行到第3行的追加操作调用，都可以通过补上**“返回** false”这一行来完成；任何未执行到第13行的 PROVE 操作调用，都可以通过补上**“返回** false”这一行来完成；任何执行到第3行的追加操作调用，都可以通过补上第4行来完成；任何执行到第13行的 PROVE 操作调用，都可以通过补上第14行来完成。

APPEND、PROVE 和读取操作的线性化点分别是第 3 行、第 13 行和第 6 行。为方便起见，我们将 H 返回“false”的任何操作称为无效操作。我们验证 H 中的每个操作都符合规范：

- 进程在 H 中执行的任何 p 无效操作，都是仅修改 p 的内部状态、且由故障进程调用，或由没有写入权限来调用该操作的进程调用的操作。

因此，这些无效操作不会影响允许列表对象的有效性和 Progress 属性。

- 如果进程 p 在 H 中调用的追加操作返回“true”，则意味着 p已执行到第 3 行。因此， p 将值 v 追加到了数组 列出值 的索引 p 处。进程 p 是唯一能够在此索引处写入的进程。由于更新操作是原子的，并且由于 p 是唯一能够写入 AS-LV[p]的进程，因此列出值 数组的仅追加属性得以保持。此外，添加到列出值 中的元素属于集合$\mathcal{S}$，并且追加该值的进程属于管理者集合 Π_M。因此， H 中任何返回“true”的追加操作调用都满足追加有效性属性。因此， H 中的任何追加调用都遵循允许列表规范。

- 如果进程 p 在 H 中调用 PROVE 操作并返回（$\mathcal{A}$ π），则意味着 $p \in \Pi_V$已执行到第 13 行。因此， p 将一个证明 π 追加到了证明 数组的索引 p处，并且该证明是 $v \in \mathcal{A}$的有效证明。进程 p 是唯一允许在该索引处修改证明 数组的进程。写入操作不存在并发。此外，集合$\mathcal{A}$是 AS-LV 数组的子集（第 9 行）。由于向 AS-LV 数组添加元素的唯一方式是通过追加操作，并且我们将 PROVE 操作的线性化点视为第 13 行，因此 PROVE 有效性属性得以确保。得益于原子快照对象的原子性，进展属性得以确保。如果某个进程在时间 t_1执行追加操作的第 3 行，那么任何在时间 $t_2 > t_1$ 执行到 PROVE(x) 操作第 8 行的正确进程都将是有效的。因此， H 中的任何 PROVE 调用都遵循允许列表规范。

- READ 操作总是返回在第 6 行执行之前线性化的 AS-PROOF 数组中的值，这得益于原子快照对象的原子性。此外，返回值始终是成功的 PROVE 操作的集合（AS-PROOF）。该集合由与调用该操作的进程名称相关联的证明组成。因此，READ 有效性属性得以保证。因此，READ 在 H 中的任何调用都遵循允许列表规范。

中的所有操作都遵循 H 允许列表规范。因此， H 是允许列表对象类型的合法历史，并且 H 是可线性化的。总之，算法1中给出的算法是允许列表对象类型的无等待实现。◀

▶**推论5**。 允许列表对象类型的共识数为1。

6 拒绝列表对象的共识数

接下来，我们提出两种无等待实现，以确定拒绝列表对象类型的共识数。在本节及后续内容中，我们将带有$|\Pi_V| = k$ 的拒绝列表称为 k-拒绝列表对象。对这一参数 k 的分析是本文研究的核心。由于该参数是静态定义的，了解这一参数可以通过减少为完成证明而需要同步的进程数量，提高拒绝列表实现的效率。

6.1 下界

算法 2 展示了如何使用具有 $\Pi_M = \Pi_V = \Pi$ 和 $|\Pi| = k$ 的 k-拒绝列表对象来实现 k-共识对象。它使用原子快照对象 AS-LIST，允许进程提议值。AS-LIST 充当一种帮助机制 [15]。此外，该算法还利用了

k-拒绝列表的 PROVE 操作的进展属性和反闪烁属性，以强制实现 k-共识一致性性质。提议操作的执行过程如下。首先，一个进程 p 尝试通过调用 PROVE(0) 来证明元素 0 未被撤销。然后，如果前一个操作成功，p 通过调用 APPEND(0) 撤销元素 0。接着，p 等待 APPEND 生效。通过反复调用 PROVE 操作，直到其中一次操作无效，来完成此验证。k-拒绝列表对象的进展属性确保了这一行为。一旦进展发生，p 就能确定没有其他进程可以再调用有效的 PROVE(0) 操作。因此，p 可以确定，READ 操作返回的集合不再增长。实际上，READ 操作返回的是在其调用之前发生的有效 PROVE 操作所对应的集合。如果无法调用有效的 PROVE(0) 操作，则 READ 操作返回的集合（就元素 0 而言）是固定的。此外，Π 中的所有进程都共享对该集合的相同视图。

最后，p 调用 READ()，以获取调用了有效 PROVE(0) 操作的进程集合。READ 操作的响应将包含所有调用了有效 PROVE 操作的进程，而对于调用 PROPOSE 操作的所有 Π 进程而言，该集合都相同。因此，直到第 7 行为止，该算法解决了集合共识问题。要解决共识问题，我们使用一个额外的确定性函数 $f_i : \Pi^i \to \Pi$，它以任意一个大小为 i 的集合为输入，并从该集合中输出一个值。

为简化算法的表示，我们还使用 separator() 函数，该函数以证明集合 ($\{(p \in \Pi, \{\mathcal{S} \subseteq \mathcal{S}, \mathrm{P} \in \mathcal{PC}_S)\}$) 为输入，输出进程集合，即执行这些证明的进程集合，也就是每个元组的第一个分量。

共享变量 3: k-dlist.APPEND(0);k-dlist $\leftarrow$ k-拒绝列表对象; 4: **执行**AS-LIST $\leftarrow$ 原子快照对象，初始为$\{\emptyset\}^k$ 5: ret $\leftarrow$ k-dlist.PROVE(0);**操作**PROPOSE(v) **为** 6: **直到** (ret $\neq$ false); 1: AS-LIST.update(v, p); 7: 进程 $\leftarrow$ separator(k-dlist.READ()); 2: k-dlist.PROVE(0); 8: **返回**AS-LIST.Snapshot()[$f_{|processes|}$(进程)]。

算法 2 使用一个 k-拒绝列表对象和一个原子快照实现 k-共识对象

►**定理 6.** 算法 *2* 无等待地实现一个 k-共识对象。

证明。 固定算法 2 中所述算法的一次执行 E 。-拒绝列表对象的进展属性保证第 4 行的 while 循环只进行有限次迭代——在循环开始前已调用追加(0)，因此提议(0) 操作最终会失效。每次提议操作调用都由有限个本地操作、原子快照对象访问以及 k-拒绝列表对象访问组成，并且这些操作均假定为原子的。因此，每个进程都会在有限个自身步内完成提议操作。令 H 为 E的历史。我们将 H的已完成历史定义为 H：对于未执行到第 8 行的提议调用，以一条“return false”语句补全。第 8 行是该算法的线性化点。为方便起见，将任何返回 false 的提议调用称为失败调用；否则，称为成功调用。

现在证明 H 中的所有操作都遵循 k-共识规范：调用了 p 中失败的 PROPOSE 操作的进程 H 是有故障的——根据定义，该进程在执行到第8行之前就提前停止了。因此，p 无法作出决定这一事实不会影响 k-共识对象的终止性和一致性属性。

- 成功的 PROPOSE 操作返回 AS-LIST.Snapshot()[$f_{|processes|}$(进程)]。此外，某个进程在第 1 行提议了该值。所有调用 PROPOSE 的进程都会执行 APPEND(0) 操作，并通过第 4 至第 6 行的 while 循环等待该操作生效。得益于 k-拒绝列表对象的抗闪烁性质，当 APPEND 操作对一个进程生效时——也就是说，Progress 发生了，换言之，PROVE(0) 操作无效——，对于任何其他将调用 PROVE(0) 操作的进程，该操作也会生效。因此，得益于抗闪烁性质，当某个进程在第 5 行从 PROPOSE(0) 操作中获得无效响应时，它就知道没有其他进程能够调用有效的 PROVE(0) 操作。这意味着，第 7 行执行的 READ 操作将返回一个固定的进程集合，而所有到达该行的进程都会看到相同的集合。此外，由于每个进程都会在第 3 行的 APPEND(0) 之前调用 PROPOSE(0)，因此至少调用过一个有效的 PROPOSE(0) 操作。因此，进程集合不为空。由于每个进程最终都得到相同的集合 进程，并且得益于函数 f_i 的确定性，所有正确进程都会输出相同的值 v（一致性属性和非平凡值）。值 v 来自原子快照对象，该对象由授权进程提议的值组成（有效性属性）。因此，成功的 PROPOSE 操作遵循 k-共识对象规范。

