arXiv:2302.04744v1 [cs.DC] 9 Feb 2023

利用 Setchain 数据类型提升区块链可扩展性

MARGARITA CAPRETTO，IMDEA Software Inst.，西班牙和马德里理工大学，西班牙
MARTÍN CERESA，IMDEA Software Inst.，西班牙ANTONIO FERNÁNDEZ ANTA，IMDEA Networks Inst.，西班牙Antonio Russo，IMDEA Networks Inst.，西班牙和马德里卡洛斯三世大学，西班牙CÉSAR SÁNCHEZ，IMDEA Software Inst.，西班牙

区块链技术正面临可扩展性挑战，必须克服这一挑战，才能确保该技术得到更广泛的应用。这一可扩展性问题源于使用共识算法来保证区块链（以及每个区块内交易）的全序。然而，全序往往并非完全必要，因为智能合约的一些重要高级应用并不要求所有操作都遵循全序。如果能够利用更宽松的顺序（而非全序），就有望实现更高的可扩展性。

本文提出了一种新颖的分布式并发数据类型，称为 *Setchain*，能够显著提升可扩展性。Setchain 实现了一种只增集合，其中的元素没有顺序，这与传统的区块链操作不同。Setchain 还允许在适当时强制设置同步屏障，为最近添加的一部分元素永久分配一个经共识达成一致的纪元编号。因此，同一纪元中的两个操作没有顺序关系，而不同纪元中的两个操作则按照各自的纪元编号排序。我们介绍了 Setchain 的多种拜占庭容错实现，证明了它们的正确性，并报告了原型实现的实证评估结果。结果表明，Setchain 的速度比基于共识的账本快几个数量级，因为它实现的是带有纪元同步的只增集合，而非全序。

由于 Setchain 屏障可以与底层区块链同步，Setchain 对象可用作侧链，以实现多种去中心化解决方案，其操作速度远快于直接在区块链之上实现的方案。

最后，我们还介绍了一种算法，它在单个进程中涵盖拜占庭服务器的组合行为；通过在具体实现中对一般攻击者进行编码，该算法简化了正确性证明。

其他关键词和短语：分布式系统、区块链、拜占庭分布式对象、共识、Setchain。

1 引言

1.1 问题

分布式账本（也称为区块链）最初由 Nakamoto 于 2009 [21]提出，并在比特币的实现中得到应用，作为一种消除电子支付系统中可信第三方的方法。现代区块链包含智能合约 [28, 33], ，它们是存储在区块链中的不可变状态程序，用于描述交易的功能，包括加密货币的交换。智能合约能够描述复杂的功能，从而支持去中心化金融（DeFi）等众多应用[1]、去中心化治理、Web3 等。

[1]据 Statista 估计，截至2021年12月，去中心化金融中锁定的货币价值约为 1000 亿美元 https://www.statista.com/statistics/1237821/defi-market-size-value-crypto-locked-usd/。

作者地址：Margarita Capretto，margarita.capretto@imdea.org，IMDEA Software Inst.，波苏埃洛-德阿拉尔孔，马德里，西班牙，以及马德里理工大学，马德里，西班牙；Martín Ceresa，martin.ceresa@imdea.org，IMDEA Software Inst.，波苏埃洛-德阿拉尔孔，马德里，西班牙；Antonio Fernández Anta，antonio.fernandez@imdea.org，IMDEA Networks Inst.，莱加内斯，马德里，西班牙；Antonio Russo，antonio.russo@imdea.org，IMDEA Networks Inst.，莱加内斯，马德里，西班牙，以及马德里卡洛斯三世大学，莱加内斯，马德里，西班牙；César Sánchez，cesar.sanchez@imdea.org，IMDEA Software Inst.，波苏埃洛-德阿拉尔孔，马德里，西班牙。

所有分布式账本的主要组成部分都是“区块链”。区块链是一种分布式对象，其中包含按区块打包的、严格有序的交易列表，这些交易代表用户执行 [13, 14]。区块链对象由多台服务器维护，无需中央权威机构，并采用能够抵御拜占庭攻击的共识算法。

目前阻碍区块链技术更快普及的一大主要障碍，是其可扩展性有限，这是拜占庭共识算法固有的吞吐量受限所致 [8, 31]。以太坊 [33], 作为最受欢迎的区块链之一，每分钟只能产生不到 4 个区块，每个区块包含不到两千笔交易。比特币 [21] 的吞吐量更低。这些数据比许多去中心化应用所需的速度慢了几个数量级，最终可能危及该技术在许多前景广阔领域的应用。吞吐量受限会推高每次操作的价格，因为执行操作的需求很高。因此，人们越来越关注提升区块链可扩展性的技术 [20, 35]。相关方法包括：

- 开发更快的共识算法 [32];
- 采用分片等并行技术 [10];
- 开发具备跨区块链通信能力的特定应用区块链 [19, 34];
- 在链下执行智能合约，并仅进行维持区块链保证所必需的最低限度同步，这类方法称为“第二层”（L2）方法 [17]。不同的 L2 方法包括：(1) 在链下计算零知识证明 [2], , 只需在链上进行验证（希望能更高效） [1], ; (2) 采用有限但实用的功能，例如通道（如 Lightning [22]）；或 (3) 部署乐观汇总（如 Arbitrum [18]），其做法是避免在服务器上运行合约（仅在需要标注声明和解决争议时运行）。

本文提出了一种提升区块链可扩展性的替代方法，其利用了以下观察。传统上，人们一直认为，加密货币需要全序才能保证不会发生双重支付。然而，许多有用的应用和功能（包括加密货币的某些用途 [16]）能够容忍较为宽松的保证，即操作仅部分有序。本文提出了一种拜占庭容错的分布式只增集合 [5, 27],实现，并为其配备了一项额外操作，用于引入屏障同步点（在该点上，所有服务器对集合内容达成一致）。在屏障之间，分布式集合中的元素可能暂时只有部分服务器知晓。我们将这种分布式数据结构称为 Setchain。实现 Setchain 的区块链 B（以及区块）可以将 B 中区块的整合与屏障同步对齐，从而获得一个非常高效的侧边数据类型集合对象，其拜占庭容错保证与 B 本身提供的保证相同。

在区块链背景下，带纪元集合的两种极端实现是 :

完全链下实现。 完全链下实现的主要缺点是，从底层区块链的角度来看，最终实现并不具备区块链所提供的可信度和问责性保障。这种方法的一个例子是内存池。内存池（memory pools 的简称）是一种 P2P 数据类型，大多数区块链都用它来维护待处理交易的集合。内存池有两个目的：（1）防止针对负责挖掘区块的服务器发起分布式攻击；（2）提供一个交易请求池，供区块生产者从中选择操作。如今，内存池受到广泛关注，因为它们缺乏问责性，也是攻击的来源 [25, 26], , 包括抢先交易 [9, 24, 30]。我们提出的数据结构 Setchain 具有强得多的问责性，因为它能够抵御拜占庭攻击，而且 Setchain 维护的集合元素是公开的，无法伪造。

完全链上解决方案。 考虑以下实现（使用一种类似 Solidity 的语言），其中 添加用于添加元素，而 **EpochInc** 用于增加时期。

```
contract Epoch { uint public epoch = 0; set public the_集
```

这种实现的一个问题是，每次添加一个元素，该_集合都会变大，从而影响执行合约所需的成本。另一个更重要的问题是，添加元素的速度很慢——慢得就像与区块链交互一样——而我们的主要目标是提供一种比区块链快得多的数据结构。

我们的方法速度更快，可以独立于底层区块链部署，并与区块链节点同步。因此，Setchain 介于上述两种极端方案之间。

对于给定的区块链 B，我们提出一种 Setchain 实现，它（1）比直接在 B 中实现和执行操作高效得多；（2）能够像 B 一样提供抵御拜占庭攻击的去中心化保障；（3）能够与 B 的演进同步，因此合约有可能检查 Setchain 的内容。简而言之，这些目标通过以下方式实现：对于非同步元素插入，使用更快的操作来协调服务器；对于纪元变更，则仅使用共识类算法。

1.2 Setchain 的应用

推动 Setchain 开发的潜在应用包括：

1.2.1 内存池。大多数区块链会在矿工选择用户的交易请求之前，将其存储在“内存池”中；而交易一旦被挖出，内存池中的信息便会丢失。若要记录并研究内存池的演变，就需要额外的对象充当可靠的内存池日志系统，且该系统必须足够快速，能够记录每一次与内存池交互的尝试，同时不影响区块链的性能。Setchain 可以充当这样的可信日志系统，在这种情况下无需在纪元与区块之间进行同步。

1.2.2 通过二层乐观汇总实现可扩展性。乐观汇总，例如 Arbitrum， [18], 利用了这样一个事实：计算可以在区块链之外执行，链上只需发布关于交易执行效果的声明。通过这种方式，Arbitrum 的维护者会提出执行若干笔交易后达到的下一个状态。一段时间后，链上安装的仲裁智能合约会假定所提议的步骤正确，因为该状态尚未受到质疑，并执行附注的效果。合约中还包含冲突解决算法，用于解决争议。该协议不要求严格的全序，只需记录所提议的操作即可。此外，冲突解决可以简化为声明验证，这项工作可以由 Setchain 的维护者完成，从而无需仲裁。

1.2.3 侧链数据。最后，Setchain 也可以用作通用侧链服务，以与区块同步的方式存储和修改数据。对于需要

仅更新智能合约存储空间中的信息（如数字登记簿）的应用而言，可以受益于更快速（因此成本更低）的存储操作方式，而无需调用昂贵的区块链操作。

1.3 贡献。

总而言之，本文的贡献如下：

- 名为 Setchain 的侧链数据结构的设计与实现，
- Setchain 的多个实现，提供不同层次的抽象，并改进算法实现，
- 对原型实现进行的实证评估，结果表明 Setchain 的速度比共识快几个数量级，
- 介绍如何将 Setchain 用作分布式对象：良好的客户端需要联系多台服务器来注入元素，并获取 Setchain 的正确视图；
- 介绍一种效率高得多的 Setchain 乐观服务：客户端只需联系一台服务器，即可添加元素并获取正确状态。
- 将多台拜占庭服务器的组合行为简化为一个单一的非确定性进程，从而简化对组合分布式系统的推理。

本文其余部分安排如下。第2 节介绍初步模型和假设。第3 节描述 Setchain 的预期性质。第4节介绍 Setchain 的三种不同实现，并采用渐进式方法。算法Basic 和Slow 用于说明我们如何得出算法Fast，它是速度最快、鲁棒性最强的算法。第5节证明了这三种算法的正确性。第6 节讨论了对不同算法原型实现的实证评估。第7 节说明如何增强 Setchain 在面对拜占庭服务器时的鲁棒性。第8 节介绍一种能够刻画拜占庭行为的非确定性算法。最后，第9 节总结全文。

2 预备知识

我们现在介绍 计算模型以及我们在 Setchain 算法中使用的构建模块 .

2.1 计算模型

分布式系统由进程（客户端和服务器）以及底层通信网络组成，每个进程都可以通过该网络与其他所有进程通信。通信通过消息传递进行。每个进程独立运行，速度各异，且其他进程无法了解其内部情况。消息传输延迟任意但有限，进程也无法获知这些延迟。我们的目标是让服务器彼此通信，以实现具有特定保证的分布式数据类型；客户端则可以与服务器通信，以使用该数据类型。

进程可能以任意方式发生故障，但发生故障的（拜占庭）服务器数量上限为 f，服务器总数 n 至少为 $3f+1$。我们假设非拜占庭（正确）进程之间存在可靠信道，因此消息不会丢失、重复或被篡改。每个进程（客户端或服务器）都有一对公钥和私钥。公钥已可靠地分发给所有可能相互交互的进程。因此，我们排除了虚假或伪造进程的可能性。我们假设消息经过身份验证，因此，拜占庭进程篡改或伪造的消息会被正确进程检测并丢弃 [7]。因此，正确进程之间的通信默认是可靠但异步的。不过，我们用作基本构建模块的集合共识服务需要部分同步 [6, 15], ，如下所述。部分同步仅适用于实现

集合共识的协议中的消息传递和计算。最后，我们假设存在一种机制，使客户端能够创建服务器可以在本地检查的“有效对象”。在区块链的语境中，这是通过公钥密码学实现的。

2.2 构建模块

我们使用四个构建模块来实现 Setchain：

2.2.1 拜占庭可靠广播（BRB）。 BRB 服务 [3, 23] 允许向一组进程广播消息，并保证正确进程发送的消息最终会被所有正确进程接收，且所有正确进程最终都会接收到相同的一组消息。BRB 服务提供一个原语 BRB。广播(m) 用于发送消息，以及一个事件 BRB。交付(m) 用于接收消息。我们列出证明 Setchain 性质所需的 BRB 相关性质（第5节）：

- **BRB 有效性：** 如果正确进程 p_i 执行 BRB。交付(m)，那么 m 曾由正确进程 p_j 发送，而该进程此前执行了 BRB。广播(m)。
- **BRB 终止性（本地）：** 如果正确进程执行了 BRB。广播(m)，那么它就会执行 BRB。交付(m)。
- **BRB 终止性（全局）：** 如果某个正确进程执行了 BRB.Deliver(m)，那么所有正确进程最终都会执行 BRB.Deliver(m)。

请注意，BRB 服务不保证将消息按相同顺序交付给两个不同的正确参与者。

2.2.2 拜占庭原子广播（BAB）。 BAB 服务 [11]在 BRB 的基础上扩展，增加了一项保证：消息按全序交付。BAB 服务提供与 BRB 相同的操作和事件，我们将其分别重命名为 BAB.Broadcast(m) 和 BAB.Deliver(m)。然而，除了有效性和终止性外，BAB 服务还提供：

- **全序：** 如果两个正确进程 p 和 q 都执行了 BAB.Deliver(m)，并且 BAB.Deliver(m')，那么 p 交付 m 先于 m'，当且仅当 q 交付 m 先于 m'。

BAB 已经 已证明其难度与共识相当 [11], , 因此也受到相同的限制 [15] .

