第六章　共识算法

目录

01 共识算法概述

1.1 分布式系统 -- 概念

- 区块链系统首先是一种分布式系统，每个节点独立运行交易。不同的节点可能运行在地理上分离的区域，有些节点可能发生故障。**区块链系统作为一个整体应该对外提供可靠、可用、一致性的服务，但处理结果的一致性是基础。**

- 分布式系统分为同步和异步分布式系统。

1.1 分布式系统 -- 同步分布式系统

同步分布式系统需要严格满足：
（1）节点运行指令的时间有严格的上限和下限；
（2）消息能够在有限的时间内传输到目标节点；
（3）每个节点有一个本地时钟，与实际时间的偏移率在一个已知的范围内。

对于运行在广域网上的区块链系统来说，上述约束条件很难满足，因为区块链网络中的节点可能在空间上的距离不等。因此**区块链系统一般采用异步分布式系统**。

1.1 分布式系统 -- 异步分布式系统特征

1.1 分布式系统 -- 异步分布式系统限制

01

每一个动作在不同的节点运行时间任意。

02

消息可以在任意长时间后接收到。

03

时钟漂移率可以是任意的。

异步模型对运行速度和消息延迟没有任何假设，这就要求区块链系统中的模型和算法不应立足于某些限制进行设计。

1.1 分布式系统 -- 一致性问题

- **处理结果一致（状态一致）。**指分布式系统中多个节点就某一处理结果达成一致，基本方法是**状态复制**（State Replication）。如传统数据库的发布订阅模式，就实现了数据库的一致性。

- **输入的一致性。**指从多个节点的输入值中，选取其中一个作为决策值，并作为分布式系统共同的输入。这也是区块链中通常所说的**共识**。

1.1 分布式系统 -- 可靠性问题

可靠性（可用性）是描述系统可以提供服务能力的重要指标，通常用两个时间指标MTBF（ Mean Time Between Failures ，平均故障间隔时间）和MTTR（ Mean Time to Repair ，平均修复时间）来衡量。**一个高可用性系统应该具有尽量长的MTBF和尽量短的MTTR。**

1.2 共识 -- 概念

- 假设系统中有n个节点，其中最多有f个节点可能崩溃，在剩下的$n-f$个好的节点中，从节点i提交一个区块B_i（输入值）开始，所有节点要遵循相同的协议从全部提交的区块中最终选择一个区块（一致性结果，决策值），并用该区块提交到区块链系统。

- 从多个节点输入值中选取一个决策值就称为**共识**，达成共识过程中所遵循的协议称为**共识算法**。

1.2 共识 -- 共识满足的条件

01

一致性（Agreement）。
所有好节点的决策值必定相同。

02

可终止性（Termination）。
所有好节点在有限的时间内结束决策过程。

03

有效性（Validity）。
选择出的决策值必须是某个节点的输入值。

1.2 共识 -- 故障容错

- 故障容错是指区块链系统中节点和通信信道都有可能出现故障，导致部分节点不可用或者偏离被认为正确的结果或行为。

- 不同故障的影响：**“遗漏故障”，“随机故障”，“时序故障”**（时序故障主要针对同步分布式系统，对执行时间、通信时间和时钟漂移均有限制）。

1.2 共识 -- 遗漏故障

- 遗漏故障是指节点或通信信道不能完成它应该做的动作，**也称为CFT**。

- 在异步环境下，异步系统对节点运行速度和消息传递延迟都没有限制，因此也无法通过遗漏故障表明某个节点崩溃，这种情况称之为**可变消息延迟（Variable Message Delay）**。区块链系统中共识算法的设计应基于可变消息延迟的模式。

1.2 共识 -- 遗漏故障

节点遗漏故障

主要是**崩溃**，意味着节点停止并不再执行任何动作，也不再对消息进行响应。**一般用一段固定时间等待故障节点对消息的应答来检测**，如果其他节点能确切检测到节点已经崩溃，那么这个节点崩溃称为故障-停止。**但对异步系统，超时只能表明节点没有响应**，这可能是节点崩溃了，也可能是节点执行速度慢甚至是消息还没有到达。

通信遗漏故障

指通过通信信道，节点A不能把消息m成功传送到节点B，也叫**信息丢失**。在存在消息丢失的消息传递模式下，任何一条消息都不能保证可以安全地到达消息的接收者。**存在两种情况，一种是消息到达节点B，但消息内容已经损坏**，这种可以通过消息签名检测；**另一种情况就是消息丢失**。

1.2 共识 -- 随机故障

- 随机故障也称为**拜占庭故障**，用于描述可能发生的任何类型故障，如故意返回错误结果或者处理的结果本身就是一个错误的。

- **节点的随机故障**通常是指节点进程随机地省略要做的处理步骤或执行一些不需要的步骤，从而导致产生错误的执行结果。这类故障**不能通过查看节点是否应答来检测**，因为它可能随机地应答或者应答错误的结果。

- **通信通道也会出现随机故障**，如消息内容可能被损坏或者传递不存在的消息，也可能多次传递同样的消息。这类故障**可以通过校验、消息签名或时间戳等机制进行检测**。

- **因此在基于拜占庭故障的共识算法在设计时，除了考虑拜占庭节点是否返回结果，还需要考虑返回错误结果的情况**，因此基于拜占庭的共识算法通常要求不超过1/3的出错节点，就是考虑到即使1/3的正常节点和1/3的故障节点对冲后，仍然有多数1/3能够使算法形成共识。

1.3 FLP原理

- FLP不可能原理：当允许存在节点失效的情况下，不存在一个确定性的共识算法总能在异步模型下达成一致。
- FLP原理证明：
  - 对于区块链网络，假设当前所有节点的状态的集合为S，节点初始状态值为0，当一个交易到达节点时状态变为1（认为一个节点只要接收到交易必然执行成功），没有到达节点时，状态依然为0。
  - 初始的S为全部是0值的集合[0,0,0, ... ,0]。
  - 当系统接收到一个交易t，不存在失效节点的情况下，经过一段时间后，系统所有节点的状态集合S'为全部是1值的集合[1,1,1, ...,1]。

1.3 FLP原理

- 初始状态S和执行一个交易后的状态S'都是单价的，即所有的节点状态都是一致的。
- 假设存在一个失效节点，必然存在一个节点没有收到交易，即状态依然为0，则经过一段时间后所有节点的状态集合S"为包含0或1两个值的集合[1,1,1,...,0,...,1]。
- S"是二价的即所有的节点状态是不相同的。
- 在满足可变消息延迟的模式下，不得不假设某个节点崩溃了，否则共识算法不能满足可终止性的要求。
- 共识算法终止后S"则不能满足一致性的要求

因此当允许存在失效节点的情况下，是不存在一个确定性算法能够使系统达到一致性的要求的。

1.3 FLP原理

- 同步系统则不同，同步系统对节点处理时间、消息传递、时钟都有要求，很容易判断是否失败，从而决定是否重传或者其他方式进行故障排除。

- **FLP原理实际上说明对于允许节点失效的情况下，纯粹的异步系统无法确保一致性在有限的时间内完成。**即便对于非拜占庭错误的前提下，包括Paxos、Raft等共识算法都存在无法达成共识的情况。

- 在工程实践中，在付出一些代价的情况下，获得高效的共识算法还是很有必要的。具体付出什么代价，共识算法能达到什么程度，往往通过CAP原理进行衡量。

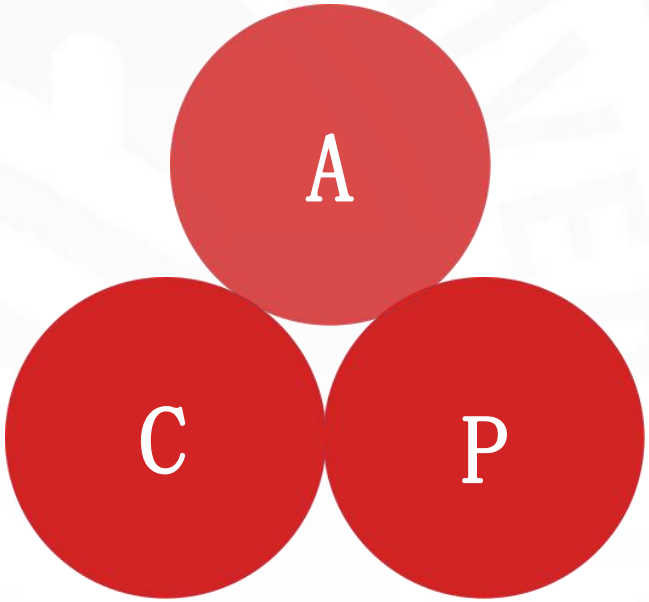
1.4 CAP原理

- CAP 原理是指在一个分布式系统中，一致性（Consistency）、可用性（Availability）、分区容错性（Partition tolerance），三者不可兼得。

- 该原理最早是2000 年由Eric Brewer 在ACM 组织的一个研讨会上提出猜想，后来Lynch 等人进行了证明。

1.4 CAP原理

在有限时间内，任何非崩溃节点都能应答请求。



分布式系统中所有的节点保存的数据副本，在同一时刻是否具有同样的值。

对通信的时限要求。系统如果不能在时限内达成数据一致性，就意味着发生了分区的情况，必须就当前操作在C和A之间做出选择。

1.4 CAP原理

- 基于CAP三种特性不能兼得的原理，分布式系统在设计时必然要弱化对某个特性的支持。那么可能存在下面三个方面的应用场景：

1. 弱化一致性

- 对结果一致性不敏感的应用，可以允许新的数据副本产生后经过一段时间后，所有节点才更新成功，期间不保证一致性。如简单分布式p2p协议Gossip。

- 不同的区块链系统采用不同的机制来确保新的交易所基于的区块链数据库是最新状态。如比特币要求6个区块确认，Fabric要求背书节点对交易背书。

1.4 CAP原理

2. 弱化可用性

- 对结果一致性很敏感的应用，例如银行取款机，当分布式系统对处理不能达到一致结果时会拒绝服务。
- Paxos 、Raft 等共识算法，主要处理这种情况。在Paxos 类算法中，可能存在着无法提供可用执行结果的情形，同时允许少数节点离线。

3. 弱化分区容错性

- 现实中，网络分区出现概率较小，但较难完全避免。两阶段的提交算法，主要考虑了这种设计。
- 实践中，网络可以通过双通道等机制增强可靠性，达到高稳定的网络通信。

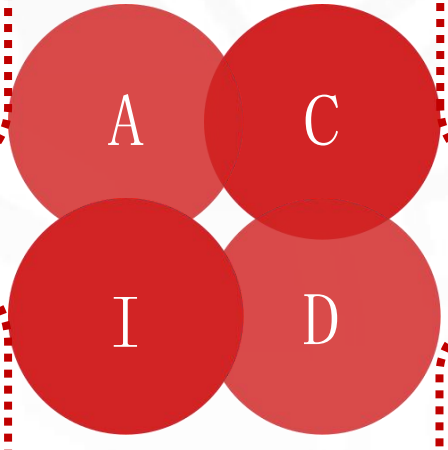
1.5 ACID原则

- ACID 也是一种比较出名的描述一致性的原则，是关系型数据库事务处理的原则。在分布式数据库领域， ACID 原则描述了分布式数据库需要满足的一致性需求，同时允许付出可用性的代价。

- ACID 原则指的是： Atomicity （原子性）、Consistency （一致性）、Isolation （隔离性）、Durability （持久性），用了四种特性的缩写。

1.5 区块链系统中的ACID原则

每次对系统的更新操作都是原子的，要么成功，要么不执行。对于区块链系统，**一个原子操作是一个区块**，因此区块的提交应满足原子性要求。



一个操作开始和结束时，数据库的状态是一致的，无中间状态。对于区块链系统，**不同节点在一个区块提交前后的状态应该是一致的**。

并发操作时，彼此不需要知晓对方存在，执行互不影响，需要串行化执行，有时间顺序。在区块链系统中，**要根据并发交易到达的顺序逐个添加到区块中，在区块提交时按照排序顺序执行**。

状态的改变是持久的，不会失效，也不会无缘无故回滚。区块链系统中，**一个区块一旦达成共识，会永久附加到原有的区块链中，不可篡改**。

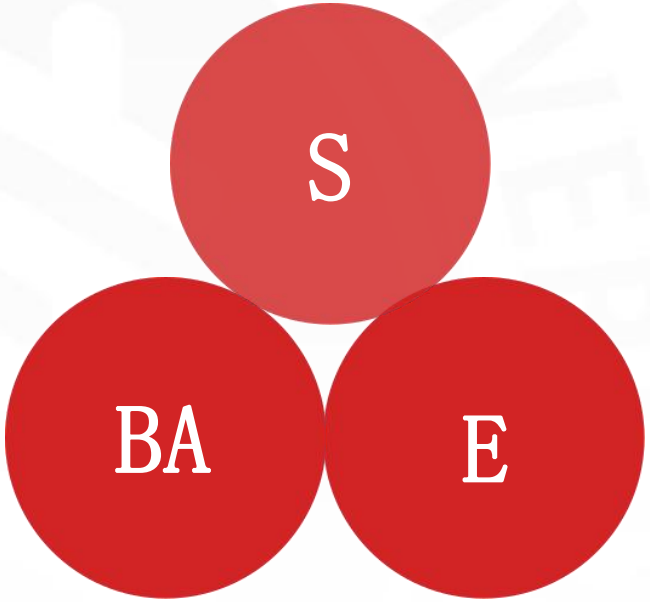
1.6 BASE理论

- 与ACID 相对的一个原则是BASE (Basic Availability, Soft-state, Eventual Consistency)原则，牺牲掉对一致性的约束（但实现最终一致性），来换取一定的可用性。

- BASE是Basically Available（基本可用）、Soft state（软状态）和Eventually consistent（最终一致性）三个短语的缩写。

1.6 BASE理论

允许系统中的数据存在中间状态，并认为该中间状态的存在不会影响系统的整体可用性，即允许系统在不同节点的数据副本之间进行数据同步的过程存在延时。



分布式系统在出现不可预知故障的时候，**允许损失部分可用性**，如响应时间上的损失或系统功能上的损失。

所有的数据副本在经过一段时间的同步之后，最终都能够达到一个一致的状态。因此本质是需要保证**最终达到一致**，而不需要实时保证强一致性。

1.6 BASE理论

- BASE理论是对CAP中一致性和可用性权衡的结果，其来源于对大规模互联网系统分布式实践的总结，是基于CAP定理逐步演化而来的。
- BASE理论的核心思想是：即使无法做到强一致性，但每个应用都可以根据自身业务特点，采用适当的方式来使系统达到最终一致性。
- BASE理论面向的是大型高可用可扩展的分布式系统，和传统的事务ACID特性是相反的，它完全不同于ACID的强一致性模型，而是通过牺牲强一致性来获得可用性，并允许数据在一段时间内是不一致的，但最终达到一致状态。

在实际的分布式场景中，不同业务单元和组件对数据一致性的要求是不同的，因此在具体的分布式系统架构设计过程中，ACID特性和BASE理论往往又会结合在一起。

02 Paxos算法

2.1 Paxos算法背景

Paxos算法是基于消息传递且具有高度容错的一致性算法，是目前公认的解决分布式一致性问题最有效的算法之一。该算法能够解决分布式网络中存在**遗漏故障**，并且**故障节点数小于总节点数的1/2**的场景下的共识问题。

1990年Leslie Lamport 在论文《The Part-time Parliament》中提出的Paxos共识算法，在工程角度实现了一种最大化保障分布式系统一致性（存在极小的概率无法实现一致）的机制。

2006年谷歌发表的论文《The Chubby lock service for loosely-coupled distributed system》中称Paxos算法是到目前为止遇到的解决异步共识问题的最实用算法，从而使得Paxos算法在工业界得到广泛应用。

2.1 Paxos算法背景

- 算法的故事背景是古希腊Paxon岛上的多个法官在一个大厅内对一个议案进行表决，如何达成统一的结果。他们之间通过服务人员来传递纸条，但法官可能离开或进入大厅，服务人员可能偷懒去睡觉。**通过消息传递来逐步消除系统中的不确定状态。**

- Paxos基于“两阶段提交”算法并进行泛化和扩展，是后来不少共识算法（如Raft ， ZAB 等）设计的基础。Paxos 算法被广泛应用在Chubby 、ZooKeeper 这样的分布式系统中。