¯ H 中的所有操作都遵循 k-共识规范。总之，算法 2 中给出的算法是 k-共识对象类型的无等待实现。◀

▶**推论 7.** k-拒绝列表对象类型的共识数至少为 k。

6.2 上界

本节基于 PROOF-LIST 对象给出拒绝列表对象规范，并利用该规范分析此对象类型的共识数上界。

我们将 DenyList 对象实例化为一个 PROOF-LIST 对象，其定义如下：

$$\forall y \in \mathcal{S}, y \in \mathcal{L}_{\mathcal{A}} \Leftrightarrow (\mathcal{A} \subseteq \mathcal{S} \wedge y \notin \mathcal{A}).$$

并且其中：

$$\mathsf{C}(y, \widehat{\mathcal{S}}) = \begin{cases} 1, & \text{if } \forall \mathcal{A}' \in \widehat{\mathcal{S}}, y \notin \mathcal{A}' \\ 0, & \text{otherwise.} \end{cases}$$

换句话说，第一个等式确保 y 不属于集合$\mathcal{A}$，而第二个等式则确保该对象满足抗闪烁性质。因此，此实例化支持对列出值进行集合非成员关系证明。为这种语言定义的 PROOF-LIST 对象遵循 DenyList 的规范。为支持这一说法，我们提供了该对象的一种实现。

为了构建一个 k-拒绝列表对象，使其满足匿名性，需要构建一种能够保护匿名性的高效帮助机制。若不泄露用户身份，就无法直接公开被证明的值。因此，我们假设调用 PROVE(v) 操作的进程 p 能够确定性地为值 v构建一个密码学承诺。令 C_v 为对值 v的承诺。那么，任何调用 PROVE(v) 操作的进程$p' \neq p$ 都可以推断出 C_v 是使用值 v构建的。然而，未调用 PROVE(v) 的进程无法得知 C_v 关联的是哪个值。如果

39:14 访问控制对象的同步能力

如果目标应用不要求用户匿名，则可以使用明文v 作为帮助值。

算法 3 展示了如何使用 k-共识对象和原子快照实现 k-拒绝列表对象。追加和读取操作与算法1中的操作类似。

另一方面，PROVE 操作必须实现抗闪烁性质。为此，使用一组 k-共识对象，以及一种基于承诺的帮助机制。

当进程调用 PROVE(v) 操作时，它会使用原子快照对象发布 C_v，即针对 v 的密码学承诺。该承诺会与时间戳 [16] 一同发布，时间戳定义如下。本地时间戳（p，c）由进程标识符 p 和本地计数器值 c构成。计数器 c 在再次使用前总会递增。因此，每个时间戳都是唯一的。此外，我们构造严格全序关系 $\mathcal{R}$，使得（p，c）$\mathcal{R}$（p'，c'）$\Leftrightarrow$（$c < c'$）$\vee$（（$c = c'$）$\wedge$（$p < p'$））。时间戳与帮助值 C_v 配合使用，以确保终止性。调用 PROVE(v) 操作的进程 p 必须解析其他进程提出的所有值。如果进程 p' 调用的 PROVE(v') 操作早于进程 p调用的操作（按照关系 $\mathcal{R}$），那么 p 必须通过共识对象影响进程 p' 的 PROVE 操作所使用的集合“val”。集合“val”通过读取 AS-LV 对象获得。AS-LV 对象是仅追加的——没有任何操作会从该对象中删除元素。此外，集合“val”通过共识对象分配。因此，这种机制确保 PROVE 操作所作用的集合始终不断增长。

此外，进程使用计数器p 变量依次解析 CONS-ARR。此行为结合共识的性质，可确保所有进程都以相同的顺序看到相同的元组 (winner, val)。

最后，如果进程 p 观察到进程 $p' \neq p$ 执行的 PROVE 操作所关联的承诺 C_v 等同于由 p提议的承诺，那么 p 就会生成相对于 v 的集合非成员证明，并将其关联到以其名义分配给 p' 的集合 ”val”。我们认为，当该集合非成员证明在第 19 行被添加到 AS-PROOF 时，有效的 PROVE 操作即被线性化。因此，当 p 生成自己的证明时——或者另一进程以其名义生成证明时——它可以确定，所有与 v 相关且在 CONS-ARR 中索引低于其自身索引的 PROVE 操作，都已发布到 AS-PROOF 原子快照对象中。因此，抗闪烁性质得以保证。确实，由于分配的集合 ”val” 总是在扩大，并且 CONS-ARR 数组会产生全序，如果 p 执行到第 25 行，那么它此前已以每个进程 $p' \neq p$ 的名义向 AS-PROOF 添加了证明；这些进程调用了 PROVE(v) 操作，且被分配的集合在 CONS-ARR 中的索引低于 p 。因此， p'的操作先于 p的操作被线性化。

PROVE 操作总是可以通过其已发布时间戳来识别。此外，当证明被添加到 AS-PROOF 对象时，总是会将其添加到索引 counter$_{p_w}$。因此，如果多个进程针对标记为 counter$_{p_w}$ 的 PROVE 操作执行第 19 行，AS-PROOF 对象将只登记一个唯一值。

此外，我们使用一个名为 ”Proof” 的函数，该函数以集合 $\mathcal{S}$ 和元素 x 作为输入，输出一个证明，证明 $x \notin \mathcal{S}$。此函数作为黑盒使用，可以输出显式证明——显式证明可以是元组 (x,$\mathcal{S}$)——，也可以输出零知识证明。

▶**定理 8.** 算法 *3* 以无等待方式实现一个 k-拒绝列表对象。

证明。固定算法 E 3 的一次执行。用于优先安排对 CONS-ARR 数组的访问的严格序关系 $\mathcal{R}$ 意味着，每个

共享变量
AS-LV ← N维度为 $\emptyset$ 的原子快照对象，初始值为 $\{\emptyset\}^N$;
AS-Queue ← N维度为 $\emptyset$ 的原子快照对象，初始值为 $\{\emptyset\}^N$;
CONS-ARR$_p$ ← 一个由 k 个共识对象组成的数组，大小为 $l > 0$;
AS-PROOF ← l维原子快照对象，初始$\{\emptyset\}^l$;
局部变量
对于每个 $p \in \Pi_V$:
已评估$_p$ ← 一个大小为 $l > 0$ 的数组，初始 $\{\emptyset\}^l$;
计数器$_p$ ←一个正整数，初始值为 0;
操作追加(v) **是**
1: **如果** $(v \in \mathcal{S}) \wedge (p \in \Pi_M)$ **则**
2: 本地值 ←AS-LV 快照()[p];
3: AS-LV 更新(本地值 $\cup$ v, p);
4: **返回** true;
5: **否则返回**false;
操作PROVE(v) **是**
6: **如果** $p \notin \Pi_V$ **，那么**
7: **返回** false;
8: C_v ←承诺(v);
9: cnt ←计数器$_p$;
10: AS-Queue.更新(((cnt, p), C_v), p);
11: 队列 ← AS-Queue.Snapshot()\ 已评估$_p$;
12: **当**(cnt, p) ∈ 队列 **执行**
13: 最早的 ← R 下队列中最小的时钟值;
14: prop ← (最早的, AS-LV 快照());
15: (winner, val) ←CONS-ARR[计数器$_p$].propose(prop);
16: ((计数器 $_{p_w}$, p_w), C^*) ← 获胜者;
17: **如果** $C^* = C_v \wedge v \notin$ 值 **则**
18: π_{SNM} ← 证明($v \notin val$);
19: AS-PROOF.更新((p_w, 值 π_{SNM}, 获胜者), 计数器 $_{p_w}$);
20: 已评估$_p$ ←已评估$_p$ ∪ 获胜者;
21: 队列 ← 队列 \ 获胜者;
22: 计数器$_p$ ← 计数器$_p$ + 1;
23: **如果** $v \notin$ 值**则**
24: **返回** (值 π_{SNM});
25: **否则返回**false;
操作读取() **是**
26: **返回** AS-PROOF.快照();

算法 3 k-拒绝列表对象类型的实现，使用 k-共识对象和原子快照对象。

进入第 12 行 while 循环的进程都只会迭代有限次。此外，我们假设 k-共识对象和原子快照对象都是原子的。因此，每个进程都会在有限个自身步骤内返回 PROVE、追加或读取操作。