2.2.3 拜占庭分布式只增集合（DSO） [5]。 集合是计算机科学中最基本、最重要的数据结构之一，通常包含添加和删除元素的操作。添加和删除操作不可交换，因此，分布式实现需要额外机制来保持副本同步，防止本地状态发生冲突。一种解决方案是只允许添加。因此，只增集合是一种只能添加元素而不能删除元素的集合，它可以实现为无冲突复制数据类型 [27]。

令 A 为一个值字母表。只增集合 GS 是一种并发对象，它维护一个内部集合 $GS.S \subseteq A$，并为任何进程 p 提供两种操作：

- GS。add(r)：将一个元素 $r \in A$ 添加到集合 $GS.S$。
- GS。get()：获取内部元素集合 $GS.S$。

初始时，集合 $GS.S$ 为空。拜占庭分布式只增集合对象（DSO）是一种以分布式方式实现、能够容忍拜占庭攻击的并发只增集合 [5]。我们列出与 Setchain（第 5节）相关的性质：

- **拜占庭完备性**：由正确进程调用的所有 get() 和 add(r) 操作最终都会完成。
- **DSO-添加获取**：所有 add(r) 操作最终都会使 r 出现在所有 get() 返回的集合中。
- **DSO-获取添加**：由 get() 返回的每个元素 r，过去都曾通过 add(r) 添加。

2.2.4 集合拜占庭共识（*SBC*）。RedBelly 中引入的 SBC [6],是一种拜占庭容错分布式问题，与共识类似。在 SBC 中，每个参与者都会提出一个元素集合（在 RedBelly 的特定情形下，是一个交易集合）。SBC 结束后，所有正确服务器都会就一个有效元素集合达成一致，并保证该集合是拟议集合并集的子集。直观而言，SBC 会高效地运行二元共识，对每个参与者提出的集合达成一致；若结果为肯定，则所提出的集合会被纳入最终的集合共识中。我们列出与 Setchain 相关的性质（第5节）：

- **SBC 终止性**：每个正确进程最终都会决定一个元素集合。
- **SBC 一致性**：不存在两个正确进程决定不同的元素集合。
- **SBC有效性**：已决定的交易集合是拟议集合并集的子集。
- **SBC非平凡性**：如果所有进程都正确且提出相同的集合，则该集合就是已决定集合。

RedBelly算法 [6] 在部分同步系统中解决了SBC问题：存在一个未知的全局稳定时间，在此之后通信将是同步的。（其他SBC算法可能具有不同的部分同步假设。）随后， [6] 提出使用 SBC替代区块链中的共识算法，以寻求提升可扩展性，因为下一 个区块中要包含的所有交易都可以通过执行一次SBC算法来决定。在RedBelly中，每台服务器都通过应用一个对已决定的交易集合进行全序排列的确定性函数来计算相同的区块，并移除无效交易或冲突交易。

我们使用 SBC 的方式不同于实现区块链。我们用它来同步分布式只增集合各局部视图之间的屏障。为确保所有元素最终都被分配到某个纪元，我们使用的 SBC 服务需要满足以下性质。

- **SBC 抗审查性**：存在一个时间 τ，在此之后，如果所有正确进程的拟议集合都包含同一个元素 e，那么 e 将出现在已决定集合中。

在 RedBelly 中，这一性质成立，因为在全局稳定时间之后，所有集合共识轮次都会决定来自正确进程的集合 [6,定理 3]。

3 Setchain 分布式数据结构

Setchain 的一个关键概念是纪元编号，它是分布式数据结构维护的全局计数器。同步屏障通过纪元变更来实现：纪元编号递增，且只增集合中尚未分配到先前纪元的元素会被标记为新的纪元编号。

3.1 Setchain 的运行方式

在介绍 Setchain 的 API 之前，我们先分析一个元素在 Setchain 中应当经历的预期路径，以及我们希望该结构具备的性质。Setchain 侧链数据结构的主要目标是利用集合内部无序这一特点所带来的效率优势，通过快速的只增集合，让用户能够添加记录（未解释的数据），从而获得这些记录存在的证明。用户可以定期或主动触发纪元变更，以生成成员资格证明，或清晰地划分 Setchain 的演进阶段。因此，Setchain 提供三种方法：添加，获取和 epoch_inc。

在集中式实现中，我们期望执行添加操作后立即能找到插入的元素。换言之，执行 add(e) 后，我们期望元素 e 出现在 get 的结果中。在分布式环境中，这种要求过于严格，因此我们转而期望元素最终会出现在集合中。此外，实现还必须保证不同正确节点之间的一致性，也就是说，它们不能相互矛盾。

3.2 Setchain 的 API 和服务器状态

我们考虑一个元素全集 U，客户端进程可以将其中的元素注入集合中。我们还假设服务器可以在本地验证元素 $e \in U$。**Setchain** 是一种分布式数据结构，其中一组服务器节点 $\mathbb{D}$ 负责维护：

- 一个集合，用于存储已添加的元素_set $\subseteq U$；
- 一个自然数纪元 $\in \mathbb{N}$；
- 一个映射 history：$[1..\text{epoch}] \to \mathrm{P}(U)$，用于描述已标记纪元编号的元素集合（$\mathrm{P}(U)$ 表示 U 的幂集）。

每个服务器节点 $v \in \mathbb{D}$ 都支持三种操作，任何客户端进程均可使用：

- v.add(e)：请求将 e 添加到_集合中。
- v.获取()：返回集合_集合、历史记录和纪元，以及 v[2]所感知到的值。
- v.纪元_inc(h)会触发纪元变更（即同步屏障）。必须满足h = 纪元 + 1。

简单来说，客户端进程 p 会在节点 v 上调用 v.get()操作，以获取（S，H，h），即v对集合 v_集合和映射 v.历史记录的视图，其定义域为 $[1 \ldots h]$。进程 p 调用 v.add(e)，将新元素 e 插入 v.集合_中，并调用 v.epoch_inc(h + 1)来请求纪元递增。在服务器 v上，集合 v_集合包含 v 关于已添加元素的知识，包括尚未分配纪元的元素；而 v.历史记录仅包含已分配纪元的元素。典型情况下，元素 $e \in U$ 最初会被 v 感知为属于_集合，之后在纪元递增时被标记并复制到历史记录中。不过，正如我们将看到的，某些实现允许以其他方式插入元素，在这种情况下，v 会在纪元变更期间首次获知 e。epoch_inc()操作会启动一个过程：收集各节点_集合中的元素，并协同决定哪些元素要盖上当前纪元的标记。

最初，每个正确服务器中的_集合、历史记录和纪元 = 0 都为空。请注意，客户端进程可以通过 Add() 向_集合插入元素，但只有服务器能够决定如何更新历史记录；客户端进程只能通过调用纪元_inc() 来影响这一过程。

在某个时间点，不同服务器所看到的_集合可能并不相同。我们提出的算法只提供最终一致性保证，其定义见下一节。

3.3 期望的性质

现在，我们规定 Setchain 正确实现的性质。首先，我们给出一个底层规范，该规范假定客户端与正确的服务器进行交互。尽管客户端无法确定所联系的服务器是否正确，我们仍将在第7 节利用这些性质构建两个正确的客户端：(1) 一个悲观客户端，它会联系多台服务器，以确保其中有足够多的服务器正确；(2) 一个乐观客户端，它只联系一台服务器（希望该服务器是正确的），并可在之后检查操作是否成功。

我们首先要求 Setchain 满足：在正确服务器上发出的每个添加、获取和 epoch_inc 操作最终都会终止。若元素 e 属于 H，则存在一个纪元 i，使得$e \in H(i)$。若元素 e 在历史记录 H 中的纪元 i 内（例如，由一次 get 调用返回），则称该元素处于该纪元中 $e \in H(i)$。第一条性质指出，纪元中只包含来自只增集合的元素。

性质 1（一致集合）。设$(S, H, h) = v$.获取() 是对正确服务器 v 的一次调用的结果，其中 e 是有效元素。那么，对于每个 $i \le h, H(i) \subseteq S$。

[2]In 在实践中，我们会提供其他查询操作，因为 get() 操作返回的值可能会不断增长 e.

第二个性质指出，添加到正确服务器的每个元素最终都会在此后对同一服务器发出的所有获取操作中返回。

性质 2（添加-获取-本地）。 设 v.添加(e) 是对正确服务器 v发起的一项操作。那么， 最终所有调用 $(S, H, h) = v$.获取() 都满足 $e \in S$。

下一个性质指出，存在于正确服务器上的元素会传播到所有正确服务器。

性质 3（获取-全局）。 设 v和 w是两台正确服务器，设 $e \in U$，并设 $(S, H, h) = v$.获取()。如果 $e \in S$，那么最终所有调用 $(S', H', h') = w$.获取() 都满足 $e \in S'$。

在本文余下部分中，我们假设在每个时间点之后，都存在一个未来时刻，此时会调用并完成 epoch_inc()。在任何现实的实际场景中，这都是一个合理的假设，因为可以轻松地利用超时机制来保证这一点。接下来的性质指出，所有添加的元素最终都会被分配一个纪元。

性质 4（最终获取）。设 v 是一台正确服务器，令 $e \in U$，且 $(S, H, h) = v$.获取()。若$e \in S$，则最终所有调用 $(S', H', h') = v$.获取() 都满足 $e \in H'$。

前述三个性质蕴含以下性质。

性质 5（添加后获取）。设 v.添加(e) 是在正确服务器 v 上调用的一项操作，且$e \in U$。那么，最终所有在正确服务器 w 上的调用 $(S, H, h) = w$.获取() 都满足 $e \in H$。

一个元素至多属于一个时期；即使历史集合是通过对两个不同（正确）服务器的 get 调用获得的，也没有元素会属于两个不同的时期。

性质 6（唯一纪元）。设 v 是一台正确服务器，$(S, H, h) = v$.获取()，并设 $i, i' \leq h$ 且$i \neq i'$。那么， $H(i) \cap H(i') = \emptyset$。

所有正确服务器进程都对纪元内容达成一致。

性质 7（一致获取）。设 v、 w 为正确服务器，设 $(S, H, h) = v$.获取() 和 $(S', H', h') = w$.获取()，并设 $i \leq \min(h, h')$。则 $H(i) = H'(i)$。

性质 7 指出，两个 获取 调用返回给正确服务器的历史记录，必有一个是另一个的前缀。然而，由于两个元素 e和 e' 可以插入到两个不同的正确服务器中——而传播需要时间——，因此，从两个正确服务器获取的该_集合部分获取可能彼此不包含。

最后，我们要求历史记录中的每个元素都来自客户端添加该元素所产生的结果。

性质 8（先添加后获取）。设 v 为正确服务器，$(S, H, h) = v$.获取()，且 $e \in S$。那么，过去某个服务器 w 上曾发生过操作 w.添加(e)。

Prop质1、6、7 和8 是安全性质。性质2、3、4 和5 是活性性质 s.

4 实现

本节介绍满足第3节所述性质的 Setchain 实现。我们先介绍一种集中式顺序实现，以建立直观理解，然后介绍三种分布式实现。第一种分布式实现使用拜占庭分布式增长型集合对象（DSO）来维护该_集合，并使用拜占庭原子广播（BAB）来实现纪元递增。第二种分布式实现同样基于 DSO，但将 BAB 替换为拜占庭可靠广播（BRB），用于宣布纪元递增，并使用集合拜占庭共识

Central 算法 单服务器实现。

```
1:  初始化：纪元 ← 0,        历史记录 ← ∅
2:  初始化：该_集合 ← ∅
3:  函数获取( )
4:      返回 （the_集合，历史记录，纪元）
5:  函数添加(e)
6:      断言 valid（e）
7:      the_集合 ← the_集合 ∪ {e}
8:  函数EpochInc(h)
9:      断言 h ≡ 纪元 + 1
10:     提案 ← 该_集合 \ ⋃纪元 k=1 历史记录(k)
11:     历史记录 ← 历史记录 ∪ {h，提案}
12:     纪元 ← 纪元 + 1
```

（SBC）处理纪元变更。最后，我们用本地集合替换 DSO，并使用 BRB 广播元素和纪元递增公告，使用 SBC 处理纪元变更，从而得到速度最快的实现。

4.1 顺序实现

Alg. Central 展示了一种集中式解决方案，它维护两个本地集合：_集合——用于记录添加的元素——以及历史记录，其中保存着一组对 h, A，其中 h 是纪元编号，A 是元素集合。我们用 history(h) 表示历史记录中对 h, A 的 A。每当出现一个新纪元时，自然数纪元编号就会递增。操作包括：Add(e)，它会检查元素 e 是否有效，并将其添加到_集合中；以及 Get()，它返回（_集合、历史记录、纪元编号）。

添加元素只有一种方式，即使用操作 Add()。由于算法 Central不维护分布式数据结构，而是维护集中式数据结构，因此不存在拜占庭节点。因此，客户端每次与唯一的服务器交互时，该服务器都是正确的。以下实现均为分布式实现，因此必须纳入某种机制，以防止拜占庭节点污染或操纵 Setchain。

4.2 分布式实现

4.2.1 第一种方法。DSO 和 BAB。Basic 算法使用两种外部服务：DSO 和 BAB。我们用消息名称后跟其内容的形式表示消息，例如“epinc(h, proposal i)”。变量 the_set 不再是本地集合，而是在第 2 行通过 Init() 初始化为空的 DSO。函数 Get() 调用 DSO Get() 函数（第 4 行）以获取元素集合。函数 EpochInc(h) 触发纪元递增所需的机制，并就属于纪元 h 的元素达成共识。共识过程首先计算本地提议集合，即所有已添加但尚未加盖标记的元素（第 10 行）。随后，通过 BAB 服务广播该提案集合，同时发送纪元编号 h 和服务器节点标识 i（第 11 行）。然后，服务器等待接收恰好 $2f + 1$ 个提案，并保留至少出现在 $f + 1$ 个提案中的元素集合 E，从而保证每个元素 $e \in E$ 都至少由一个正确服务器提出。BAB 的使用保证，正确服务器发送的每条消息最终都会按相同顺序到达其他每个正确服务器，因此所有正确服务器都拥有相同的 $2f + 1$ 提案集合。于是，所有正确服务器都会得出相同结论，并在第 16 行将集合 E 作为纪元 h 添加到历史记录中。

算法基础 服务器 i 实现，使用 DSO 和 BAB

1: **初始化：** 纪元 $\leftarrow 0$，　历史记录 $\leftarrow \emptyset$
2: **初始化：** 该_集合 $\leftarrow$DSO.Init()
3: **函数**获取( )
4: 　**返回** （该_集合.Get() $\cup$ 历史记录，历史记录，纪元）
5: **函数**添加(e)
6: 　**断言** 有效(e)
7: 　该_集合.添加(e)
8: **函数**EpochInc(h)
9: 　**断言** $h \equiv$ 纪元 $+1$
10: 　提案 $\leftarrow$ 此_集合。获取() $\setminus \bigcup_{k=1}$纪元　历史记录(k)
11: 　BAB.广播(epinc(h，提案 i))
12: **当** (BAB.交付(epinc(h，提案 j))
13: 　　来自 $2f+1$ 不同的服务器 j 对于相同的 h) **执行**
14: 　**断言** $h \equiv$ 纪元 $+1$
15: 　$E \leftarrow \{e:\ e \in$ 至少针对 $f+1$ 个不同 j 的提案$\}$
16: 　历史记录 $\leftarrow$ 历史记录 $\cup \{h,\ E\}$
17: 　纪元 $\leftarrow$ 纪元 $+1$

慢速算法服务器 i 实现，使用 DSO、BRB 和 SBC。

7: ...　▷ Get 和 Add 与 Basic 算法中的相同
8: **函数**EpochInc(h)
9: 　**断言** $h \equiv$ 纪元 $+1$
10: 　BRB.Broadcast(epinc(h))
11: **当** (BRB.Deliver(epinc(h)) 且 $h <$ 纪元 $+1$) **执行**
12: 　**drop**
13: **当** (BRB.Deliver(epinc(h)) 且 $h \equiv$ 纪元 $+1$) **执行**
14: 　**断言** $prop[h] \equiv null$
15: 　$prop[h] \leftarrow$ 此_集合。Get() $\setminus \bigcup_{k=1}$纪元 历史记录(k)
16: 　SBC[h].Propose($prop[h]$)
17: **当**(SBC[h].SetDeliver($propset$) 且 $h \equiv$ 纪元 $+1$) **执行**
18: 　$E \leftarrow \{e:\ e \in propset[j],\ valid(e) \wedge e \notin$历史记录$\}$
19: 　该_集合 $\leftarrow$ 该_集合.$Add(E)$
20: 　历史记录 $\leftarrow$ 历史记录 $\cup \{h,\ E\}$
21: 　纪元 $\leftarrow$ 纪元 $+1$