2.2 Paxos算法三种角色

1. Proposer（**提案者**）。提出一个提案，等待大家批准（chosen）为结案（value）。系统中提案都拥有一个自增的唯一票号。往往由**客户端**担任该角色。

2. Acceptor（**接受者**）。负责对提案进行判断，是否接受（accept）提案，接受记录票号。往往由**服务端**担任该角色。

3. Learner（**学习者**）。获取批准结果，帮忙传播，不参与投票过程。

2.3 Paxos算法两个重要概念

1. Ticket（票）。弱化形式的锁，算法中称为票号。

- 票可以重新发布，当提案者有新的提案产生时，可以生成新的票号。
- 票可以过期，当提案者使用一张票t向接受者发送提案时，只有t是最新的票时，接受者才会接收。

2. 提案（Propose）。

- 是一个包含票号和输入值的一个序列对[N, V]，N是票号，V是提案者提交的输入值。
- V可以是任意类型，可以是一个命令，甚至可以是区块链系统的一个交易。

2.4 Paxos算法三阶段

1. 票号申请

- Proposer选择一个提案编号t，然后向所有的Acceptor发送票号为t的票号申请请求。

```
//阶段 1 准备阶段 - 申请票号
1. t = t + 1 票号递增
2. Broadcast(t, all) 向所有 Acceptor 发送消息，请求得到编号为 t 的票
```

- 如果一个Acceptor收到一个编号为t的票号请求，且t大于该Acceptor已经响应过的所有票号申请请求的票号，那么它就会将它已经接受过的票号最大的提案（如果有的话）作为响应反馈给Proposer，同时该Acceptor承诺不再接受任何票号小于t的提案。

2.4 Paxos算法三阶段

```
//初始化
v = null          当前提交的提案值
t = 0             当前请求的票号
P_accept = [0, null] 当前 Acceptor 端接收的提案

//阶段 1 准备阶段 - 申请票号
1. t = t + 1 票号递增
2. Broadcast(t, all) 向所有 Acceptor 发送消息，请求得到编号为 t 的票
```

票号申请对于形成共识的意义：

（1）解决票号全局一致的问题，使得最终选择的提案排序全局一致；

（2）通过获取票号，使得同一时间仅仅一个Proposer能够发送提案，从而把并行的提案串行化，形成一个串行化的提案序列，并在所有的节点串行化执行，从而实现状态复制；

（3）Proposer通过竞争票号获取提出提案的机会，这样该Proposer提出的提案是最新的提案，即票号最大。

2.4 Paxos算法三阶段

2. 发送提案

- Proposer：收到半数以上Acceptor对其发出的票号为t的票号申请的响应，那么它就会发送一个针对[t,v]提案的Propose消息给所有响应的Acceptor。

- 注意：v就是收到的响应中编号最大的提案的value，如果响应中不包含任何提案，那么v就由Proposer自己决定。

```
//阶段 2 接受阶段 - 发送提案
3. 等待 Acceptor 的响应
4. If 过半数 Acceptor 回复 ok then
5.     Paccept = [taccept, vaccept]，其中 taccept 是所有接收到的回复 ok 的消息中票号最大的提案
6.     If vaccept 不为空 then
7.         v = vaccept
8.     End if
9.     Broadcast( Propose( t, v ), 回复 ok 的 Acceptor)
10. End if
```

2.4 Paxos算法三阶段

- Acceptor：收到一个针对编号为t的提案的Proposer消息，只要该Acceptor没有对编号大于t的票号申请请求做出过响应，它就接受该提案。

```
//阶段 2 接受阶段 - 接受提案
5. If t = t_req then     Accepter 仅接受提案票号是票号申请阶段的票号，并且没有新的票号产生
6.       t_accept = t    保存接受的提案票号和提案的值
7.       v_accept = v
8.       回复 accepted 消息给 Proposer
9. End if
```

- **发送提案对于形成共识的意义：**
（1）所有Proposer都支持最新接受的提案原则，使得所有获取票号的Proposer都支持最新提案；
（2）每个Acceptor仅接受最新的提案号。

2.4 Paxos算法三阶段

3. 提交选定值，形成共识

- Proposer：广播形成共识的值。

```
//阶段 3 提交阶段
11. 等待 Acceptor 的 accepted 消息响应
12. If 过半数 Acceptor 接受该提案 then
13.     Broadcast( v, all ) 向所有节点广播该选定值
14. End if
```

- Acceptor：提交数据并为下一次共识进行准备。

```
//阶段 3 提交阶段 - 记录数据并重新等待接受提案
10. 提交数据
11. t_accept = 0
12. v_accept = null
```

2.5 Paxos算法描述

Proposer端（PA）

```
//初始化
v = null            当前提交的提案值
t = 0               当前请求的票号
P_accept = [0, null]  当前 Acceptor 端接收的提案

//阶段 1 准备阶段 - 申请票号
1. t = t + 1  票号递增
2. Broadcast(t, all)  向所有 Acceptor 发送消息，请求得到编号为 t 的票

//阶段 2 接受阶段 - 发送提案
3. 等待 Acceptor 的响应
4. If 过半数 Acceptor 回复 ok then
5.      P_accept = [t_accept, v_accept]，其中 t_accept 是所有接收到的回复 ok 的消息中票号最大的提案
6.      If v_accept 不为空 then
7.          v = v_accept
8.      End if
9.      Broadcast( Propose( t, v ), 回复 ok 的 Acceptor)
10. End if

//阶段 3 提交阶段
11. 等待 Propose 消息响应
12. If 过半数 Acceptor 接受该提案 then
13.     Broadcast( v, all ) 向所有节点广播该选定值
14. End if
```

2.5 Paxos算法描述

Acceptor端（AA）

```
//初始化
t_req = 0            当前接收到的票号申请的最大票号
t_accept = 0         当前接受的提案的票号
v_accept = null      当前接受的提案的值

//阶段 1 准备阶段 - 接收来自 Proposer 的票号申请请求 t
1. If t > t_req then   仅仅响应申请票号是 Accepter 已接收到的票号申请都要大的票号
2.     t_req = t       更改当前 Accepter 保存的最大票号
3.     回复 ok( t_req, v_accept) 回复票号申请成功消息，并把当前 Accepter 接受的提案返回给 Proposer
4. End if

//阶段 2 接受阶段 - 接受提案
5. If t = t_req then    Accepter 仅接受提案票号是票号申请阶段的票号，并且没有新的票号产生
6.     t_accept = t     保存接受的提案票号和提案的值
7.     v_accept = v
8.     回复 accepted 消息给 Proposer
9. End if

//阶段 3 提交阶段 - 记录数据并重新等待接受提案
10. 提交数据
11. t_accept = 0
12. v_accept = null
```

2.6 Paxos算法分析

1. 运行前提

- 不能超过一半的节点崩溃，超过半数节点无响应，则导致无法申请到票号（PA-4行），也无法形成接受的提案（PA-12）。最终无法对输入的值达成共识。

2. 解决节点宕机问题

- **Acceptor节点宕机：**只要宕机节点数不超过半数，算法能够正常运行并形成共识

- **Proposer节点宕机：**
（1）一个Proposer节点在得到票号之前宕机，不影响其他Proposer申请新的票号；
（2）Proposer得到票号之后宕机，依然不影响其他Proposer节点，这些节点可以申请新的票号，当新的票号大于Acceptor节点保存的已申请最大票号时，Proposer节点依然可以申请到新的票号，并用新的票号发送提案。

2.6 Paxos算法分析

3. 解决票号全局一致的问题

- **如果全局票号不一致，产生的问题：**如果一个Proposer A申请了一个票号t，另外一个Proposer B也可能会申请到票号t，A和B就可能在阶段2提交票号相同但提案值不同的提案，这会导致一部分Acceptor保存A的提案值，一些Acceptor保存B的提案值，最终导致分布式系统的不一致性。

- **如果每个Acceptor独自产生本地票号，产生的问题：**
（1）在PA-4行收到的所有票号中，不同的Proposer可能接收到的最大票号不同，并且可能不是最大的票号；
（2）在某些Acceptor这个票号可能已经过期，也会导致最终的提案值是不一致的。

2.6 Paxos算法分析

- **Paxos算法采用的方法：**让每个Proposer独自产生票号。
（1）Proposer产生票号t后向所有的Acceptor请求票号为t的票，如果该Proposer成功申请票号t，则说明超过一半的Acceptor的最大票号更新为t；
（2）其他Proposer申请同样的票号t或者小于t的票号，都无法申请成功，只能申请大于t的票号；
（3）Paxos算法中Acceptor需要保存“已接收到的最大票号”，对于小于或等于最大票号的申请，Acceptor都不予以支持，从而确保票号在所有节点顺序增加并且不会重复。

2.6 Paxos算法分析

4. 解决并发问题串行化的问题

- 通过准备阶段获取票号，使得同一时间仅仅一个Proposer能够发送提案，从而把并行的提案串行化，形成一个串行化的提案序列，并在所有的节点串行化执行，从而实现状态复制。

- 接受阶段，Proposer一旦获取某个票号t，说明超过一半的Acceptor的最大票号是t（PA-4行），除非有新的Proposer申请大于t的票号t’，那么就形成类似锁的机制，使得Acceptor不会接受其他的票号申请（AA-1行）。

2.6 Paxos算法分析

5. 解决共识问题

- Paxos算法能够对各个Proposer输入的提案值v达成共识，并且会从输入的提案值v中选择一个作为选定值，从而满足共识的要求

- **三个规则：**
  （1）Proposer通过竞争票号获取提出提案的机会，这样该Proposer提出的提案是最新的提案，即票号最大；
  （2）所有Proposer都支持最新接受的提案原则，使得所有获取票号的Proposer提出支持的最新提案；
  （3）一旦一个Proposer发送一个提案Propose(t, v)并被超过半数的Acceptor接受，则提案值v将是所有节点最终接受并提交的提案值，这样就对输入的提案值形成了共识

2.6 Paxos算法分析

（1）假设在阶段2，一个Proposer A发送了提案Propose(t, v)，接受该提案的Acceptor的集合为A。
（2）另外一个Proposer B在阶段三提交之前（PA-12行之后）发送了提案Propose(t’, v’)，接受该提案的集合为A’。
（3）票号申请通过必然意味着集合A和A’包含超过一半的Acceptor，同时意味着t’大于t。
（4）那么集合A和集合A’必然存在一个非空的Acceptor交集C既接收到提案Propose(t, v)也接收到提案Propose(t’, v’)。
（5）基于接受最新提案的原则。
（6）在阶段1节点B申请票号时，集合C中的Acceptor因为已接受Propose(t, v)，因此AA-3行代码会返回（t’, v），节点B会用已接受的提案值v作为B的提案值，而不是B决定的提案值提交提案，因此PA-9行代码会向A’集合中的Acceptor发送Propose(t’, v)。因此v’和v的提案值是相同的。

2.6 Paxos算法分析

（7）假设节点B在PA-12之前发送提案Propose(t’，v’)，这时存在两种情况，一种是提案Propose(t，v)已经被半数的Acceptor接受，则v’与v值相同。另一种是Propose(t，v)没有被半数以上的Acceptor接受，则意味着C中的部分Acceptor在B申请票号t’时返回null的存储值，而Acceptor也会根据AA-5代码接受最新票号的提案Propose(t’，v’)，这时v和v’并不一致。这时Learner会帮助传播提案并**最终选择最新提案v’**作为最终接受的提案。

2.7 Paxos算法运行模式

根据Proposer和Acceptor的多少，可以有不同的运行模式，也可以根据不同的运行模式对算法进行优化。

01

单Proposer多Acceptor（1对多）：

（1）由Proposer维护票号，无需向Acceptor申请。

（2）模式本身是串行化运行，不存在并行化问题；

（3）但Proposer一旦发生故障，则系统无法运行。

02

多Proposer单Acceptor（多对一）：

（1）Acceptor一旦接受某个提案则迅速形成共识。

（2）可直接改进Acceptor第一阶段算法，如有接受提案则重新申请票号。

（3）容易形成Acceptor的单点故障，包括Acceptor接受的提案的发送者Proposer或者Acceptor节点发生故障。

03

多Proposer多Acceptor（多对多）：

是区块链点对点网络中常用的模式，也是Paxos算法的解决重点。

2.8 Paxos算法改进 -- 问题

在Paxos算法中存在某些情况下使得算法失去活性即无法终止。

1. **初始提交提案（t，v）的Proposer在阶段3（过半Acceptor接受提案）崩溃，那么系统无法接受新的提案v’。**该情况下，由于已有过半Acceptor接受提案，其他Proposer提交新的提案v’时，总是会用已经接受的最新提案v作为提案在阶段2提交，那么新的提案无法提交给系统。

2. **如果两个Proposer依次提出最新的提案，则最终会死循环。**假设Proposer A发送提案（t，v），在过半数Acceptor接受之前，Proposer B申请了新的票号t’（根据Acceptor阶段1的代码，由于A的提案没有被半数Acceptor接受，B能够申请到新的票号并且返回的提案为空）并发送了最新的提案（t’，v’）。这时Acceptor会忽略A的提案，接受B的提案。A发现后，在B被半数Acceptor接受之前重新申请票号t”，然后重新发送提案。这样A和B就不断的阻止对方的提案值被接受并形成共识，导致算法无法终止。

2.8 Paxos算法改进 -- 解决方法

1. **增加等待时间 - 解决情景2的问题。**申请票号失败或者没有过半Acceptor接受提案后，算法增加等待时间，则可以减少进入死循环的概率。

2. **选取一个主的Proposer发送提案。**简化多对多模式为1对多模式，尽管会有单点故障问题，但是保证了算法的活性。

3. **Acceptor中增加Proposer故障检测机制 - 解决情景1的问题。**为避免Proposer发送的提案被过半节点接受后，因为出现故障导致算法活性，可以在Acceptor中增加故障检测机制，当检测到发送接受提案的Proposer节点一段时间无响应时，可以选择一个主的Proposer执行阶段3-提交已接受的提案工作。

课堂作业

- 为什么是半数?
- 只有一个值，性能低下，连续多个值怎么运行，Multi-Paxos
- 如果每个竞争到票号的提案者都提交自己的提案，算法会出现什么样的情况?

03 Raft算法

3.1 Raft算法背景

Raft与Paxos一样都是基于CFT解决共识问题，同样要保证n/2+1节点正常就能够提供服务。Raft与Paxos相比，有着基本相同运行效率，但是更容易理解，也更容易被用在系统开发上。

斯坦福大学的Diego Ongaro、John Ousterhout教授ATC2014论文《In Search of an Understandable Consensus Algorithm》中提出了以易懂（Understandability）为目标的新的共识算法Raft。

3.2 Raft算法概述

- Raft算法将共识问题分成三个子问题：选举（Leader election）、日志复制（Log replication）、安全性（Safety）三个子问题。

- Raft首先从全部节点中选择一个领导者（leader），然后由Leader负责从客户端接收输入数据，并负责协调把数据复制到其他节点，从而达到一致性的目标。这个中心节点Leader也容易成为系统瓶颈。