设 H 为 E的历史。我们定义 H，即 H的已完成历史。我们为 H中的所有待处理调用关联一个特定响应。关联的响应如下：

- 任何未执行到第 3 行的追加操作调用，都可以通过执行语句**“返回** false”来完成。任何未执行到第 10 行的 PROVE 操作调用，都可以通过执行语句**“返回** false”来完成。进程 p 发起的任何已执行到第 10 行的待处理 PROVE 操作调用，如果 AS-PROOF 数组中存在 (p, value π_{SNM}, winner)，且进程 p 在第 10 行添加的值为“winner”，则通过执行语句**“返回** (val π_{SNM});”来完成。否则，该操作通过执行语句**“返回** false”来完成。任何已执行到第 3 行的待处理追加操作调用，都可以通过执行第 4 行来完成。

追加和读取操作的线性化点分别位于第 3 行和第 26 行。考虑进程 p 调用的一个有效 PROVE 操作，该操作在 CONS-ARR 数组的 counter$_{p_w}$ 索引处被赋予元组 (winner, val)。我们称该操作在线性化时刻完成：任意进程执行第 19 行中第一个标记为 counter$_{p_w}$ 的 AS-PROOF.Update 时。

为方便起见，我们将返回 false 的操作称为无效操作。每个进程至多访问 CONS-ARR 中的共识对象一次。只有 $k = |\Pi_V|$个进程可以访问这些对象。因此，数组中的 k-共识对象始终会返回一个不同于 $\emptyset$的值。现在我们证明， H 中的所有操作都遵循拒绝列表规范：

- H 中的无效追加操作只会修改进程的内部状态。该操作不会修改共享对象的状态。它要么由在第 1 行失败的未授权进程调用，要么由故障进程调用。该操作遵循规范；H 中的无效 PROVE 操作是指在第 7 行或第 25 行返回 false 的操作。在

第一种情况下，该进程无权提议证明。在第二种情况下，进程使用的值 v 已经位于共识在第 15 行为该进程指定的集合“val”中。该集合由添加到 AS-LV 对象中的值生成。该对象初始为空集合，且只能使用追加操作向该集合中添加值。因此，PROVE 有效性属性得以保证。

- 如果 H 中的追加操作调用返回 true，则意味着进程 p 在第 3 行的索引 p 处向列出值数组追加了值 v 。由于写入操作是原子的，并且只有进程 p 能够在 AS-ACC$[p]$ 中写入，因此 列出值数组的仅追加属性得以保持。因此，成功的追加操作符合规范。

- 如果 H 中的 PROVE 操作调用返回 True，则意味着：1）进程 p 在第 15 行被赋予了一个 k 共识集合 ”val”；2）根据第 17 行， $v \notin$ val；3）证明 $v \notin$ ”val” 已被添加到 AS-PROOF 对象中，这一操作可能由 p 执行，也可能由另一个进程通过帮助机制执行。首先，为了证明进展属性，我们假设存在这样一段历史：某个进程 p' 首先从 APPEND(v) 操作中获得肯定响应。之后，进程 p 调用 PROVE(v) 操作。因此，此时值 v 已包含在 AS-LV 对象中，因为 p' 从其调用中收到了肯定响应。在调用 p 之后执行 PROVE 操作第 14 行的任何进程，都会提议一个包含 v 的集合。因此，通过第 15 行的共识分配给 p 的集合 ”val” 将包含 v。由 p 调用的 PROVE(v) 操作将无效。进展属性得以保证。

  其次，帮助机制以及第 10 行至第 22 行使用的 k- 共识对象确保了抗闪烁性质。在 Π_V 中调用 PROVE(v) 操作的进程将使用 k-共识对象集合，依次为每个证明进程赋予一个“val”集合。此外，这种顺序赋值会考虑 AS-LV 对象的演变。因此，与对象 CONS-ARR$[i-1]$ 关联的集合始终包含于与对象 CONS-ARR$[i]$ 关联的集合中。

  此外，每个调用 PROVE 操作的进程都会按顺序遍历 CONS-ARR 数组。因此，如果进程 p 以时间戳 t调用 PROVE(v) 操作，而该调用最终无效，那么 p 仍会在操作返回前，将所有时间戳低于 t 的 PROVE(v) 操作线性化。因此，所有有效的 PROVE(v) 操作都会在线性化于 p 的调用响应之前，而在 p 的调用响应之后发生的任何 PROVE(v) 操作调用都会失败。

  第三，PROVE 有效性属性直接由第 (2) 点和抗闪烁性质得出。因此，成功的 PROVE 操作符合该规范。
- 读取操作总能返回，并且得益于原子快照对象的原子性，它总能返回 AS-PROOF 数组的最新版本。 H 中的所有操作都遵循 k-拒绝列表规范。因此，算法 3 中提出的算法是 k-拒绝列表对象类型的无等待实现。◀

以下推论由定理 6 和定理 8 得出。

▶**推论 9**。 k-拒绝列表对象类型的共识数为 k。

7 讨论

本节介绍了若干应用，其中可使用允许列表和 k-拒绝列表来确定更复杂对象的共识数。更重要的是，对

这些应用场景中的共识进行分析，可以确定实际实现是否在同步方面达到了最佳效率。如果没有，我们便利用对允许列表和拒绝列表对象共识数的认识，探讨如何构建更实用的实现。更具体地说，允许列表和拒绝列表对象的共识数（在大多数情况下）小于 n，这意味着大多数实现都可以减少实现此类分布式对象时需要同步的进程数量。只有当网络进入同步时段时，许多共识协议的活性才能得到保证。因此，减少需要同步的进程数量可以提高系统进入此类同步时段的概率，从而提升此类协议的有效性。

7.1 可验证凭证的撤销

我们首先使用拒绝列表对象 [4] 分析 Sovrin 的可验证凭证撤销方法。Sovrin 是一种隐私保护型分布式身份管理系统（DIMS）。在该系统中，用户拥有由称为发行者的实体签发的凭证。用户可以使用此类凭证向验证者证明自己具备某些特征。发行者可能希望提前撤销用户的凭证。为此，发行者维护一个仅追加的已撤销凭证列表。当用户希望证明其凭证有效时，必须向验证者提供一个有效的集合非成员关系零知识证明（ZKP），证明其凭证未被撤销，即不在拒绝列表中。在此应用中，管理者集合 Π_M 仅由该凭证的签发者构成。因此，该证明仅涉及验证者和用户。Sovrin 实现这种验证交互的方式，是在每次交互中为用户和验证者创建一个临时的点对点共识实例。即使由此产生的拒绝列表的共识数为 2，Sovrin 仍使用区块链支持的账本上的单写多读对象来实现追加操作（这要求同步系统的 N 进程）。我们的结果表明，Sovrin 的撤销机制可以不使用区块链，而仅通过成对共识来实现。

7.2 匿名资产转移对象

匿名资产转移对象是拒绝列表和允许列表对象的另一种应用。如附录 B 所述，可以使用这些对象来实现 [9] 中描述的资产转移对象。我们的工作将 Guerroui 等人 [9] 的结果推广了。Guerraoui 等人证明，联合账户的共识数为 k，其中 k 是可以从该账户提取资金的代理数量。我们很容易就能证明这一结果，只需注意到，从联合账户提取资金需要使用拒绝列表记录已花费的代币。尽管如此，我们支持零知识证明的构造使得我们可以证明：即使资产转移对象中的用户是匿名的，且其交易无法相互关联，该对象的共识数仍为 k，其中 k 是用户保持匿名的进程数量。匿名资产转移的两种主要实现方式 ZeroCash 和 Monero [17, 18]，都使用区块链作为主要的双重支付防范机制。前者在整个网络中提供匿名性，而后者仅在系统中参与的一部分进程之间提供匿名性。因此，后一种实现方式可以相应地降低其同步要求。

此外，我们的实现使用允许列表作为无中生有代币创建的防止机制。因此，匿名资产转移对象的两项安全属性可以分别得到保障。鉴于允许列表的共识数为 1，这一结果意味着，协议的这一部分可以更高效地处理。例如，

ZeroCash 协议中最消耗空间和资源的部分是无中生有代币创建的防止机制。协议的这一部分以同步方式实现，也就是说，所有进程都同步执行这一机制；正如我们所证明的，这种方式并非最优。因此，这项分析可能有助于更高效地实现类 ZeroCash 资产转移协议。

7.3 分布式电子投票系统

最后，DenyList 对象的另一个直接应用，是基于盲签名的电子投票系统，其共识数为 k，其中 k 是投票服务器的数量；我们将在附录 C 中介绍该系统。此类系统的大多数分布式实现也使用区块链，但实际需要同步的仅是参与其中的一部分进程。

8 相关工作

比特币与区块链

尽管分布式共识算法早已得到广泛研究 [19, 20, 21, 22, 23]，以太坊的兴起及其灵活多样的智能合约所带来的可能性，催生了将已有应用去中心化的新思路。其中，电子投票和 DIMS [4] 就是两个例子。

