《网络存储技术》课程作业三

Consensus 技术概述

付容天 学号 2020211616

班级 2020211310

计算机科学与技术系

计算机学院（国家示范性软件学院）

2022 年 10 月 21 日

目录

1 引言

分布式共识（Consensus）是怎么达成分布式共识维护一致性帐本的一种技术。

分布式计算（Distributed Computing）和多代理系统中的一个基本问题是在存在许多故障进程的情况下实现整体系统可靠性。这通常需要协调进程以达成共识，或就计算期间所需的某些数据值达成一致。共识的示例应用包括同意以何种顺序提交到数据库的事务、状态机复制和原子广播。通常需要共识的实际应用包括云计算、时钟同步、PageRank、智能电网、无人机控制、负载平衡、区块链等。

共识问题需要多个进程（或代理）就单个数据值达成一致。某些进程（或代理）可能会失败或以其他方式变得不可靠，因此共识协议必须具有容错性（或弹性）。这些过程必须以某种方式提出它们的候选值，相互交流，并就一个单一的共识值达成一致。

共识问题是多智能体系统控制中的一个基本问题。产生共识的一种方法是让所有进程（代理）就多数值达成一致。在这种情况下，多数需要至少一半以上的可用票数（其中每个进程都获得一票）。然而，一个或多个错误的过程可能会扭曲结果结果，从而可能无法达成共识或达成不正确的共识。

解决共识问题的协议旨在处理数量有限的错误进程。这些协议必须满足一些有用的要求。例如，一个简单的协议可以让所有进程输出二进制值 1，但是这显然没有实际意义，因此修改了要求，使得输出必须以某种方式依赖于输入。也就是说，一个共识协议的输出值必须是某个进程的输入值。另一个要求是一个过程只能决定一个输出值一次，并且这个决定是不可撤销的。如果一个过程没有经历失败，那么它在执行中被称为正确的。一个容忍停止失败的共识协议必须满足以下属性：

- Termination：所有正确的进程最终都会认同某一个值
- Integrity：所有正确的进程认同的值都是同一个值
- Agreement：如果正确的进程都提议同一个值，那么所有处于认同状态的正确进程都选择该值

2 Consensus 概要

2.1 为什么需要 Consensus

我们首先需要明确分布式系统达成共识的动机。在应用场景下，分布式系统往往面对下面的困难：

- 网络问题：分布式系统中的多个节点以网络进行通信，但是网络并不保证什么时候到达以及是否一定到达，在以下情况中可能会出错：
  - 请求可能已经丢失（可能有人拔掉了网线）
  - 请求可能正在排队，稍后将交付（也许网络或收件人超载）
  - 远程节点可能已经失效（可能是崩溃或关机）
  - 远程节点可能暂时停止了响应（可能会遇到长时间的垃圾回收暂停），但稍后会再次响应
  - 远程节点可能已经处理了请求，但是网络上的响应已经丢失（可能是网络交换机配置错误）
  - 远程节点可能已经处理了请求，但是响应已经被延迟，并且稍后将被传递（可能是网络或者你自己的机器过载）
- 时钟问题：消息通过网络从一台机器传送到另一台机器需要时间，但由于网络中的可变延迟，我们不知道到底花了多少时间。这个事实有时很难确定在涉及多台机器时发生事情的顺序。由于网络上的每台机器都有自己的时钟，因此可能不同的机器之间有快慢的区别
- 部分失效问题：单机系统上的程序要么工作，要么出错。在分布式系统中，系统的某些部分可能会以某种不可预知的方式宕机。这被称为部分失效（partial failure）

Lamport 的论文《Time, Clocks and the Ordering of Events in a Distributed System》阐述了在分布式系统中，我们无法判断事件 A 是否发生在事件 B 之前，除非 A 和 B 存在某种依赖关系。由此还引出了分布式状态机的概念。在分布式系统中，共识就常常应用在这种多副本状态机（Replicated state machines）上，状态机在每台节点上都存有副本，这些状态机都有相同的初始状态，每次状态转变、下个状态是什么都由相关进程共同决定，每一台节点的日志的值和顺序都相同。每个状态机在“哪个状态是下一个需要处理的状态”这个问题上达成共识，这就是一个共识问题。

最终，这些节点看起来就像一个单独的、高可靠的状态机。Raft 在他著名的论文中提到，使用状态机我们就能克服上述的三个分布式系统往往要面对的问题：

- 网络问题：通过在所有非拜占庭条件下确保安全（不会返回错误结果），包括网络延迟、分区、丢包、重复和重排序，网络问题可以得到解决
- 时钟问题：我们可以实现不依赖于时序的效果
- 部分失效问题：通过高可用机制，只要集群中的大部分节点正常运行，并能够互相通信且可以同客户端通信，这个集群就完全可用。例如，拥有 5 个节点的集群可以容忍其中的 2 个节点失败。假使通过停掉某些节点使其失败，稍后它们会从持久化存储的状态进行恢复，并重新加入到集群中

不仅如此，达成共识还可以解决分布式系统中的以下经典问题：

- 互斥（Mutual exclusion）：哪个进程进入临界区访问资源？分布式锁？
- 选主（Leader election）：在单主复制的数据库，需要所有节点就哪个节点是领导者达成共识。如果一些由于网络故障而无法与其他节点通信，可能会产生两个领导者，它们都会接受写入，数据就可能会产生分歧，从而导致数据不一致或丢失
- 原子提交（Atomic commit）：跨多节点或跨多分区事务的数据库中，一个事务可能在某些节点上失败，但在其他节点上成功。如果我们想要维护这种事务的原子性，必须让所有节点对事务的结果达成共识：要么全部提交，要么全部中止/回滚

总而言之，在共识的帮助下，分布式系统就可以像单一节点一样工作，所以共识问题是分布式系统最基本的问题。

2.2 Consencsus 面临的问题

理论上，如果分布式系统中各节点都能以十分“理想”的性能（瞬间响应、超高吞吐）稳定运行，节点之间通信瞬时送达（如量子纠缠），则实现共识过程并不十分困难，简单地通过广播进行瞬时投票和应答即可。

可惜地是，现实中这样的“理想”系统并不存在。不同节点之间通信存在延迟（光速物理限制、通信处理延迟），并且任意环节都可能存在故障（系统规模越大，发生故障可能性越高）。如通信网络会发生中断、节点会发生故障、甚至存在被入侵的节点故意伪造消息，破坏正常的共识过程。