3.3 Raft算法三个角色（状态）

1. Follower（追随者）。刚启动时所有节点为Follower状态，主要负责：（1）响应Leader的日志同步请求；（2）响应候选者的选举请求；（3）把来自客户端的请求转发给Leader。

2. Candidate（候选者）。负责选举投票。Raft刚启动时或者无法接收到Leader的心跳时由一个节点从Follower转为Candidate发起选举，选举出Leader后从Candidate转为Leader。

3. Leader（领导者）。负责：（1）日志的同步管理；（2）处理来自客户端的请求；（3）Follower保持心跳联系。

3.3 Raft算法三个角色（状态）

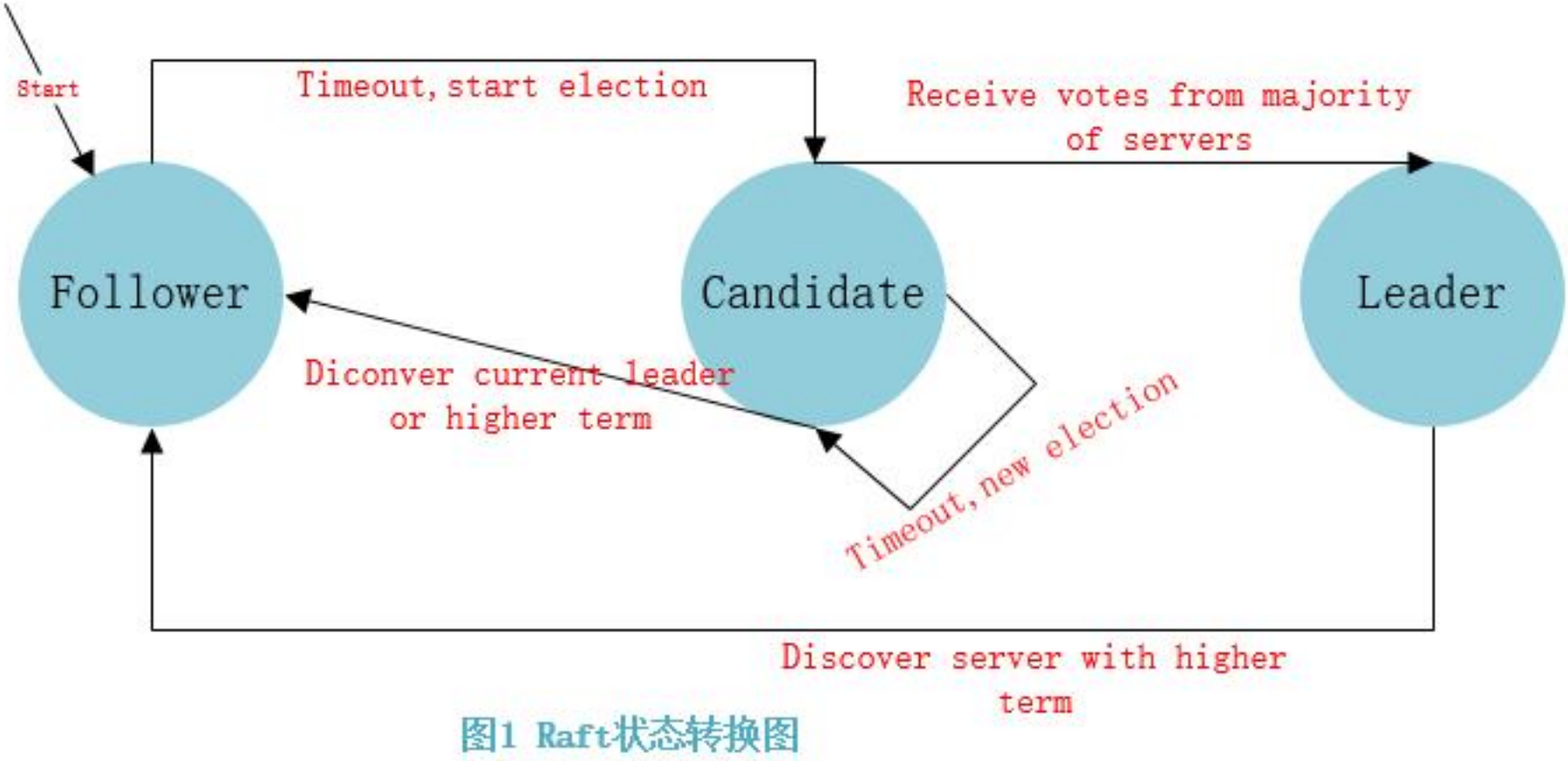


图1 Raft状态转换图

3.4 Raft算法两个消息

1. RequestVote – 请求选票。

- 包含当前节点的任期（Term）、当前节点的名称（CandidateName）、当前节点最新日志条目的任期及日志索引号[LastLogTerm, LastLogIndex]等信息。

2. AppendEntries – 同步日志

- 日志条目同步消息是Leader节点定时发出的心跳信号，同时也作为日志同步的消息和日志提交消息。

- 消息内容主要包括：Leader节点名称（LeaderName）、当前任期（Term）、已提交索引号（CommitIndex）、需同步日志条目的第一个索引号（PrevLogIndex）及已同步日志条目的最新任期信息（PrevLogTerm）、需同步的日志条目列表（可以一次同步多个）（Entries，LogEntry结构的数组）。

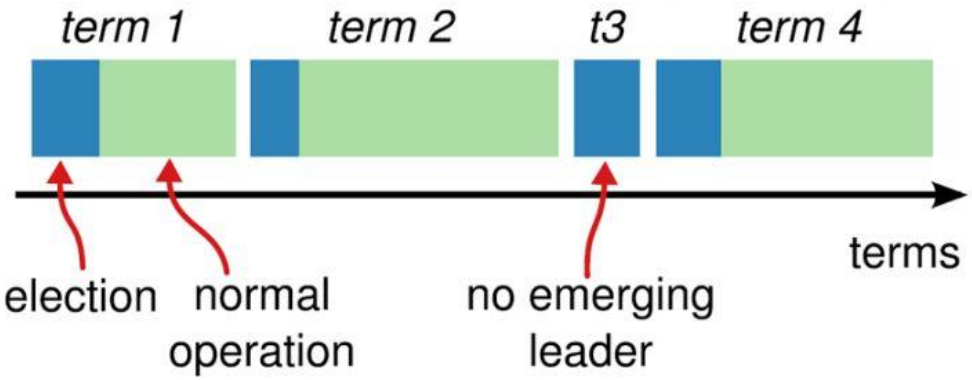
3.4 Raft算法两个消息

- 当Leader节点初始启动时，会以Leader节点的最新日志条目[LastLogTerm, LastLogIndex]作为首次日志同步消息的PrevLogIndex，最新日志条目的前一个日志条目的任期号作为PrevLogTerm并将最新日志条目作为同步的日志条目列表形成AppendEntries请求消息。

- 当Follower节点中PrevLogIndex之前的日志条目的任期与PrevLogTerm不一致，则说明存在PrevLogIndex之前的日志条目与Leader不一致，需要同步PrevLogIndex之前的一个日志条目，直到找到Follower节点的日志条目与Leader的日志条目一致的索引号。

3.5 Raft算法重要概念 -- 任期

- 在Raft算法中引入了任期（Term）的概念，可以理解为任期（如总统的第几届），每个任期都是一个连续递增的编号。
- 每个任期会选举出一个Leader，直到Leader异常才会开始一个新的任期。
- Raft算法开始时所有的节点任期值为1，当某个节点转变为Candidate后，本节点的任期值变更为2，然后开始选举。



3.5 Raft算法重要概念 -- 任期

开始选举后，可能会出现的情况：

1. 没有选举出Leader

- 当本任期没有选举出Leader时，所有的Follower节点在某个随机时间结束后都有机会转变为Candidate，然后任期值递增，开始新一期的选举过程。

2. 本任期选举出Leader，经过一段时间后，Leader宕机

- 在Leader异常或者网络分区的情况下，只要没有出现超过一半的节点异常，系统都会开始新一个任期的运行过程，并且往届Leader在节点或者网络恢复后，会变更为Follower节点并从最新一个任期的Leader中进行数据复制，保持与Leader的一致。

3.5 Raft算法重要概念 -- 任期

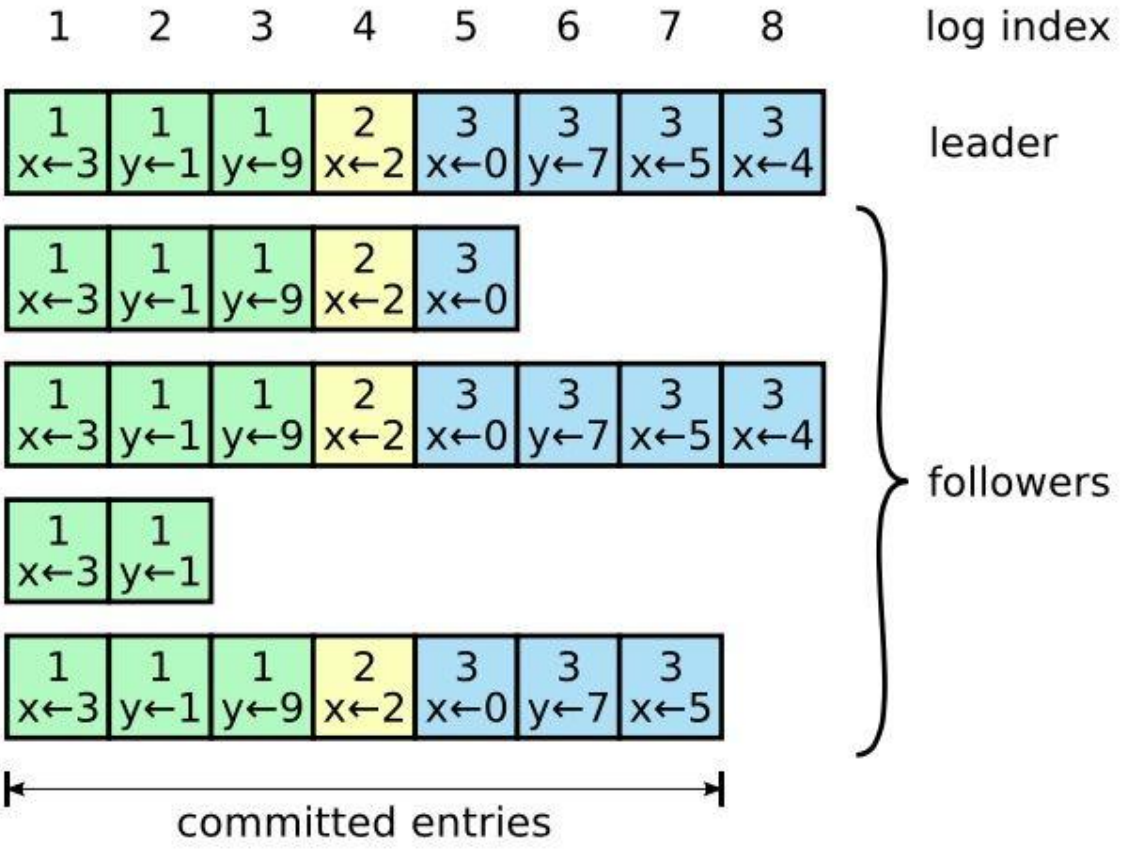
3. 当Leader或Candidate的任期值比别的Follower小时，Leader或Candidate转变为Follower并且任期值递增。

4. 当Follower发现本节点任期值比Leader或Candidate小时，更新本节点的任期值为Leader或Candidate的任期值。

3.6 Raft算法重要概念 -- 日志（数据）

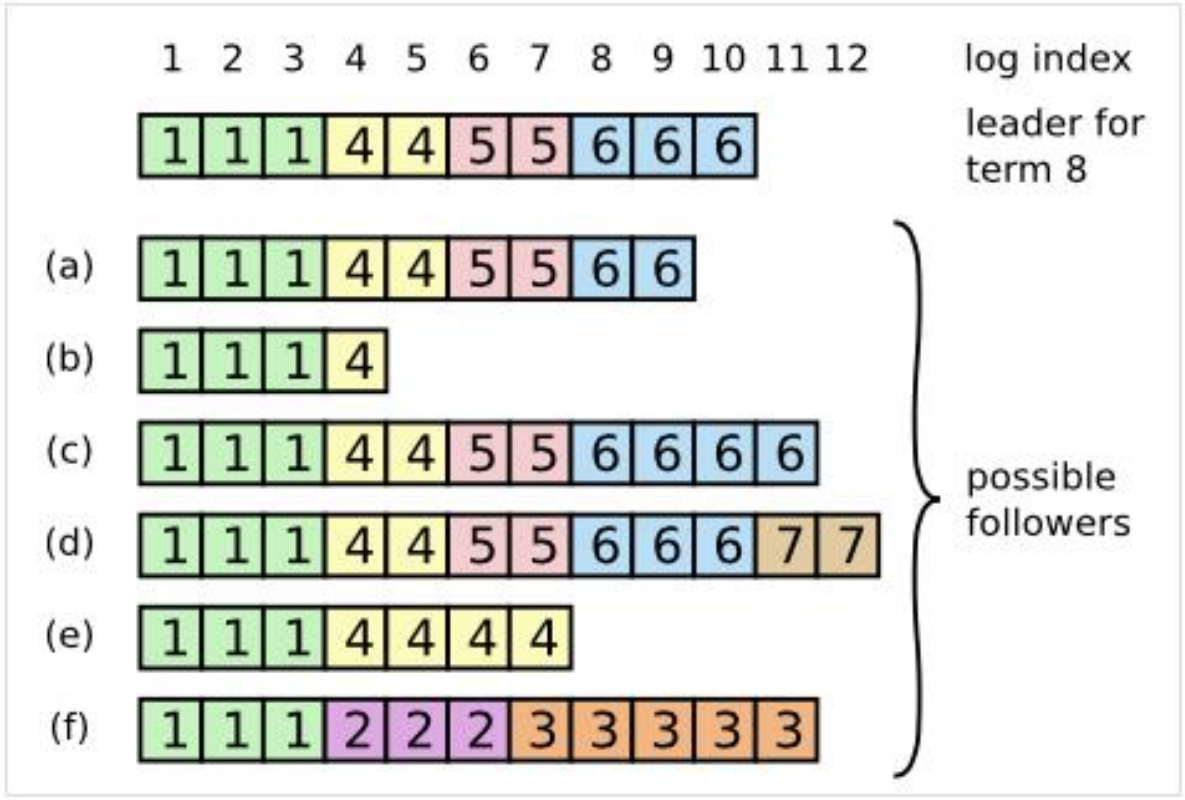
- 日志由有序编号（log index）的日志条目组成。每个日志条目包含它被创建时的任期号（term），和日志中包含的数据组成，日志包含的数据可以为任何类型，从简单类型到区块链的区块。
- logindex是一个自1开始递增的顺序号，也可以称为日志索引号。
- 每个日志条目可以用[term, index, data]序列对表示，其中term表示任期，index表示索引号，data表示日志数据。
- 当一个日志条目被多半的Follower接受，则认为该日志条目被提交（Commit）了。

3.6 Raft算法重要概念 -- 日志（数据）



日志视图

3.6 Raft算法重要概念 -- 日志（数据）



日志同步的例子

3.6 Raft算法重要概念 -- 日志（数据）

日志同步的过程：

1. leader为每个follower维护一个nextIndex，表明下一个将要发送给follower的log entry。

2. 当leader刚上任时，会把所有的nextIndex设置成其最后一个log entry的index加1，如上图，则是11。

3. 当follower的日志和leader不一致时，一致性检查会失败，那么会把nextIndex减1。

4. 最终nextIndex会是leader和follower相同log entry的index加1，这时候，再发送AppendEntries会成功，并且会把follower的所有之后不一致的日志删除掉。

3.7 Raft算法共识特性

1. 一个节点包含日志条目[term, index, data]，另外的节点中包含日志条目[term’,index’, data’]，当term = term’并且index=index’时，data必然等于data’。

- Raft算法中每个任期仅仅存在一个Leader或者没有，而Leader在每个任期内为每个索引号仅仅创建一个日志条目，那么其他节点的日志条目[term’,index’, data’]，如果term = term’并且index=index’，如果data<>data’，则该日志条目必然来自同一任期的不同Leader，这与算法每个任期至多有一个Leader相矛盾，故应该data = data’。

3.7 Raft算法共识特性

2. 如果节点A包含日志条目[term, index, data]，节点B包含日志条目[term’,index’, data’]，两个日志条目的term = term’并且index=index’，则节点A和节点B的index之前的所有日志条目都是相等的。

- 该特性源于日志复制（AppendEntries消息）的一个简单的一致性检查，当Leader向Follower发送一个日志复制请求时，Leader会把新日志条目及前一个日志条目的索引号和任期都包含在里面。如果Follower没有在它的日志序列中找到索引号和任期都相同的日志，它就会拒绝新的日志条目。然后Leader会再发送前一个日志条目的复制请求，往前逐个尝试，直到成功找到日志条目一致的索引号，以此为复制起点，从前往后逐个复制日志条目，直到最新的日志条目为止。

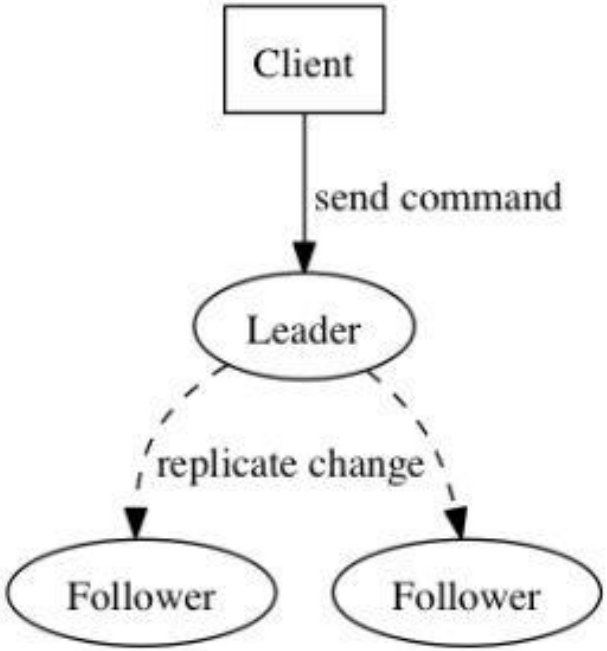
3.8 Raft算法运行过程（以区块链系统为例）

1. 选举Leader

- Raft的选举由定时器（Election Timeout）来触发，每个节点的选举定时器时间都是不一样的，是一个随机数。
- 开始时某个Follower节点的定时器触发选举，然后任期递增，角色由Follower转为Candidate，向其他节点发起请求投票（RequestVote）请求
- 在一个任期每个节点只能投票一次，当投给某个节点后，就不能投给其他节点。
- 当有多个Candidate存在时就会出现每个Candidate发起的选举都存在接收到的投票数都不过半的问题，这时每个Candidate都将任期值递增并重新发起选举。由于每个节点中定时器的时间都是随机的，所以就不会多次存在有多个Candidate同时发起投票的问题。