区块链提升了人们对现有算法的分布式版本的兴趣。然而，这些系统的开发通常很少关注其所依赖的理论基础。一个很好的例子是无需信任的货币转账协议或加密货币。底层的分布式资产转移对象直到最近才得到研究。一项理论研究证明，安全的资产转移协议不需要网络节点之间保持同步 [9]。在此之前，所有提出的方案都使用了共识协议，而共识协议无法在异步网络中确定性地实现 [24]。因此，许多现有协议都可以由更高效的基于可靠广播 [23]的算法取代。这项工作为资金转账协议提出了更高效的实现方案 [10]。随后，Alpos 等人将这项研究扩展到了以太坊 ERC20 智能合约 [12]。后续这篇论文聚焦于智能合约的资产转移能力。此外，所描述的对象具有动态共识数，该数值取决于获准从某个账户转移资金的进程。此外，这项工作以及 Guerraoui 等人 [9] 的工作都分析了一种特定对象，而该对象并非用于确定其他应用的共识数。相比之下，我们的工作旨在作为一种通用工具，用于确定众多系统的共识数。

电子投票

拒绝列表的一个绝佳应用示例是实现基于盲签名的电子投票系统 [25]。盲签名是一种数字签名，签发者可以在不知道消息内容的情况下对其签名。签发者对用户生成的消息的密码学承诺进行签名——密码学承诺是一种密码学方案，爱丽丝在隐藏某个值的同时仍受该值约束 [26]——因此，签发者不知道实际签署的消息内容。随后，用户可以撤销对消息的承诺，并向验证者出示针对明文消息的签名。验证者随后会将这条消息添加到拒绝列表中。列入拒绝列表的签名不再有效。这类签名应用于某些电子投票系统 [27, 28]。在这种情况下，盲签名使投票操作具有匿名性。这正是我们在本文中研究的电子投票机制，

我们研究的机制可以通过使用拒绝列表来实现，以防止用户多次投票。附录 C 探讨了这种方法

还有两种方法可以为电子投票系统的用户提供匿名性。第一种是使用混合网络 [29, 30, 31]。此处使用混合网络来打破投票者与其投票之间的关联。最后，还可以通过同态加密技术实现匿名性 [32, 33]。

每种技术都有其自身的优缺点，具体取决于电子投票系统的特性。我们选择分析基于盲签名的电子投票系统，因为它是本文形式化的分布式拒绝列表对象的直接应用。

匿名货币转账

区块链最初的实现目的是支持无需信任的货币转账协议。这类协议的一项重大缺陷是，它们只为用户提供假名性。因此，任何人都可以查看转账和账户余额，从而泄露用户的敏感信息。后续研究提出了在防止欺诈的同时隐藏用户身份的方法。其主要保障包括防止双重支付——即同一用户不能重复转移同一代币——以及防止凭空 创建——即用户不能凭空创造货币。Zcash [18] 和 Monero [17] 是匿名货币转账协议的典型代表。前者使用允许列表避免资产创建，并使用拒绝列表禁止双重支付；后者则使用环签名。我们将在附录 B 中证明，拒绝列表和允许列表对象可以实现匿名货币转账对象，并由此定义系统中各进程的同步要求。

9 结论

本文首次给出了分布式允许列表和拒绝列表对象类型的形式化定义。这些定义使得分析它们的共识数成为可能。分析得出，实现允许列表对象不需要共识。另一方面，对于拒绝列表对象，所有能够提议集合非成员证明的进程都必须进行同步，这使得拒绝列表的实现需要更多资源。

将允许列表和拒绝列表定义为分布式对象，使我们能够深入研究其他可将允许列表和拒绝列表用作构建模块的分布式对象。例如，我们讨论了 Sovrin DIMS 中的授权列表和撤销列表。我们还在附录中提供了其他几个示例。特别是，我们在附录 B 中证明，拒绝列表对象与允许列表对象的组合能够实现匿名资产转移协议，而且这种实现的同步能力是最优的。该结果也可推广到任何资产转移协议，其中进程充当钱包所有者的代理。在这种情况下，只需在可能代表某个钱包所有者转移资金的进程之间进行同步。

参考文献

1 Gaby G. Dagher、Praneeth Babu Marella、Matea Milojkovic 和 Jordan Mohler。Broncovote：使用以太坊区块链的安全投票系统。载于 ICISSP，2018。 2 以太坊名称服务文档。在线 - https://docs.ens.domains/ - 访问日期：23/11/2022。 3 Harry A. Kalodner、Miles Carlsten、Paul Ellenbogen、Joseph Bonneau 和 Arvind Narayanan。Namecoin 的实证研究及其对去中心化命名空间设计的启示。载于信息安全经济学研讨会，2015。 4 Sovrin：一种用于自我主权身份和去中心化信任的协议与代币。技术报告，Sovrin基金会，2018。 5 Lundkvist、Heck、Torstensson、Mitton 和 Sena。Uport：自我主权身份平台。技术报告，Uport。 6 Ming K. Lim、Yan Li、Chao Wang 和 Ming-Lang Tseng。区块链技术在供应链中的应用综述：对主题、方法和行业的全面分析。《计算机与工业工程》，154：107133，2021。 7 Jyoti Grover。利用区块链保障车载自组织网络安全：全面综述。《车载通信》，34：100458，2022。 8 Maurice P Herlihy 和 Jeannette M Wing。线性一致性：并发对象的一种正确性条件。《ACM编程语言与系统汇刊》，12(3)：463–492，1990。 9 Rachid Guerraoui、Petr Kuznetsov、Matteo Monti、Matej Pavlovič 和 Dragos-Adrian Seredinschi。加密货币的共识数。PODC ’19，第 307–316 页，2019。 10 Alex Auvolat 、 Davide Frey 、 Michel Raynal 和 François Taïani。简化资金转账：规范、通用算法及其证明。《欧洲理论计算机科学协会公报》，132，2020 年 10 月。 11 Maurice Herlihy。无等待同步。《ACM编程语言与系统汇刊》，13(1)：124–149，1991 年 1 月。 12 Orestis Alpos、Christian Cachin、Giorgia Azzurra Marson 和 Luca Zanolini。论代币智能合约的同步能力。载于第 41 届 IEEE ICDCS，第 640–651 页，2021。 13 Yehuda Afek、Hagit Attiya、Danny Dolev、Eli Gafni、Michael Merritt 和 Nir Shavit。共享内存的原子快照。JACM，40(4)：873–890，1993 年 9 月。 14 Daniel Benarroch、Matteo Campanelli、Dario Fiore、Kobi Gurkan 和 Dimitris Kolonelos。集合成员关系的零知识证明：高效、简洁、模块化。载于《金融密码学与数据安全》。施普林格·柏林·海德堡，2021。 15 Keren Censor-Hillel、Erez Petrank 和 Shahar Timnat。救命！PODC ’15，第 241–250 页，2015。 16 Leslie Lamport。分布式系统中的时间、时钟与事件排序。《ACM通讯》21(7)：558–565，1978 年 7 月。 17 Nicolas van Saberhagen。CryptoNote v 2.0，2013 年 10 月。 18 Eli Ben Sasson、Alessandro Chiesa、Christina Garman、Matthew Green、Ian Miers、Eran Tromer 和 Madars Virza。ZeroCash：基于比特币的去中心化匿名支付。载于 2014 IEEE安全与隐私研讨会，第 459–474 页，2014 年 5 月。 19 Miguel Castro 和 Barbara Liskov。实用拜占庭容错。OSDI ’99，第 173–186 页，1999。 20 Leslie Lamport。《兼职议会》，第 16 卷，1998。 21 Miguel Castro 和 Barbara Liskov。拜占庭容错系统中的主动恢复。载于 OSDI 2000，2000 年 10 月。 22 Pierre-Louis Aublin、Sonia Ben Mokhtar 和 Vivien Quéma。RBFT：冗余拜占庭容错。载于 IEEE 第 33 届分布式计算系统国际会议，第 297–306 页，2013。 23 Gabriel Bracha。异步拜占庭一致协议。《信息与计算》，75(2)：130–143，1987。