2.3 Consensus 系统模型

在考虑如何达成共识之前，我们需要考虑分布式系统中有哪些可供选择的计算模型。评价模型往往主要有以下几个方面：

- 网络模型：
  - 同步（Synchronous）：响应时间是在一个固定且已知的有限范围内
  - 异步（Asynchronous）：响应时间是无限的
- 故障类型：
  - Fail-stop failures：节点突然宕机并停止响应其它节点
  - Byzantine failures：源自“拜占庭将军问题”，是指节点响应的数据会产生无法预料的结果，可能会互相矛盾或完全没有意义，这个节点甚至是在“说谎”，例如一个被黑客入侵的节点

- 消息类型：
  - 口头消息（oral messages）：消息被转述的时候是可能被篡改的
  - 签名消息（signed messages）：消息被发出来之后是无法伪造的，只要被篡改就会被发现

我们接下来从常见的角度来展开讨论，即分别讨论在同步系统和异步系统中的共识。在同步通信系统中达成共识是可行并且较为容易的，但是，在实际的分布式系统中同步通信是不切实际的，我们不知道消息是故障了还是延迟了。异步与同步相比是一种更通用的情况。一个适用于异步系统的算法，也能被用于同步系统，但是反过来并不成立。

2.4 异步系统中的 Consensus 与 FLP 不可能性

FLP 不可能性（FLP Impossibility）是分布式领域中一个非常著名的结果，该定理的论文是由 Fischer、Lynch 和 Patterson 三位作者于 1985 年发表，他们凭借该定理获得了分布式计算中最具影响力的 Dijkstra 论文奖。

FLP 不可能性给出了一个令人吃惊的结论：在异步通信场景，即使只有一个进程失败，也没有任何算法能保证非失败进程达到一致性。因为同步通信中的一致性被证明是可以达到的，因此在之前一直有人尝试各种算法解决以异步环境的一致性问题，最终 FLP 不可能性的提出使这样的尝试终于有了答案。

在同步的系统中，达成共识是可以被解决的。因为在同步系统中，当有进程出现故障，或者响应超时，我们可以认定它已经崩溃。

在异步的系统中，当一个进程出现故障，或者响应丢失时，是无法检测到的。在这样的条件下，如果其中有任意一个进程出现问题，没有任何一个分布式算法，可以让所有的非故障进程，达成一致性共识。

因为有 FLP 不可能性的限制，大部分区块链项目的共识算法都把大部分节点认为是诚实的，并以一定的同步性作为前提。例如，POW 认为 51% 的节点是诚实的，并且有一定的同步性；POS 和 PBFT 也认为大部分节点（66%）是诚实的。

人们开始意识到一个分布式共识算法需要具有的两个属性：安全性（safety）和活性（liveness）。安全性意味着所有正确的进程都认同同一个值，活性意味着分布式系统最终会认同某一个值。每个共识算法要么牺牲掉一个属性，要么放宽对网络异步的假设。FLP 不可能性给后续的研究提供了新的思路：不再尝试寻找异步通信系统中共识问题完全正确的解法。FLP 不可能性是指无法确保达成共识，并不是说如果有一个进程出错，就永远无法达成共识。这种不可能的结果来自于算法流程中最坏的结果：

- 一个完全异步的系统
- 发生了故障

- 最后，不可能有一个确定的共识算法

针对这些最坏的情况，可以找到一些方法，尽可能去绕过 FLP 不可能性，能满足大部分情况下都能达成共识。《分布式系统：概念与设计》一书中提到，有三种办法可以解决这些情况：

- 故障屏蔽（Fault masking）：既然异步系统中无法证明能够达成共识，我们可以将异步系统转换为同步系统，故障屏蔽就是第一种方法。故障屏蔽假设故障的进程最终会恢复，并找到一种重新加入分布式系统的方式。如果没有收到来自某个进程的消息，就一直等待直到收到预期的消息。例如，两阶段提交事务使用持久存储，能够从崩溃中恢复。如果一个进程崩溃，它会被重启（自动重启或由管理员重启）。进程在程序的关键点的持久存储中保留了足够多的信息，以便在崩溃和重启时能够利用这些数据继续工作。换句话说故障程序也能够像正确的进程一样工作，只是它有时候需要很长时间来执行一个恢复处理
- 使用故障检测器（Failure detectors）：将异步系统转换为同步系统的第二个办法就是引入故障检测器，进程可以认为在超过一定时间没有响应的进程已经故障。一种常见的故障检测器的实现思路是超时（timeout）机制。但是，这种办法要求故障检测器是精确的。如果故障器不精确的话，系统可能放弃一个正常的进程；如果超时时间设定得很长，进程就需要等待（并且不能执行任何工作）较长的时间才能得出出错的结论。这个方法甚至有可能导致网络分区。解决办法是使用“不完美”的故障检测器。Chanadra 和 Toueg 在《The weakest failure detector for solving consensus》中分析了一个故障检测器必须拥有的两个属性：
  - 完全性（Completeness）：每一个故障的进程都会被每一个正确的进程怀疑
  - 精确性（Accuracy）：正确的进程没有被怀疑

  同时，他们还证明了，即使是使用不可靠的故障检测器，只要通信可靠，崩溃的进程不超过 $N/2$，那么共识问题是可以解决的。我们不需要实现强完全性（Strong Completeness）和强精确性（Strong Accuracy），而只需要一个最终弱故障检测器（eventually weakly failure detector），并且该检测器具有如下性质：
  - 最终弱完全性（eventually weakly complete）：每一个错误进程最终常常被一些正确进程怀疑
  - 最终弱精确性（eventually weakly accurate）：经过某个时刻后，至少一个正确的进程从来没有被其它正确进程怀疑
- 使用随机性算法（Non-Determinism）：这种解决不可能性的技术是引入一个随机算法，随机算法的输出不仅取决于外部的输入，还取决于执行过程中的随机概率。因此，给定两个完全相同的输入，该算法可能会输出两个不同的值。随机性算法使得“敌人”不能有效地阻碍达成共识。和传统选出领导节点再协作的模式不同，像区块链这类共识是基于哪个节点最快计算

出难题来达成的。区块链中每一个新区块都由本轮最快计算出数学难题的节点添加，整个分布式网络持续不断地建设这条有时间戳的区块链，而承载了最多计算量的区块链正是达成了共识的主链（即累积计算难度最大）。比特币使用了 PoW（Proof of Work）来维持共识，一些其它加密货币（如 DASH、NEO）则使用 PoS（Proof of Stake），还有一些（如 Ripple）使用分布式账本（ledger）。但是，这些随机性算法都无法严格满足安全性。攻击者可以囤积巨量算力，从而控制或影响网络的大量正常节点，例如控制 50% 以上网络算力即可以对 PoW 发起 Sybil 攻击。只不过前提是攻击者需要付出一大笔资金来囤积算力，实际中这种风险性很低，如果有这么强的算力还不如直接挖矿赚取收益