3.8 Raft算法运行过程（以区块链系统为例）

1. 选举Leader



3.8 Raft算法运行过程（以区块链系统为例）

2. 当节点为Candidate，等待成为Leader时

- 该Candidate接收到过半数节点的投票，从Candidate转为Leader，并向其他节点发送心跳（heartBeat）以使得其他节点知道Leader正常运转。
- 该Candidate节点收到其他Leader节点发送过来的数据复制请求，如果该Leader节点的任期值大则当前节点转为Follower，否则保持Candidate拒绝该请求。
- 该Candidate节点收到其他节点的投票请求，如果其他节点的任期值大，则当前节点转为Follower节点并更新任期值为投票请求节点的任期值并投票给其他节点，否则保持Candidate拒绝该请求。
- 一定时间后仍然未收到超过半数的投票，则任期值递增，重启选举定时器并重新发起选举。

3.8 Raft算法运行过程（以区块链系统为例）

3. 发起选举的情况

- Raft初次启动，不存在Leader，定时器计数结束发起选举。
- Leader宕机或Follower没有接收到Leader的心跳信号，定时器计数结束发起选举。

4. Leader执行交易并形成区块数据

- 选举出Leader后，由Leader接收客户端或其他Follower节点转发的交易，处理后根据系统参数形成区块。
- 作为串行化器实现并发交易的串行化和序列化的目标。

3.8 Raft算法运行过程（以区块链系统为例）

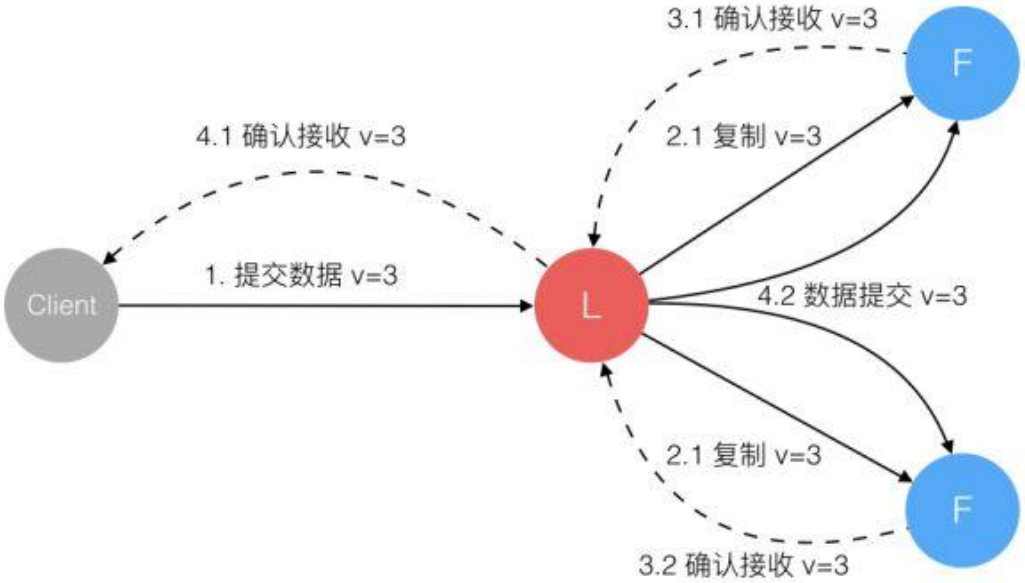
5. 日志复制

- Raft算法中日志复制（Log Replication），主要作用是用于保证节点数据的一致性。Leader节点接收交易请求并按接收的顺序形成区块数据，通过复制使得所有节点按照Leader节点产生的交易序列更新本地数据，从而实现数据一致性的目标。

- 日志复制分为两个阶段：
（1）接受阶段。Leader节点向所有Follower节点发送日志复制请求（AppendEntries），Follower节点接受后，进行检查，没有问题，向Leader返回成功消息；
（2）提交阶段。Leader节点接收到超过半数的Follower成功接受消息后，向所有Follower节点发送提交消息，通知Follower节点数据状态已提交。并返回客户端交易已成功消息。

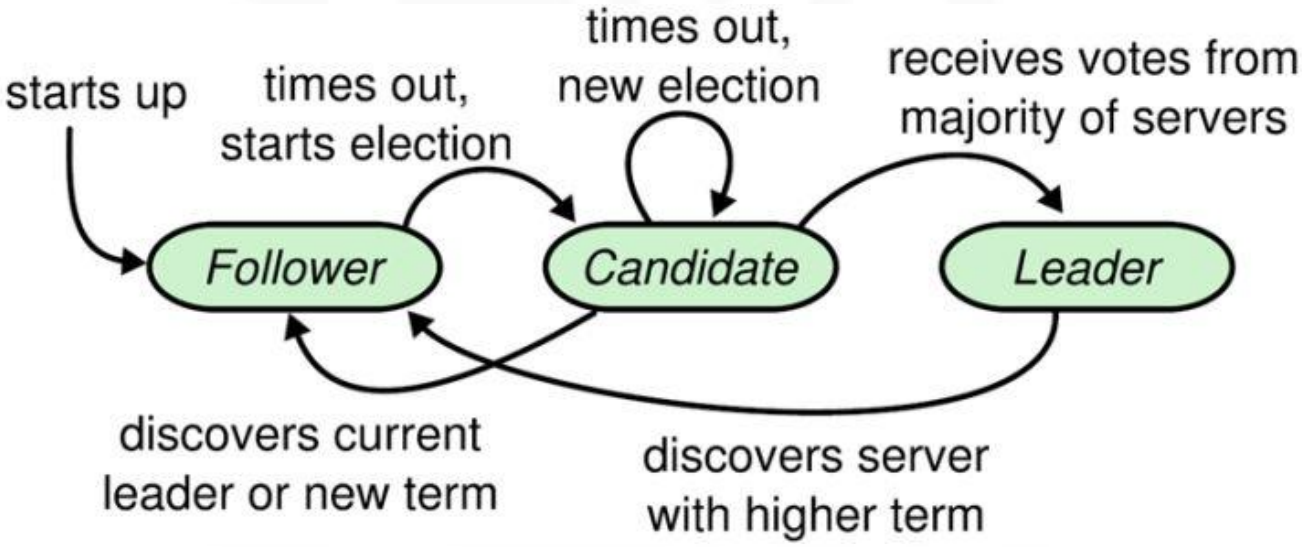
3.8 Raft算法运行过程（以区块链系统为例）

5. 日志复制



3.9 Raft算法描述

Raft算法中每个节点的角色变换，可以设计成一个状态机，每个角色分别表示成节点的一种状态，就形成如图所示的状态转换图。



3.9 Raft算法描述 -- 节点Start

State = Initialized	初始状态为 Initialized，在此期间不接受其他节点的消息
ElectionTimeout = 150s	超时选举时间，超过一定的时间没有收到消息，则重新开始选举
HeartbeatInterval = 50s	Leader 的心跳频率
Init()	读取日志数据及其他相关初始化工作
CurrentTerm =	最后一个日志条目的任期值
CommitIndex=	最新已接受日志条目的索引号
State = Follower	设置状态为 Follower，准备接收其他节点的消息

3.9 Raft算法描述 -- 节点是Follower - FA

日志同步请求。

```
3.      If msg.Type == AppendEntriesRequest then
4.          If msg.Term < CurrentTerm then     说明消息来自于一个已过期 Leader
5.              返回 Failed 以及当前节点的最新任期、最新索引号及提交索引号
6.          End if
7.          更新当前节点的任期并设置 Leader 为 msg 中的 Leader
8.          If  没有在当前日志中找到[msg.PrevLogIndex, msg.PrevLogTerm] then
9.              //说明本节点最新索引号之前的日志条目与 Leader 不一致，需要先进行复制
10.             返回 Failed 以及当前节点最新索引号，及处于提交状态的日志条目的索引号
11.         End if

12.         删除本地日志中 PrevLogIndex 之后的日志条目（如果已经提交则不能删除）
13.         在本节点日志条目最后附加新的日志条目
14.         （如果新的日志条目为空，则仅仅完成后续提交）
15.         更改本地的 CommitIndex 为 Leader 的 CommitIndex 值
16.         （一般后续的消息中会把本次附加的日志条目变更为已提交）
17.     End if
```

3.9 Raft算法描述 -- 节点是Follower - FA

选票请求。

```
18.     If msg.Type == RequestVoteRequest then
19.         If msg.Term < CurrentTerm then
20.             拒绝投给相应节点，并返回 Failed 响应及当前节点的最新任期
21.         End if

22.         If 最新日志条目的索引号 > msg.LastLogIndex
23.             or 最新日志条目的任期 > msg.LastLogTerm then
24.             拒绝投给相应节点，并返回 Failed 响应及当前节点的最新任期
25.         End if

26.         If 当前节点已投票且票没有投给 msg 的发送节点 then
27.             拒绝投给相应节点，并返回 Failed 响应及当前节点的最新任期
28.         End if

29.         记录已投票节点信息并返回投票成功响应
30.     End if
```

3.9 Raft算法描述 -- 节点是Candidate - CA

发送选票请求消息。

```
1.  VotesGranted = 1      候选者支持票数，默认为 1，本节点投票给自己

2.  While State == Candidate do
3.      CurrentTerm = CurrentTerm + 1      任期累加
4.      向网络中其他节点发送 RequestVoteRequest 消息，消息包括任期、最新日志条目[index, term]及
    节点名
```

过半数投票支持。

```
27.     If 得到超半数节点投票支持 then
28.         State = Leader
29.     End if
30. End while
```

3.9 Raft算法描述 -- 节点是Candidate - CA

接收选票响应消息。

```
18.    If msg.Type == RequestVoteRequest then
19.        If msg.Term > CurrentTerm then    说明当前节点的任期已经过期
20.            CurrentTerm = msg.Term        更新本地任期为最新任期
21.            State = Follower              更新状态为 Follower，退出 Candidate 状态
22.            记录已投票节点信息并返回投票成功响应
23.        else
24.            拒绝投给相应节点，并返回 Failed 响应及当前节点的最新任期
25.        End if
26.    End if
```

3.9 Raft算法描述 -- 节点是Candidate - CA

接收到日志同步请求消息。

```
11.    If msg.Type == AppendEntriesRequest then
12.        If msg.Term < CurrentTerm then     说明消息来自于一个已过期 Leader
13.            返回 Failed 以及当前节点的最新任期、最新索引号及提交索引号
14.        End if
15.        State = Follower       节点状态变更为 Follower
16.        按照 Follower 状态对 AppendEntriesRequest 消息进行处理
17.    End if
```

3.9 Raft算法描述 -- 节点是Leader- LA

定时发送心跳消息 - 日志同步消息。

```
3.      For each peer in 节点列表 do
4.          Peer.PrevLogIndex = 本节点最新的日志条目索引号    初始认为其他节点已经和 Leader 同步
5.          Start 心跳线程 do
6.              定时发送 AppendEntriesRequest 消息，消息包括已提交索引号（CommitIndex）、前一个
    日志条目的索引号及任期信息、需同步的日志条目列表（可以一次同步多个）
7.          end
8.      End for
```

客户端的命令。

```
9.      等待接收到消息 msg
10.     If msg.Type == Command then   消息为来自客户端的命令
11.         执行相关的命令
12.         生成新的日志条目并附加到日志序列中（下次心跳会发送给所有的节点）
13.     End if
```

3.9 Raft算法描述 -- 节点是Leader- LA

- 算法需要为每个节点维护一个状态信息，如PrevLogIndex，该值和节点缓存的CommitIndex保持同步。通过这个参数表明PrevLogIndex及其前面的日志条目已经和Leader节点保持一致，那么Leader节点每次给相应节点仅需发送PrevLogIndex到最新日志索引号之间的日志条目。

- 在Raft的Go语言实现中有一个简便的算法来计算CommitIndex的值：
（1）对所有节点的PrevLogIndex按逆序（从大到小）进行排列；
（2）获取1/2位置的PrevLogIndex值；
（3）如果该值大于CommitIndex，该值即为Leader的CommitIndex。

- 算法思路：以1/2位置的PrevLogIndex为基准，前面的1/2个节点的PrevLogIndex比基准值大，说明PrevLogIndex日志条目被一半的节点接受，加上leader即为超过一半的节点已提交。

3.9 Raft算法描述 -- 节点是Leader- LA

日志同步响应消息。

```
14.  If msg.Type == AppendEntriesResponse then
15.      If msg.Fail then                节点复制失败
16.          //失败则分许多情况，分情况处理 1）说明当前 leader 已经是过期 leader
17.          If msg.Term > CurrentTerm then
18.              更新 CurrentTerm 为 msg.Term
19.               State   = Follower              将当前 leader 变更为 Follower
20.               Continue;
21.           End if

22.           //2）说明节点已经提交成功，但是响应消息丢失
23.           If msg.CommitIndex > Peer.PrevLogIndex
24.               更新  Peer.PrevLogIndex 为 msg.CommitIndex
25.               Continue;
26.           End if
              //3）节点日志没有与 Leader 同步
27.           PrevLogIndex 递减，等待心跳线程根据新的 PrevLogIndex 发送 AppendEntriesRequest
28.           Continue;
29.      End if

30.      更新相应节点的 PrevLogIndex 为同步条目的最后一个条目的索引号
31.       If  复制的日志条目的最后一个条目的任期 = CurrentTerm then
32.           节点附加成功
33.       End if
34.
35.       If  超过半数的节点附加成功 then      索引条目被超过半数的节点接受并附加成功
36.           计算并更新 Leader 的 CommitIndex，并在下次心跳时通知节点
37.       End if
38.  End if
```

3.9 Raft算法描述 -- 节点是Leader- LA

日志同步消息。

```
39.     If msg.Type == AppendEntriesRequest then
40.         If msg.Term < CurrentTerm then     说明消息来自于一个已过期 Leader
41.             返回 Failed 以及当前节点的最新任期、最新索引号及提交索引号
42.         End if
43.         State = Follower       节点状态变更为 Follower
44.         按照 Follower 状态对 AppendEntriesRequest 消息进行处理
45.     End if
```

请求选票消息。

```
46.     If msg.Type == RequestVoteRequest then
47.         If msg.Term > CurrentTerm then    说明当前节点的任期已经过期
48.             CurrentTerm = msg.Term        更新本地任期为最新任期
49.             State = Follower              更新状态为 Follower
50.             记录已投票节点信息并返回投票成功响应
51.         else
52.             拒绝投给相应节点，并返回 Failed 响应及当前节点的最新任期
53.         End if
54.     End if
```

3.10 Raft算法分析

1. 运行前提

- 不能超过一半的节点崩溃，超过半数节点无响应，则导致无法选举出新的Leader（CA-23行），也无法更新日志条目的CommitIndex（LA-30）。

2. 解决节点宕机问题

- Leader或Candidate节点宕机：Leader节点宕机后，在超时选举时间后，某个Follower节点会自动变更为Candidate节点，选举出新的Leader节点；

- Follower节点宕机：只要宕机的节点数不超过半数节点，对算法的运行没有影响。

3.11 Raft算法共识分析

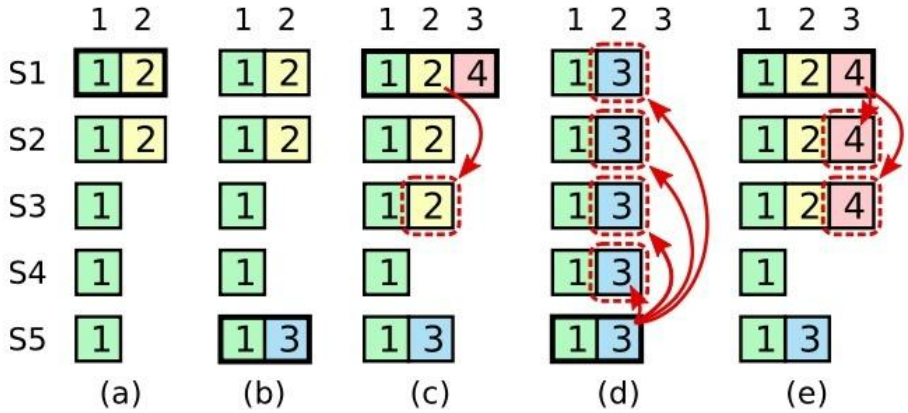
Raft算法中提出安全性的概念，安全性是Raft算法保证每个节点的日志数据一致性的安全机制，主要包括如下的规则：