24 Michael J. Fischer、Nancy A. Lynch 和 Michael S. Paterson。单个进程发生故障时分布式共识的不可能性。《ACM 杂志》，32(2):374–382，1985 年 4 月。25 David Chaum。用于不可追踪支付的盲签名。载于《密码学进展》，第 199–203 页，1983 年。26 Torben Pryds Pedersen。非交互式、信息论安全的可验证秘密共享。载于《密码学进展》——CRYPTO '91，第 129–140 页，1992 年。27 Atsushi Fujioka、Tatsuaki Okamoto 和 Kazuo Ohta。适用于大规模选举的实用秘密投票方案。载于 AUSCRYPT '92，第 244–251 页，1993 年。28 Miyako Ohkubo、Fumiaki Miura、Masayuki Abe、Atsushi Fujioka 和 Tatsuaki Okamoto。对实用秘密投票方案的改进。载于《信息安全》，第 225–234 页，1999 年。29 Ari Juels、Dario Catalano 和 Markus Jakobsson。抗胁迫电子选举。WPES '05，第 61–70 页，2005 年。30 Markus Jakobsson、Ari Juels 和 Ronald L. Rivest。通过随机部分检查增强电子投票混合网络的健壮性。载于第 11 届 USENIX 安全研讨会，2002 年 8 月。31 Michael R. Clarkson、Stephen Chong 和 Andrew C. Myers。Civitas：迈向安全投票系统。IEEE SSP，第 354–368 页，2008 年。32 Olivier Baudron、Pierre-Alain Fouque、David Pointcheval、Jacques Stern 和 Guillaume Poupard。实用的多候选人选举系统。PODC '01，第 274–283 页，2001 年。33 Ronald Cramer、Rosario Gennaro 和 Berry Schoenmakers。安全且最优高效的多机构选举方案。载于 EUROCRYPT '97，第 103–118 页，1997 年。34 Gang Wang、Zhijie Jerry Shi、Mark Nixon 和 Song Han。Sok：区块链分片。载于第一届 ACM 金融科技进展大会论文集，第 41–61 页，2019 年。35 中本聪。比特币：一种点对点电子现金系统。技术报告，2009 年 3 月。36 Andreas Pfitzmann 和 Marit Hansen。匿名性、不可关联性、不可检测性、不可观测性、假名性与身份管理——术语的综合提案。v0 版本，31，2007 年 1 月。

A 列出值数组的变体

在前面的章节中，我们假设列出值数组是仅追加的。某些用例可能需要为此数组采用不同的配置。在本节中，我们将探讨列出值数组不再仅追加的情况。

仅单进程

我们首先探讨一种受限情形：进程只能移除自己写入的值。在这种情况下，追加和移除操作之间不存在冲突。列出值数组可以视为由 $|\Pi_V|$ 个值组成的数组。进程 p_i 可以写入列出值数组的第 i 个索引。它是唯一会修改此数组的进程。因此，写入时不会发生冲突。我们需要为允许列表和拒绝列表对象添加一个移除操作。由于有了这个移除操作，允许列表可以充当拒绝列表。事实上，假设管理者首先将可能标识符全集中的所有元素都添加到允许列表中。那么，这个允许列表就可以实现一个拒绝列表对象，其中允许列表的移除操作等同于拒绝列表的追加操作。因此，允许列表对象需要具备抗闪烁性质，以防止并发的 PROVE 操作产生冲突的结果。这意味着，带有移除操作的允许列表对象

等同于拒绝列表对象，其共识数为 k，其中 k 是 Π_V 中的进程数。

多进程

先前单写入者-移除者列出值 数组的推广形式，是一种列出值数组，其中 k_{AR} （用于追加/删除的 AR）进程可以删除由进程 p_i追加的值。我们假设每个进程 p 都获准在其“自己的”寄存器上执行追加和删除操作。此外，每个进程 p_i 都有一个预定义的授权集合 $\mathcal{A}_i \subseteq \Pi_M$，用于规定哪些进程可以在 p_i 的寄存器上执行追加或删除操作。我们始终有 $p_i \in \mathcal{A}_i$。如果 $p_j \in \mathcal{A}_i$，那么 p_j 可以“覆盖”（删除） p_i 写入的任何内容。在这种情况下，所有获授权的进程都需要进行同步，才能向列出值 数组写入一个值。更确切地说，我们可以区分两种情况。

第一种情况是“完全共享数组”情况，即所有进程共享同一个$\mathcal{A}_i = \Pi_M$。一个进程 p_i 对列出值 数组所做的任何修改，都可能与其他任一进程 $p_j \in \Pi_M$发生冲突。因此，管理者集合中的所有进程必须进行全面同步，才能修改列出值 数组。在需要这种行为时，允许列表和拒绝列表都需要至少 $|\Pi_M|$ 个进程通过解决共识问题来实现追加和删除操作。

第二种情况是“集群”情形：一部分进程共享一个可供它们写入的子数组。在这种情况下，给定集群中的每个进程都必须先同步，然后才能写入（或删除）一个值。所需的同步仅限于该集群中经过 k_{AR} 授权的进程。这在某种程度上对应于分片网络 [34]。

B 匿名资产转移对象类型

去中心化货币转移协议因比特币 [35] 而广为人知。Guerraoui 等人提出了一项理论分析 [9] ，证明如果每个账户仅由一个进程拥有，那么底层对象——资产转移对象——的共识数为 1。这一结果表明，在消息传递环境中，比特币实现所采用的高成本工作量证明（POW）是一种过度设计的解。基于可靠广播原语的成本较低的解同样有效 [10]。Guerraoui 等人的论文还研究了多个进程共享账户的情况。在这种情况下，所得对象的共识数为 k，即共享同一账户的进程数量的最大值。

这些研究有助于深入理解资产转移问题，但它们只研究假名系统，在这类系统中，所有交易都可以关联到同一个假名。随着人们对隐私增强技术的兴趣日益浓厚，加密货币社区试图开发匿名且不可链接的货币转移协议 [18, 17]。随之而来的问题是，底层分布式对象所对应的共识数是多少。本文对允许列表和拒绝列表对象进行的形式化，使回答这一问题成为可能。本节将专门介绍这一证明。

B.1 问题形式化

资产转移对象类型定义

资产转移对象类型允许一组进程通过分布式网络交换资产。我们重新表述 Guerraoui 等人提出的定义 [9] ，以描述此对象：

▶**定义 10.** （假名）资产转移对象类型提供两种操作：转移操作 和 余额。该对象类型定义于进程集合 Π 和账户集合 W 上。账户由其在时间 t 持有的资产数量定义。每个账户的初始资产数量均为 $v_0 \in \mathbb{Z}^{+*}$。我们定义映射 $\mu : W \to \{0,1\}^{|\Pi|}$，将每个账户关联到有权为这些钱包调用转移操作 的进程。资产转移对象类型支持两种操作：转移操作 和 余额。对于转移操作(i, j, v)，$i \in W$ 称为发起方，$j \in W$ 称为接收方，而 $v \in \mathbb{N}$ 称为转账金额。令 $T(i, j)_t$ 表示在时间 t 之前由进程 i 发起、并由进程 j 接收的所有有效转移操作 之和。这些操作满足以下三个性质：

- （终止性）正确进程调用的转移操作和余额操作总会返回。（转移有效性）进程 p 在时间t 调用的操作 *TRANSFER*(x, y, v) 的有效性以递归方式定义。如果在时间 t 之前没有调用过 *TRANSFER*(x, i, v)，$\forall i \in W$，那么当 $v \leq v_0$ 且 $p \in \mu(x)$ 时，该操作有效。否则，当 $v \leq v_0 + \sum_{i \in W} T(i, x)_t - \sum_{j \in W} T(x, j)_t$ 且 $p \in \mu(x)$ 时，该操作有效。（余额有效性）在时间 t 调用的余额操作有效，当且仅当对于每个账户 x，它都返回 $v_0 + \sum_{i \in W} T(i, x)_t - \sum_{j \in W} T(x, j)_t$。

资产转移对象被认为需要具备防止双重支付属性。定义 10 中的 TRANSFER 有效性属性体现了这一属性。事实上，防止双重支付属性旨在避免凭空创造货币。在无等待实现中，有效的转移操作是原子的。因此，双重支付已得到防止。TRANSFER 操作会考虑来自同一账户的所有先前转移。

Guerraoui 等人的论文 [9] 告诉我们，此类对象的共识数取决于映射 μ。如果 $\sum_{i \in \{0,\cdots |\Pi|\}} \mu(w)[i] \leq 1, \forall\, w \in \mathcal{W}$，那么该对象类型的共识数为 1。否则，共识数为 $\max_{w \in \mathcal{W}}(\sum_{i \in \{0,\cdots |\Pi|\}} \mu(w)[i])$。换言之，此类对象类型的共识数，就是能够代表给定钱包调用 TRANSFER 操作的不同进程的最大数量。

从连续余额到基于代币的资产转移

Guerraoui 等人提出的定义采用连续表示来表示每个账户的余额。要用这种表示实现匿名资金转账，需要一种隐藏交易金额 [18] 的机制。由于这种机制不会影响匿名资金转移对象的同步属性，我们通过考虑基于代币的表示来简化问题。这意味着算法只能转移具有预定义权重的代币。为了从一种表示转换到另一种表示，我们在基于账户的货币连续表示的有限历史与基于代币的表示之间建立双射。