2.5 同步系统中的 Consensus

我们之前提到的解决方法的中的第一个方法和第二个方法都想办法让系统达到一定程度的“同步”。Paxos 在异步系统中，由于活锁的存在，并没有完全解决共识问题（liveness 不满足）。但 Paxos 被广泛应用在各种分布式系统中，就是因为在达成共识之前，系统并没有那么“异步”，还是有极大概率达成共识的。Dolev 和 Strong 在论文《Authenticated Algorithms for Byzantine Agreement》中证明了: 同步系统中, 如果 N 个进程中最多有 f 个会出现崩溃故障, 那么经过 $f+1$ 轮消息传递后即可达成共识。Fischer 和 Lynch 的论文《A lower bound for the time to assure interactive consistency》证明了该结论同样适用于拜占庭故障。

基于此，大多数实际应用都依赖于同步系统或部分同步系统的假设。

Leslie Lamport、Robert Shostak 和 Marshall Pease 在论文《The Byzantine General’s Problem》中讨论了 3 个进程互相发送未签名（口头的）的消息的情况，并证明了只要有一个进程出现故障，就无法满足拜占庭将军的条件。但如果使用签名的消息，那么 3 个将军中有一个出现故障，也能实现拜占庭共识。Pease 将这种情况推广到了 N 个进程，也就是在一个有 f 个拜占庭故障节点的系统中，必须总共至少有 $3f+1$ 个节点才能够达成共识。即 $N \geq 3f+1$。

虽然同步系统下拜占庭将军问题的确存在解, 但是代价很高, 需要 $O(N^{(f+1)})$ 的信息交换量，只有在那些安全威胁很严重的地方才会考虑使用。

PBFT（Practical Byzantine Fault Tolerance）算法顾名思义是一种实用的拜占庭容错算法，由 Miguel Castro 和 Barbara Liskov 发表于 1999 年。PBFT 算法也是通过使用同步假设保证活性来绕过 FLP 不可能性。PBFT 算法容错数量同样也是 $N \geq 3f+1$，但只需要 $O(n^2)$ 信息交换量，即每台计算机都需要与网络中其他所有计算机通讯。虽然 PBFT 算法已经有了一定的改进，但在大量参与者的场景还是不够实用，不过在拜占庭容错上已经作出很重要的突破，一些重要的思想也被后面的共识算法所借鉴。

2.6 CAP 原理

在 2000 年的分布式计算原则研讨会（PODC）上，计算机科学家 Eric Brewer 针对分布式计算系统的一致性（Consistency）、可用性（Availability）和分区容错性（Partition-tolerant）提出了猜想。在 2002 年，他的猜想得到了来自麻省理工学院的两位教授 Nancy Lynch 和 Seth Gilbert 的证明，并被称为 CAP 定理。

CAP 定理证明了：当网络存在分区时，提供可靠的原子一致性数据是不可能的，但是想要实现一致性、可用性、分区容错性，三个属性中的两个是可行的。在异步通信系统中，当没有锁提供时，如果出现消息丢失，即使允许过时的数据返回，提供一致性数据也是不可能的。在同步通信系统中，可以在一致性和可用性间取得一定的平衡。

2.6.1 Consistency

CAP 理论的论证中，把一致性定义限定在原子数据对象上，这和其他大多数正式定义一致性服务的方法相同。满足一致性条件的系统，对所有操作都有统一记录，这些操作记录看起来像是一个单独的实例完成的。这要求分布式系统的所有请求必须进行同步，然后才能执行操作。最终呈现的结果，像是同一个节点在同一时间响应，然后执行的一样。

这是提供给用户理解的，最简单的一致性保障模型，也是给设计分布式客户端应用的人理解的最简单的模型，简单来讲就是所有节点都能在同一时间返回同一份最新的数据副本。

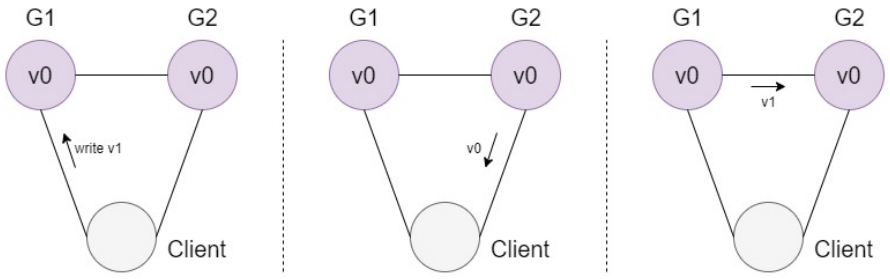

图 1: Consistency 概念图

某条记录是 v0，用户向 G1 发起一个写操作，将其改为 v1。用户有可能向 G2 发起读操作，由于 G2 的值没有发生变化，因此返回的是 v0。G1 和 G2 读操作的结果不一致，这就不满足一致性了。为了让 G2 也能变为 v1，就要在 G1 写操作的时候，让 G1 向 G2 发送一条消息，要求 G2 也改成 v1。

2.6.2 Availability

为了能让分布式系统持续可用，每个请求会被发送给一个系统中的正常节点，并收到响应。这是任何分布式服务使用的算法必须要满足的。CAP 理论的论

证中，将可用性定义为每次请求都能够返回非错误的响应，具体可以分为两种情况：

- 弱可用性：在终止之前，算法运行的多久是没有边界的，因此允许没有边界的计算
- 强可用性：当服务网络发生错误时，每个请求也必须被响应

在弱可用性条件下，系统对响应时间可以不做保证，但是必须做出响应，当系统出现错误时，并不保证对请求做出响应。而在强可用性条件下，即使系统出现错误，请求也必须得到响应。

用户可以选择向 G1 或 G2 发起读操作。不管是哪台服务器，只要收到请求，就必须告诉用户，无论是 v0 还是 v1，否则就不满足可用性。

2.6.3 Partition-tolerant

CAP 理论的论证中，分区指的是，网络中允许丢失从一个节点发送到另一个节点的任意数量的消息。这意味着当网络中出现分区时，从一个分区中的节点发送给另外一个分区的节点的消息将会全部丢失。

分区容错指的是，在出现分区时，系统依然能够满足以上定义中的一致性和可用性，也就是说服务器间的通信即使在一定时间内无法保持畅通也不会影响系统继续运行。原子性要求意味着每一个响应将会是原子性的，尽管任意作为算法的一部分的消息可能不会被传递。可用性要求意味着，每个节点收到的客户端请求必须被响应，尽管任意的消息都可能丢失。