（1）**选举权规则**。拥有较新日志条目的Follower才有资格成为Leader。所谓较新的日志条目是指日志条目的[term, index]与一半节点的最后一个日志的条目[term1, index1]比较，term >= term1并且index>=index1。满足该条件的节点才可能成为leader，否则拒绝给相应节点投票。[Commit index]

（2）**Leader日志完整性规则（Leader Completeness Property）**。仅Leader当前任期的日志条目被提交，才能更改日志条目的状态为提交状态。

3.11 Raft算法共识分析

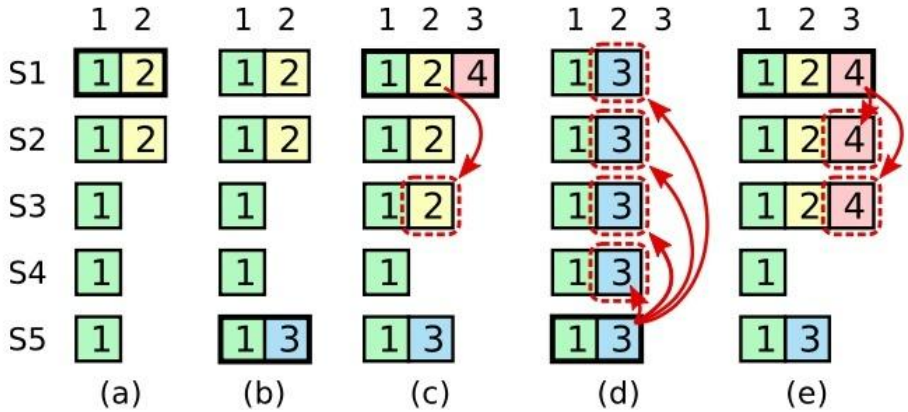
- 根据选举权规则，在某些场景下会出现已提交的日志又被覆盖的情况。



- 假设网络包含S1到S5共5个节点，每次复制一个日志条目。
- 在阶段a，term为2，S1是Leader，且S1写入日志条目[term, index]为[2, 2]，并且日志被同步写入了S2。
- 在阶段b，S1离线，触发一次新的选主，假设此时S5被选为新的Leader，此时系统term为3，且写入了日志[term, index]为[3，2]。

3.11 Raft算法共识分析

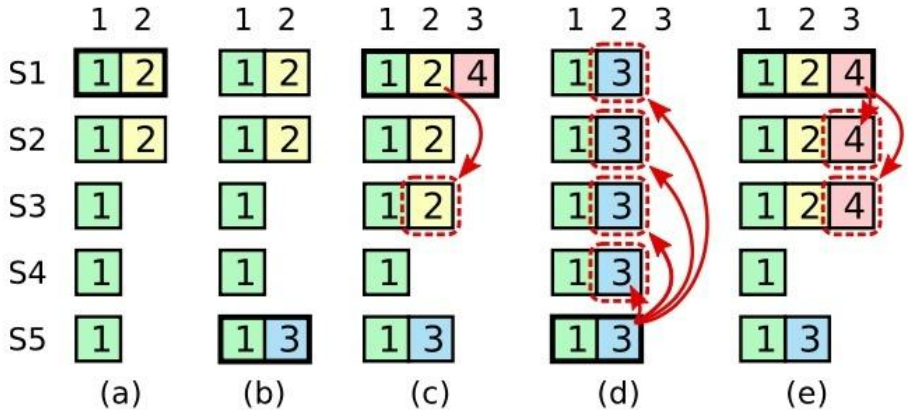
- 在阶段c，S5离线，进而触发了一次新的选主，S1重新变成Leader，此时系统term为4，S1会将自己的日志条目[2，2]同步到了S3，此时该日志条目已经被同步到了多数节点（S1，S2，S3）。



- 假设认为日志条目复制到多数节点即认为已提交，则可能出现以下情况：

3.11 Raft算法共识分析

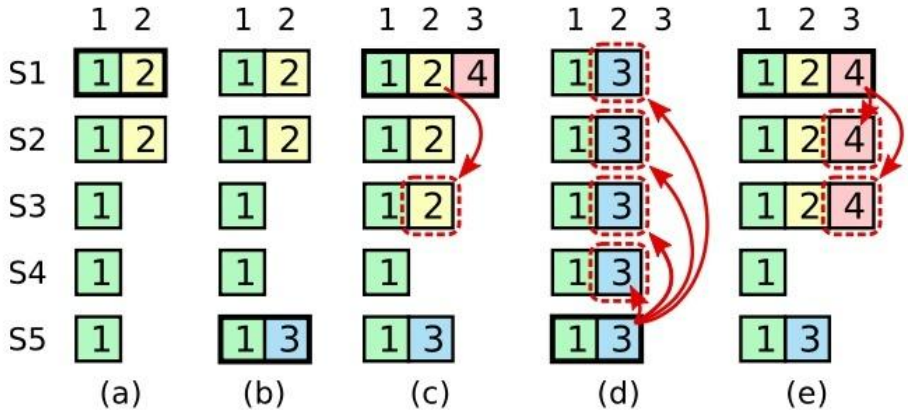
- 如果在同步日志条目[4, 3]之后S1离线，由于[4, 3]条目已经复制到S1、S2、S3节点，S5的任期为3，根据选举规则，S5不会成为Leader，Leader只能在S2、S3中间产生。



- 如果在同步日志条目[4, 3]之前S1离线，由于S5的日志条目为[3, 2]，比S2、S3、S4的任期新，根据选举规则，S5可以被选为Leader，然后S5会将自己的日志更新到Followers，于是S2、S3中已经被提交的日志[2, 2]被截断了，出现阶段d的情况。在日志条目被多数接受即认为提交的原则下，就会出现阶段d的情况，被提交的日志条目[2, 2]被[3, 2]覆盖了，为解决这个问题，Raft算法增加了一个完整性规则。

3.11 Raft算法共识分析

为了避免这种致命错误，需要对协议进行一个微调：只允许主节点提交包含当前term的日志。



针对上述情况就是：即使日志（2，2）已经被大多数节点（S1、S2、S3）确认了，但是它不能被Commit，因为它是来自之前term(2)的日志，直到S1在当前term（4）产生的日志（4， 3）被大多数Follower确认，S1方可Commit（4，3）这条日志，当然，根据Raft定义，（4，3）之前的所有日志也会被Commit。此时即使S1再下线，重新选主时S5不可能成为Leader，因为它没有包含大多数节点已经拥有的日志（4，3）。

3.12 Raft算法与Paxos比较

- Raft算法是Paxos算法简化版，运行环境相似，算法本身有许多相似的地方，也有一些Raft算法特殊的地方。

- Raft与Paxos相似的概念：

Raft	Paxos
任期（Term）	票（Ticket）
领导者（Leader）	提案者（Proposer）
选举阶段（RequestVote）	准备阶段（Prepare）
复制阶段（AppendEntries）	接受阶段（Accept）

3.12 Raft算法与Paxos比较

Raft与Paxos的不同：

	Raft	Paxos
领导者	某个任期唯一Leader	一个票一个提案者，但可以支持其他人的提案或者提出新提案
领导者选举权	具有最新的日志条目	任意
瓶颈	Leader会成为瓶颈	多提案者，没有瓶颈

04 POW系列算法

4.1 POW系列算法概述

- POW（Proof of Work）共识算法主要是各个节点通过付出一定的计算资源**求解Hash难题**来竞争创建新区块的权利。Hash难题的求解需要消耗一定计算资源和时间，从而保证一段时间内只有求解出Hash难题的少量节点获得向区块链系统提交新区块的权利即合法提案。

- 少量的提案在全网广播后，其他节点接收并验证，验证通过则附加在最长的链上。由于同时有多个提案存在，系统不可避免的存在**分叉（fork）**，但最终所有节点都会**接受链长最长的**一个链条，从而保证系统的一致性。

4.1 POW系列算法概述

- 最长的链是合法的区块链这一特性，使得**攻击区块链网络变得困难**，一个节点要篡改一个区块数据只能重做包含这个交易的区块，以及这个区块之后的所有的区块，创建一个比目前诚实区块链更长的区块链。只有网络中的大多数节点都转向攻击者创建的区块链，攻击者的攻击才算成功了。

- 中本聪证明，六个区块后攻击者追赶上最长链的可能性降低到0.0002428%，再过四个或更多区块后这个可能行会降到0.0000012%，每新增一个区块，攻击的可能性就会以指数形式下降。在实际中，比特币交易会在**六个区块后被确认**，因为在这种情况下，攻击者追赶上的可能性已经非常低了，可以认为这个交易是有效的，不再会被修改。

4.1 POW系列算法概述

- 一旦攻击者达到全网算力1/3，区块链网络就存在被破坏的风险了。当达到全网算力的1/2，从概率上，攻击者就能掌控整个网络了，但是要实现这么大的算力，攻击者将需要付出巨大的经济成本。

- POW的关键理念是通过**按某种资源的比例来选择节点**，从而最终达到近似随机节点选择，消除中心化和垄断的目的。POW的资源是计算能力，根据资源的不同形式形成不同的变种，如POS和DPOS。POS的资源是节点持有的货币所有权，而DPOS的资源则是节点持有的股权。

- POW算法**假设网络中的多数节点是诚实节点**，节点依据持有的资源证明向网络提交候选区块，候选区块被多数节点接受（即链长最长，CPU投票），则记入账本，否则区块会存在分叉中。

4.2 Hash难题

- POW中的Hash难题是以区块中的交易清单、上一区块的Hash值以及一个随机数（nonce）为整体计算一个Hash值，这个Hash值必须小于一个特定的目标值（target）。找出满足下列不等式的随机数，即说明解决Hash难题。不等式如下：
  H(nonce || prev_hash || tx || tx || ... || tx) < target

- 不等式中前一区块Hash值（prev_hash）、交易列表都是固定的，目标值（target）则根据难度值进行调整，nonce则是随机变化的。Hash难题的求解过程就是找到一个nonce，该nonce与prev_hash和交易列表一起求hash值，当hash值小于target时，nonce值就求解出来。从而证明节点消耗了一定的计算资源。

4.2 Hash难题

每个节点**独立进行**Hash难题的计算，直到某个幸运节点找到这个满足不等式的随机数，那么这个节点被选中并发布区块。没有人知道哪个节点会被选中，完全通过Hash难题的求解达到随机选择节点的目标。

根据系统设计需要，这个Hash函数应该满足以下特性：

（1）**难以计算（Difficult to compute）**：区块之间有一定的时间间隔。

（2）**参数化成本（Parameterizable cost）**：间隔时间固定。

（3）**简单验证（Trivial to verify）**：其他节点快速验证

4.2 Hash难题 -- 难以计算

- 通常用难度值来衡量这一难易程度。随着时间的变化和全网计算能力的提升而变化，难度值决定了节点大约需要经过多少次哈希运算才能产生一个合法的区块。
- 产生一个区块会获得一定的奖励，因此有些节点愿意通过提升计算能力来竞争创建新区块的权利。
- 反复试图解决Hash难题的过程称之为**挖矿**，参与挖矿的节点称之为矿工。即使从技术上讲任何节点都可以是一个矿工，但是由于技术的发展，计算能力的提升，甚至采用专用挖矿的硬件，导致采矿成本过高，也导致区块链网络中存在一些日趋集中的算力中心。

4.2 Hash难题 -- 参数化成本

- 当采用POW共识算法时，不同的区块链系统根据不同的性能要求，会给出一个产生一个区块的平均时间。这个时间尽管不同的系统不同，但同一个系统往往是固定的。

- 实现的方法是算法中的所有节点能够自动计算目标值。目标值的大小决定了Hash目标空间的大小，也决定了Hash值命中目标空间的难易程度即难度值。

- 以比特币网络为例，每产生2016个区块，算法会重新计算目标值，从而使得连续产生的区块之间的间隔时间大约为10分钟。2016个区块计算需要2周时间，也就是说比特币每隔2周对目标值计算一次。

4.2 Hash难题 -- 简单验证

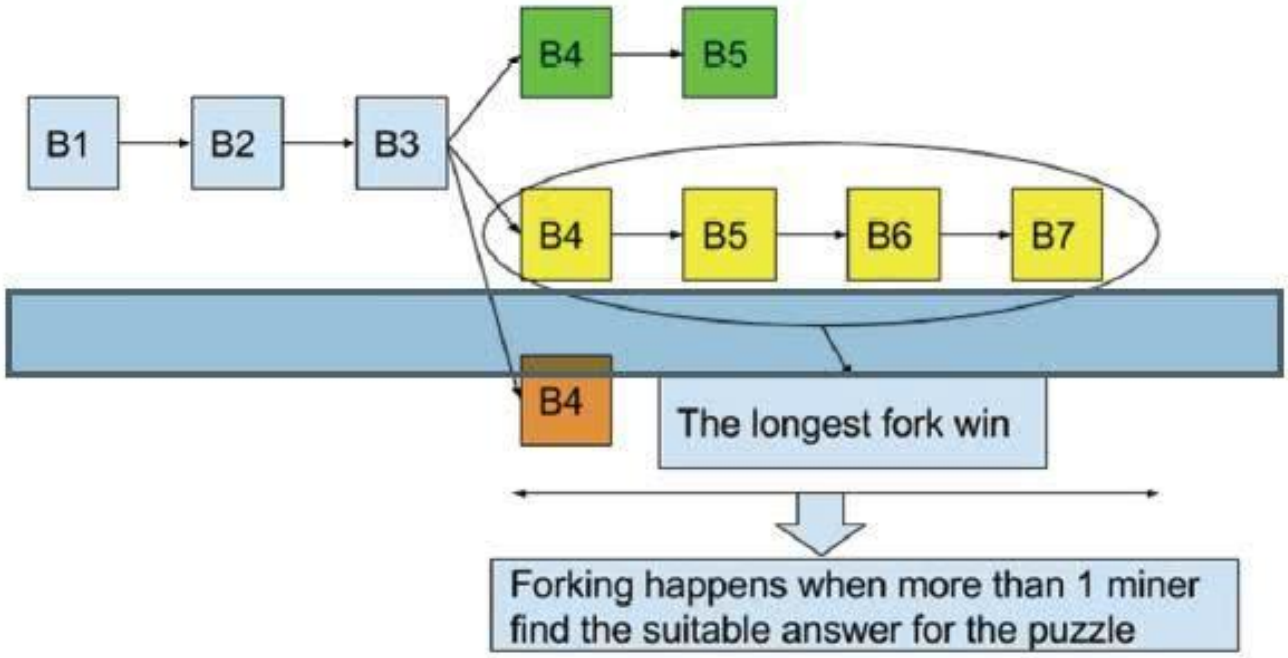
- 当新区快打包时，会把nonce值及难度值打包进区块头部，其他节点根据区块头部中的难度值计算目标值，然后判断是否满足Hash难题的不等式。

- 这个计算过程所消耗的资源是微不足道。这一特性使得任何节点或矿工都可以迅速验证其他节点或矿工产生的区块是否满足工作量证明属性即是否正确求解nonce值。

4.3 挖矿

- 在公有链中，当用户向网络中发布交易后，需要有节点将交易进行确认，形成区块并提交到区块链结构中。因此在比特币网络中首次引入了“挖矿”的概念。在比特币网络中运行POW共识算法的节点称为矿工。找随机数nonce的过程称为挖矿。
- 如果一个矿工解决了某个时间段的难题，表示该矿工挖到一个区块。有可能在极短的时间内，不同的矿工都挖出新的区块，矿工会在各自的分叉链上挖矿，直到其中一条链的长度超过其他链的长度。一般超过其他链6个区块的长度，其他链基本不再可能超越最长的链。
- 最长链就是区块链网络的共识链，共识链中的交易是有效的交易。

4.3 挖矿


B1
B2
B3
B4
B5
B4
B5
B6
B7
B4
The longest fork win
Forking happens when more than 1 miner find the suitable answer for the puzzle

4.3 挖矿

- 当挖出一个新的区块，区块链系统会奖励矿工一定币值的金钱，这笔收益足够大，足以刺激矿工进行挖掘。

- If 挖矿收益 > 挖矿成本 then 矿工盈利。

  - 挖矿收益包括（1）挖出新区块的奖励。找到新区块的速率越大，获得的奖励越多，这个速率依赖于本地算力和全网算力的比例，比例越大，找到新区块的速率也越大。（2）打包交易提供的交易费。矿工在打包交易时会选择交易费多的交易打包。