算法 Basic虽然易于理解且正确性证明简单，但效率不高。首先，要完成一次纪元递增，至少需要向不同服务器调用 $3f+1$ 次 EpochInc(h)，因此至少会接收到$2f+1$ 个提案（f 个拜占庭服务器可能不会提出任何提案）。低效的另一个来源是使用现成的构建模块。例如，每次调用 DSO Get() 时，都要交换许多消息来计算集合 [5]的可靠本地视图。同样，每次纪元变更都需要在第 () 行调用 DSO Get10 以创建提案。此外，第 13 行要求等待 $2f+1$ 次原子广播交付完成。BAB 服务最自然的实现方式是每交付一条消息就执行一次共识（见 [4]中的图 7），这会使该算法非常缓慢。我们通过两种替代算法解决这些问题。

快速算法 使用本地集合、BRB 和 SBC 的服务器实现。

1: **初始化：** 纪元 $\leftarrow 0$, 历史记录 $\leftarrow \emptyset$
2: **初始化：** 该_集合 $\leftarrow \emptyset$
3: **函数**获取( )
4: **返回** (该_集合,历史记录,纪元)
5: **函数**添加(e)
6: **断言** $valid(e)$ 且 $e \notin$ 该_集合
7: BRB.Broadcast(add(e))
8: **当**(BRB.Deliver(add(e))) **执行**
9: **断言** $valid(e)$
10: 该_集合 $\leftarrow$ 该_集合 $\cup \{e\}$
11: **函数**EpochInc(h)
12: **断言** $h \equiv$ 纪元 + 1
13: BRB.Broadcast(epinc(h))
14: **当** (BRB.Deliver(epinc(h)) 且 $h <$ 纪元 + 1) **执行**
15: **drop**
16: **当** (BRB.Deliver(epinc(h)) and $h \equiv$ 纪元 + 1) **执行**
17: **断言** $prop[h] \equiv \emptyset$
18: $prop[h] \leftarrow$ 该_集合 $\setminus \bigcup_{\text{纪元 } k=1}$ 历史记录(k)
19: SBC[h].Propose($prop[h]$)
20: **当** (SBC[h].SetDeliver($propset$) and $h \equiv$ 纪元 + 1) **执行**
21: $E \leftarrow \{e$: $e \in propset[j]$, $valid(e) \wedge e \notin$历史记录$\}$
22: 这个_集合 $\leftarrow$ 这个_集合 $\cup E$
23: 历史记录 $\leftarrow$ 历史记录 $\cup \{h, E\}$
24: 纪元 $\leftarrow$ 纪元 + 1

4.2.2 第二种方法：避免 *BAB*。 算法 Slow提升了算法 Basic 的性能，具体如下。首先，它使用 BRB 来传播纪元递增。其次，用 SBC 算法取代 BAB 以及等待第 $2f + 1$ 条消息到达的做法（见算法 13 中的第 Basic 行），从而可以同时解决多个共识实例。

理想情况下，当触发 EpochInc(h) 时，每个正确服务器本地 的_集合 中的未盖章元素都应盖上新的纪元编号，并添加到集合历史记录中。不过，我们需要确保每个纪元的集合历史记录 在每个正确服务器上都相同。算法 Basic 使用 BAB 并统计足够数量的已接收消息来实现这一点。算法 Slow 则使用 SBC 同时解决多个独立共识实例，每个实例对应一个参与者提案。第 10 行广播纪元变更邀请，促使正确服务器构建拟议集合，并通过 SBC 提议该集合。每次纪元变更都对应一个 SBC 实例h，由 SBC[h]标识。SBC 服务保证每个正确服务器都决定相同的提案集合（其中每个提案都是一个元素集合）。随后，每个节点对相同的提案集合应用相同的函数，从而就如何更新 历史记录(h)得出相同结论。该函数会保留有效且未盖章的元素。值得注意的是，这使得我们可以在纪元变更期间直接提议元素来添加它们，而无需事先广播。这项优化将在第 6 节中加以利用，以进一步加快算法。最后需要指出的是，算法 Slow 允许拜占庭服务器绕过 Add() 来提议元素，只要元素有效，就会被接受。这等同于客户端通过 Add()操作提议一个元素，随后该元素在纪元变更期间成功传播。请注意，算法Slow 仍会在每台服务器上触发一次 DSO Get() 调用，以构建本地提案。

4.2.3 最终方案。*BRB* 和 *SBC*，不使用 *DSO*。 算法 Fast 通过剖析 DSO Get() 调用的内部机制，并将其中的内部步骤直接纳入 Setchain 算法，避免了 DSO Get() 调用所需的一连串消息。这个思路利用了这样一个事实： 一个正确的 Setchain 服务器 本身就是 DSO 的正确客户端，因此 DSO 无需采取防御措施。这一观察表明，使用拜占庭容错构件并不能实现高效组合，但探讨这一一般性思路超出了本文的范围。

算法 Fast 通过该_集合 使用本地集合（第2行）。在 Add(e) 中接收的元素会通过 BRB 进行传播。在任意时刻，两个正确服务器的本地集合可能不同（这是由于 BRB 交付待处理），但在一个服务器中添加的每个元素最终都会被所有其他服务器知晓。局部变量 历史记录 仅在第 23 行因 SBC 轮次而更新。因此，所有正确服务器都会对由某些服务器提出的未盖时间戳元素所组成的相同集合达成一致。此外，算法 Fast 会更新该_集合 以纳入服务器尚未拥有的元素（第 22 行），从而保证 历史记录 中的所有元素也都包含在该_集合中。

5 正确性证明

现在，我们证明第 4 节中介绍的分布式算法满足第3节所引入的期望性质。

以下引理讨论元素如何被赋予时间戳。它们分别直接蕴含性质 4(最终获取)、6 (唯一纪元) 和 7 (一致获取)。我们将在以下小节中证明算法满足这些引理。

引理 1. 设 v 和 w 为正确服务器。如果 $e \in v$.该_集合。那么最终 e 会出现在 w.历史记录 中 .

引理 2。 设 v 为正确服务器， h、 h' 为两个纪元编号，且 $h \neq h'$。如果 $e \in v$.历史记录(h)，那么 $e \notin v$.历史记录(h')。

引理 3。 设 v 和 w 为正确服务器。设 h 满足 $h \leq v$.纪元 且 $h \leq w$.纪元。那么 v.历史记录(h) $= w$.历史记录(h)。

很容易看出，性质 5 (添加后获取) 直接由性质2、3 和4推出。

5.1 算法正确性。基本

对于 Basic 算法，性质 1（一致集合）显然成立，因为 Get() 的响应是元组（the_set.get()∪历史记录,history, 纪元）（见第 4 行）。性质 2（添加-本地获取）、性质 3（全局获取）和性质 8（先添加后获取）直接由 DSO-添加获取和 DSO-获取添加性质推出，因为这些性质讨论的都是如何向_集合中添加元素，而该集合由 DSO 实现。

现在我们证明引理 1、2 和3。我们先证明引理 3，该引理指出，两个正确服务器 v和 w在都到达 h 之后，对纪元 h 的内容达成一致。

命题 5.1。引理 *1*、*2* 和*3* 对于 *Basic* 算法均成立。

证明。我们先证明引理 3，再证明引理1 ，最后证明引理 2。

- 我们首先证明引理 3。该引理指出，两个正确服务器 v和 w在都到达 h之后，对纪元 h的内容达成一致。变量 history 仅在第 16 行更新。由于 v和 w是正确服务器，且计算了 history(h)，它们都必须从 $2f + 1$ 个不同的服务器接收形如 epinc(h, *proposal*, j) 的 BRB.Deliver 消息。BRB 终止性（全局）和全序性质保证两个服务器收到的是相同的消息。 v和 w都以相同的方式筛选元素，只保留由某条已交付消息中的 $f + 1$ 服务器提出的元素。因此，在第 16 行， v和w更新的 history(h) 相同。由此，引理 3 得证。

• 现在证明引理 1。设 e 是某个正确服务器的_集合中的一个元素。由性质 3 可知，存在一个时间点 t，此后所有正确服务器上的_set.get() 返回的集合都包含 e。请注意，我们假设纪元总会递增，特别是，在 t之后会有一次新的 EpochInc(h)。如果 e 已经在 $h' < h$的 history(w.') 中，那么证明完成。否则，根据引理 3，对于任何正确服务器 z，e 都不会出现在其 history(z.') 中，其中 $h' < h$。于是，在第 10 行计算纪元 h的提案时，所有正确服务器都会将 e纳入其集合。为了计算纪元 h，服务器 w 会等待从 $2f + 1$ 个不同的服务器接收形如 epinc(h, *proposal*, j) 的 BRB.Deliver 消息（见第 13 行），其中至少有 $f+1$ 条消息来自正确服务器。因此，收到的提案中至少有 $f+1$ 包含 e，这意味着服务器 w 会将e 纳入其历史集合 history(h)。由此，引理 1 得证。

• 最后，我们使用引理 3 来证明引理 2。证明采用反证法。假设 $e \in v$.history(h')。更新 history(h') 的唯一位置是第 16 行，在此处，history(h') 被定义为恰好包含不同服务器为纪元 h'提出的元素。因此，鉴于拜占庭服务器至多有 f 台，至少有一台正确服务器为纪元 h' 提出了一个包含 e 的集合。令 w 为其中一台正确服务器。服务器 w 计算纪元 h' 的提案（见第 10 行）时，将其设为_set.Get() $\setminus \bigcup_{k=1}^{h_r-1}$ history(k)。但是，根据引理 3，w.history(h) = v.history(h)，这意味着$e \in \bigcup_{k=1}^{h_r-1}$ history(k)，因为 $h \leq h' - 1$。因此，$e \notin$ the_set.Get() $\setminus \bigcup_{k=1}^{h_r-1}$ history(k)，也就是说，e并未由 w 为纪元 h' 提出。这与假设 $e \in v$.history(h')相矛盾。因此，引理 2 对 Basic 算法成立。

□

5.2 Alg.Slow的正确性

我们可以用与证明 Alg. Slow相同的推理，证明其满足所有引理；只需将 BRB 和 BAB 的性质替换为 SBC 的相应性质。为了证明引理 1，我们不再依赖足够多的消息已通过 BRB.Deliver 交付，而是使用 **SBC-Censorship-Resistance**，它保证已决定集合将包含元素 e。引理 3的证明与 Alg.Basic 的证明类似，但我们不使用性质 **BRB 终止性（全局）**和 **全序**，而是使用性质 **SBC 一致性**，该性质保证正确服务器对已决定集合达成一致。

5.3 Alg.Fast的正确性

现在我们证明 Alg. Fast的正确性。首先，性质 2 (添加-本地获取) 可直接由 Add()、Get() 的第 4 行以及性质 **BRB 终止性（本地）**推出。其次，我们证明所有带有时间戳的元素都在 the_ 集合中，这意味着性质 1 (一致集合)。

引理 4. 在每个函数结束时以及执行 *Alg. Fast*时，$\bigcup_h v$.历史记录(h) $\subseteq v$.中的_集合，对于每个正确服务器 v均成立。

证明。 设 v 是一个正确服务器。向 v.历史记录 添加元素的唯一方式是在第 23行，该行前面是第 22 行，后者将相同的元素添加到 v.该_集合中。唯一其他会修改 v.该_集合 的指令是第 10 行，它只会使集合增大。 □

第三，以下引理表明，正确服务器中的元素会传播到所有正确服务器，这等价于性质3(全局获取)。

引理 5. 设 v 是一台正确服务器，e 是 v.该_集合中的一个元素。那么，对于每台正确服务器 w，e 最终都会在w.该_集合 中。

证明。最初，v._集合为空。有两种方式可以向v._集合添加元素e：(1) 在第 10 行，因此e有效，并通过 BRB.Deliver(add(e)) 接收。根据 BRB 终止性（全局），每台正确服务器w最终都会执行 BRB.Deliver(add(e))，随后（由于 e 有效），w 会在第 10 行将其添加到w._集合中。(2) 在第 22 行，因此元素 e 有效，并作为 propset 中某个集合的元素，通过 SBC[h].SetDeliver(propset) 接收，其中$h = v$.纪元为 $+1$。根据 SBC 终止性和 SBC 一致性，所有正确服务器对同一组提案达成一致。于是，w最终也会收到 SBC[h].SetDeliver(propset)。因此，如果 v 添加了 e，那么 w 要么也会添加它，要么它已经在自己的 w.历史记录中；根据引理 4，这意味着$e \in w$._集合。在这两种情况下，e最终都会在 w._集合中。□

最后，我们证明本节开头提出的引理，这些引理讨论了元素如何被标记。

命题 5.2。 引理 *1*、引理 *2* 和引理 *3*对 *Alg.Fast*成立。

证明。我们分别证明每个引理。

- 引理 1 （在正确服务器中添加的元素最终会在所有正确服务器中被标记）的证明与 Alg.Slow的证明类似。
- 引理 2 可直接由以下检查得出： e 不会在 v.历史记录(h) 处被注入 $e \in v$.历史记录 在第 23行中。

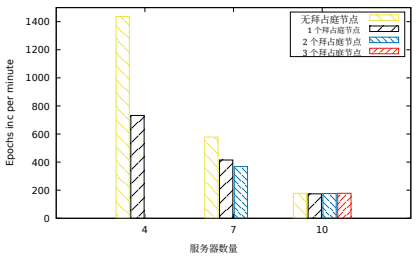
- 接下来，我们证明两个正确服务器 v 和 w 对双方计算出的历史记录前缀达成一致，即引理 3。证明通过对纪元编号 epoch 进行归纳来展开。基础情形为 epoch $= 0$，显然成立，因为 v.history(0) $= w$.history(0) $= \emptyset$。变量 epoch 仅会在第 24 行递增 1；此前，在第 23 行，当 $h =$ epoch$+1$时，history(h) 已发生变化。在该行中，v和 w处于 SBC 的同一阶段（针对相同的h）。根据 SBC 一致性，v和 w接收相同的属性集，二者都会以相同方式验证所有元素；并且（根据归纳假设），对于每个 $h' \leq$ epoch，都有 $e \in v$.history(h') 当且仅当 $e \in w$.history(h')。因此，在第 23 行，v 和 w 会以相同方式更新 history(h)；而在第 24 行之后，有 v.history(epoch) $= w$.history(epoch)。

□

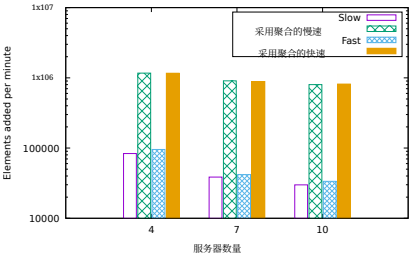
最后，我们讨论性质 8 (先添加后获取)。如果有效元素只能由客户端创建并通过 Add 添加(e)，则该性质显然成立。如果有效元素也可能由拜占庭服务器等创建，那么它们无需使用 Add，就能将元素注入._正确服务器的集合和历史记录中()。这些服务器既可以直接执行 BRB.Broadcast，也可以在纪元轮次中通过 SBC 直接执行。在这些情况下，算法 Fast满足（先添加后获取）的一个较弱版本，即 Get() 返回的元素要么由 Add()添加，要么通过 BRB.Broadcast 添加，或在 SBC 阶段注入。