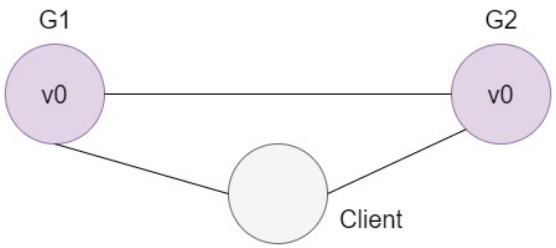

图 2: Partition-tolerant 概念图

G1 和 G2 是两台跨区的服务器。G1 向 G2 发送一条消息，G2 可能无法收到。系统设计的时候，必须考虑到这种情况。一般来说，分区容错无法避免，因此可以认为 CAP 的分区容错总是成立。而 CAP 定理的一致性和可用性的矛盾则告诉我们，剩下的 C 和 A 无法同时做到。

2.6.4 Consistency 和 Availability 的矛盾

需要注意的是，一致性和可用性，可能同时成立，也可能出现矛盾。如果保证 G2 的一致性，那么 G1 必须在写操作时，锁定 G2 的读操作和写操作。只有数据同步后，才能重新开放读写。锁定期间，G2 不能读写，没有可用性。如果保证 G2 的可用性，那么势必不能锁定 G2，所以一致性不成立。综上所述，G2 无法同时做到一致性和可用性。

系统设计时只能选择一个目标。如果追求一致性，那么无法保证所有节点的可用性；如果追求所有节点的可用性，那就没法做到一致性。

2.6.5 CAP 与区块链不可能三角

在区块链领域中，安全性、可扩展性、去中心化，三者被称作区块链的“不可能三角”，意思是说，在同一个区块链系统中，想要同时做到三者，并且都达到足够高的要求，是不可能做到的。基本论证思路是，CAP 理论在分布式系统中成立，区块链属于分布式系统，区块链必然遵守 CAP 理论，只要能证明 CAP 理论中的一致性（Consistency）、可用性（Availability）和分区容错性（Partition tolerance），与区块链的不可能三角存在相应的逻辑关系，即可证明区块链不可能三角不可突破。

第一，在 CAP 理论的证明过程中，证明了在异步网络模型中，实现一个读/写数据对象同时具备可用性和一致性是不可能的。我们将该结论和证明过程，对应到区块链系统中。在 CAP 理论限定的网络环境中，我们假设一个区块链系统中，存在 A 和 B 两个节点，其中 A 和 B 同时记录了一个地址 H 的加密货币余额为 X1，此时 A 和 B 出现了分区。当用户在 A 节点所在的分区发起一笔交易时，地址 H 中的余额将会发生变化，成为 X2。当用户在 B 节点所在的分区发起一次余额查询操作时，地址 H 中的余额依然是 X1。由此我们说区块链系统中，出现了账本不一致的情况。当区块链系统中出现不一致状态时，我们认定这样的区块链系统是不安全的。在这样的定义下，一致性是区块链系统安全的基本前提。或者说区块链的安全性是比分布式系统的一致性更加严格的需求。

所以安全性大于一致性。

第二，在 CAP 理论的可用性定义中，分为弱可用性和强可用性，但是这两种可用性都要求，系统可以对所有正常请求做出响应。从技术的角度来讲，即是可以实现正常的可读可写。在区块链系统中，可扩展性指的是，每秒可以处理的交易量。从技术的角度来讲，高可扩展性即是实现每秒钟高频次的可读可写操作。从逻辑上，我们可以看出，可用性是比可扩展性更基础的网络要求，不能实现可用性的区块链系统，是不能实现可扩展性的，即是可用性是可扩展性的前提。或者说区块链的可扩展性是比分布式系统的可用性更加严格。

所以可扩展性大于可用性。

第三，在 CAP 理论中，分区被认为是分布式系统必然存在的。事实也的确如此，在真实分布式环境中，不可能保证系统中的每个节点，都不会出现任何故

障。去中心化作为区块链的基本特征，意味着区块链系统必然是分布式的，也就是说去中心化必定导致发生分区的可能。

所以分区容错性是实现去中心化的前提。

通过对以上对一致性与安全性、可用性与可扩展性、分区容错性与去中心化的逻辑关系推导，我们可以得出以下结论：

- 一致性是安全性的必要条件
- 可用性是可扩展性的必要条件
- 分区容错性是去中心化的必要条件

又通过 CAP 理论可以知道一致性、可用性、分区容错性是不能同时满足的，所以我们可以得出：在 CAP 理论限定的条件下，安全性、可扩展性、去中心化不能同时满足，即是区块链的不可能三角不可突破。

3 Consensus 实现技术介绍

3.1 Paxos

Pasox 是一系列协议，用于在不可靠或易出错的处理器网络中解决 Consensus（共识）问题。共识是一组参与者就一个结果达成一致的过程。当参与者或其通信可能遇到故障时，此问题变得困难。Leslie Lamport 和 Fred Schneider 的研究表明：共识协议是分布式计算的状态机复制方法的基础。状态机复制是一种将算法转换为容错分布式实现的技术。

Paxos 协议于 1989 年首次提交，后来在 1998 年作为期刊文章发表。Paxos 协议系列包括在处理器数量、学习商定值之前的消息延迟数量、单个参与者的活动级别、发送的消息数量和故障类型之间的一系列权衡。Paxos 通常用于需要持久性的地方（例如复制文件或数据库），其中持久状态的数量可能很大。即使在有限数量的副本无响应期间，该协议也会尝试取得进展。还有一种机制可以删除永久失败的副本或添加新副本。

3.1.1 Paxos 发展历史

1988 年，Lynch、Dwork 和 Stockmeyer 证明了 Consensus 在广泛的“部分同步”系统家族中的可解性。Paxos 与“viewstamped replication”中用于协议的协议有很强的相似性，该协议由 Oki 和 Liskov 于 1988 年在分布式事务的上下文中首次发布。

可重构状态机与支持动态组成员资格的可靠组多播协议的先前工作有着密切的联系，例如 Birman 在 1985 年和 1987 年关于虚拟同步 gbcast 协议的工作。

但是，gbcast 在支持持久性和解决分区故障方面是不同寻常的。Lamport、Malkhi 和 Zhou 在一篇论文中详细说明了大多数可靠的多播协议缺少这些属性，而这些属性正是状态机复制模型的实现所必需的。

3.1.2 Paxos 技术细节

为了简化 Paxos 的表述，下面的假设和定义是明确的：

- 处理器方面：
  - 处理器以任意速度运行
  - 处理器可能会出现故障
  - 具有稳定存储的处理器可能会在发生故障后重新加入协议（遵循崩溃恢复故障模型）
  - 处理器不会串通、撒谎或以其他方式试图破坏协议（也就是说，不会发生拜占庭式故障）
  - Consensus 算法可以使用 $n = 2F + 1$ 数量的处理器，并且非故障进程的数量必须严格大于故障进程的数量（即故障进程数量不超过 F）。也可以采用一种协议，该协议可以在任意数量的总故障中幸存下来，只要同时失败的次数不超过 F。对于 Paxos 协议，这些配置可以作为单独的配置来进行处理
- 网络方面：
  - 处理器可以向任何其他处理器发送消息
  - 消息是异步发送的，并且可能需要任意长的时间来传递
  - 消息可能会丢失、重新排序或重复
  - 消息传递时没有损坏（也就是说，不会发生拜占庭式故障）