设$\mathcal{W}$为账户集合。令 $\mu : \mathcal{W} \to \{0,1\}^{\Pi}$ 为所有者映射，并令$D_{(I,V)}$ 为离散化函数，使得 $D_{(I,V)} : \mathcal{S} \subseteq \mathbb{R}^+ \to \mathbb{N}^I$，其中 $I \in \mathbb{N}^*$为某个值。$D_{(I,V)}$ 定义如下：

$$\forall i \in \{0, \cdots, I\},\ \forall x \in \mathcal{S} : \sum_{i=1}^{I} D_{(I,V)}(x)[i] = x$$
$$\text{where } D_{(I,V)}(x) = \left((D_{(I,V)}(x)[i] = V) \vee (D_{(I,V)}(x)[i] = 0)\right).$$

我们需要定义 S，使$D_{(I,V)}$成为双射。设数组$\{InitBal_1, \cdots \quad InitBal_{|\mathcal{W}|}\}$为 |W| 个账户的初始余额，其中 $InitBal_i \in \mathbb{R}^+, \forall i \in \{1, \cdots \quad p\}$。固定¯资产转移对象的一次执行 E。设 H 为 E 的历史， H 为该历史的线性化。假设转移金额属于 $\mathbb{N}^{*+}$。令 T 为转移金额数组。令 V 为 T 中的所有元素与初始余额数组中所有元素的最大公约数。令 TotalTrans$_i$ 为¯账户在 H中收到的交易总数 i 。令 $balance(i, j)$为账户 i 在第 j次资产¯接收后的余额。令 I 为 H 中账户所持有的最大金额除以 V，即 $I = \max_{i \in \{1, \cdots, |\mathcal{W}|\}}(max_{j \in \{1, \cdots, TotalTrans_i\}}(balance(i,j)))_V$。因此，我们可以将$D_{(I,V)}$应用于初始余额数组中的每个元素。我们可以将 S 定义为无限集合$\{0, V, 2V, 3V, \cdots\}$。¯使用这样的 S 时，映射 $D_{(I,V)}$ 应用于 H后是双射。

由于存在这种双射，所有交易都可以视为转移了给定数量的“硬币”。一枚硬币对应给定数量的资产 V。每个账户持有的硬币数量由一个大小为 I 的数组表示，其值为 V 或 0。因此，我们得到资产转移对象的离散化版本，并且连续设置与离散化设置之间存在双射。

下文中，我们使用资产转移对象的离散版本来讨论匿名资产转移。具体而言，在代币化版本中，价值为 kV 的一次转移由 k 个 TRANSFER 操作组成，每个操作转移一个价值为 V 的代币。

匿名集

设 S 为一个参与者集合。我们将“匿名性”定义为：从观察者 $o \notin S$ 的角度来看，参与者 $a \in S$ 的操作 v 无法与任何其他参与者 $a' \in S$ 的操作区分开来。我们将 S 称为参与者 a 针对操作 v [36] 的匿名集。

实现匿名资产转移需要隐藏代币与拥有它的账户或进程之间的关联。如果“代币所有者”从同一账户转移令牌两次，这两笔交易就可以被关联起来，不再具有匿名性。因此，我们假设“代币所有者”持有令牌的离线所有权证明。这些证明与共享在线元素相关联，使其他进程能够验证交易的有效性。我们将特定用户拥有的离线证明集合称为钱包 。我们将拥有该钱包的个人称为钱包所有者。需要注意的是，一个钱包所有者可以拥有多个钱包，而我们假设一个钱包仅由一位所有者拥有。此外，我们假设每个进程都可以代表多位钱包所有者调用 TRANSFER 操作。否则，单个进程（在大多数情况下通过其 IP 地址或公钥来识别）将关联到单个钱包。这样一来，钱包就会关联到唯一标识符，其执行的交易也就无法匿名。依照同样的推理，我们可以假设钱包所有者可以请求多个进程代表自己调用 TRANSFER 操作。否则，该设置提供的就不是“网络匿名性”，而只是“联邦匿名性”，即钱包仅在连接到同一进程的所有其他钱包中保持匿名。在我们的模型中，进程充当代理。

匿名资产转移对象类型

我们给出了一个将匿名性纳入考量的资产转移对象类型新定义。假名资产转移对象类型与匿名对象类型之间的第一个区别在于不存在余额操作。钱包所有者可以使用不属于分布式

对象的本地余额函数计算自己钱包的余额。TRANSFER 操作也做了些许修改。考虑一个希望将代币 T_O 转移给接收方的发送方。接收方创建一个带有关联离线密码学证明的新代币 T_R（实际上，T_R 可以由发送方使用接收方的公钥创建）。具体而言，接收方为其关联一个私钥。该私钥只有接收方知晓：事实上，知晓该私钥就代表拥有该代币。在转移操作之前，接收方将代币 T_R 发送给发送方。发送方销毁代币 T_O 并激活代币 T_R。销毁操作防止双重支付，而创建操作则使代币能够在隐藏接收方身份的同时转移给新的所有者。此外，这种销毁与创建的过程还能够解除最终作为唯一代币的各次使用之间的关联。

每个代理都维护一个本地钱包，其中包含该代理所有的代币（及其相关的离线证明）。钱包的所有者 w 可以使用 $\mu(w)$ 中的任意进程来调用 TRANSFER 操作。从进程 p 为钱包 w 执行的转移操作关联一个匿名集 $\mathcal{AS}_p^w$，其大小等于与进程 p关联的钱包数量：$|\mathcal{AS}_p^w| = \sum_{i \in \mathcal{W}} \mu(i)[p]$。每笔交易都具有最大匿名集的设置，是一种匿名资产转移对象，其中每个钱包都可以从任意进程执行 TRANSFER 操作：即，$\mu(i) = \{1\}^{|\Pi|}, \forall i \in \mathcal{W}$。基于代币的匿名资产转移对象类型定义如下：

▶**定义 11.** 匿名资产转移对象类型仅支持一种操作：TRANSFER 操作。它定义在进程集合 Π 和钱包集合 W 上。账户由其在时间 t所控制的代币数量定义。初始时，每个账户都会分配数量为 v_0 的代币。我们定义映射 $\mu : W \to \{0,1\}^{|\Pi|}$，它将每个钱包映射到可代表该钱包调用 TRANSFER 的进程。当考虑 TRANSFER(T_O, T_R) 操作时，T_0 是发起方的密码学材料，用于证明某个代币 T的存在；T_R 是接收方生成的密码学材料，用于创建新代币。TRANSFER 操作满足以下三项性质：

- （终止性）正确进程调用 转移操作 后，总会收到返回值。（转移操作 有效性）在时间 t 调用的 转移操作(T_O, T_R) 有效，当且仅当满足以下条件：（存在性）代币 T_O 在交易之前已经存在，即它要么是最初创建的代币之一，要么是在时间 $t' < t$ 调用的有效 转移操作(T'_O, T_O) 期间创建的。（防止双重支付）在时间 $t'' < t$，没有调用过 转移操作(T_O, T'_R)。（匿名性）进程 p 调用 转移操作(T_O, T_R) 时，不会泄露有关所有者 w 和 w' 的信息，也不会泄露有关 T_O 和 T_R的信息，但会透露 w 属于匿名集 AS_p^w这一事实。

让我们从这个定义中提取信息。TRANSFER 有效性属性意味着钱包所有者可以向网络提供存在性证明和防双花证明。这意味着，同一匿名集中的任何其他所有者，只要拥有相同的密码学材料（随机性和关联元素），就可以要求转移同一代币。

我们知道，生成 TRANSFER 证明所需的材料存储在钱包中。此外，我们可以假设，给定钱包所有者使用的所有随机性均由随机数预言机生成，该预言机通过种子获取随机数。每个钱包都有唯一的种子。我们假设，预言机输出的数字对于外部观察者而言似乎是随机的，但共享同一种子的两个进程将按相同顺序获得同一组随机数。

最后，必须通过 TRANSFER 操作向其他进程和钱包所有者公布一笔交易。因此，转移证明是公开的。我们知道，这些证明是

借助确定性随机预言机模型确定性计算得出的。此外，每个转移操作只关联一个发送方和一个接收方。因此，公开证明在不透露发送方和接收方身份的情况下，通过密码学方式将发送方与交易绑定起来。因此，公开证明是一种密码学承诺，发送方或任何掌握与发送方相同信息的其他参与者都可以将其打开。

为了研究该对象的共识数，我们假设钱包所有者可以将其密码学材料与整个网络共享，从而放弃匿名性。在匿名系统中，这样做毫无意义，但它为推理该对象的共识数提供了有价值的工具。此共享过程可以通过原子寄存器实现（因此不会影响共识数，正如我们稍后将讨论的那样）。