6 实证评估

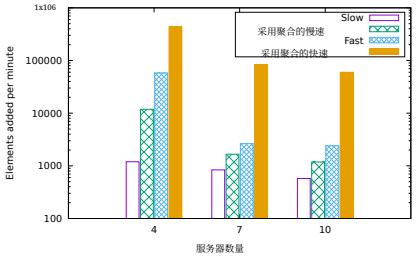
我们实现了算法 Slow 和算法 Fast的服务器端代码，并自行实现了 DSO、BRB 和 SBC。我们的原型使用 Go语言编写 [12] 1.16，消息传递采用 ZeroMQ [29]，运行于 TCP 之上。测试平台使用 Docker，运行在一台配备 2 颗 3GHz 英特尔至强中央处理器、共 36 个核心和 256GB 内存的服务器上，操作系统为 Ubuntu 18.04 Linux64。每台 Setchain 服务器都封装在 Docker 容器中，且不限制中央处理器或内存的使用。算法 Slow将 Setchain 和 DSO 实现为两个独立的可执行文件，并通过 Docker 容器内部的环回网络接口进行远程过程调用。RPC服务器和客户端采用 Go语言标准库提供的实现。对于算法 Fast，所有内容都包含在单个可执行文件中。对于这两种算法，我们分别评估了两个版本：一个版本中，每次插入元素



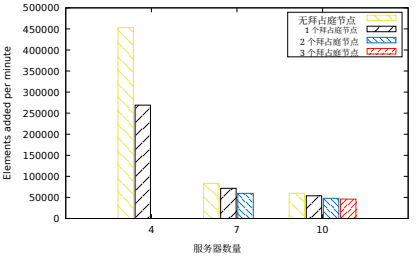
(a) 最大纪元变更次数



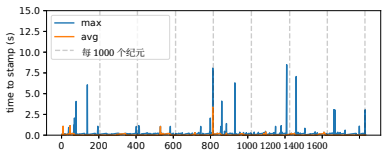
(b) 最大添加元素数（无纪元）



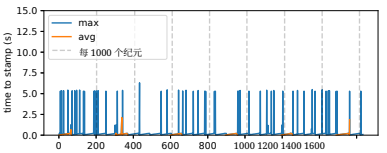
(c) 最大添加元素数（有纪元）



(d) 添加操作中的静默拜占庭效应（算法：采用聚合的快速）



（e）盖章时间（算法 Fast）



（f）盖章时间（算法 采用聚合的快速）

图1。实验结果。算法 Slow+集合和算法 采用聚合的快速 是采用聚合的算法版本。拜占庭服务器仅保持静默。

都会触发一次广播；另一个版本中，服务器会在本地聚合插入的元素，直到达到最大消息大小（ 10^6 个元素）或最长元素超时时间（5秒）。在所有情况下，元素大小均为 116–126 字节。算法 采用聚合的快速 实现了算法 Fast的聚合版本。

我们对以下假设进行了实证评估：

- （H1）：可插入元素的最大速率远高于纪元最大速率。
- （H2）：算法 Fast 的表现优于算法Slow。
- (H3)：聚合版本的表现优于对应的基础版本。
- (H4)：静默拜占庭服务器不会显著影响性能。
- (H5)：性能不会随时间推移而下降。

为评估假设 H1 至 H5，我们进行了下文所述的实验，结果见图 1。所有情况下，操作均由运行在同一 Docker 容器中的客户端注入。驻留内存始终充足，因此没有任何实验需要操作系统进行磁盘交换。所有实验均考虑部署 4、7 或 10 个服务器节点的情况，每项报告的实验结果均取自 10 次执行的平均值。

我们首先测试了 Setchain 实现每分钟能够处理多少个纪元。在这些测试中，我们没有添加任何元素，而是逐步提高纪元速率，以找出最小的

采用聚合的快速算法服务器实现使用本地集合、BRB 和 SBC。

初始化： 纪元 $\leftarrow 0$, 历史记录 $\leftarrow \emptyset$
初始化： 该_集合 $\leftarrow \emptyset$, to_广播 $\leftarrow \emptyset$
函数获取( )
  返回 (the_集合,历史记录,纪元)
函数添加(e)
  断言 $valid(e)$ 且 $e \notin$ the_集合
  到_广播 $\leftarrow$ 到_广播 $\cup \{e\}$
当 (BRB.Deliver(添加(s)))**时**
  断言 $valid(s)$
  该_集合 $\leftarrow$ 该_集合 $\cup s$
  向_广播 $\leftarrow$ 向_广播 $\setminus s$
  .....。 ▷ EpochInc 和 BRB。交付(epinc(h)) 如算法 Fast
当 (SBC[h].SetDeliver($propset$) 且 $h \equiv$ 纪元 $+ 1$) **执行**
  $E \leftarrow \{e:\ e \in propset[j],\ valid(e)\ \wedge e \notin$历史记录$\}$
  历史记录 $\leftarrow$ 历史记录 $\cup \{h,\ E\}$
  该_集合 $\leftarrow$ 该_集合 $\cup E$
  向_广播 $\leftarrow$ 向_广播 $\setminus E$
当 (|向_广播| > 1000000或 向_广播.$oldest$ > 5s) **执行**
  BRB.Broadcast(添加(to_broadcast))
  to_广播 $\leftarrow \emptyset$

相邻两个纪元之间的延迟。我们分别在包含 4、7 和 10 个节点的情况下运行测试，并分别测试有无拜占庭服务器的情况。结果如图1（a）所示。

在第二个实验中，我们通过实证估算了在不进行任何纪元递增的情况下，使用 Setchain 的四种不同实现（Alg.Slow 和 Alg. Fast，采用或不采用聚合）每分钟可以添加多少个元素。结果如图1(b) 所示。在该实验中，Alg. Slow和 Alg. Fast 的表现相近。采用聚合时，Alg. Slow 和 Alg. Fast 的表现也相近，但比不采用聚合时高一个数量级，这证实了 (H3)。图1(a) 和 (b) 合在一起表明，集合的速度比纪元变更快三个数量级，这证实了 (H1)。

第三个实验比较了我们的实现同时进行纪元递增和插入元素时的性能。我们将纪元速率设为每秒变更一个纪元，并计算添加操作的最大比率。结果如图 1(c) 所示，其中 Alg. Fast 优于 Alg.Slow。事实上，Alg. 采用聚合的快速 甚至优于 Alg. Slow+：对于 4 个节点，集合提升约 5 倍；对于 7 个和 10 个节点，则提升约 2 倍。Alg. 采用聚合的快速处理的元素数量是 Alg. Fast 的 8 倍（4 个节点）和 30 倍（7 个和 10 个节点）。随着节点数量增加，Alg. 采用聚合的快速 相较于 Alg. Fast 的优势也随之扩大，因为 Alg. 采用聚合的快速 避免了广播元素；广播会产生数量与网络中节点数平方成正比的消息。该实验证实了 (H2) 和 (H3)。在之前的实验（没有纪元变更）中，无法观察到 Alg. Fast 和 Alg. Slow 之间的差异，因为二者的主要区别在于纪元变更期间服务器收集元素以进行投票的方式。

接下来的实验探究静默拜占庭服务器对以下方案的影响：采用聚合的快速。我们实现了静默拜占庭服务器，并在纪元变更比例为每秒 1 个纪元的条件下，分别使用 4、7 和 10 个节点运行实验，计算最大添加速率。结果如图 1(d) 所示。静默拜占庭服务器会降低 4 个节点时的速度，因为在这种情况下，实现会非常频繁地将该服务器视为静默服务器

算法 5 DPO 的正确客户端协议（适用于算法 Slow 和Fast）。

函数DPO.Add(e)
  在 $f+1$ 个不同的服务器中调用 Add(e)。
函数DPO.Get( )
  在至少 个不同的服务器中调用 Get()。
  等待 $2f+1$ 响应 s.(该_集合,历史记录,纪元)
  $S \leftarrow \{e | e \in s.$至少在 $f+1$ 台服务器 s上的_集合$\}$
  $H \leftarrow \emptyset$
  $i \leftarrow 1$
  $N \leftarrow \{s : s.$纪元 $\geq i\}$
  当 $\exists E: |\{s \in N: s.$历史记录$(i) = E\}| \geq f+1$ **执行**
    $H \leftarrow H \cup \{i, E\}$
    $N \leftarrow N \setminus \{s : s.$历史记录$(i) \neq E\}$
    $N \leftarrow N \setminus \{s: s.$纪元 $= i\}$
    $i \leftarrow i+1$
  返回 $(S, H, i-1)$
函数DPO.EpochInc(h)
  调用 EpochInc(h)，在 $f+1$ 台不同的服务器上。

并处于验证阶段，但可以观察到，服务器数量越多，这种影响就越小，从而验证了 (H4)。

在最后一项实验中，我们使用 4 台服务器运行较长时间（30 分钟），设置每秒 5 个纪元的纪元比率，并以最大速率的 50% 添加请求。我们计算客户端请求添加元素的时刻与元素被盖上时间戳的时刻之间经过的时间。图 1(e) 和 (f) 展示了最后一秒内插入元素所花费的最长时间和平均时间。对于算法 Fast，在 30 分钟实验期间，最坏情况下耗时约 8 秒，但大多数元素在 1 秒或更短时间内便完成插入。对于算法 采用聚合的快速，长时间运行期间多次出现最长耗时 5 秒的情况（5 秒是触发广播的超时时间）。这是因为某个元素未能通过集合共识插入，最终只能通过广播来插入。两种情况下，长时间运行都不会导致性能下降，这证实了（H5）。

鉴于纪元变更本质上属于集合共识，我们的实验表明，在 Setchain 中插入元素的速度比执行共识快三个数量级。不过，要全面验证这一假设，就必须在高性能流言协议上完整实现 Setchain，并将其与可比的共识实现进行比较。

7 客户端协议

第 5 节中的所有性质和证明都考虑客户端联系正确服务器的情况，而第 4 节中的实现并不向客户端保证服务器是否正确。由于客户端进程无法知道自己联系的是拜占庭服务器还是正确服务器，因此需要客户端协议来保证其正确使用 Setchain 的接口。

本节将介绍两种这样的协议。首先，我们介绍一种受 [5], DSO客户端启发的客户端协议，每次操作都需要联系多台服务器。之后，我们将介绍一种基于尝试并检查的效率更高的“乐观”方案，它只需对服务器进行简单修改。

算法 6 扩展算法。 Fast 添加新的纪元哈希，并进行密码学签名。

```
22: ...                                                    ▷ 与算法中前面的行相同。 Fast
23: 当 (SBC[h].SetDeliver(propset) and h ≡ 纪元 + 1) 执行
24:     E ← {e: e ∈ propset[j], valid(e) ∧ e ∉ 历史记录}
25:     历史记录 ← 历史记录 ∪ {h, E}
26:     该_集合 ← 该_集合 ∪ E
27:     纪元 ← 纪元 + 1
28:     添加(epoch_signature(h, sign(hash(h, E))))
```

算法 7 DPO 的乐观客户端协议（适用于算法 Slow 和 Fast）。

```
1: 函数添加并检查(e)
2:     调用在 1 个服务器中添加(e)。
3:     等待 Δ_y
4:     调用在 1 个服务器中获取()。
5:     等待响应（该_集合，历史记录，纪元）
6:     如果 ∃E, i: 历史记录(i) = E ∧ e ∈ E ∧ (h, hash(E))
7:       由 f + 1 个不同的服务器签署后，位于_集合则
8:         返回 确定。
9:     else
10:        返回 失败。
```

7.1 Setchain 作为分布式偏序对象（DPO）

客户端协议的基本思路是与足够多的服务器交互，以确保有足够多的服务器执行所需操作。算法5 展示了客户端协议。对于添加和 EpochInc，为了确保至少联系到一台正确服务器，我们需要向不同的服务器发送 $f+1$ 个请求。请注意，根据协议的不同，每条消息可能会触发不同的广播。

函数 Get 的包装算法可分为两部分。首先，客户端协议联系$3f+1$ 个节点，并等待至少 $2f+1$ 个响应（f个拜占庭服务器可能拒绝响应）。来自服务器 s的响应是（s.该_集合的 s.历史记录 s.纪元）。随后，协议将 S计算为至少有 $f+1$ 个服务器（其中至少有一个正确服务器）知晓属于该_集合的那些元素。为计算 H，代码按纪元逐步处理，直到纪元 i，此时同意集合 E中的元素属于 i的服务器少于 $f+1$ 个（注意，如果有 $f+1$ 个服务器同意E 是纪元 i中的元素集合，情况确实如此）。我们还会从 N中移除那些不知道纪元 i的服务器（无论是运行缓慢的进程还是拜占庭服务器），以及错误报告了与 E 不同内容的服务器（拜占庭服务器）。此过程结束后，客户端返回 S、H以及计算出的最新纪元。可以保证历史记录 $\subseteq$ 该_集合。

7.2 Fast乐观客户端

我们还提出了一种速度更快的乐观客户端。在这种方法中，正确服务器还会在每个纪元 i，对 i中的元素集合进行密码学签名，生成其哈希，并将此哈希作为一个元素插入 Setchain，参见算法6。服务器向 Setchain 添加的额外元素数量与纪元数呈线性关系，其数量级远小于 Setchain 中的元素数量。

算法 7 展示了下文所述的乐观客户端。要向 Setchain 中插入一个元素 e，乐观客户端会向一台服务器发送一次 单独的 添加(e) 请求，并希望该服务器是正确服务器。

等待一段时间后，客户端会向**一台**服务器（同样可能是正确服务器或拜占庭服务器）调用获取操作，并检查 e 是否出现在某个纪元中，且该纪元的哈希由至少 $f+1$ 台不同服务器签名。请注意，仅仅收到一份历史记录，其中 e 位于某个纪元内，并不足以保证 e 已被添加或盖上时间戳，因为拜占庭服务器可能会撒谎。然而，由于密码签名无法伪造，如果 $f+1$ 台服务器对某个纪元的哈希进行签名（客户端也会计算该哈希），那么至少有一台正确服务器验证了该纪元的内容。相比之下，乐观客户端的延迟可能高于算法 5中的 DPO，因为乐观客户端在添加元素后需要等待，以检查该元素是否已插入（或重试）。另一方面，在通常能够联系到正确服务器的情况下，该客户端每次添加只需发送一条消息，每次获取也只需发送一条消息，从而大幅减少了消息交换量。请注意，每个添加请求都会触发一次 BRB.Broadcast，并产生消息级联，其数量与服务器数量的平方成正比。因此，这两种客户端在延迟和吞吐量之间有所取舍。

8 拜占庭行为建模

形式化证明拜占庭容错分布式算法的性质[3] 是一项极具挑战性的任务。本节介绍一种非确定性进程（见算法 8），它抽象了所有拜占庭进程的组合行为（算法 Fast）。这一技术将一个包含 $n-f$ 个正确服务器和 f 个拜占庭服务器的场景，转换为一个等效的场景，其中包含 $n-f$ 个正确服务器和一个非确定性服务器，从而能够利用许多近期技术，形式化证明（非拜占庭）分布式算法的性质。我们的非确定性进程抽象了拜占庭行为，即使拜占庭进程能够彼此即时通信以实施协同攻击，也同样适用。

Setchain 假设拜占庭进程无法伪造有效元素（第 2.1 节），只有在正确服务器发送这些元素，或客户端插入这些元素时，它们才会得知这些元素的存在。我们假设，一旦某个拜占庭进程接收到一个有效元素 e，所有其他拜占庭进程也都 e 知晓该元素。

我们为所有算法扩展一个新的原语 SBC[h].Inform(命题)。当服务器调用 SBC[h].Propose(命题)时会触发该原语，并满足以下性质：

- **SBC-Inform-Validity**：如果某个进程执行了 SBC[h].Inform(命题)，那么过去一定有其他进程执行过 SBC[h].Propose(命题)。

我们利用这一性质，使拜占庭服务器能够发现正确服务器在集合共识期间提出、但尚未分配纪元的元素。请注意，拜占庭服务器可以在响应 Get() 调用时返回已提出但尚未分配纪元的元素。然而，如果不包含原语 SBC[h].Inform，我们就无法知道这些元素的来源。