并且，在 Paxos 中有“角色”的概念：

- Client：向分布式系统发出请求，并等待响应。例如，对分布式文件服务器中的文件的写入请求
- Acceptor（Voters）：充当协议的容错“内存”。Acceptor 被收集到称为 Quorum 的组中。发送给 Acceptor 的任何消息都必须发送给 Acceptor 的 Quorum。除非收到来自 Quorum 中的每个 Acceptor 的副本，否则从 Acceptor 收到的任何消息都会被忽略
- Proposer：提出客户端请求，并试图说服 Acceptor 同意它，并在发生冲突时充当协调者以推动协议向前发展

- Learner：充当协议的复制因子。一旦 Acceptor 同意客户端请求，Learner 可以采取行动（即：执行请求并向客户端发送响应）。为了提高处理的可用性，可以添加额外的 Learner
- Leader：Paxos 需要 Proposer 才能取得进展，某些进程可能自认为他们是 Leader，但协议只有在最终选择其中一个时才能保证进展。如果两个进程认为他们是领导者，他们可能会通过不断提出冲突更新来停止协议

其中，Quorum 是为了保证 Paxos 的安全（或一致性）属性而被提出的。Quorum 被定义为一组 Acceptor 的子集，这样任何两个 Quorum 至少共享一个成员。通常，Quorum 是参与 Acceptor 的任何“多数”。例如，给定一组 Acceptor$\{A, B, C, D\}$，多数 Quorum 将是任意三个 Acceptor：$\{A, B, C\}$、$\{A, C, D\}$、$\{A, B, D\}$、$\{B, C, D\}$。更一般地，可以将任意正权重分配给 Acceptor；在这种情况下，Quorum 可以定义为总权重大于所有 Acceptor 总权重一半的 Acceptor 的任何子集。

提案编号（proposal number）和商定值（agreed value）是指在每次尝试定义一个约定的值 v 时，都会使用 Acceptor 可能接受或不接受的提议来执行。对于给定的 Proposer，每个提案都有唯一的编号。因此，例如，每个提议可以是 (n, v) 的形式，其中 n 是提议的唯一标识符，而 v 是实际提议的值。对应于编号提案的值可以作为运行 Paxos 协议的一部分进行计算，但这并不是必须的。

在 Paxos 的大多数部署中，每个参与进程扮演三个角色：Proposer、Acceptor 和 Learner，这在不牺牲正确性的情况下显著降低了消息的复杂性。通过合并角色，协议“折叠”成高效的客户端-主-副本风格的部署。

基础 Paxos 的实现阶段：

- 阶段 1a【准备】：Proposer 创建一条消息，我们称之为“准备消息”，用数字 n 标识。请注意，n 不是要提议的值，也不是要达成一致的值，而只是一个数字，它唯一地标识 Proposer 的初始消息（将发送给接受者）。数字 n 必须大于此 Proposer 在之前的任何准备消息中使用的任何数字。然后，它将包含 n 的准备消息发送到至少一个 Acceptor 的 Quorum 中。请注意，准备消息仅包含数字 n（也就是说，它不必包含例如建议的值，通常用 v 表示）。Proposer 决定谁在 Quorum 中。如果 Proposer 不能与 Acceptor 的 Quorum 通信，则不应启动 Paxos
- 阶段 1b【承诺】：任何 Acceptor 都等待来自任何 Proposer 的准备消息。如果一个 Acceptor 收到了 Prepare 消息，那么 Acceptor 必须查看刚刚收到的准备消息的标识符号 n，有以下两种情况：
  - 如果 n 大于 Acceptor 从任何 Proposer 收到的每个先前的提案编号，则 Acceptor 必须向 Proposer 返回一条我们称之为“承诺消息”的消息，以忽略所有未来比 n 小的提案编号。如果 Acceptor 在过去的某个时间点接受了一个提案，它必须在其对 Proposer 的响应中包含先前的提案编号
  - 如果 n 小于或等于 Acceptor 从任何 Proposer 收到的任何先前的提案编号，则 Acceptor 可以忽略收到的提案。在这种情况下，它不必回答

Paxos 就可以工作。但是，为了优化，发送拒绝响应会告诉 Proposer，它可以停止与提案 n 建立共识的尝试

- 阶段 2a【接受】：如果 Proposer 从 Acceptor 的 Quorum 中收到承诺消息，则它需要为其提案设置一个值 v。如果任何 Acceptor 之前接受过任何提案，那么他们将把他们的值发送给 Proposer，Proposer 现在必须将其提案的值 v 设置为与 Acceptor 报告的最高提案编号相关联的值，我们称之为 z。如果到目前为止没有任何 Acceptor 接受提案，那么 Proposer 可以选择它最初想要提议的值，比如 x。Proposer 向 Acceptor 的 Quorum 发送一条 Accept 消息 (n, v)，其中包含为其提案 v 选择的值和提案编号 n（与先前发送到准备消息中包含的编号相同），因此，Accept 消息是 $(n, v = z)$ 或者（如果之前没有任何 Acceptor 接受过值）是 $(n, v = x)$
- 阶段 2b【接受】：如果 Acceptor 从 Proposer 接收到 Accept 消息 (n, v)，那么当且仅当它尚未承诺（在阶段 1b 中）只考虑标识符大于 n 的提案时，必须接受它，即：将值 v 注册为（协议的）接受值，并发送一个已接受消息给 Proposer 和每个 Learner（通常可以是 Proposer 自己）

3.2 Raft

Raft 是一种 Consensus 算法，旨在替代 Paxos 系列算法。通过逻辑分离，它比 Paxos 系列算法更易于理解，而且它也被正式证明是安全的，并提供了一些额外的功能。Raft 提供了一种在计算系统集群中分布状态机的通用使用方法，确保集群中的每个节点都同意相同的一系列状态转换。它有许多开源参考实现，在 Go、C++、Java 和 Scala 等语言中都有完整且规范的实现。

3.2.1 Raft 基本概念

Raft 通过选举的领导者达成共识。Raft 集群中的服务器要么是领导者，要么是追随者，并且在选举的精确情况下可以成为候选人（领导者不可用）。领导者负责将日志复制到追随者。它通过定期发送消息通知追随者它的存在。每个追随者都有一个超时时间（通常在 150 到 300 毫秒之间），在该超时时间内它期望来自领导者的定期消息。接收到定期消息时会重置超时。如果没有收到定期消息，追随者将其状态更改为候选人并开始领导选举。