  - 挖矿成本包括硬件成本和运行成本（电力、降温等），此外还有沉没成本，即竞争失败的矿工所挖区块的成本。沉没成本在比特币中完全不考虑，而在以太坊中当分叉的区块包含在规范区块链的叔父中时会给予一定的补偿。

4.4 POW变种

- POW算法造成能量浪费和安全问题。
  - 大量的矿工通过硬件的升级来扩大自己挖到区块的概率。
  - 巨大的算力除了用于保护比特币网络安全外，并没有对社会产生其他作用。
  - ASIC矿池的运用导致中心化问题。

4.4 POW变种

- 以太坊使用了一种称为Ethash的共识算法。Ethash算法的设计使得挖矿过程更多的依赖于内存和带宽，以此来弱化具有大算力矿工的优势。Ethash算法每猜测一次随机数，都需要循环的从内存中的DAG随机地抽取一段数据用于计算。因此，挖矿依赖于内存和带宽。

- 2015年John Tromp引入了另外一个POW算法。算法采用了Cuckoo哈希函数来代替PoW算法，Cuckoo哈希函数允许矿工付出更少的努力，且赋予矿工更容易挖到区块的权利。

- King提出了一种思想：利用网络中的算力来寻找最长的素数链。找Cunningham链的PoW算法除了将算力用于挖矿之中，还产生了一些有助于数学研究的素数。

4.5 双花攻击

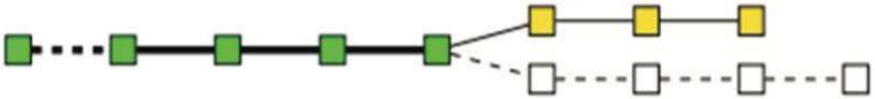
- POW的缺点之一就是一个双重花费攻击问题。简而言之，攻击者试图通过分叉并在其中一条分叉链上发起一笔交易（Tx2），以达到撤销一笔刚刚上链（另外一条分叉链）的交易（Tx1）。接着，攻击者将会努力使得有交易Tx2的分叉链长度超过另外一条分叉链的长度（即，交易Tx1无效）。

- 为了达到这样的目的，攻击者需要拥有全网51%的算力。因此，双重花费攻击也被称为51%攻击。对于单个用户来说，拥有全网51%的算力是件极其困难的事情。

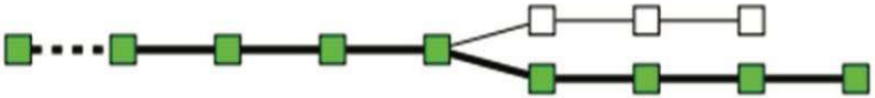
4.5 双花攻击

(a) Initial state of the blockchain in which all transactions are considered as valid.

(b) Honest nodes continue extending the valid chain by putting yellow blocks, while the attacker secretly starts mining a fraudulent branch.



(c) The attacker succeeds in making the fraudulent branch longer than the honest one.



(d) The attacker's branch is published and is now considered the valid one.

4.5 双花攻击

- 随着所谓的矿池出现，攻击者们容易发起51%攻击。
  - 在矿池挖矿的过程中，不是单个矿工试图猜测一个Hash难题的随机数nonce，而是所有的矿工聚集在一起，对Hash难题进行拆分，每一个矿工完成其中一小部分任务。
  - 解决一个Hash难题的工作变得容易。
  - 当矿池挖到一个区块时，每个矿工都将获得一定比例（相同或者根据矿工完成的任务难易比例）的奖励。
  - 在规模经济的影响下，大量的矿工不断地加入矿池。

4.5 双花攻击

- 为了阻止大型矿池的产生，Miller等人提出一种新的机制：将POW的Hash难题设计成不可外包（non-outsourceable puzzles）。

- **不可外包Hash难题的大致工作原理是每猜测一次随机数，都需要外包者的私钥对区块信息签名一次。**如果外包者想加快解决Hash难题的速度，则会把Hash难题外包给其他矿工一起合作挖矿。其他矿工每猜测一次随机数，都需要外包者的私钥对消息进行签名，因此，**外包者必然将自己的私钥告诉他们**。

- 一旦外包者将自己的私钥告诉跟他有合作的矿工，那么其他任何跟他有合作的矿工（都有外包者的私钥）都能轻松获取整个区块的报酬。对于外包者来说，并不愿意其他矿工能够获得该区块所有的报酬。因此这种不可外包的Hash难题能够有效的阻止矿工们合作起来，形成大型矿池进行挖矿。

05 PBFT算法

5.1 PBFT算法概述

- PBFT是Practical Byzantine Fault Tolerance的缩写，意为实用拜占庭容错算法。该算法是Miguel Castro（卡斯特罗)和Barbara Liskov（利斯科夫）在1999年提出来的，**解决了原始拜占庭容错算法效率不高的问题，将算法复杂度由指数级降低到多项式级，使得拜占庭容错算法在实际系统应用中变得可行**。

- PBFT算法是基于状态机复制（State Machine Replication）理念设计的一个实用的并能解决拜占庭容错的算法，当存在f个失效节点时必须保证存在至少3f+1个副本，才能保证在异步分布式系统中达成共识，并且算法满足安全性和活性要求，即**提供了(n-1)/3的容错性**。

5.1 PBFT算法概述

- 采用PBFT算法的区块链系统中，每个节点维护服务在本节点的状态（区块链）及相关的服务操作（交易），这里的服务被称为在系统中不同节点之间复制（消息传输）的状态机，包含状态和操作。

- 整个系统共同维护一个状态，并在不同节点之间复制，所有的节点采取一致的行动，从而满足分布式系统的一致性要求。状态机在每个节点的复制体称为副本，所有副本的集合用R表示，每个副本用R_i表示，i是0到|R|-1的整数。

- 同所有的状态机复制技术一样，PBFT对每个副本节点提出了两个限定条件：
（1）所有节点必须是确定性的。也就是说，在给定状态和参数相同的情况下，操作执行的结果必须相同；
（2）所有节点必须从相同的状态开始执行。

5.2 重要概念

- 视图（View）- 一次成功的主节点变换
  - 是一次成功的节点配置轮换过程，在一个视图中会根据算法确认一个副本作为主节点（primary），其他副本作为备份（backups）。
  - 视图编号是连续的。
  - 与RAFT算法竞争Leader节点不同，PBFT算法中主节点由公式 p = v mod |R|计算得到，这里v是视图编号，p是副本编号，|R|是副本集合的个数。
  - 当主节点失效的时候就需要启动视图更换（view change）过程。

5.2 重要概念

- 主节点（primary）
  - 主要用于分配基于本视图的消息序列号（全局编号），从而对分布式系统的消息进行全局排序
  - 进而所有副本按照视图编号、序列号的顺序处理消息，变更状态，使得所有副本的状态保持一致。

5.2 重要概念

- 消息（message）
  - 使用加密和签名技术来防止欺骗攻击和重放攻击，以及检测被破坏的信息。
  - 但对消息本身的错误、消息的延迟或者不响应则通过共识算法来防止。
  - 因此算法中消息包含了公钥签名、消息摘要（message digest）等安全信息及检测消息重复的消息验证编码（MAC）。
  - 算法中使用m表示消息，m_i表示由节点i签名的消息，D(m)表示消息m的摘要。

5.3 客户端（消息发起者）

- 客户端c向主节点发送操作请求<REQUEST, o, t, c>，请求执行状态机操作o。
  - 其中t是时间戳，用来保证客户端请求只会执行一次。
  - 客户端c发出请求的时间戳是全序排列的，不可逆的，后续发出的请求比以前发出的请求拥有更高的时间戳。
  - 对于客户端，请求发起的时间戳可以是本地时钟值。

5.3 客户端（消息发起者）

- 副本发给客户端的响应为<REPLY, v, t, c, i, r>，其中v是视图编号，t是时间戳，i是副本节点的编号，r是请求执行的结果。
- 每个由副本节点发给客户端的消息都包含了当前的视图编号，使得客户端能够跟踪视图编号，从而进一步推算出当前主节点的编号。
- 客户端通过点对点消息向它自己认为的主节点发送请求，然后主节点自动将该请求向所有备份节点进行广播。

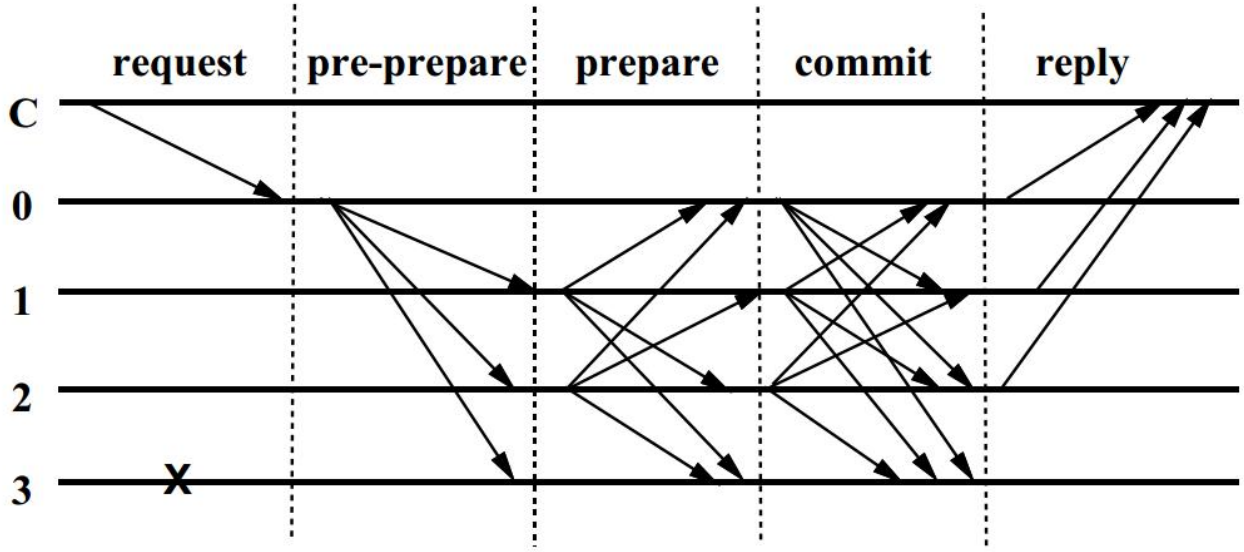
5.3 客户端（消息发起者）

- 客户端等待从不同副本节点得到的响应，当有f+1个响应相同时，客户端把r作为正确的执行结果。
- 所谓的响应相同是指：签名正确；具有同样的时间戳t；执行结果r相同。
- 因为失效的副本节点不超过f个，所以f+1个副本的一致响应必定能够保证结果是正确有效的。
- 如果客户端在有限时间内没有收到f+1个相同的响应，客户端重新将该请求向所有副本节点进行广播。
- 这时如果副本节点已经处理过该请求，则直接把上次发送的响应重新发送给客户端。如果没有处理过该请求，则副本节点进行处理请求的同时，会将该请求转发给主节点。

5.4 三阶段提交协议

- 主节点接收到来自客户端的请求后，按照三阶段协议向全网广播该消息。主要包括预准备（pre-prepare）、准备（prepare）和确认（commit）三个阶段。
  - 预准备和准备两个阶段用来确保同一个视图中请求发送的时序性。
  - 准备和确认两个阶段用来确保在不同的视图之间的已接受的请求命令是严格排序的。

5.4 三阶段提交协议



副本0是主节点，副本3是失效节点，C是客户端

5.5 预准备阶段

- 在预准备阶段，主节点基于当前视图v分配一个序列号n给收到的客户端请求，然后向所有备份节点群发送预准备消息。
- 预准备消息的格式为<〈PRE-PREPARE，v，n，d〉，m〉，这里v是视图编号，m是客户端发送的请求消息，d是请求消息m的摘要。
- 客户端请求本身是不包含在预准备的消息里面的，其目的有两个：
  - 每个由副本节点发给客户端的消息都包含了当前的视图编号，使得客户端能够跟踪视图编号，从而进一步推算出当前主节点的编号。
  - 客户端通过点对点消息向它自己认为的主节点发送请求，然后主节点自动将该请求向所有备份节点进行广播。

5.5 预准备阶段

- 备份节点接收到预准备消息后进行验证和检查，验证和检查的内容包括：
  - 请求和预准备消息的签名正确，并且d与m的摘要一致；
  - 检查预准备消息中的视图编号v是否为当前视图编号；
  - 节点是否接收过视图编号为v，序号为n，但消息摘要与d不同的消息（<v，n>相同的消息，消息必须相同）；
  - 消息的序号n必须在水线（watermark）上下限h和H之间。h和H的含义在后续检查点协议中会详细说明。
- 一旦某个备份节点i接受了预准备消息< <PRE-PREPARE，v，n，d>，m>，则协议进入准备阶段。

5.6 准备阶段

- 节点i向所有副本节点发送准备消息<PREPARE, v, n, d, i>，并将预准备消息和准备消息在节点i进行保存。

- 当节点i收到来自其他节点的准备消息，当满足下一页的条件时，副本节点i将（m, v, n, i）保存到本地并标记为Prepared状态，记为**prepared(m, v, n, i)**。其中m是预准备消息中的消息m，v是预准备消息中的视图编号，n是预准备消息中的序号。

- 当变更为prepared(m, v, n, i)后，节点i向其他副本节点广播格式为<COMMIT, v, n, D(m), i>的确认消息，于是协议进入确认阶段。

5.6 准备阶段

- 变更为prepared(m, v, n, i)的条件：
  - 接收到超过2f个节点的准备消息；
  - 接收到的2f个与预准备消息一致的准备消息，一致的检查内容包括：视图编号v，消息序号n，摘要d。

5.7 确认阶段

- 每个副本节点接受确认消息并写入消息日志，具体条件为：
  - 签名正确；
  - 消息的视图编号与节点的当前视图编号一致；
  - 消息的序号n满足水线条件，在h和H之间。

5.7 确认阶段

- 当节点i接受到2f+1个m的确认消息后并满足相应条件后，消息m变更为committed-local状态，记为committed-local(m, v, n, i)。具体条件为：
  - 节点i的prepared(m, v, n, i)为真；
  - 节点i已经接受了2f+1个确认消息（包括自身在内）与预准备消息一致。一致的检查内容包括：视图编号v，消息序号n，摘要d。

5.7 确认阶段

- 当消息m在节点i的状态为committed-local(m, v, n, i)时，节点执行m的请求，节点i的状态能够确保所有编号小于n的请求依次顺序执行。
- 在完成m请求的操作之后，每个副本节点都向客户端发送回复。
- 副本节点会把时间戳比已回复时间戳更小的请求丢弃，以保证请求只会被执行一次。
- 当节点i的消息m变更为committed-local(m, v, n, i)时，对于分布式系统来说，消息m变更为committed(m, v, n)，也就意味着任意f+1个正常副本节点集合中的所有副本节点i其prepared(m,v,n,i)为真。这就确保了所有正常节点以同样的顺序执行所有请求，确保了节点状态进行复制并保证了算法的正确性。

5.8 检查点协议

- 为了节省存储，系统需要一种将日志中的无异议消息记录删除的机制。
- 算法设置周期性的检查点协议,将系统中的服务器同步到某一个相同的状态，同时可以定期地处理日志、节约资源并及时纠正服务器状态。
  - 在PBFT算法中，会保存所有接收到的消息并记录到日志中，如果日志不能及时清理，会导致系统资源被大量的日志占用并影响系统的性能和可用性。
  - 另一方面由于拜占庭节点的存在，算法不能保证每个节点执行了相同序列的请求，即所有节点状态可能不一致。

5.8 检查点协议

- 执行过程：
  - 当副本节点i接收到的请求消息的序号可以被某个常数（如100）整除时会周期性向其他节点广播检查点消息<CHECKPOINT, n, d, i>，这里n是最近一个影响状态的请求序号，d是状态的摘要。
  - 每个副本节点都默默地在各自的日志中收集并记录其他节点发过来的检查点消息，直到收到来自2f+1个不同副本节点的具有相同序号n和摘要d的检查点消息。
  - 这2f+1个消息就是这个检查点的正确性证明。
- 算法将这些请求执行后得到的状态称作**检查点（checkpoint）**，并且将具有证明的检查点称作**稳定检查点（stable checkpoint）**。每个副本节点保存了服务状态的多个逻辑版本，包括最新的稳定检查点，零个或者多个非稳定的检查点，以及一个当前状态。