我们假设存在函数 *havoc_子集*、*havoc_划分*、*havoc_元素*、*havoc_数字*和 *havoc_无效_元素*，它们分别从给定集合中生成随机子集、元素和划分，以及随机数和随机无效元素。我们不关注这些函数的语义，而是用它们来模拟拜占庭进程从已知值集合中任意选取元素组成集合。

我们在算法中对所有拜占庭进程的组合行为进行建模。8，其中它维护一个本地集合知识，用于记录“拜占庭”服务器所知的所有有效元素。它提供与算法相同的接口。Fast，并额外提供一个函数 Start。进程启动时会调用函数 Start，并在任意时间非确定性地发出消息（第21-27行），使用 BRB 和 SBC 原语，因为只有这些消息可能对正确服务器产生影响。请注意

[3]我们指的是严谨、可由机器复现或验证的证明。

算法 8 捕获拜占庭行为

```
1: 初始化：知识 ← ∅
2: 函数获取( )
3:     返回 (havoc_子集(知识 ∪生成_无效_元素()),
4:             havoc_划分(havoc_子集(知识 ∪ 生成_无效_元素())), havoc_数量())
5: 函数添加(e)
6:     断言 valid(e)
7:     知识 ← 知识 ∪ {e}
8: 当(BRB.Deliver(add(e))) 执行
9:     断言 valid(e)
10:    知识 ← 知识 ∪ {e}
11: 函数EpochInc(h)
12:    返回
13: 当 (BRB.Deliver(epinc(h)))执行
14:    无操作
15: 当(SBC[h].SetDeliver(propset)) 执行
16:    知识 ← 知识 ∪{e： e ∈ propset ∧ valid(e)}
17: 当(SBC[h].通知(prop)) 执行
18:    知识 ← 知识 ∪{e： e ∈ prop ∧ valid(e)}
19: 函数Start
20:    当真 执行
21:        BRB.广播(add(任意_元素(知识 ∪ 生成_无效_元素())))
22:        ||
23:        BRB.广播(epinc(任意_数字()))
24:        ||
25:        SBC[任意_数字()].提议(任意_子集(知识 ∪ 生成_无效_元素()))
26:        ||
27:        无
```

这些消息可能包含无效元素，也可能包含该非确定性进程已知的有效元素。同样，当客户端调用获取()时，它会返回$(s_v \cup s_i,\ partition(s'_v \cup s'_i)\ h)$，其中$s_v$和$s'_v$是来自知识的有效元素集合，而$s_i$和$s'_i$是无效元素集合（第2-4行）。收到消息后，非确定性进程会将所有新发现的有效元素记录到其本地集合知识中。

我们证明，任何由以下服务器维护的 Setchain 执行 n 实现算法Fast 且其中至多有 $1 \le f < n/3$ 个拜占庭服务器，都可以映射为由以下服务器维护的 Setchain 执行$n - f$**正确的**服务器实现算法Fast 以及一**个**服务器实现算法8，反之亦然。

我们用 Γ 来表示模型，该模型描述由以下进程维护的 Setchain 的执行 n这些进程实现算法Fast ，其中 $1 \le f < n/3$个是拜占庭服务器。

我们用 $\mathbb{C} \subset \mathbb{D}$ 表示 $n - f$正确服务器的集合，用 $\{b_0、\ldots\ldots、\ b_{f-1}\} = \mathbb{B} \subset \mathbb{D} \setminus \mathbb{C}$ 表示拜占庭服务器的集合。

事件。我们用事件表示用户和服务器与 Setchain 之间可能发生的不同交互，以及 Setchain 达成共识这一内部事件。

- get() 表示对函数 Get 的调用，
- add(e) 表示对函数 Add(e) 的调用，
- BRB.Broadcast(x) 表示通过网络广播添加或 epinc 消息，分别为x = add(e) 或 x = epinc(h)，

- BRB.Deliver(x) 表示消息 x的接收,
- EpochInc(h) 表示函数 EpochInc(h) 的调用,
- SBC[h].Propose(命题) 表示在 h SBC 实例中提出命题,
- SBC[h].Inform(命题) 表示收到 SBC[h].Propose(命题),
- SBC[h].SetDeliver(属性集) 表示属性集是 h SBC 实例的结果,
- SBC[h].Consensus(属性集) 是一个内部事件，表示已达成纪元 h 的共识,

- 空操作表示什么都没有发生的事件。

事件 Consensus(属性集) 表示集合共识最终确定的时间点，届时所有正确服务器都会就集合 属性集达成一致。

除 Consensus 和空操作外，所有事件ev都发生在某个特定服务器 s 上，因此它们具有一个属性ev.server，该属性返回 s。

网络。我们将网络建模为从服务器映射到形如 (sent, pending, received) 的元组。对于服务器 s， $\Delta(s)$.sent 是从服务器 s 发送到其他服务器的消息序列， $\Delta(s)$.pending 是一个多重集，其中包含从其他服务器发送而来且 s 尚未处理的所有消息，而 $\Delta(s)$.received 是服务器 s接收的消息序列。服务器发送或接收消息时，网络状态会发生变化。当 s 使用 BRB.Broadcast(m) 发送消息时， m 会被添加到 s 的已发送消息序列中，并添加到所有其他服务器的待处理多重集中。类似地，当服务器在 SBC 的 h 实例中提出命题 时，消息 $m = (h, prop)$会被添加到其已发送消息序列中，并添加到所有服务器的待处理多重集中:

$$\text{send}(\Delta, m, s)(s') = \begin{cases} (\Delta(s).\text{sent} +\!\!+ m, \Delta(s).\text{received}, \Delta(s).\text{pending} \cup \{m\}) & \text{if } s = s' \\ (\Delta(s').\text{sent}, \Delta(s').\text{received}, \Delta(s').\text{pending} \cup \{m\}) & \text{otherwise} \end{cases}$$

当服务器接收到消息 m 时，会从其待处理多重集中移除 m，并将其插入已接收序列:

$$\text{receive}(\Delta, m, s)(s') = \begin{cases} (\Delta(s).\text{sent}, \Delta(s).\text{received} +\!\!+ m, \Delta(s).\text{pending} \setminus \{m\}) & \text{if } s = s' \\ \Delta(s') & \text{otherwise} \end{cases}$$

模型 Γ 的配置 $\Phi = (\Sigma,\ \Delta,\ H,\ K)$ 由以下部分组成：一个将正确服务器映射到其本地状态的状态映射 Σ；一个包含服务器间发送消息的网络 Δ；一个从纪元编号映射到元素集合的部分映射 H（即已达成共识的历史记录）；以及一个有效元素集合K，其中的元素已披露给某个拜占庭服务器（也就是说，我们假设拜占庭服务器之间可以即时通信）。

初始配置 $\Phi_0 = (\Sigma_0,\ \Delta_0,\ H_0,\ K_0)$ 满足：$\Sigma_0(s)$ 是每个正确进程的初始状态， Δ_0 是空网络， H_0 是空映射， K_0 是空集。

我们如下定义事件ev在配置 $(\Sigma,\ \Delta,\ H,\ K)$ 中是否可执行:

- get() 和空操作始终可执行,
- add(e) 可执行的条件是 e 有效，且满足以下任一条件： $s = \text{add}(e).\text{server}$ 是拜占庭服务器，或者 $e \notin \Sigma(s).S$,
- BRB.广播(x) 在以下条件下启用：BRB.广播(x).服务器是拜占庭服务器，且$x = \text{epinc}(h)$ 或 $x =$ 添加(e) 与 $e \in K$ 或 e 无效,
- BRB.交付(add(e)) 在以下条件下启用：add(e) $\in \Delta(s)$.待处理消息，且 e 有效,
- EpochInc(h) 在以下条件之一成立时启用： $s =$ EpochInc(h).服务器是拜占庭服务器，或 $h = \Sigma(s)$.纪元 $+1$,
- BRB。交付(epinc(h))在满足以下条件时启用：epinc(h) $\in \Delta(s)$。待处理消息，并且 $s = ev$。服务器是拜占庭服务器， $h < \Sigma(s)$。纪元 $+1$ 或 $h = \Sigma(s)$。纪元 $+1$ 加上 s 尚未为 SBC 的第 h 个实例提出任何内容： $\nexists$命题 : $(h, prop) \in \Delta(s).sent$,

• SBC[h].提议(属性集)在满足以下条件时启用：SBC[h].提议(属性集)。服务器是拜占庭服务器，并且拜占庭服务器已知属性集 中的所有有效元素：$\{e \in$ 属性集 $: valid(e)\} \subseteq K$,

• SBC[h].Inform(命题) 在满足以下条件时启用：$h, prop \in \Delta(s)$。待处理消息,

• SBC[h].SetDeliver(属性集) 在满足以下条件时启用：$H(h) =$ 属性集 且满足以下条件之一：$s = ev$。服务器是拜占庭服务器，或 $h = \Sigma(s)$。纪元 $+1$,

• SBC[h].Consensus(属性集) 在满足以下条件时启用：$H(h-1)$ 已定义，$H(h)$ 未定义，至少有一个进程在 ($\exists s, h,$ 命题 : $(h,$ 命题$) \in \Delta(s)$。已发送) 之前提议了一个集合，并且 属性集 是为 hSBC 实例提议的所有元素的并集的子集 (属性集 $\subseteq \bigcup_r \{p : (h, p) \in \Delta(r)$。已发送),

然后，我们将事件 的效果 定义为：事件 ev作用于配置 (Σ, Δ, H, K)，其中 ev在 (Σ, Δ, H, K) 中启用时，得到以下配置 $(\Sigma', \Delta', H\prime K')$。

• 对于集合 $K\prime$，如果事件 ev.server 是拜占庭服务器，则 $K\prime = K \cup$ 有效_元素(ev)，否则 $K\prime = K$。

其中

$$\text{有效_元素(ev)} = \begin{cases} \{e\}, \text{ 对于 ev } = \text{添加}(e) \wedge valid(e) \{e\}; \text{ 对于 ev } = \text{BRB.Deliver(添加}(e)) \wedge valid(e)\ S; \text{ 对于 ev } = \text{SBC}[h]\text{.通知} \\ \text{(prop)} \wedge S = \{e \in prop : valid(e)\}\ S; \text{ 对于 ev } = \text{SBC}[h]\text{.SetDeliver(propset)} \wedge S = \{e \in propset : valid(e)\}\ \emptyset; \text{ 否则} \end{cases}$$

• 对于网络 Δ'，– 如果事件 ev 是添加(e)，且 $s =$ ev.server 是正确服务器，则 $\Delta' =$ 发送$(\Delta,$ 添加(e) $s)$；– 如果事件 ev 是 BRB.Deliver(添加(e))，则 $\Delta' =$ 接收$(\Delta,$ 添加$(e),$ ev.server)；– 如果事件 ev 是 EpochInc(h)，且 $s =$ ev.server 是正确服务器，则$\Delta' =$ 发送$(\Delta,$ epinc(h) $s)$；– 如果事件 ev 是 BRB.Deliver(epinc(h))，且 $s =$ ev.server 是拜占庭服务器，或 $h <\Sigma(s)$.epoch $+1$，则 $\Delta' =$ 接收$(\Delta,$ epinc(h) $s)$；– 如果事件 ev 是 BRB.Deliver(epinc(h))，且 $s =$ ev.server 是正确服务器，并且 $h = \Sigma(s)$.epoch $+1$，则令 ps 为 s 中不带纪元的元素集合，$ps = \Sigma(s).S \setminus \bigcup_k^{h-1} \Sigma(s).H(k)$，在新网络中 $\Delta' =$ 发送(接收$(\Delta,$ epinc(h) $s),$ (h, ps) $s)$；– 如果事件 ev 是 BRB.Broadcast(m)，则 $\Delta' =$ 发送 $(\Delta, m,$ ev.server)；– 如果事件 ev 是 SBC[h].Propose$(prop)$，则 $\Delta' =$ 发送$(\Delta,$ $(h, prop),$ ev.server)；– 如果事件 ev 是 SBC[h].Inform$(prop)$，则 $\Delta' =$ 接收$(\Delta,$ $(h, prop),$ ev.server)；– 否则 $\Delta' = \Delta$。

• 对于状态映射 $\Sigma\prime$：– 如果事件ev.服务器是拜占庭的,
则 $\Sigma\prime = \Sigma$.– 如果事件 $s = ev$.服务器是正确服务器，则

∗ $ev =$ BRB.Deliver(添加(e))，则 $\Sigma\prime = \Sigma \oplus \{s \mapsto (\Sigma(s).S \cup \{x : x = e \wedge$ 有效$(x)\}$ $\Sigma(s).H,\Sigma(s).$纪元$)\}$.

∗ $ev =$ SBC[h].SetDeliver(属性集)，则 $\Sigma\prime = \Sigma \oplus \{s \mapsto (\Sigma(s).S \cup E, \Sigma(s).H \cup \{h, E\}\ h)\}$，其中 $E = \{e :$ $e \in$ 属性集, 有效$(e) \wedge e \notin \Sigma(s).H\}$

∗ 否则 $\Sigma\prime = \Sigma$.

• 对于纪元映射 $H\prime$：– 如果事件为 SBC[h].共识(属性集)，则 $H\prime(h) =$ 属性集，并且 $H\prime(x) = H(x)$，其中$x \neq h$。– 否则 $H\prime = H$。

如果事件 ev在配置 (Σ, Δ, H, K) 中启用，并且 $(\Sigma', \Delta', H', K')$ 是对 (Σ, Δ, H, K) 应用事件 ev的效果后得到的配置，则记作 $(\Sigma, \Delta, H, K) \xrightarrow{ev} (\Sigma', \Delta', H', K')$。

定义 1（Γ的有效轨迹）。 Γ 的有效轨迹是一个无限序列 $(\Sigma_0, \Delta_0, H_0, K_0) \xrightarrow{ev^0} (\Sigma_1, \Delta_1, H_1, K_1) \xrightarrow{ev^1} \ldots$，满足 $(\Sigma_0, \Delta_0, H_0, K_0)$ 是初始配置。

定义 2（Γ中的可达配置）。配置 Φ 在 Γ 中是可达的，当且仅当它是初始配置 $\Phi = \Phi_0$，或者它是在可达配置中应用一个已启用事件后的结果：$\exists \Psi$，ev：Ψ 可达 $\wedge\, \Psi \xrightarrow{ev} \Phi$。

我们用 Γ′ 表示模型，该模型描述由以下服务器维护的 Setchain 的执行：$n - f$个实现算法的正确服务器 Fast 以及一个 b 个实现算法8的服务器。模型 Γ′ 中的配置是一个元组 $(\Sigma, \Delta, H, \mathcal{T})$ 其中 $\mathcal{T}$ 是服务器 b的本地状态，而 Σ, Δ 和 H与模型 Γ中相同。服务器 b 的本地状态由存储模型 Γ中所有拜占庭进程获取的知识构成。

初始配置 $\Phi'_0 = (\Sigma_0, \Delta_0, H_0, \mathrm{T}_0)$ 满足：$\Sigma_0(s)$ 是每个正确进程的初始状态，T_0 是空集，Δ_0 是空网络，且 H_0 是空映射。

我们按照模型 Γ 中相同的思路，在 Γ′ 中定义有效轨迹。对于正确进程，我们采用与前一个模型相同的规则。进程 b 的规则与前一个模型中拜占庭进程的规则类似。区别在于，此处进程 b 消耗一个事件时，会将该事件中包含的所有有效元素存储在其本地状态 T 中，以便之后能够非确定性地使用有效元素来生成事件。