进程可以使用本地“解除承诺”函数，根据共享信息推导出发送者身份。“解除承诺”函数以预言机、随机种子、令牌元素以及代币转移的“链上”证明作为输入；如果这些元素有效，则输出钱包所有者标识。否则，输出 $\emptyset$。

B.2 匿名资产转移对象类型的共识数

下界

算法 4 给出了一种算法，该算法仅使用 k 个匿名资产转移对象和单写多读寄存器，实现一个 k-共识对象。 k-匿名资产转移对象中的 k 在此指最大 $\mu(w)$ $\forall\, w \in \mathcal{W}$ 的大小。

共享变量:
一个 AT ← k-Anonymous-AT 对象，初始化时包含 $k+1$ 个钱包，前 k 个钱包中的每一个都拥有这些元素这些元素是转移一个共享令牌所必需的，第 $k+1$ 个钱包是转移操作的接收方，不受任何进程控制；
RM-LEDGER ← 原子快照对象，初始为 $\{\emptyset\}^k$;
V-LED ← 原子快照对象，初始为 $\{\emptyset\}^k$;
O ← 一个随机预言机;
TokenMat ←与唯一代币关联的秘密;
局部变量:
种子 ← 随机数;

操作提议(v) **是**:
1: RM-LEDGER[p].更新(种子，p);
2: V-LED[p].更新(v，p);
3: 结果 ← AT.transfer(TokenMat, O, 种子，$k+1$);
4: RML ← RM-LEDGER.快照();
5: VL ← V-LED.快照();
6: **对于** i，取遍 $\{1, \cdots k\}$ **执行**:
7: **如果** uncommit(O, RML[i], TokenMat, res) $\neq \emptyset$ **那么**:
8: **返回** VL[i];
9: 返回 False;

算法 4 使用 k个 Anon-AT 对象实现的 k-共识对象

▶**定理 12**。 *算法 4 以无等待方式实现 k-共识。*

证明。 固定算法的一次执行 E 。每个提议操作只需要有限次 AT 转移操作，以及有限次对原子快照对象的访问。假设这两类对象都是原子的。钱包所有者的数量有限。因此，每个进程都能在有限个自己的¯步内完成对提议操作的调用。令 H 为 E的历史。我们定义 H 的完成历史 H，其中：

- 任何未执行到第 3 行的提议操作调用 H ，均以执行“return False”这一行完成；

- 在 H 中，任何执行到第 3 行的提议调用，都会通过执行第 4 至第 8 行而完成。我们将返回 false 的操作称为失败操作，其他操作称为¯成功操作。我们验证已完成历史 H 遵循一个k-共识对象的规范：

- ¯任何调用失败的提议操作的进程都是故障进程。该进程无法决定一个值这一事实，不会影响有效性、一致性或终止性。

- ¯在 H 中，任何成功的 PROPOSE 操作调用都会返回成功转移 TokenMat 关联代币的唯一进程所提议的值。如果某个进程在时间 t 到达第 4 行，则在时间 $t' < t$ 必定有且仅有一次 TRANSFER 操作成功。因此，取消提交操作至少返回一次且至多返回一次 1，从而确保有效性属性。由于没有两个进程能够花费同一代币，因此保证了一致性。此外，与 TokenMat 关联的代币会转移到一个无法调用 TRANSFER 操作的虚拟账户。因此，一致性性质得以满足。

¯在 H 中，所有提议操作调用均遵循规范，且算法 4 中给出的算法是无等待的。总之，所提出的 k-共识对象实现是可线性化的。◀

▶**推论13.** *Anon-AT* 对象的共识数至少为 k。

上界

我们给出一种仅使用原子快照对象、拒绝列表对象和允许列表对象来实现 Anon-AT 对象的方法。每位钱包所有者都可以向 k 个不同的进程请求执行 TRANSFER 操作。该实现使用一次性代币，这些代币要么在系统初始化时创建，要么在代币转移过程中创建。销毁某个代币后，可以创建一个新代币，而该代币的新所有者是唯一知晓与这个新代币相关联的密码学材料的人。下文中，我们使用拒绝列表和允许列表对象类型的零知识版本，其中所有集合成员（非成员）证明都使用零知识设置。此外，我们使用允许列表对象来确保代币确实存在（不会凭空创建），并使用拒绝列表对象来确保代币尚未被花费（双重支付保护）。

本文不讨论所使用的底层密码学对象。不过，我们假设实现采用 ZeroCash [18] 密码学实现；这是一种健全的匿名资产转移协议。更确切地说，我们将使用其链下函数的高层定义。需要指出的是，使用 ZeroCash 实现时，可以通过一种称为“铸造”的特殊交易，将价值从假名资产转移对象转移到匿名对象。为简化构造，我们假设每个钱包创建时都带有初始数量的代币 v_0，并且我们的对象不允许跨链转移。因此，我们没有“铸造”操作。

ZeroCash 使用一种名为$pour$ 的 TRANSFER 操作，该操作会执行转移，并销毁和创建相关的密码学材料。这里，我们使用修改版的$pour$，它不执行转移或任何非本地操作。这是一个黑盒本地函数，用于生成证明源代币（T_O）已销毁且目标代币（T_R）已创建所需的密码学材料。修改后的$pour$函数以源代币、发送方的私钥（sk_s）和接收方的公钥（pk_r）为输入：$pour(T_O, \mathsf{pk}_r, \mathsf{sk}_s) \to tx$, tx 是用于销毁 T_O 并创建 T_R的密码学材料。

可能有多个进程同时转移令牌。因此，我们定义一个确定性的本地函数 ChooseLeader($\mathcal{A}$ tx)，它接受任意集合 $\mathcal{A}$ 和交易 tx作为输入，并输出一个调用了 BL.PROVE(tx) 的参与者 p。[7]

[7] 实际上，ChooseLeader 的签名会更加复杂，因为该函数需要 T_O, pk_r, sk_s

共享变量：5: DL.APPEND(tx); DL ← k-拒绝列表对象，初始值为($\emptyset$, $\emptyset$); 6: 执行：AL ← 允许列表对象，初始值为({$(token_{(i,j)})_{i=1}^{t}$}$_{j=1}^{k}$, $\emptyset$) 7: ret ← DL.PROVE(tx); 操作 TRANSFER(T_O, pk$_r$, sk$_s$)：1: $tx \leftarrow$ Pour(T_O, pk$_r$, sk$_s$) 2: 如果 verify(tx) 且 $tx \in$ AL 和 $tx \notin$ DL，则：3: AL.PROVE(tx); 8: 当 ret $\neq$ 为 false 时；9: 如果 ChooseLeader(DL.READ(), $tx.T_R$)=p，则：10: AL.append($tx.T_R$); 11: 返回 $tx.T_R$; 4: DL.PROVE(tx); 12: 返回 False;

算法 5 使用单写多读寄存器、允许列表对象和拒绝列表对象实现 Anon-AT 对象。

▶**定理 14.** 算法 *5* 以无等待方式实现 *Anon-AT* 对象。

证明。 固定算法的一次执行 E。 E 仅使用拒绝列表对象和单写多读寄存器（允许列表对象的共识数为 1，且可使用单写多读寄存器实现）。每个 TRANSFER 操作仅需要有限次 DenyList.PROVE 和 DenyList.APPEND 操作，并假设这些操作是原子的。每个进程都能在有限个自身步内完成 TRANSFER 的调用。令 H 为 E的历史。¯我们定义 H H的完成历史，其中：

- 将 H 中任何尚未到达第 11 行的待处理 TRANSFER 调用补全为执行 "return False"；将 H 中任何已到达第 11 行的待处理 PROPOSE 调用补全为执行第 12 行。
-

我们将返回 false 的操作称为失败操作。其他操作称为¯成功操作。接下来，我们分析已完成历史 H 是否符合匿名资产转移对象规范：

- ¯在 H 中调用 TRANSFER 操作时失败的任何进程都是故障进程。它无法转移资金、创建代币或进行双重支付。因此，这并不违背匿名资产转移对象的属性。这个故障进程可能会丢失代币（销毁代币而不转移），但由于我们假设崩溃是永久性的，它的所有代币无论如何都会丢失。
- ¯由进程 p 在 H 中调用并成功执行的 TRANSFER(T_O, T_R) 操作，向网络证明代币 T_O 存在且尚未被花费。为新代币生成密码学材料并销毁旧代币后，该操作会验证这些材料的正确性，以及新旧代币的有效性（第 2 行）。第 3 行的证明操作必定成功，因为 $tx \in AL$。如果另一个进程正在对同一个 T_O执行并发转账操作，第 4 行的证明操作则可能失败。此潜在冲突由 ChooseLeader 函数解决，该函数以 DenyList 上成功的证明操作列表作为输入。ChooseLeader 函数的确定性以及 READ 操作所提供的一致性，确保即使多个进程针对同一个 T_O发起并发且相互冲突的 TRANSFER 操作，所有进程也会在第 10 行添加相同的 $tx.T_R$ 。前述机制可防止双重花费。不可创建属性则由 PROVE 操作确保