Raft 协议可以使得一个集群的服务器组成复制状态机。一个分布式的复制状态机系统由多个复制单元组成，每个复制单元均是一个状态机，它的状态保存在一组状态变量中，状态机的变量只能通过外部命令来改变。简单理解的话，可以想象成是一组服务器，每个服务器是一个状态机，服务器的运行状态只能通过一行行的命令来改变。每一个状态机存储一个包含一系列指令的日志，严格按照顺序逐条执行日志中的指令，如果所有的状态机都能按照相同的日志执行指令，那么它们最终将达到相同的状态。因此，在复制状态机模型下，只要保证了操作日志的一致性，我们就能保证该分布式系统状态的一致性。

3.2.2 Raft 与 Consensus

Raft 通过领导者方法实现 Consensus，领导者可以在不咨询其他服务器的情况下决定新条目的放置和建立它与其他服务器之间的数据流。一个领导者领导直到它失败或断开连接，在这种情况下，一个新的领导者被选举出来。

并且，在 Raft 协议中，时间被分成了一些任意长度的时间片，称为 term，term 使用连续递增的编号的进行识别。每一个 term 都从新的选举开始，candidate 们会努力争取称为 leader。一旦获胜，它就会在剩余的 term 时间内保持 leader 状态，在某些情况下选票可能被多个 candidate 瓜分，无法形成 Quorum，因此 term 可能直至结束都没有 leader，下一个 term 很快就会到来重新发起选举。term 也起到了系统中逻辑时钟的作用，每一个 server 都存储了当前 term 编号，在 server 之间进行交流的时候就会带有该编号，如果一个 server 的编号小于另一个的，那么它会将自己的编号更新为较大的那一个；如果 leader 或者 candidate 发现自己的编号不是最新的了，就会自动转变为 follower；如果接收到的请求的 term 编号小于自己的当前 term 将会拒绝执行。

综上所述，共识问题在 Raft 中被分解为下面列出的两个相对独立的子问题：

- Leader election：当现有领导者失效或算法初始化时，需要选举新的领导者。在这种情况下，集群中会开始一个新 term，即服务器上需要选举新领导者的任意 term。每个 term 都以领导人选举开始。如果选举成功完成（即选出了一个领导者），则该 term 将继续进行由新领导者精心策划的正常操作。如果选举失败，则新的 term 开始，新的选举开始。领导者选举由候选服务器启动。如果服务器在称为选举超时的一段时间内没有收到领导者的通信，它就会成为候选人，因此假设不再有代理领导者。它通过增加任期计数器来开始选举，将自己投票为新的领导者，并向所有其他服务器发送请求投票的消息。服务器每个 term 仅投票一次，先到先得。如果候选人从另一台服务器接收到一个任期号大于候选人当前任期的消息，则候选人的选举失败，候选人变为追随者，并认为领导者是合法的。如果候选人获得多数票，那么它将成为新的领导者。如果两者都没有发生，例如，由于投票分歧，那么新的 term 开始，新的选举开始
- Log replication：领导者负责日志复制，它接受客户端请求，每个客户端请求都包含一个要由集群中的复制状态机执行的命令。在作为新条目附加到领导者的日志后，每个请求都作为 AppendEntries 消息转发给追随者。在追随者不可用的情况下，领导者无限期地重试 AppendEntries 消息，直到日志条目最终被所有追随者存储。一旦领导者从其大多数追随者那里收到该条目已被复制的确认，领导者就会将该条目应用于其本地状态机，并且该请求被视为已提交。此事件还提交领导者日志中的所有先前条目。一旦追随者得知日志条目已提交，它将将该条目应用于其本地状态机。在领导者崩溃的情况下，日志可能会不一致，来自旧领导者的一些日志没有通过集群完全复制。新的领导者将通过强制追随者复制自己的日志来处理不一致。为此，对于每个追随者，领导者将其日志与追随者的日志进行比较，找到他们同意的最后一个条目，然后删除追随者日志中此关键条目之后的所有

条目，并将其替换为自己的日志条目。此机制将在发生故障的集群中恢复日志一致性

并且，在日志复制过程中，领导者进行日志复制的步骤为：

1. leader append log entry
2. leader issue AppendEntries RPC in parallel
3. leader wait for majority response
4. leader apply entry to state machine
5. leader reply to client
6. leader notify follower apply log

可以看到日志的提交过程有点类似两阶段提交（2PC），不过与 2PC 的区别在于，leader 只需要大多数（majority）节点的回复即可，这样只要超过一半节点处于工作状态则系统就是可用的。leader 只需要日志被复制到大多数节点即可向客户端返回，一旦向客户端返回成功消息，那么系统就必须保证 log（其实是 log 所包含的 command）在任何异常的情况下都不会发生回滚。这里有两个词：commit（committed），apply（applied），前者是指日志被复制到了大多数节点后日志的状态；而后者则是节点将日志应用到状态机，真正影响到节点状态。

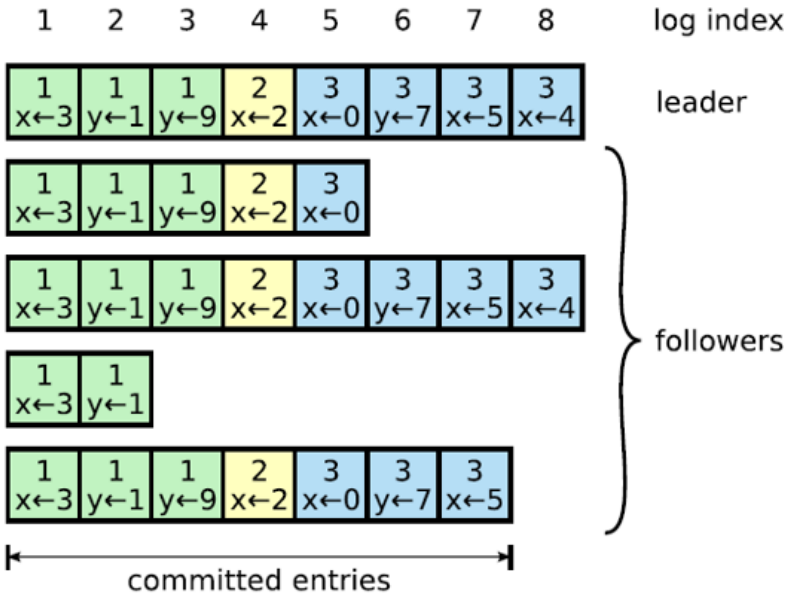

图 3: 日志复制概念图

在上图中，logs 由顺序编号的 log entry 组成，每个 log entry 除了包含 command，还包含产生该 log entry 时的 leader term。从上图可以看到，五个节点的日志并不完全一致，Raft 算法为了保证高可用，并不是强一致性，而是最终一致性，leader 会不断尝试给 follower 发 log entries，直到所有节点的 log entries 都相同。