在 Γ′ 中，事件 ev的作用现定义如下。给定一个配置 $(\Sigma, \Delta, H, \mathrm{T})$，其中 ev在 $(\Sigma, \Delta, H, \mathrm{T})$ 中已启用，则 ev的作用是得到一个配置 $(\Sigma', \Delta', H', \mathrm{T}')$，满足：

- 对于 Δ'、Σ' 和 H'，其效果如 Γ所示。
- 对于 T'：– 若事件 ev.server $= b$，则 $\mathrm{T}' = \mathrm{T} \cup$ 有效_元素(ev)。– 否则 $\mathrm{T} = \mathrm{T}'$

请注意，现在唯一的“拜占庭”进程只会标注它发现的所有事件。有效轨迹和可达配置的定义与 Γ中的定义类似。

定义3（Γ′的有效轨迹）。 Γ′ 的有效轨迹是一个无限序列 $(\Sigma_0, \Delta_0, H_0, \mathcal{T}_0) \xrightarrow{ev^0} (\Sigma_1, \Delta_1, H_1, \mathcal{T}_1) \xrightarrow{ev^1} \ldots$，满足 $(\Sigma_0, \Delta_0, H_0, \mathcal{T}_0)$ 是初始配置。

与轨迹 Γ有效的主要区别在于，现在我们定义了“拜占庭”服务器可以执行什么操作。8，并相应地更新其状态。

定义4（Γ′中的可达配置）。配置 Φ' 在 Γ′ 中可达，当且仅当它是初始配置 $\Phi' = \Phi'_0$，或者是在某个可达配置中应用一个已启用事件后得到的结果：$\exists \Psi'$，ev：Ψ' 可达 $\wedge\, \Psi' \xrightarrow{ev} \Phi'$。

鉴于服务器 b 接收到消息时，其有效元素会被添加到本地状态 T 中，根据归纳法可知，在可达配置中，服务器 b 接收多重集里出现的所有有效元素也都在 T 中：

引理 6。 设 $(\Sigma, \Delta, H, \mathcal{T})$ 是 Γ′ 中的一个可达配置，则：

$$\bigcup_{ev \in \Delta(b).\mathtt{received}} valid_elements(ev) \subseteq \mathcal{T}$$

我们证明模型 Γ 和 Γ' 观察等价，换言之，在假设拜占庭进程可能共享信息的情况下，外部用户无法区分这两个模型。为证明这一点，我们证明：对于一个模型中的每条有效轨迹，另一个模型中都存在一条有效轨迹，使得对应的配置无法区分。直观地说，两个配置 $\Phi = (\Sigma,\ \Delta,\ H,\ K)$ 和 $\Phi' = (\Sigma',\ \Delta',\ H',\ \mathrm{T}')$ 观察等价，$\Phi \sim \Phi'$，当且仅当：(1) 每个正确进程具有相同的本地状态；(2) 网络观察等价；(3) T 和 K 包含相同的元素；(4) 达成共识的历史相同。

核心思想是，既然拜占庭进程在网络之外共享了它们的知识，我们就可以用一个非确定性进程来替代它们，该进程能够执行拜占庭进程可能执行的所有操作。因此，我们对观察等价的定义基于以下映射：将模型 Γ 中的每个拜占庭操作映射为模型 Γ' 中进程 b 的一个可能操作，并将模型 Γ' 中的每个单独操作 b 映射为模型 Γ 中各进程的一系列操作（每个拜占庭进程对应一个操作）。

定义 5（网络观察等价）。设 Δ 是模型 Γ 的一个网络，Ω 是模型 Γ' 的一个网络。当且仅当以下条件成立时，我们称 Δ 与 Ω 观察等价，$\Delta \sim \Omega$：

（*1*）正确进程的状态完全相同：$\forall s \in \mathbb{C}:\ \Delta(s) = \Omega(s)$；（*2*）消息由进程 b 发送，当且仅当它由拜占庭进程发送：$\bigcup_{s\in\mathbb{B}} multiset(\Delta(s).sent) = multiset(\Omega(b).sent)$；（*3*）进程 b 的所有待处理消息都在所有拜占庭进程中处于待处理状态：$\forall s \in \mathbb{B}$: $\Omega(b).pending \subseteq \Delta(s).pending$；（*4*）消息由进程 b 接收，当且仅当它至少由一个拜占庭进程接收：$multiset(\Omega(b).received) \subseteq \bigcup_{s\in\mathbb{B}} multiset(\Delta(s).received)$；（*5*）发送给拜占庭进程的所有消息（无论处于待处理还是已接收状态）也发送给了 b：$\forall s \in \mathbb{B}:\ multiset(\Delta(s).received) \cup \Delta(s).pending = multiset(\Omega(b).received) \cup \Omega(b).pending$；（*6*）如果消息由 b 接收，那么拜占庭进程也接收了该消息：$\forall s \in \mathbb{B}:\ multiset(\Delta(s).received) \subseteq multiset(\Omega(b).received)$。

以下引理表明，函数 send 和 receive 保持网络之间的观察等价关系。

引理 7. 设 Δ 是模型为 Γ的网络，Δ' 是模型为 Γ' 的网络，且 s 是正确服务器。那么，

- 发送$(\Delta, m, s) \sim$(发送(Δ', m, s),
- 接收$(\Delta, m, s) \sim$(接收(Δ', m, s).

证明。首先，我们将证明：如果某个正确服务器在两个观察等价的网络中发送相同的消息，那么这两个网络仍然观察等价。

设 s_0 是正确服务器，那么 Δ 与 Δ' 之间的观察等价意味着$\Delta(s_0) = \Delta'(s_0)$。由于函数 send 仅通过向所有服务器的待处理组件以及发送方的已发送组件添加消息 m 来修改网络（两种情况下都是同一服务器 s），因此可得 (1) $\forall s_0 \in \mathbb{C}$: send$(\Delta,\ m,\ s)(s_0)$ =send$(\Delta',\ m,\ s)(s_0)$。

现在，我们考虑发送函数在拜占庭服务器和服务器 b中的影响：
(2)$\bigcup_{s_b\in\mathbb{B}} multiset$（发送$(\Delta, m, s)(s_b)$.sent） $= \bigcup_{s_b\in\mathbb{B}} multiset(\Delta(s_b)$.sent) $=multiset(\Delta'(b)$.sent) =发送$(\Delta', m, s)(b)$.sent，因为 sent 字段仅对服务器 s进行修改。(3) 设 s_b 是拜占庭服务器。由于 m 被添加到所有服务器的待处理多重集中，因此可得：发送$(\Delta', m, s)(b)$.pending $= \Delta'(b)$.pending $\cup\{m\} \subseteq \Delta(s_b)$.pending $\cup\{m\}$ =发送$(\Delta, m, s)(s_b)$.pending。

(4) 发送(Δ′, *m*, *s*)(*b*).received = *multiset*(Δ′(*b*).received)⊆ $\bigcup_{s_b \in \mathbb{B}}$ *multiset*(Δ(s_b).received) = $\bigcup_{s_b \in \mathbb{B}}$ 发送(Δ, *m*, *s*)(s_b).received，因为发送函数不会为任何服务器修改 received 字段。(5) 设 s_b 是拜占庭服务器。由于 *m* 被添加到所有服务器的待处理多重集中：*multiset*(发送(Δ, *m*, *s*)(s_b).received) ∪发送(Δ, *m*, *s*)(s_b).pending = *multiset*(Δ(s_b).received) ∪ Δ(s_b).pending ∪ {*m*}= *multiset*(Δ′(*b*).received) ∪ Δ′(*b*).pending ∪ {*m*}= *multiset*(发送(Δ′, *m*, *s*)(*b*).received) ∪发送(Δ′, *m*, *s*)(*b*).pending。(6) 设 s_b 是拜占庭服务器。received 字段不会为任何服务器修改，因此：*multiset*(发送(Δ, *m*, *s*)(s_b).received) = *multiset*(Δ(s_b).received)⊆ *multiset*(Δ′(*b*).received) = *multiset*(发送(Δ′, *m*, *s*)(*b*).received)。

因此，发送(Δ， *m*， *s*) ∼发送(Δ′， *m*， *s*)。

接下来，我们将证明，如果正确服务器在两个观察等价的网络中接收到相同的消息，那么这两个网络仍然观察等价。注意，只需证明 (a) 接收(Δ， *m*， *s*)(*s*)。received =接收(Δ′， *m*， *s*)(*s*)。received，以及 (b) 接收(Δ， *m*， *s*)(*s*)。pending =接收(Δ′， *m*， *s*)(*s*)。pending，因为这些是接收函数会修改的唯一字段，因此其余条件可由关系Δ ∼ Δ′直接推出：

(a) 接收(Δ, *m*, *s*)(*s*).received = Δ(*s*).received ∪{*m*}= Δ′(*s*).received ∪{*m*} =接收(Δ′, *m*, *s*)(*s*).received (b) 接收(Δ, *m*, *s*)(*s*).pending = Δ(*s*).pending \ {*m*}= Δ′(*s*).pending \ {*m*} =接收(Δ′, *m*, *s*)(*s*).received

在这两种情况下，第一个和最后一个等式由接收函数的定义得出，而第二个等式则由观察等价 Δ ∼ Δ′得出。

因此，接收(Δ， *m*， *s*) ∼接收(Δ′， *m*， *s*)。

□

引理 8。 设 Δ 是模型 Γ的网络， Δ′ 是模型 Γ′ 的网络，且 *s* 是一个拜占庭服务器。那么，

- 发送(Δ， *m*， *s*) ∼(发送(Δ′, *m*， *b*)，
- 如果 *m* ∈ Δ(*s*)。待处理消息 且 *m* ∈ Δ′(*b*)。待处理消息 则 接收(Δ， *m*， *s*) ∼接收(Δ′*m*， *b*)。
- 如果 *m* ∈ Δ(*s*).待处理消息 且 Δ′(*b*).已接收.计数(*m*) > Δ(*s*).已接收.计数(*m*) 则接收(Δ， *m*， *s*) ∼ Δ′。

证明。 如果拜占庭服务器发送一条消息 *m*，且所在网络与一个服务器 *b* 发送相同消息 *m* 的网络观察等价，那么所得网络也观察等价。其证明与前一个引理7 的证明 *m* 类似（其中 *m* 由两个网络中的同一个正确服务器发送），但情况 (2) 除外：在这种情况下，将 *m* 添加到已发送字段中，同时保持 *b* 和 *s* 之间的关系：

$\bigcup_{s_b \in \mathbb{B}}$ *multiset*(send(Δ， *m*， *s*)(s_b)。sent) = $\bigcup_{s_b \in \mathbb{B}}$ *multiset*(Δ(s_b)。sent) ∪{*m*} =*multiset*(Δ′(*b*)。sent) ∪{*m*} =send(Δ′， *m*， *s*)(*b*)。sent

因此，send(Δ， *m*， *s*) ∼(send(Δ′， *m*， *b*)，若 *s*是拜占庭服务器。

接下来，我们将证明：如果拜占庭服务器 s_b 在一个网络中接收消息 *m*，而该网络观察等价于另一个网络，在后者中服务器 *b* 接收相同的消息 *m*，那么只要两台服务器的待处理多重集中都包含 *m*，这两个网络仍然观察等价：

(1) 令 s_0 为正确服务器：receive(Δ, m, s_0)(s) = $\Delta(s_0)$ = $\Delta'(s_0)$ =receive(Δ', m, b)(s_0)，由于接收消息 m 的服务器不是正确服务器，因此对于 s_0，网络不会发生变化。

(2) $\bigcup_{s_b \in \mathbb{B}}$ *multiset* 对于任意服务器，接收函数都不会修改 sent 字段，因此 (receive(Δ, m, s)(s_b)。sent) = $\bigcup_{s_b \in \mathbb{B}}$ *multiset*($\Delta(s_b)$。sent)= *multiset*($\Delta'(b)$。sent) = *multiset*(receive(Δ', m, b)(b)。sent)

(3) 设 s_b 为一台拜占庭服务器：

- 若 $s_b = s$: receive(Δ,m, b)(b)。pending = Δ,(b)。pending \ {m} $\subseteq$ $\Delta(s)$。pending \ {m} =接收函数 receive(Δ, m, s)(s)。pending，因为这两种情况下，m 都从包含它的多重集 pending 中移除，因此多重集之间的包含关系得以保持。
- 若 $s_b \neq s$: receive(Δ,m, b)(b)。pending = Δ,(b)。pending \ {m} $\subset$ Δ,(b)。pending $\subseteq\Delta(s_b)$。pending =receive(Δ, m, s)(s_b)。pending。

(4) 接收(Δ', m, b)(b)。已接收 = *multiset*($\Delta'(b)$。已接收) $\cup${m}$\subseteq$ $\bigcup_{s_b \in \mathbb{B}}$ *multiset*($\Delta(s_b)$。已接收) $\cup${m} = $\bigcup_{s_b \in \mathbb{B}}$ 接收(Δ, m, s)(s_b)。已接收，因为消息 m 被添加到 b 和 s双方的已接收多重集中。

(5) 设 s_b 是一个拜占庭服务器。对于 $s_b \neq s$ 接收函数 receive(Δ, m, s)，不会修改字段 receive 或 pending；对于 $s_b = s$，它会将 m 从其待处理多重集移至其已接收序列（根据假设 $m \in \Delta(s)$.pending）。因此，*multiset*(receive(Δ, m, s)(s_b).received) $\cup$receive(Δ, m, s)(s_b).pending= *multiset*($\Delta(s_b)$.received) $\cup$ $\Delta(s_b)$.pending。类似地，对于 b，接收函数 receive(Δ', m, b) 会将 m 从其待处理多重集移至其已接收序列（根据假设$m \in \Delta'(b)$.pending）。因此，*multiset*($\Delta'(b)$.received) $\cup$ $\Delta'(b)$.pending= *multiset*(receive(Δ', m, s)(b).received) $\cup$ receive(Δ', m, s)(b).pending。由此可得，*multiset*(receive(Δ, m, s)(s_b).received) $\cup$r eceive(Δ, m, s)(s_b).pending= *multiset*(receive(Δ', m, s)(b).received) $\cup$receive(Δ', m, s)(b).pending。

(6) 设 s_b 是一个拜占庭服务器。消息 m 被添加到执行接收函数的服务器的已接收序列中，而其余服务器的已接收序列保持不变：*multiset*(receive(Δ, m, s)(s_b)。received) $\subseteq$ *multiset*($\Delta(s_b)$。received) $\cup$ {m}$\subseteq$ *multiset*($\Delta'(b)$。received) $\cup${m} = *multiset*(receive(Δ', m, s)(b)。received)。

接下来，我们将证明：如果拜占庭服务器 s_b 在一个网络中接收消息 m，而该网络与另一个服务器 b 已经接收相同消息 m 的网络观察等价，那么这两个网络仍然观察等价：

(1) 设 s_0 是一个正确服务器：receive(Δ, m, s_0)(s) = $\Delta(s_0)$ = $\Delta'(s_0)$，因为接收消息 m 的服务器不是正确服务器，因此对于 s_0，网络不会被修改。

(2) 接收函数不会修改任何服务器的 sent 字段，因此

$$\bigcup_{s_b \in \mathbb{B}} multiset(\text{receive}(\Delta, m, s)(s_b).\texttt{sent}) = \bigcup_{s_b \in \mathbb{B}} multiset(\Delta(s_b).\texttt{sent}) = multiset(\Delta'(b).\texttt{sent})$$

.