除此之外还有 tx。这些额外元素使得解除承诺 tx成为可能，从而使 PROVE 操作的值与 $tx.T_R$相匹配。请注意，这不会造成匿名性威胁，因为这是由 sk_s的所有者调用的本地函数。为简化说明，以下内容中省略这些细节。

该操作在第 3 行对允许列表执行。最后，通过使用允许列表和拒绝列表对象的零知识证明版本来确保匿名性。

在 H 中，TRANSFER 操作的所有调用均遵循匿名资产转移对象的规范。总之，所提出的算法是匿名资产转移对象的无等待实现。◀

▶**推论15**。 *k-anon-AT* 对象的共识数上界为 k。利用这一推论和前一个推论，我们进一步推得， *k-anon-AT* 对象的共识数为 *1*。

C 使用 DenyList 对象实现电子投票系统

本节将说明，DenyList 对象可以为复杂对象的共识数提供上界。作为示例，我们研究一个电子投票系统。电子投票系统必须满足与“现场”投票相同的属性。现场投票系统必须确保四项安全属性，其中两项针对组织者，另外两项针对投票者。首先，投票组织者必须确保每个投票者都有投票资格。其次，每位投票者只能投票一次。第三，投票者必须验证自己的投票已计入最终计票结果。第四，一项可选属性是投票者匿名性。根据投票类型，投票者可能希望隐藏自己的身份。

我们希望设计一种分布式电子投票系统，让投票者能够将自己的选票提交给多个不同的投票服务器，同时确保其选票的唯一性。我们假设每台服务器都是分布式基础设施中的一个进程。投票者充当客户端，向服务器提交投票请求。我们假设通过令牌来确保“投票权”这一属性。令牌是一次性使用的假名，用于将选票与投票者关联起来。用户从签发者处获取令牌。所有投票服务器都信任该签发者。投票者和签发者都无法访问电子投票对象，除非某个参与者同时承担多个角色。根据这些规范，我们将电子投票对象类型定义如下：

▶**定义16**。 电子投票对象类型支持两种操作：投票*(*令牌，v*)* 和投票计数*()*。令牌是投票者的标识符，而 v 是选票。此外，这些操作支持三个强制属性和一个可选属性：

1. （投票有效性）如果满足以下条件，则在时间 t 调用的投票*(*令牌，v*)*有效：令牌是由投票服务器信任的签发者签发的有效令牌；且不存在在时间 $t' < t$调用的有效投票*(*令牌，v'*)* 操作，其中 $v' \neq v$。**2.** （投票计数有效性）投票计数*()*操作返回已调用的有效投票操作集合。**3.**（可选——匿名性）即使投票服务器和签发者串通，令牌也不会将选票与投票者的身份关联起来。

接下来，我们分析一种基于签名的电子投票系统。签发者向投票者签发签名。该签名的消息是一个随机数。元组（签名，随机数）用作代币。当投票者发出投票请求时，服务器会验证签名的有效性，并在签名有效时进行投票。

使用盲签名 [25]，可以轻松地为基于签名的电子投票系统添加匿名性。盲签名算法是一种数字签名方案，签发者无法获知其所签署的值。由于被签署的值通常是随机数，签发者无需验证该值——恶意选择的值不会赋予投票者超出预期的权限。

39:30 访问控制对象的同步能力

形式上，盲签名算法由元组 (Setup, Commit, Sign, Uncommit, Verify) 定义，其中 Setup 会创建该方案的公共参数（生成元、共享随机性等）以及所有签发者的公钥/私钥对。公钥会与系统中的所有参与者共享。Commit 是一种具有隐藏性和绑定性的承诺方案；它会输出对某个值（即由用户随机选择的随机数）的承诺。Sign 算法以对随机数的承诺和某位签发者的私钥作为输入，并输出对该承诺的签名。Uncommit 算法以对承诺的签名和签发者的公钥作为输入，并输出对解除承诺后的消息（原始消息）的同一签名。最后，如果解除承诺后的签名是某位签发者对该消息的有效签名，Verify 算法就输出 1 m。

算法 6 使用任意签名方案、一个 k拒绝列表对象和一个单写多读原子快照对象，为电子投票系统提供无等待实现。在这里， k的值对应投票服务器的数量，并且 k=$|\Pi_V| = |\Pi_M|$。该算法的思路是使用追加操作，表明某个代币已被用于投票。为了获得无等待实现，我们使用存储在原子快照对象 AS-prevote 中的辅助值 [15] 。投票操作按如下方式进行：投票服务器 V 在 AS-prevote 对象中传达其将投出的选票。然后， V 执行 PROVE(代币) 操作，以证明该代币尚未被使用。接着， V 调用 APPEND(代币) 操作，并等待 APPEND 生效——即第 8 至 10 行的 do-while 循环。最后， V 使用读取操作验证自己是否是唯一一个为该特定代币提议选票的进程。如果是，则将该选票加入投票数组——AS-vote 数组。否则，只有当使用同一代币投票的其他服务器在第 4 行提议了相同的选票时，才会将该选票加入投票数组。

如果同一投票者向不同服务器请求了两个值不同的并发交易，也可以提出其他实现方案。例如，可以修改算法 6，使用确定性函数从所有可能的投票中选出一个值。此修改不会影响该实现的性质。

现在，我们对这一实现的线性一致性给出非正式证明。 k-拒绝列表对象的抗闪烁性质确保 while 循环能够终止。因此，该实现是无等待的。同一性质还确保所有持有相同代币的投票者，其 投票值变量都相同，从而确保投票唯一性。第 1 行的签名验证确保了投票授权。若使用盲签名方案，匿名性也能得到满足。因此，无需使用匿名 DenyList 对象。由此可得， k-DenyList 对象类型的共识数是电子投票系统共识数的上界。

还可以使用一个电子投票对象来构建 k-共识对象的无等待实现。该实现见算法 7。该算法的思路是，每个进程都尝试使用同一个代币进行投票。由于只能接受一票，电子投票对象只会采纳第一个投票者的投票。最终，每个进程都会在投票对象中看到相同的值。该值就是共识的结果。

每次调用都是由有限个本地操作和对电子投票对象的访问组成的序列，且这些操作和访问均假定为原子的。因此，每个进程都能在有限个自身步骤内完成提议操作。只有当某个进程提出了某项投票时，该投票才会被计入，这确保了有效性属性。一致性性质源于投票唯一性。最后，共识对象的非平凡性得以保证，因为决定的值是获胜进程所选择的任意值 v 。

共享变量：11：votes ← k-dlist.READ();k-dlist ← k-DenyList 对象；12：客户端投票 ← votes 中所有 token 为 token 的值；AS-prevote ← 原子快照对象，初始为 $\{\emptyset_k$ 13$\}$；AS-vote ← 原子快照对象，初始为 $\{\emptyset\}_k$ 14：投票者 ← client-votes 中的所有进程；：vote-values ← AS-prevote 中所有值 操作 VOTE((signature, pk, token, v) 为：‘其中 token 为 token，且添加该值的进程属于投票者；1：如果 Verify(signature, token, pk) $\neq 1$ 则：15：如果 vote-values $= \{v\}^l, \forall\, l \geq 1$ 则：2：返回 false；16：previous-votes ←AS-vote.SNAPSHOT()$[p]$；3：AS-prevote.update((token, v), p)；17：AS-vote.UPDATE(previous-votes ∪ (token, v, voters), p)；4：ret ← k-dlist.PROVE(token) 18：返回 true；5：如果 ret $=$ 为 false，则：19：否则返回 false；6：返回 false；操作 VOTE-COUNT() 为：7：　k-dlist.APPEND(token)；20：votes ← AS-vote.Snapshot() 中的所有值。8：执行：‘每个元组 (token, v, voters) 仅保留一个；9：ret ← k-dlist.PROVE(token)；21：返回 votes；10：当 ret $\neq$ 为 false 时，

算法 6 使用一个 k-拒绝列表对象和原子快照实现电子投票对象

共享变量：vote-obj ←电子投票对象，其中唯一获授权的代币为 0；σ ← 受信任签发者对 0 的签名；**操作**提议(v) **是**：1：vote-obj.投票((σ, 0), v); 2：{(winner-token, value)} ←vote-obj.投票计数(); 3：**返回** value;

算法 7 使用一个电子投票对象实现 k-共识对象

基于盲签名的电子投票系统的共识数，一方面受 k-共识对象的共识数限制，另一方面受 k-拒绝列表对象的共识数限制。因此，基于盲签名的电子投票对象类型的共识数为 k。