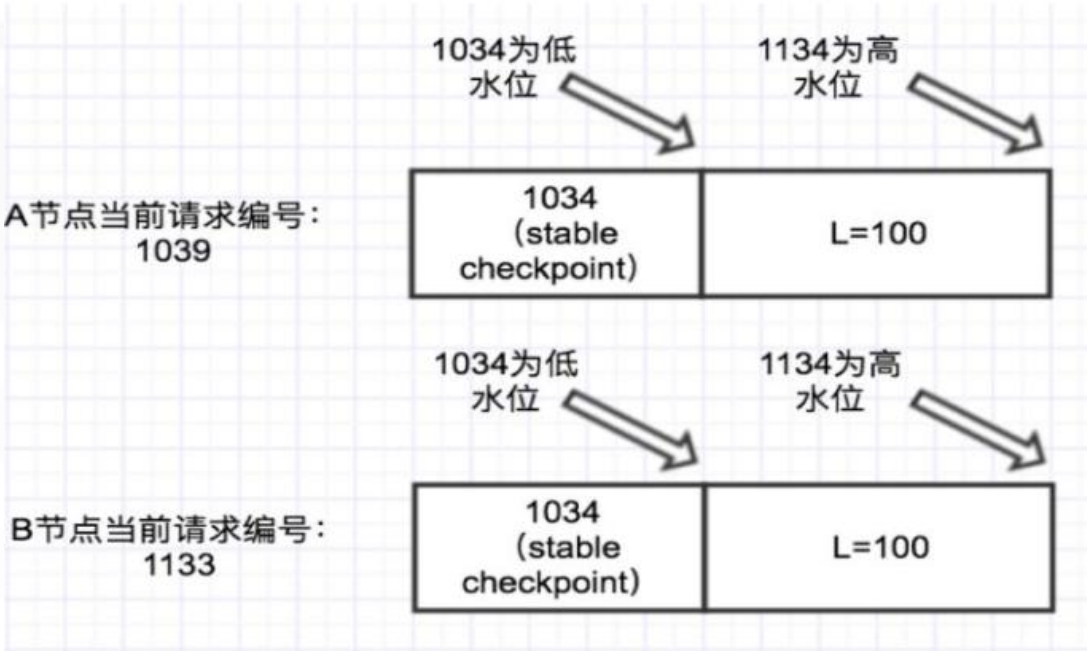
5.8 检查点协议

- 垃圾回收：
  - 节点确认某个检查点为稳定检查点后，然后副本节点就可以将所有序号小于等于n的预准备、准备和确认消息从日志中删除。
  - 同时也可以将之前的检查点和检查点消息一并删除。
- 更新水线（watermark）的高低值（h和H）：
  - 这两个高低值限定了可以被接受的消息。
  - 水线的低值h与最近稳定检查点的序列号相同，而水线的高值H=h+k，k需要足够大才能使副本不至于为了等待稳定检查点而停顿。

    假如检查点每100个请求产生一次，k的取值可以是200。

5.9 水线（watermark）

图中A节点的当前请求编号是1039，即checkpoint为1039；B节点的 checkpoint 为1133。当前系统 stable checkpoint 为1034 。那么1034这个编号就是低水位，而高水位 H=低水位 h+L ，其中L是可以设定的数值。因此图中系统的高水位为1034+100=1134。



5.10 视图更换协议

- 在PBFT算法中主节点承担消息序号分配、请求转发等核心功能，因此一旦主节点发生错误，就导致系统无法正常运行。
- 视图更换协议确保主节点失效时算法的活性。
- 视图更换的触发可以通过备份节点中一个请求的超时执行触发。
- 一个节点i的超时产生时，会启动视图更换，并将视图编号v变更为v+1，同时不再接受除检查点消息（checkpoint）、视图更换消息(view-change)和新视图消息（new-view）外的其他消息请求。

5.10 视图更换协议

- 节点i向其他副本节点广播视图更换消息<VIEW-CHANGE, v+1, n, C, P, i>。
  - 其中n是节点i的最新稳定检查点s的序号，
  - C是证明s是稳定检查点的2f+1个检查点消息，
  - P是所有序号大于n的所有prepared(m, v, n, i)为真的消息P_m，包括请求消息m的有效的预准备消息和与预准备消息一致的2f个准备消息。

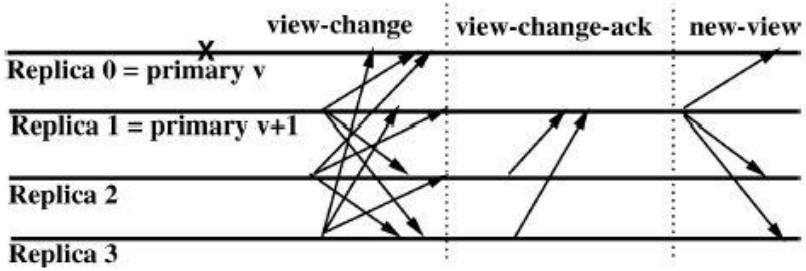


Fig. 2. View-change protocol: the primary for view v (replica 0) fails causing a view change to view $v+1$.

5.10 视图更换协议

- 通过公式p = v mod |R|计算得到主节点p，当主节点p收到2f个来自其他复制节点的有效的视图更换消息后，节点p向其他复制节点广播新视图消息<NEW-VIEW, v + 1, V, Q>
  - V包含主节点收到的2f+1个有效的视图更换消息（包括主节点p本身发送的视图更换消息）
  - Q包含有效的预准备消息（不包括请求消息），其范围通过以下算法获得：从集合V中获取最小的稳定检查点序号min_s和最大的稳定检查点max_s，主节点p为min_s到max_s中间的每个序号n在新的视图v+1中创建新的预准备消息。

5.10 视图更换协议

- 这时序号n存在两种情况：min-s和max-s之间，是否存在P消息集合
  - 至少存在一个V中的视图变更消息的集合P中包含序号n，这说明存在一个预准备消息m，则主节点向其他备份节点广播新的预准备消息<PRE-PREPARE, v+1, n, d>；
  - V中所有的视图变更消息的集合P中都不包含序号n，则主节点向其他备份节点广播预准备消息<PRE-PREPARE, v+1, n, d^{null}>，d^{null}是对null消息的签名，null消息执行空动作。

5.10 视图更换协议

- 主节点以min_s为最新的稳定检查点，并更新水线的h为min_s。
- 备份节点接收到新视图消息后，主要完成以下工作：
  - 采用主节点相同的算法校验新视图消息中的Q，校验通过后将Q中的预准备消息写入本节点的日志；
  - 为Q中的每个预准备消息创建准备消息并向其他复制节点进行广播并将准备消息写入本节点日志；
  - 更新视图编号为v+1。

5.10 视图更换协议

- 副本节点会对min_s到max_s中间的消息重新执行三阶段提交协议，在运行阶段，协议会通过本地存储的回复客户端的消息进行校验，已经回复的消息不会重新执行。

- 通过视图更换协议使得即使拜占庭节点称为主节点，也能在视图变换后确保分布式系统的一致性。

5.11 共识问题

- 在PBFT算法中通过以下四个方面确保分布式系统的共识：
  - 由单一主节点对来自多个客户端的请求分配序号n，从而对分布式系统的消息进行全局排序；
  - 任一复制节点i的消息m进入状态prepared(m, v, n, i)，则意味着所有正常节点对同一个视图中的请求序号达成一致，即当任意两个正常节点，v和n相同时，消息m也相同。

5.11 共识问题

- 任意f+1个正常副本节点集合中的所有副本i其prepared(m, v, n, i)为真，意味着消息m已成功提交给分布式系统即committed(m, v, n)。在三阶段提交协议的确认阶段，任一节点i上消息m满足committed-local(m, v, n, i)为真，则committed(m, v, n)，进而节点i可以执行消息并向客户端返回结果，其他节点i也会按照相同的顺序执行消息m，从而实现状态复制。

- 当客户端收到f+1个从不同副本的同样响应时，则意味着f+1个正常节点已成功执行消息。当f+1个副本节点成功执行消息，就确保了所有正常的节点以同样的顺序执行所有请求。

5.12 RAFT和PBFT协议的比较

对比点	Raft	Pbft
适用环境	私链	联盟链
算法通信复杂度	O（n）	O（n^2）
最大故障和容错节点	故障节点：2f+1<=N	容错节点：3f+1<=N
流程对比	1.初始化leader选举（谁快谁当） 2.共识过程 3.重选leader机制	1.初始化leader选举（按编号依次轮流做主节点） 2.共识过程 3.重选leader机制

06 DAG算法

6.1 DAG算法概述

- 自2013年，首次由以色列希伯来大学学者提出基于DAG的共识算法后，逐渐成为区块链项目的技术热点之一。DAG（Directed Acyclic Graph，有向无环图）原本是计算机领域一种常用数据结构，因为独特的拓扑结构所带来的优异特性，经常被用于处理动态规划、导航中寻求最短路径、数据压缩等多种算法场景。
- DAG则是没有包含循环的有向图，故称之为有向无环图。
  - 图（Graph）是一个顶点和边的集合，其中顶点是通过边连接的无结构化对象，表示为G(V, E)。
  - 有向图则是从一个顶点到另外一个顶点的边是有方向的。
  - 有向图中顶点u到顶点v的路径是从顶点u开始经过一些列的边到达v的路径。当存在一个顶点v到顶点u的路径时，则有向图包含了一个循环。

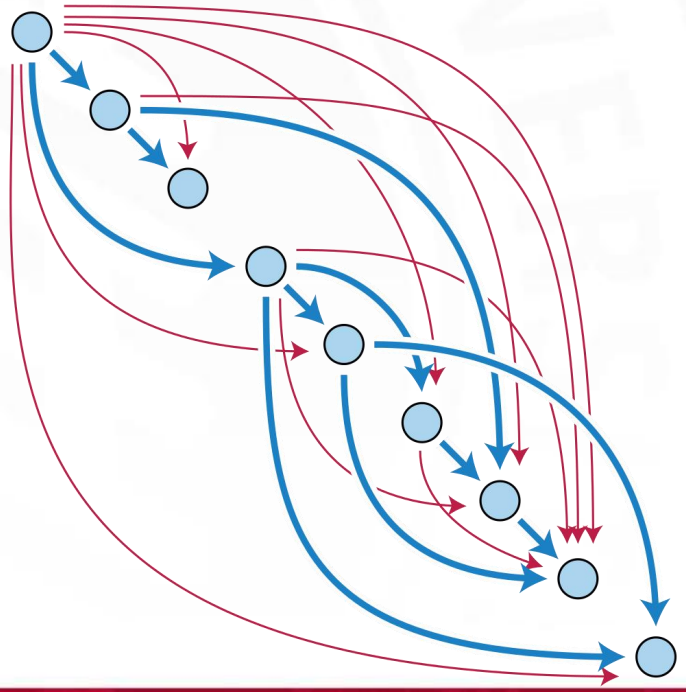
6.1 DAG算法概述

任何DAG中的可达关系可以形式化为DAG顶点上的偏序≤。当存在从顶点u到顶点v的有向路径时，可以用偏序表示为u≤v。但是不同的DAG可能会产生相同的偏序。如图G_1包含两个边a→b和b→c，产生a≤b、b≤c和a≤b≤c，图G_2具有三个边a→b，b→c和a→c，产生a≤b、b≤c、a≤c和a≤b≤c，两个图都可以产生a≤b≤c的偏序。

DAG具有可达性、传递闭包和传递归约的特性。

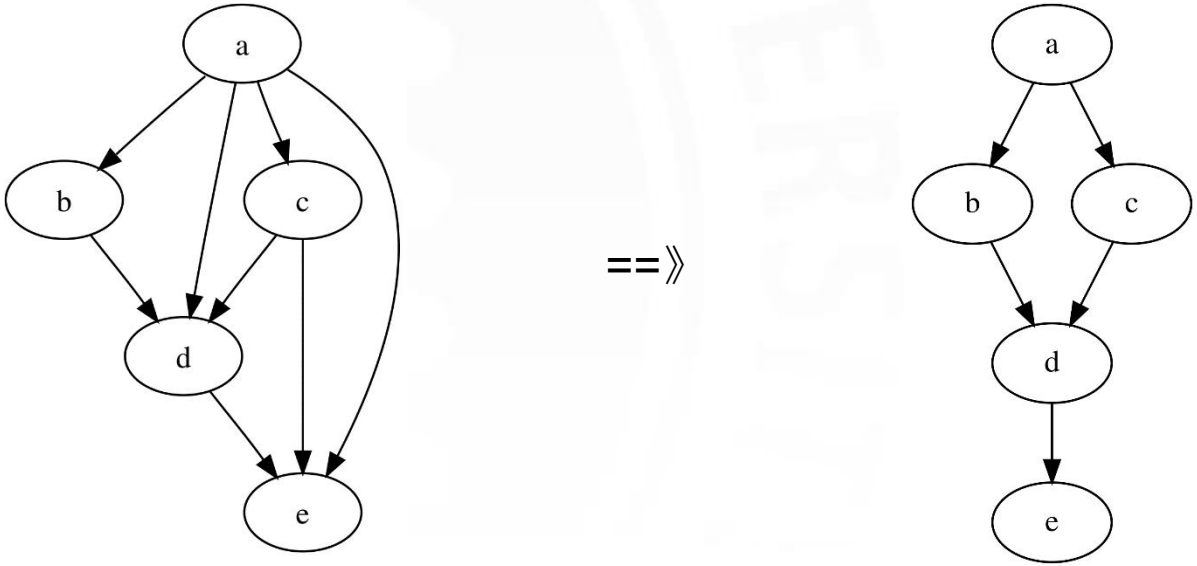
6.1 DAG算法概述

传递闭包是指具有与图G相同的可达性关系的最多的新图。用偏序集合（S, ≤）表示为图G所有的可达关系，对于集合S中任意的偏序关系u≤v，使得新图G’具有边u→v，则称为具有传递闭包的DAG图，图G’与图G具有相同的可达关系（S, ≤）。



6.1 DAG算法概述

传递归约是指与图G相同可达关系的最少的边的图，该图一般是G的子图。对于DAG图，传递归约后的子图具有唯一性。对于偏序集合（S, ≤）中任意一个偏序关系u≤v，当具有边u→v时，同时存在另外一个较长的路径，则去掉边u→v，从而形成新的图G’（V, E’），G’与G具有相同的（S, ≤）。



6.1 DAG算法概述

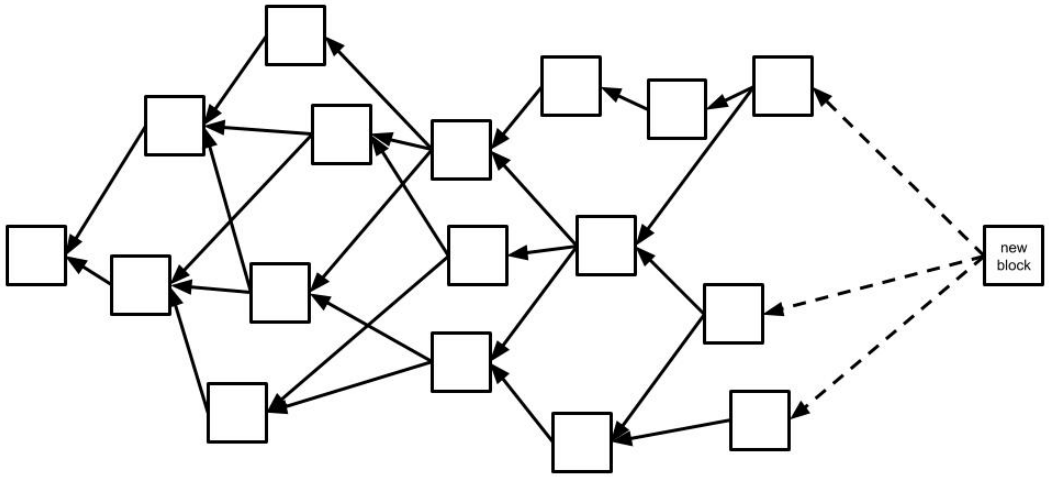
- DAG中一个重要的运算是**拓扑排序 - 区块排序的基础。**每个DAG都至少有一个拓扑排序，一般来说是不唯一的，但如果DAG中存在一个路径包含了所有顶点，则拓扑排序是唯一的。

- 拓扑排序是图G(V, E)中所有V中顶点的线性排序，对于E中任意一个边u→v，边的起始顶点u在序列中比边的结束顶点v更早出现

- 通过拓扑排序，可以对图中所有顶点进行线性排序，而通常区块链也是区块的一种特殊线性排列，实际上区块链也是一种特殊的DAG图，只是这个图的顶点是区块。

6.2 DAG共识

- 采用DAG作为区块链结构时，分叉被整合到DAG中，这称之为blockDAG，如下图所示。在blockDAG中后续block并不指向唯一的一个区块，而是可以指向多个区块。