(3) 设 s_b 是一个拜占庭服务器：

- 如果 $s_b = s$，那么结合以下事实：b 和 s 发送了相同的消息集合（*multiset*($\Delta(s_b)$.received)$\cup\Delta(s_b)$.pending = *multiset*($\Delta'(b)$.received)$\cup\Delta'(b)$.pending），以及假设 $\Delta'(b)$.received.count(m) > $\Delta(s)$.received.count(m)，可得出$\Delta'(b)$.pending.count(m) < $\Delta(s)$.pending.count(m)。因此，从 $\Delta(s)$.pending 中移除 m 不会破坏 b 与 s 的待处理集合之间的关系：$\Delta'(b)$.pending $\subseteq$ $\Delta(s)$.pending\ {m} =receive(Δ, m, s)(s).pending。
- 若 $s_b \neq s$则 Δ,(b)。待处理消息 $\subseteq$ $\Delta(s_b)$。待处理消息 =接收(Δ, m, s)(s_b)。待处理消息

(4) 消息 m 被添加到 s 的接收多重集中，则： $multiset(\Delta'(b)$。接收) $\subseteq \bigcup_{s_b \in \mathbb{B}} multiset(\Delta(s_b)$。接收) $\subset \bigcup_{s_b \in \mathbb{B}} multiset(\Delta(s_b)$。接收) $\cup \{m\} = \bigcup_{s_b \in \mathbb{B}}$ 接收$(\Delta, m, s)(s_b)$。接收。(5) 设 s_b 为一台拜占庭服务器。采用与前一种情形中证明(5)类似的推理，可以证明：$multiset$(接收$(\Delta, m, s)(s_b)$。接收) $\cup$接收$(\Delta, m, s)(s_b)$。待处理消息$= multiset(\Delta(s_b)$。接收$) \cup \Delta(s_b)$。待处理消息 $= multiset(\Delta'(b)$。接收$) \cup \Delta'(b)$。待处理消息。(6) 设 s_b 为一台拜占庭服务器。

- 如果 $s_b = s$：根据假设 $\Delta(s)$.received.count$(m) < \Delta_r(b)$.received.count(m)，因此将消息 m 添加到 s 的已接收序列中，可保持 s与 b已接收序列之间的包含关系： $multiset$(receive$(\Delta, m, s)(s_b)$.received) $\subseteq multiset(\Delta(s_b)$.received) $\cup \{m\} \subseteq multiset(\Delta_r(b)$.received)。
- 如果 $s_b \neq s$，则服务器 s_b的已接收序列不会改变，因此：$multiset$(receive$(\Delta, m, s)(s_b)$.received) $= multiset(\Delta(s_b)$.received$) \subseteq multiset(\Delta_r(b)$.received)。

□

如果一个事件 ev 发生在正确服务器中 s ，那么两个模型运行相同的算法。具体而言，事件 ev 对某一配置产生的影响，仅通过组合使用 s 发送 和 接收函数来修改本地状态和网络。因此，如果将事件 ev 的影响应用于两个观察等价的配置，那么结果配置中的正确状态的本地状态仍保持相等；引理 7 表明，结果网络观察等价，而且已达成的共识历史和拜占庭知识均未改变，因此它们在结果配置中仍保持相等。因此，正确服务器中发生的事件会保持配置之间的观察等价性，如下一个引理所述：

引理 9. 设 $\Phi = (\Sigma, \Delta, H, K)$ 和 Ψ 是模型 $\Gamma, \Phi' = (\Sigma', \Delta', H', \mathcal{T}_r)$ 中的配置， Ψ'是模型 Γ' 中的配置，并且 ev 是满足 $s = ev$.服务器为正确服务器的一个事件。若 $\Phi \sim \Phi', \Phi \xrightarrow{ev} \Psi$且 $\Phi' \xrightarrow{ev} \Psi'$ ，则 $\Psi \sim \Psi'$。

证明。由于 ev 发生在正确服务器上，其作用只会修改正确服务器的本地状态和配置中的网络组件。因此， $\Psi = (\Sigma_{ev}, \Delta_{ev}, H, K)$，其中

$$\Sigma_{ev} = \begin{cases} \Sigma \oplus \{s \mapsto (\Sigma(s).S \cup \{x : x = e \wedge \text{valid}(x)\}, \Sigma(s).H, \Sigma(s).h)\} & \text{if } ev = \text{BRB.Deliver}(\text{add}(e)) \\ \Sigma \oplus \{s \mapsto (\Sigma(s).S \cup E, \Sigma(s).H \cup \{\langle h, E\rangle\}, h)\} & \text{if } ev = \\ \text{with } E = \{e : e \in propset, \text{valid}(e) \wedge e \notin \Sigma(s).H\} & \text{SBC}[h].\text{SetDeliver}(propset) \\ \Sigma & \text{otherwise} \end{cases}$$

并且， Δ_{ev} 是通过应用仅取决于ev的发送函数和接收函数组合，由 Δ 生成的。

Similarly, $\Psi' = (\Sigma'_{ev}, \Delta'_{ev}, H', \mathcal{T}')$ with

$$\Sigma'_{ev} = \begin{cases} \Sigma' \oplus \{s \mapsto (\Sigma'(s).S \cup \{x : x = e \wedge \text{valid}(x)\}, & \text{if } ev = \text{BRB.Deliver}(\text{add}(e)) \\ \Sigma'(s).H, \Sigma'(s).h)\} & \\ \Sigma' \oplus \{s \mapsto (\Sigma'(s).S \cup E, \Sigma'(s).H \cup \{\langle h, E\rangle\}, h)\} & \text{if } ev = \text{SBC}[h].\text{SetDeliver}(propset) \\ \text{with } E = \{e : e \in propset, \text{valid}(e) \wedge e \notin \Sigma'(s).H\} & \\ \Sigma' & \text{otherwise} \end{cases}$$

并且， Δ'_{ev} 是通过应用仅取决于ev的发送函数和接收函数组合，由 Δ' 生成的。

为了证明 $\Psi \sim \Psi'$，我们需要证明：(1) $\Sigma_{ev} = \Sigma'_{ev}$、(2) $\Delta_{ev} \sim \Delta'_{ev}$、(3) $H = H'$ 以及(4) $K = \mathrm{T}'$：第一个等式 (1) 可由 Σ_{ev} 和 Σ'_{ev} 的定义，以及 Φ 与 Φ' 之间的观察等价推出，这意味着 $\Sigma = \Sigma'$。对于网络之间的观察等式 (2)，注意到 Δ_{ev} 是通过应用与从 Δ' 生成 Δ'_{ev} 时相同的发送和接收函数组合生成的。然后，由于 $\Delta \sim \Delta'$，可以应用引理 7，因此得到 $\Delta_{ev} \sim \Delta'_{ev}$。最后， Φ 与 Φ' 之间的观察等价直接推出等价关系 (3) 和 (4)。□

类似地，考虑一个事件 ev，它发生在拜占庭服务器中 s_b，在模型Γ 的一个配置上，并考虑同一事件 ev'，只不过它发生在服务器 b 中，且处于模型 Γ'的一个配置上。这两个事件对相应配置的影响仅仅是：将事件中发现的有效元素添加到拜占庭知识中，并对网络应用发送和接收函数的组合发送 和接收。因此，如果两个配置观察等价，那么正确状态的本地状态和已达成共识的历史记录不会改变，因此在生成的配置中仍然相等。引理8 表明，生成的网络观察等价；同时，向拜占庭知识集合添加了相同的有效元素集合，从而保持它们之间的相等。因此，事件ev 保持配置之间的观察等价，如下一个引理所述：

引理 10. 令 Φ 和 Ψ 为模型 Γ中的配置， Φ' 和 Ψ' 为模型 Γ'中的配置，ev 为一个事件，且 $ev.\texttt{server}$ 是拜占庭服务器；ev' 是与 ev 相同的事件，只是 $ev.\texttt{server} = b$。如果 $\Phi \sim \Phi'$，$\Phi \xrightarrow{ev} \Psi$ 且 $\Phi' \xrightarrow{ev'} \Psi'$，那么 $\Psi \sim \Psi'$。

证明。 证明与引理 9的证明类似。唯一的区别在于，ev 和 ev' 不会修改正确服务器的本地状态，而是通过添加有效$K = \mathcal{T}$（分别）有效_元素（ev）来修改它们的拜占庭知识，因此保持两者相等。□

在下面的引理中，我们将证明：如果存在两个可达配置 Φ 和 Φ'，它们观察等价，并且有一个发生在拜占庭服务器 s 上的事件，表示接收一条消息，且服务器 b 已在 Φ' 中接收了该消息，但 s 未在 Φ中接收该消息，那么将ev 的效果应用于 Φ 后得到的配置与 Φ'观察等价。

引理 *11.* 设 $\Phi = (\Sigma, \Delta, H, K)$ 和 Ψ 是模型 Γ 中的配置，且 $\Phi' = (\Sigma', \Delta', H', T')$ 是模型 Γ' 中的一个可达配置，其中 $\Phi \sim \Phi'$。考虑一个事件 ev，其中 $s = ev.server$ 是一台拜占庭服务器，且 $\Phi \xrightarrow{ev} \Psi$。如果 $ev = BRB.Deliver(m)$ 或 $ev = SBC[h].$通知$(prop)$，但 $\Delta'(b).received.count(m) > \Delta(s).received.count(m)$，那么 $\Psi \sim \Phi'$。

证明。由于 ev 发生在拜占庭服务器中，其效果只会修改配置中的网络和拜占庭知识组件。具体而言，事件 ev 表示接收消息 m，因此 $\Psi = (\Sigma, \text{receive}(\Delta, m, s)\ H, K \cup \text{有效_元素(ev)})$。所以，为了证明 $\Psi \sim \Phi'$，我们需要证明：(1) $\Sigma = \Sigma'$，(2) $\text{receive}(\Delta, m, s) \sim \Delta'$，(3) $H = H'$，以及 (4) $K \cup$有效 _ 元素 (ev) =T。 Φ 与 Φ' 之间的观察等价直接蕴含等价关系 (1) 和 (3)。由引理 8 可知，$\text{receive}(\Delta, m, s) \sim \Delta'$，因为根据假设 $\Delta'(b).\text{received.count}(m) > \Delta(s).\text{received.count}(m)$，观察等价关系 $\Phi \sim \Phi'$ 可推出这一点。最后，为了证明 (4)，注意到 $\Delta'(b).\text{received.count}(m) > \Delta(s).\text{received.count}(m) \geq 0$ 蕴含 $m \in \Delta'(b).\text{received}$。于是，根据引理 6，valid_元素(ev) $\subseteq$ T。因此， $K \cup$ 有效_元素(ev) =T $\cup$ 有效_元素(ev) = T。

□

由前述三个引理可知，对于一个模型中的每条有效轨迹，另一个模型中都存在一条有效轨迹，使得对应的配置无法区分，因为一条轨迹中发生的任何事件都可以映射到另一条轨迹中的一个（或 f）事件。

定理 8.1。对于模型 Γ 中的每条有效轨迹，模型 Γ′ 中都存在一条有效轨迹，使得相应的状态不可区分。

证明。设 σ 是模型 Γ 中的一条有效轨迹。我们需要构造模型 Γ′ 中的一条有效轨迹 σ'，使得对于所有 $i \geq 0$ 都有 $\sigma_i \sim \sigma'_i$。

我们将 $\sigma'_0 = \Phi'_0$ 定义为 Γ′ 中的初始配置，该配置的所有组件均为空，因此与 Γ、$\Phi_0 = \sigma_0$ 中的初始配置观察等价。

接下来，对于 $n \geq 0$，假设 σ' 已定义至 n，并且 $\sigma_i \sim \sigma'_i$（如果 $n > i \geq 0$）。考虑事件 ev_n，使得 $\sigma_n \xrightarrow{ev_n}{}^{n} \sigma_{n+1}$，其中 $\sigma_n = (\Sigma,\ \Delta,\ H,\ K)$。我们将证明存在一个事件 ev'_n 和一个配置 σ'_{n+1}，使得 $\sigma'_n \xrightarrow{ev'_n}{}^{n} \sigma'_{n+1}$ 且 $\sigma'_{n+1} \sim \sigma_{n+1}$，其中 $\sigma'_n = (\Sigma',\ \Delta',\ H',\ \mathrm{T}')$。

对 ev_n 进行分类讨论：

- 如果 $s = \mathrm{ev}_n.$服务器是正确服务器，则取 $\mathrm{ev}'_n = \mathrm{ev}_n$。因此，$\mathrm{ev}'_n$ 在 σ'_n 中是启用的，因为在 σ'_n 中，正确服务器上发生的事件是否启用仅取决于 s 本地状态、s 网络和已达成共识的历史记录，而由于 $\sigma_n \sim \sigma'_n$，配置中的所有这些组成部分在两种配置中都相同。因此，存在 σ'_{n+1} 使得 $\sigma'_n \xrightarrow{\mathrm{ev}'_n} \sigma'_{n+1}$，由引理 9 可得 $\sigma_n \sim \sigma'_{n+1}$。

- 如果 $s = \mathrm{ev}_n.$服务器是拜占庭服务器，则将 ev'_n 视为与 ev_n 相同的事件，但其 $\mathrm{ev}_n.$服务器为 $= b$。可能发生两种情况：ev'_n 在 σ'_n 中启用，或者未启用。在前一种情况下，存在 σ'_{n+1} 使得 $\sigma'_n \xrightarrow{\mathrm{ev}'_n}{}^{n} \sigma'_{n+1}$，由引理 10 可得 $\sigma_{n+1} \sim \sigma'_{n+1}$，如需证明。在后一种情况下，ev 必须表示接收一条在 s 中待处理但不在 b 中的消息。用符号表示，即 $\mathrm{ev} \in \Delta(s).\mathrm{pending}$，而 $\mathrm{ev}' \notin \Delta'(b).\mathrm{pending}$。由于 $\Delta \sim \Delta'$，可得 $multiset(\Delta(s).\mathrm{received}).\mathrm{count}(\mathrm{ev}) < (multiset(\Delta(s).\mathrm{received}) \uplus \Delta(s).\mathrm{pending}).\mathrm{count}(\mathrm{ev}) = (multiset(\Delta'(b).\mathrm{received}) \cup \Delta'(b).\mathrm{pending}).\mathrm{count}(\mathrm{ev}') = multiset(\Delta(s).\mathrm{received}).\mathrm{count}(\mathrm{ev})$。因此，引理 11 意味着 $\sigma_{n+1} \sim \sigma'_n$。然后，将 ev'_n 取为空操作，它在 σ'_n 中是启用的，并且 $\sigma'_{n+1} = \sigma'_n$。

- 若 $\mathrm{ev}_n = \mathrm{SBC}[h].$共识(propset)，则 $\sigma_{n+1} = (\Sigma, \Delta, H \oplus \{h \mapsto \text{属性集 } K\})$。注意，$\mathrm{ev}'_n = \mathrm{ev}_n$ 在 σ'_n 中已启用。事实上，由于 $\sigma_n \sim \sigma'_n, H = H'$ 和 $\Delta \sim \Delta'$，这意味着 $(h, p) \in \Delta(s)$。sent 当且仅当存在某个 s，使得 $(h, p) \in \Delta'(s').\mathrm{sent}$。因此，$\sigma'_n \xrightarrow{\mathrm{ev}'_n}{}^{n} (\Sigma' \Delta' H' \oplus \{h \mapsto propset\}, \mathrm{T}')$，它与 σ_{n+1} 观察等价。