在日志复制过程中，可能会出现一种特殊故障：如果 leader 崩溃了，它所记录的日志没有完全被复制，会造成日志不一致的情况，follower 相比于当前的 leader 可能会丢失几条日志，也可能会额外多出几条日志，这种情况可能会持续几个 term，如下图所示：

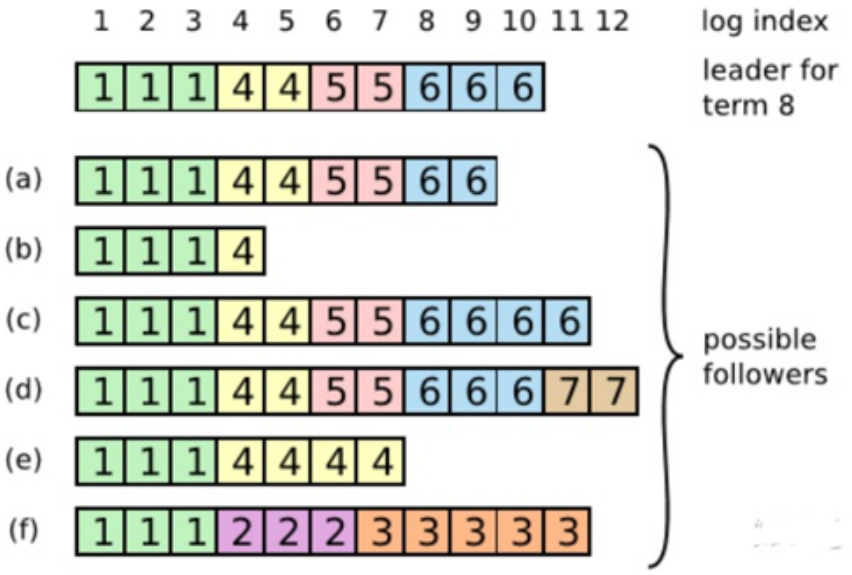

图 4: 故障概念图

图中框内的数字是 term 编号，a、b 丢失了一些命令，c、d 多出来了一些命令，e、f 既有丢失也有增多，这些情况都有可能发生。比如 f 可能发生在这样的情况下：f 节点在 term2 时是 leader，在此期间写入了几条命令，然后在提交之前崩溃了，在之后的 term3 中它很快重启并再次成为 leader，又写入了几条日志，在提交之前又崩溃了，等他苏醒过来时新的 leader 来了，就形成了上图情形。在 Raft 中，leader 通过强制 follower 复制自己的日志来解决上述日志不一致的情形，那么冲突的日志将会被重写。为了让日志一致，先找到最新的一致的那条日志（如 f 中索引为 3 的日志条目），然后把 follower 之后的日志全部删除，leader 再把自己在那之后的日志一起推送给 follower，这样就实现了一致。而寻找该条日志，可以通过 AppendEntries RPC，该 RPC 中包含着下一次要执行的命令索引，如果能和 follower 的当前索引对应，那就执行，否则拒绝，然后 leader 将会逐次递减索引，直到找到相同的那条日志。

3.2.3 Raft 的安全规则

Raft 保证了以下的安全特性：

- Election safety：在给定的任期内最多可以选举一个领导者
- Leader append-only：领导者只能将新条目附加到其日志中（它既不能覆盖也不能删除条目）

- Log matching：如果两个日志包含具有相同索引和术语的条目，则日志在通过给定索引的所有条目中都是相同的
- Leader completeness：如果一个日志条目在给定期限内提交，那么它将因为这个给定期限而出现在领导者的日志中
- State machine safety：如果服务器已将特定日志条目应用到其状态机，则其他服务器不得对同一日志应用不同的命令

对于 State machine safety 而言，如果节点将某一位置的 log entry 应用到了状态机，那么其他节点在同一位置不能应用不同的日志。简单点来说，所有节点在同一位置（index in log entries）应该应用同样的日志。但是似乎有某些情况会违背这个原则。

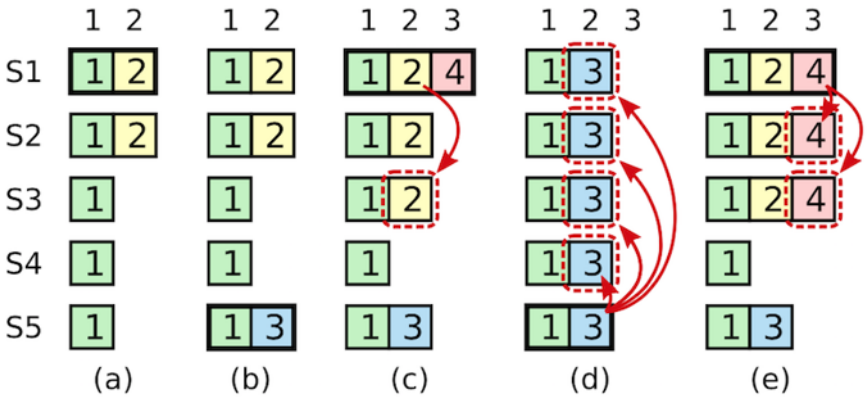

图 5: 一个可能出现的问题

上图是一个较为复杂的情况。在时刻 (a)，s1 是 leader，在 term2 提交的日志只赋值到了 s1 和 s2 两个节点就 crash 了。在时刻 (b)，s5 成为了 term3 的 leader，日志只赋值到了 s5，然后 crash。然后在 (c) 时刻，s1 又成为了 term4 的 leader，开始赋值日志，于是把 term2 的日志复制到了 s3，此刻，可以看出 term2 对应的日志已经被复制到了 majority，因此是 committed，可以被状态机应用。不幸的是，接下来 (d) 时刻，s1 又 crash 了，s5 重新当选，然后将 term3 的日志复制到所有节点，这就出现了一种奇怪的现象：被复制到大多数节点（或者说可能已经应用）的日志被回滚。

究其根本，是因为 term4 时的 leader s1 在 (c) 时刻提交了之前 term2 任期的日志。为了杜绝这种情况的发生，某个 leader 选举成功之后，不会直接提交前任 leader 时期的日志，而是通过提交当前任期的日志的时候“顺手”把之前的日志也提交了。现在的问题是：如果 leader 被选举后没有收到客户端的请求那么会发生什么？相关论文中提到，在任期开始的时候发立即尝试复制、提交一条空的 log。因此，在上图中，不会出现 (c) 时刻的情况，即 term4 任期的 leader s1 不会复制 term2 的日志到 s3。而是如同 (e) 描述的情况，通过复制-提交 term4 的日志顺便提交 term2 的日志。如果 term4 的日志提交成功，那么 term2 的日志也一定提交成功，此时即使 s1 出现了 crash 情形，s5 也不会重新当选。