- 若 ev = 空操作，则 $\sigma_{n+1} = \sigma_n$。因此，将 ev_m 视为空操作（它在 σ_m 中已启用），并且 $\sigma_{m+1} = \sigma_m$ 可推出 $\sigma_m \xrightarrow{ev_m} \sigma_{m+1}$ 以及 $\sigma_{n+1} \sim \sigma_{m+1}$。

因此，通过归纳法即可得出该定理。 □

现在我们证明另一个方向。为此，我们首先将轨迹 σ 的停顿扩展定义为轨迹 σ^{st}：它在 σ 中的每个事件之后添加 $f - 1$ 个空操作事件。

定理 8.2. 对于模型 Γ′ 中的每条有效轨迹 σ'，模型 Γ 中都存在一条有效轨迹 σ，使得 σ 中的每个状态都与 σ' 的停顿扩展中对应的状态不可区分。

证明。设 σ' 是模型 Γ′ 中的一条有效轨迹。我们需要在模型 Γ 中构造一条有效轨迹 σ，使得 $\sigma_i \sim {\sigma'_i}^{st}$ 对 $i \geq 0$ 成立，或者等价地，使得 $\sigma_{i*f+j} \sim \sigma'_i$ 对 $0 \leq j < f$ 成立。

我们首先将 $\sigma_j = \Phi_0$ 定义为 Γ 中的初始配置（对应 $0 \le j < f$），其所有组成部分均为空，因此与 Γ' 中的初始配置 $\Phi'_0 = \sigma'_0$ 观察等价。

接下来，对于 $n \ge 0$，假设 σ 在 $k = n * f + f - 1$ 之前均有定义，并且对于 $0 \le i \le n$ 和 $0 \le j < f$，$\sigma_{i*f+j} \sim \sigma'_i$。

我们将证明存在 f 个事件 ev_k、……、ev_{k+f-1} 以及 f 个配置 σ_{k+1}、……、σ_{k+f}，使得对于任意 $0 \le j < f$，都有$\sigma_{k+j} \xrightarrow{ev_{k+j}} \sigma_{k+j+1}$。

对事件 ev'_n 进行分类讨论，其中 $\sigma'_n \xrightarrow{ev'_n} \sigma'_{n+1}$：

- 如果 ev'_n 是服务器 $= b$ 的事件，且 ev'_n = 为 BRB.Deliver(m) 或 ev'_n = 为 SBC[h].Inform($prop$)，那么令 ev_{k+j} = 为 ev'_n，但将 ev_{k+j} 设为服务器 $= b_j$，对应于 $0 \le j < f$。因此，ev_k 在 σ_k 中可启用，因为$m \in \sigma'_n(b)$ 的待处理消息 $\subseteq \sigma_k(b_0)$。pending。由此可得，存在 σ_{k+1} 使得 $\sigma'_k \xrightarrow{ev} \sigma'_{k+1}$，且根据引理 10 可知 $\sigma_{k+1} \sim \sigma'_{n+1}$。由于 ev'_n 通过将 m 从 b 的待处理多重集中移至其已接收序列来修改网络，因此可得 $\sigma'_{n+1}(b)$。received.count(m) $= \sigma'_n(b)$。received.count(m) $+ 1 > \sigma'_n(b)$。received.count(m) $\ge \sigma_k(b_j)$。received.count(m) $= \sigma_{k+1}(b_j)$。received.count(m)，对于 $0 < j < f$ 均成立。最后一个等式源于以下事实：ev_k 仅修改 b_0 的待处理多重集，以及 σ_k 与 σ'_n 之间观察等价。将此结果与以下事实结合：$multiset(\sigma_{k+1}(b_1)$。received) $\cup \sigma_{k+1}(b_1)$。pending $= multiset(\sigma'_n(b)$。received) $\cup \sigma'_n(b)$。pending，可得 $m \in \sigma_{k+1}(b_1)$。pending，因此 ev_{k+1} 在 σ_{k+1} 中可启用。由此可得，存在 σ_{k+2} 使得 $\sigma_{k+1} \xrightarrow{ev} \sigma_{k+2}$，且根据引理 11 可知 $\sigma_{k+2} \sim \sigma'_{n+1}$。对 $1 < j < f$ 重复此推理，可证明存在 σ_{k+j+1} 使得 $\sigma_{k+j} \xrightarrow{ev} \sigma_{k+j+1}$ 且 $\sigma_{k+j+1} \sim \sigma'_{n+1}$。

- 否则，取 $ev_k = ev'_n$。然后，通过类似于前一定理中的分析，可以证明 ev_k 在 σ_k 中是可启用的，并且，所得配置$\sigma_k \xrightarrow{ev} \sigma_{k+1}$ 满足 $\sigma_{k+1} \sim \sigma'_{n+1}$。最后，对于 $1 < j < f$，定义 ev_{k+j} = 为空操作，它在所有配置中都是可启用的，尤其是在 σ_{k+j} 中。因此，$\sigma_{k+j} \xrightarrow{ev} \sigma_{k+j+1}$ 和 $\sigma_{k+j+1} \sim \sigma'_{n+1}$，如所愿。

因此，该定理可由归纳法得证。 □

因此，本节的主要结果是：模型 Γ 中的性质成立，当且仅当它们在模型 Γ' 中也成立。也就是说，可以证明模型 Γ' 中轨迹的性质，其中所有组件均已良好定义，而结果将直接转化到模型 Γ' 中的轨迹。

9 结论与未来工作

本文介绍了一种名为 Setchain 的新型分布式数据类型，它实现了一个带纪元的只增集合，并能够容忍拜占庭服务器节点。我们为 Setchain 提供了底层规范，定义了其理想性质，并提出了三种分布式实现，其中性能最高的一种使用拜占庭可靠广播和 RedBelly Set 拜占庭共识。实证评估表明，在 Setchain 中插入元素的速度比共识快三个数量级。此外，我们证明，拜占庭服务器节点的行为可以建模为与 BRB 和 SBC 进行的一系列简单交互，并引入了一个涵盖这些交互的非确定性进程。这种建模为使用不具备拜占庭推理能力的形式化推理来分析 Setchain 铺平了道路。

未来的工作包括开发引言中列出的应用，例如使用 Setchain 的内存池日志，以及更快速的二层乐观汇总。Setchain 还可用于缓解抢先交易攻击。内存池存储用户请求的交易，因此观察内存池便能预测未来的操作。抢先交易是指观察交易请求，并恶意注入交易，使其在被观察到的交易之前执行的行为 [9, 30]（通过向矿工支付更高手续费）。Setchain 可用于检测抢先交易，因为它可以作为一种基础机制，用于构建高效且可充当请求日志的内存池。此外，Setchain 还可作为构件来解决抢先交易问题：用户使用多重签名解密方案加密请求，参与方解密服务器会在矿工选定请求执行后、且顺序已确定时对请求进行解密。

我们还将研究如何为区块链配备 Setchain（同步区块和纪元），使智能合约能够将 Setchain 作为其存储的一部分进行访问。

一个尚待解决的重要问题是，如何设计一种支付系统，让客户端为 Setchain 的使用付费（即使费用远低于区块链本身的费用）。我们的 Setchain 利用了特定的偏序，从而放宽了区块链所强加的全序。作为未来的工作，我们将探索其他偏序及其用途，例如 Setchain 联盟、每个智能合约对应一个 Setchain 等。

参考文献

[1] Eli Ben-Sasson、Alessandro Chiesa、Christina Garman、Matthew Green、Ian Miers、Eran Tromer 和 Madars Virza。2014年。Zerocash：基于比特币的去中心化匿名支付。载于《S&P'14 论文集》。459–474。https://doi.org/10. 1109/SP.2014.36[2] Eli Ben-Sasson、Alessandro Chiesa、Eran Tromer 和 Madars Virza。2014年。面向冯·诺依曼架构的简洁非交互式零知识。载于《USENIX Sec.'14 论文集》。USENIX，781–796。https://www.usenix.org/conference/ usenixsecurity14/technical-sessions/presentation/ben-sasson [3] Gabriel Bracha。1987年。异步拜占庭一致协议。《Inf. Comput.》75，2（1987年），130–143。https: // doi.org/10.1016/0890-5401(87)90054-X [4] Tushar Deepak Chandra 和 Sam Toueg。1996年。用于可靠分布式系统的不可靠故障检测器。《J. ACM》 43, 2 1996 225 267 10 1145 226643 226647 （mar），–。https://doi.org/ . / . Vicent Cholvi 、 Antonio [5] Fernández Anta 、 Chryssis Georgiou 、 Nicolas Nicolaou 、 Michel Raynal 和 Antonio Russo。2021年。拜占庭容错分布式只增集合：规范与应用。载于《FAB'21 论文集》。2:1–2:19。[6] Tyler Crain、Christopher Natoli 和 Vincent Gramoli。2021年。Red Belly：一种安全、公平且可扩展的开放区块链。载于《S&P' *21* 466 483 10 1109 40001 2021 00087 论文集》。–。https://doi.org/ . /SP . . F. Cristian、[7] H. Aghili、R. Strong 和 D. Volev。 1995 . 原子广播：从简单的消息扩散到拜占庭一致。载于第25届国际容错计算研讨会。431–。https://doi.org/10.1109/FTCSH.1995.532668[8] Kyle Croman、Christian Decker、Ittay Eyal、Adem Efe Gencer、Ari Juels、Ahmed Kosba、Andrew Miller、Prateek Saxena、Elaine Shi、Emin Gün Sirer、Dawn Song 和 Roger Wattenhofer。2016年。论去中心化区块链的扩展。载于《金融密码学与数据安全》。Springer，106–125。[9] Philip Daian、Steven Goldfeder、T. Kell、Yunqi Li、X. Zhao、Iddo Bentov、Lorenz Breidenbach 和 A. Juels。2020年。Flash Boys 2.0：去中心化交易所中的抢先交易、矿工可提取价值与共识不稳定性。《S&P'20 论文集》（2020年），910–927。[10] Hung Dang、Tien Tuan Anh Dinh、Dumitrel Loghin、Ee-Chien Chang、Qian Lin 和 Beng Chin Ooi。2019年。通过分片实现区块链系统扩展。载于《SIGMOD'19 论文集》。ACM，123—-140。https://doi.org/10.1145/3299869. 3319889[11] Xavier Défago、André Schiper 和 Péter Urbán。2004年。全序广播与组播算法：分类与综述。《ACM Comput. Surv.》 36, 4 2004 372 421 10 1145 1041680 1041682 （dec），–。[12] https://doi.org/ . / . Alan A.A. Donovan 和 Brian W. Kernighan。 2015 .《Go编程语言》。Adison-Wesley。[13] Antonio Fernández Anta、Chryssis Georgiou、Maurice Herlihy 和 Maria Potop-Butucaru。2021年。《区块链系统原理》。Morgan & Claypool出版社。[14] Antonio Fernández Anta、Kishori Konwar、Chryssis Georgiou 和 Nicolas Nicolaou。2018年。分布式账本对象的形式化与实现。《ACM Sigact News》49，2（2018年），58–76。[15] Michael J. Fischer、Nancy A. Lynch 和 Michael S. Paterson。1985年。存在一个故障进程时分布式共识的不可能性。《JACM》32，2（1985年），374–382。https://doi.org/10.1145/3149.214121

[16] Rachid Guerraoui、Petr Kuznetsov、Matteo Monti、Matej Pavlovic 和 Dragos-Adrian Seredinschi。2019年。加密货币的共识数。载于《PODC’ *19* 307 316 10 1145 3293611 3331589 论文集》。ACM，–。[17] 2019 https://doi.org/ . / . Maxim Jourenko、Kanta Kurazumi、Mario Larangeira 和 Keisuke Tanaka。 . SoK：加密货币第二层可扩展性相关协议的分类体系。《IACR 密码学电子预印本档案》2019（2019年），352。[18] Harry Kalodner、Steven Goldfeder、Xiaoqi Chen、S. Matthew Weinberg 和 Edward W. Felten。2018年。Arbitrum：可扩展的私密智能合约。载于第27届 USENIX 安全研讨会。USENIX 协会，1353–1370。https://www.usenix.org/conference/usenixsecurity18/presentation/kalodner [19] Jae Kwon 和 Ethan Buchman。2019年。Cosmos 白皮书。[20] Zamani Mahdi、Mahnush Movahedi 和 Mariana Raykova。2018年。RapidChain：通过全分片扩展区块链。载于《CSS’ *18* 931 948 10 1145 3243734 3243853 论文集》（加拿大多伦多）。ACM，—。[21] 2009 https://doi.org/ . / . 中本聪。 . 比特币：一种点对点电子现金系统。[22] Joseph Poon 和 Thaddeus Dryja。2016年。比特币闪电网络：可扩展的链下即时支付。https: //lightning.network/lightning-network-paper.pdf [23] Michel Raynal。2018年。《容错消息传递分布式系统：算法方法》。https://doi.org/10. 1007/978-3-319-94141-7[24] Robinson、Dan 和 Konstantopoulos、Georgios。2020年。以太坊是一座黑暗森林。https://medium.com/@danrobinson/ ethereum-is-a-dark-forest-ecc5f0505dff[25] Muhammad Saad、Laurent Njilla、Charles Kamhoua、Joongheon Kim、DaeHun Nyang 和 Aziz Mohaisen。2019年。基于工作量证明的区块链系统中用于防御分布式拒绝服务攻击的内存池优化。载于《ICBC’19 论文集》。 285 292 10 1109 2019 8751476 –。[26] https://doi.org/ . /BLOC. . Muhammad Saad、My T. Thai 和 Aziz Mohaisen。2018年。海报：通过内存池优化遏制针对基于区块链的加密货币的分布式拒绝服务攻击。载于《ASIACCS’18 论文集》。ACM，809—811。https://doi.org/10.1145/3196494.3201584[27] Marc Shapiro、Nuno Preguiça、Carlos Baquero 和 Marek Zawirski。2011年。收敛且可交换的复制数据类型。《Bulletin- European Association for Theoretical Computer Science》104（2011年6月），67–88。https://hal.inria.fr/hal- 00932833[28] Nick Szabo。1996年。智能合约：数字市场的构建模块。《Extropy》16（1996年）。[29] ZeroMQ 作者。2021年。ZeroMQ。https://zeromq.org https://zeromq.org。[30] Christof Ferreira Torres、Ramiro Camino 和 Radu State。2021年。《抢先交易者琼斯与黑暗森林掠夺者：以太坊区块链抢先交易的实证研究》。载于《USENIX Sec.’21 论文集》。1343–1359。https://www.usenix.org/conference/usenixsecurity21/presentation/torres [31] Shobha Tyagi 和 Madhumita Kathuria。2021年。区块链可扩展性解决方案研究。ACM，394–401。https: //doi.org/10 1145 3474124 3474184 . / . Ke Wang 和 Hyong S. Kim。 . FastChain：通过基于信息的邻居选择扩展区块链系统。载于《IEEE Blockchain’ *19* 376 383 10 1109 2019 00058 论文集》。–。[32] 2019 https://doi.org/ . /Blockchain. . Gavin Wood。[33] 2014 . 以太坊：一个安全的去中心化通用交易账本。《以太坊项目黄皮书》151（2014年），1–32。[34] Gavin Wood。2016年。Polkadot：异构多链框架愿景。《白皮书》21（2016年）。[35] Cheng Xu、Ce Zhang、Jianliang Xu 和 Jian Pei。2021年。SlimChain：通过链下存储和并行处理扩展区块链交易。《VLDB基金会会刊》14，11（2021年7月），2314–2326。https://doi.org/10.14778/3476249. 3476283