3.2.4 Raft 的工作过程

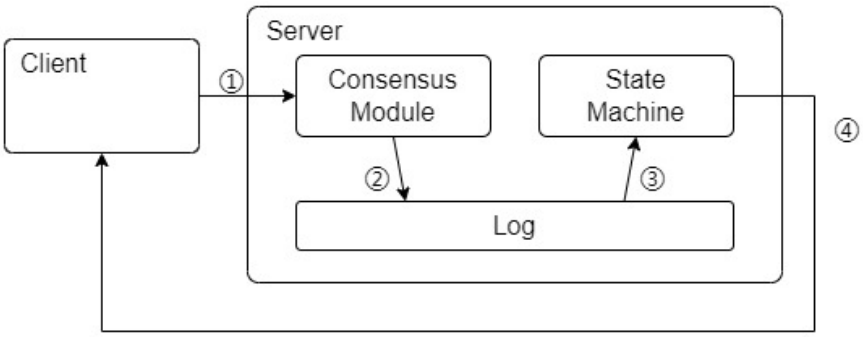

图 6: Raft 工作过程概念图

Raft 算法的工作过程如下所述：

1. Client 发起请求，每一条请求包含操作指令
2. 请求交由 Leader 处理，Leader 将操作指令（entry）追加（append）至操作日志，紧接着对 Follower 发起 AppendEntries 请求、尝试让操作日志副本在 Follower 落地
3. 如果 Follower 的 Quorum 同意 AppendEntries 请求，则 Leader 进行 commit 操作，把指令交由状态机处理
4. 状态机处理完成后将结果返回给 Client

3.3 ZAB

Zookeeper Atomic Broadcast（ZAB）协议实现了主备模式下的系统架构，保持集群中各个副本之间的数据一致性。ZAB 协议定义了选举（election）、发现（discovery）、同步（sync）和广播（Broadcast）四个阶段：

- 选举阶段：选出哪台为主机
- 发现阶段和同步阶段：当主机选出后，要做的恢复数据的阶段
- 广播阶段：当主机和从选出并同步好数据后，正常的主写同步从写数据的阶段

在 Zookeeper 集群中，有领导者（Leader）、跟随者（Follower）和观察者（Observer）三种角色；而 Zookeeper 集群中的节点又有选举状态（Looking）、领导者状态（Leading leader）和跟随者状态（Following follower）三种状态。

每次写成功的消息，都有一个全局唯一的标识，叫 zxid，是 64bit 的正整数，高 32 为叫 epoch 表示选举纪元，低 32 位是自增的 id，每写一次加一。

Zookeeper 集群一般都是奇数（如 $n = 2n + 1$）个机器，且只有一个主机 leader，其余都是从机 follower。必须要有至少 $n + 1$ 台选举相同，才能执行选举的操作。在投票优先级方面：优先比较 zxid，如果相等，再比较机器的 id，都按从大到小的顺序。

3.3.1 选举阶段

当集群新建、或者主机死机、或者主机与一半或以上的从机失去联系后，都会触发选择新的主机操作。有两种算法 fast paxos 和 basic paxos 实现选举。

默认 ZAB 采用的算法是 fast paxos 算法。每次选举都要把选举轮数加一，类似于 zxid 里的 epoch 字段，防止不同轮次的选举互相干扰。每个进入 looking 状态的节点，最开始投票给自己，然后把投票消息发给其它机器。内容为 < 第几轮投票, 被投节点的 zxid, 被投节点的编号 >。其他 looking 状态的节点收到后，进行如下判断：

- 判断票是否有效，方法为看票的投票轮数和本地记载的投票轮数是否相等：
  - 如果比本地投票轮数的小，丢弃
  - 如果比本地投票轮数的大：
    - 第一，证明自己投票过期了，清空本地投票信息
    - 第二，更新投票轮数和结果为收到的内容
    - 第三，通知其他所有节点新的投票方案
  - 如果和本地投票轮数相等，按照投票的优先级比较收到的选票和自己投出去的选票：
    - 如果收到的优先级大，则更新自己的投票为对方发过来投票方案，把投票发出去
    - 如果收到的优先级小，则忽略该投票
    - 如果收到的优先级相等，则更新对应节点的投票
- 每收集到一个投票后，查看已经收到的投票结果记录列表，看是否有节点能够达到一半以上的投票数。如果有达到，则终止投票，宣布选举结束，更新自身状态。然后进行发现和同步阶段。否则继续收集投票

而对于 basic paxos 算法来讲：

1. 每个 looking 节点先发出请求，询问其他节点的投票。其他节点返回自己的投票 < zk 的 id, zxid >，第一次都投自己
2. 收到结果后，如果收到的投票比自己投票的 zxid 大，更新自己的投票
3. 当收到所有节点返回后，统计投票，有一个节点的选举达到一半以上，则选举成功。否则继续开始下一轮询问，直到选择出 leader 结束

现在来比较 basic paxos 和 fast paxos 两种方法。fast paxos 方法是主动推送出，只要结果有更新，就马上同步给其他节点。其他节点可能还没把自己的票通知给所有节点，就发现自己投的票优先级低，要更新投票，然后更新再重新通知给所有节点。而 basic paxos 方法则要每一节点都询问完，才能知道新结果，然后再去问其他节点新的选举结果。fast 比 basic 快的地方是一个节点不用和每个节点都交换投票信息后才能知道自己的票是否要更新，这可以显著减少交互次数。

3.3.2 广播阶段

主从同步数据比较简单，当有写操作时，如果是从机接收，会转到主机。做一次转发，保证写都是在主机上进行。主先提议事务，收到过半回复后，再发提交。主收到写操作时，先本地生成事务为事务生成 zxid，然后发给所有 follower 节点。当 follower 收到事务时，先把提议事务的日志写到本地磁盘，成功后返回给 leader。leader 收到过半反馈后对事务提交。再通知所有的 follower 提交事务，follower 收到后也提交事务，提交后就可以对客户端进行分发了。

4 结语

随着互联网系统日益复杂，大多数系统都从单体架构转向分布式架构，而在区块链这样以分布式技术为基础的技术更是高度依赖数据一致性和共识机制。Consensus 技术有效地提高了分布式数据服务的性能、增加了系统的鲁棒性，这使其实际性能表现超出了我们的预期！

在本文中，我简要梳理了 Consensus 技术的一些常见内容，感觉收获颇丰，同时又惊叹于计算机工业界实践成果之富，这值得我们持续不断地学习与探索！