- 在基于POW的共识算法中，各个矿工基于本身的区块链产生新的区块，并选择最长的链作为主链，这不可避免的会产生分叉，同时降低系统的安全性。

6.2 DAG共识

- 优点：
  - 通过采取blockDAG的数据结构时，使每个节点不需要再等待其它节点形成共识区块就可以处理新的交易，从而避免因网络延迟和数据同步造成的时间浪费，使得参与DAG记账的节点很容易大幅扩展并可提高系统的交易吞吐量。
  - 这种特点使得采用DAG的区块链系统可以异步处理交易并支持多客户端并行交易
- 缺点：
  - 不可避免产生一些安全问题，如：双花问题、影子链攻击
  - 因此需要设计新的blockDAG的构造方法及共识算法，从而确保分布式系统的一致性。

6.2 DAG共识

- 双花问题：
  - DAG异步处理数据的特征导致攻击者可能利用顶点间的信息差进行双花。
  - 如果DAG中两个顶点没有偏序关系，攻击者可以分别在只看到这两个不同顶点中的一个不同网络节点处对同一笔钱进行双花。
  - 这种双花的检测只能在能同时看到这两个顶点的网络节点处或者两个顶点重新汇合到一个新顶点时才能最终判定哪一笔是双花。
  - 为防止这种双花的产生，需要制定更周密的双花检测机制。
- 影子链攻击：
  - DAG允许多重并行交易的特征，导致攻击着可以暗中生成一条影子链并且可以不时地将影子链跟主链进行对接以逃避检测算法。极端情况下，这条影子链有可能代替主链成为全网共识。

6.2 DAG共识

- 基于blockDAG这种数据结构需要新的共识算法来形成区块链系统的共识，而不是POW中的最长的链规则。对POW共识算法进行简单更改使得每个矿工基于本节点的当前blockDAG产生新的区块，并使得新的区块指向当前blockDAG的所有最顶端的区块。

- 不能把blockDAG视为一种新的共识算法或协议，它仅仅是一种新的共识协议的基础框架。blockDAG中包含来自不同分支的块，这些由不同矿工产生的区块中会包含冲突的交易，而传统区块链通过构造保持一致性（BlockChain的最长链规则或难度值最大等）。

6.2 DAG共识

- 设计一个基于blockDAG的排序协议使得所有节点对DAG中的所有区块的排序达成一致并能够消除区块之间的冲突，那么就形成了基于blockDAG的共识算法。
  - 当分布式系统中所有的节点对交易的顺序达成一致并且按照约定的顺序执行交易时，就很容易使得系统中的所有节点状态达成一致。
  - 每个blockDAG都有唯一一个拓扑排序，因此对于blockDAG已经有一个对所有区块的拓扑排序，但对于并行生成的区块（没有路径连接的区块），没有连接的边，这些区块就无法排序。那么只需对并行生成的区块定义一个偏序即可

6.2 DAG共识

- 基于此希伯来大学开发了基于blockDAG的协议，包括SPECTER和PHANTOM
  - 协议定义了各自的算法，能够对DAG中的所有区块进行排序
  - 按照顺序对区块进行迭代处理并消除与其他块的交易冲突。
  - 相对来说SPECTER协议略弱一些。

6.3 基于blockDAG共识的优势

- 极短的确认时间，blockDAG可以在几秒中内确认交易，从而避免双花和冲突的出现；
- 较大的交易吞吐量，每个节点基于自身构建区块，从而极大的提升了系统的吞吐量，使得吞吐量仅受网络和节点容量的影响，相应的也极大的降低了交易费用；
- 采矿权中心化问题，基于区块的POW共识中，矿工一方面可以形成集中化的矿场集团，另一方面，获得打包交易权的矿工拥有巨大权力。blockDAG使得每个节点都可产生区块，促使打包交易的权利分散，从而避免中心化的风险；
- 可以避免分叉，同时也规避了分叉带来的风险；
- 降低能耗，传统区块链中采用竞争挖矿，一旦竞争胜利，获得所有奖励，造成其他矿工的能源浪费及奖励的独占性。blockDAG可以消除自私性的奖励，通过把奖励分散给所有的区块，降低能耗。

6.4 顶点粒度

- DAG在区块链的应用分为blockDAG和blocklessDAG。

- blocklessDAG：
  - 2015年9月，Sergio Demian Lerner发表了 《DagCoin: a cryptocurrency without blocks》一文，提出了DAG-Chain的概念，首次把DAG网络从区块打包这样粗粒度提升到了基于交易层面
  - DagCoin的思路是，让每一笔交易都直接参与维护全网的交易顺序。交易发起后，直接广播全网，跳过打包区块阶段，达到所谓的blockless。这样省去了打包交易出块的时间。

6.5 PHANTOM

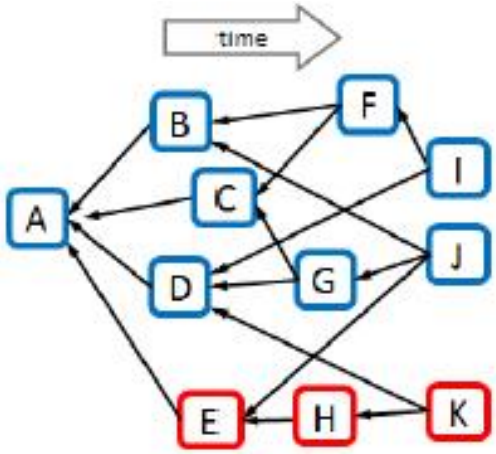
- PHANTOM仍然遵循与比特币相同的框架模型，包括交易、区块、工作量证明、计算能力受限的攻击者、基于P2P网络的块通信、基于概率的安全保证等

- 区别主要在于采用blockDAG以及与之相关的挖矿及共识问题。

07 PHANTOM共识协议

7.1 PHANTOM协议主要术语

- 设定blockDAG为G = (C, E)，其中C表示分布式系统产生的集合，E表示指向先前产生的区块的hash值。那么对于G中的任何一个区块B，定义下列术语：
  - past(B, G)：是指G中B能达到的区块集合，集合中的任意一个区块B’满足B’≤B并且B’C，表示B’早于B被矿工创建;
  - future(B, G)：是指G中能到达B的区块集合，集合中的任意一个区块B’满足B≤B’并且B’C，表示B’晚于B被矿工创建;
  - anticone(B, G)：是指即包含在past(B, G)也不包含在future(B, G)中的顶点集合，这些顶点没有直接或间接到达B的路径;



7.1 PHANTOM协议主要术语

- tips(G)：是G中入度（in-degree）为0的区块集合，这些区块是最新的区块，也就是不存在指向该区块的区块即集合中的任意一个区块B，其future(B, G)为空集合。

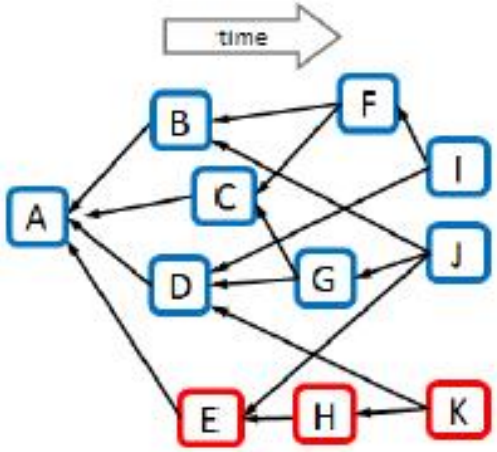
- 参数k：是anticone(B, G)集合中区块的最大个数。在PHANTOM算法中认为所有诚实节点生成的blockDAG应该满足其|anticone(B, G)|k，如果超过k则认为是恶意节点或错误节点创建的区块，这些区块将会被排序在blockDAG的拓扑排序的后面。

7.1 PHANTOM协议主要术语

- 参数k的确定：区块链网络中的节点通信时，存在通信延迟，假设D_{max}是通信延迟的上限，该值是一个系统设置的常量（如比特币的10分钟）。如果区块B在时间点t被一个诚实节点创建，则$t-D_{max}$之前其他节点创建的区块必须到达该诚实节点并包含在past(B, G)中，同时B也应该包含在$t+D_{max}$之后挖掘出的任意一个区块B’的past(B’, G)中。那么在$[t-D_{ma}, t+D_{max}]$中间挖掘出的区块就可能包含在anticone(B, G)中，同时也是该集合的最大集合。POW机制确保在$[t-D_{ma}, t+D_{max}]$时间范围内挖掘出相对稳定数量的区块，而k就是*$[t-D_{ma}, t+D_{max}]$时间范围内所挖掘出的除B以外的最大数量*。

7.1 PHANTOM协议主要术语

- k-cluster（k簇）：给定一个blockDAG G=(C, E)，C中的一个子集称之为k-cluster，当且仅当子集S中的每个区块，其|anticone(B, G)| ≤ k。因此PHANTOM算法中将blockDAG分为k-cluster区块和其他区块，对k-cluster进行拓扑排序，对其他区块则进行特殊处理。如下图所示的一个blockDAG，设定参数为3，则A、B、C、D、F、G、I、J在3-cluster的集合中，用蓝色标记，而E、H、K则不属于3-cluster用红色标记。

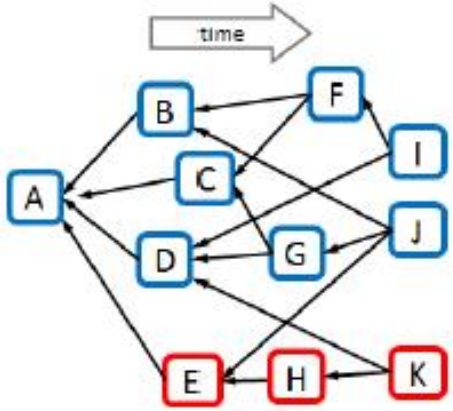


7.2 DAG挖矿协议

- 两条规则：
  - PHANTOM算法中的矿工基于当前矿工节点所观察到的blockDAG进行挖矿，挖掘出的新区块中包含当前tips(G)中所有区块的has值。
  - 挖掘出新区块后，矿工会尽可能快的将新区块广播出去。

7.3 PHANTOM排序协议

- 对下图的例子按照PHANTOM排序协议进行排序，可以得到如下的排序序列：（A，D，C，G，B，F，I，E，J，H，K）。



- PHANTOM排序协议实际应用很少，求解k-cluster最大子集是一个NP难问题。

- 变种的PHANTOM排序协议称之为GHOSTDAG，引入一种贪心算法。

7.4 GHOSTDAG排序协议

- 与PHANTOM排序协议相同，GHOSTDAG也是讲blockDAG分成蓝色部分和红色部分，但是**求解的方法不同**。
- 求解完整的完整的最长链：
  - 首先GHOSTDAG选取tips(G)中past(B, G)集合（排除红色部分区块）最大的区块B_{max}放到链头。这里的B_{max}是指tips(G)中具有最大BlueSet集合的区块
  - 然后在past(B_{max})中选择新的B'_{max}，以此类推一直到创始区块genesis。
- 表示为：Chn(G) =(genesis = $Chn_0(G)$, $Chn_1(G)$, ... , $Chn_h(G)$)。
- 基于链Chn(G)，对blockDAG中所有的block进行排序。那么当k=0时，就形成了和比特币相同的最长区块链。

7.4 GHOSTDAG排序协议

- 具体的算法说明参见下面的OrderDAG(G, k)描述。整个算法结束的唯一条件是blockDAG中仅仅包含一个genesis，整体上可以分为下列步骤：
  - 以递归调用的形式计算tips中每个区块B的past(B)子图的BlueSet和OrderList[行3]；
  - 选择tips中的B_{max}区块[行4]，并将该B_{max}的BlueSet和OrderList作为$BlueSet_G$和$OrderList_G$；[行5-6]
  - 将B_{max}添加到$BlueSet_G$和$OrderList_G$；[行7-8]
  - 然后对anticone(B, G)中的每个区块B进行迭代处理[行9-12]，检查将B添加到$BlueSet_G$后新的BlueSet是否依然满足k-cluster属性，满足则添加到$BlueSet_G$中[行11]；
  - 对于anticone(B, G)中的区块无论是否属于BlueSet，都添加在$OrderList_G$的最后。

7.4 GHOSTDAG排序协议

- 以递归调用的形式计算tips中每个区块B的past(B)子图的BlueSet和OrderList[行3]；
- 选择tips中的B_{max}区块[行4]，并将该B_{max}的BlueSet和OrderList作为$BlueSet_G$和$OrderList_G$；[行5-6]
- 将B_{max}添加到$BlueSet_G$和$OrderList_G$；[行7-8]
- 然后对anticone(B, G)中的每个区块B进行迭代处理[行9-12]，检查将B添加到$BlueSet_G$后新的BlueSet是否依然满足k-cluster属性，满足则添加到$BlueSet_G$中[行11]；
- 对于anticone(B, G)中的区块无论是否属于BlueSet，都添加在$OrderList_G$的最后。

7.4 GHOSTDAG排序协议

输入：G - blockDAG， k 参数
输出：$OrdererList_G$ - 包含 G 中所有区块的排序序列
$BlueSet_G$ - G 中蓝色部分的区块
算法：function OrderDAG(G, k)

```
1:      if G 仅包含 genesis then
2:          return [{genesis}, {genesis}]
3:      对 tips(G)中的每个区块 B 递归调用 OrderDAG(past(B), k)
4:      B_max = BlueSet 集合最大的区块
5:      BlueSet_G = BlueSet[B_max]
6:      OrdererList_G=OrdererList[B_max]
7:      将 B_max 添加到 BlueSet_G 中
8:      将 B_max 添加到 OrdererList_G 的末尾
9:      for anticone(Bmax, G)中的每个区块 B do
10:         将 B 添加到 OrdererList_G 的末尾
11:         如果添加 B 后，BlueSet_G 仍然是 k-cluster，则将 B 添加到 BlueSet_G 中
12:     end for
13:     返回[OrdererList_G, BlueSet_G]
```

7.5 客户端排序

- 客户端排序协议能够使所有的节点对交易的顺序达成一致，则节点的状态就将达成一致。
  - 基于PHANTOM的排序协议建立blockDAG中所有区块的线性排列ord(G) = (B_0, B_1, ... , $B_{|G|}$)。
  - 经过DAG排序后，blockDAG形成了一个包含所有区块的排序序列，但因为anticone(B, G)的存在，序列中可能包含了冲突的交易，这些问题的解决是通过PHANTOM的客户端排序协议解决的
  - 所谓客户端是指没有挖矿能力的节点，这些节点仅仅通过访问本地账本就可以处理冲突，而不需要与其他节点交互。
- 排序规则：
  - 对同一区块基于交易的打包顺序进行排序。
  - 不同区块冲突时，先出现的有效，后出现的无效。

08 基于DAG的区块链系统

8.1 知名DAG区块链系统

- 市场上应用DAG区块链系统，最知名的是IOTA、Byteball、Nano。

8.2 IOTA

- IOTA在2017年下半年市值冲入币值排行榜第四名，从而引起对DAG的大量关注。
- IOTA背后最主要的创新是Tangle（纠缠），这是一个基于DAG的全新设计的分布式账簿结构。Tangle的核心原则与区块链一致，与传统区块链之间的主要区别，就是Tangle数据结构以及共识机制。
- 在Tangle中，每一个顶点代表的是一个交易，没有区块的概念。
- 每一个交易都会引用过去的两条交易记录Hash，这样新的交易会证明过去两条交易的合法性并间接证明之前所有交易的合法性。
- 整个网络都参与交易合法性的验证，不像传统区块链需要矿工或PoS的权益所有人来验证交易合法性，也就没有挖矿和矿工的概念。
- IOTA的共识就是它自身内化特性，可以使它在没有交易费用的情况下进行规模化使用，整个网络的吞吐量也很高，这是IOTA的最吸引人的亮点之处。

8.2 IOTA目前的问题

- IOTA使用了自己开发的哈希算法curl，极易发生碰撞，于是就有伪造数字签名的风险；
- 共识是由全网交易确定的，那么理论上来说，如果有人能够产生1/3的交易量，他就可以将无效交易变成有效交易。另一方面，由于IOTA无手续费，所以没有矿工激励，IOTA面临着拒绝服务攻击和垃圾信息攻击可能；
- IOTA引入闭源的中心化组件Coordinator来对全网交易进行检查（例如双花），如何有效移除Coordinator并建立一个具有良性激励机制的去中心化机制，IOTA还没有给出解决方案。

8.3 Byteball

- Byteball被称为区块链3.0的代表。具有DAG体系家族中最完善的应用生态，Byteball钱包内置丰富功能，包括类似Appstore模式的BotStore，自由开发者可以在上面自由开发应用，开发者非常活跃。

- Byteball在DAGCoin的基础上，创新性引入主链与见证人概念，鼓励验证多个父辈交易单元，形成一个随着交易增长、相互验证，安全性不断加强的数字签名Hash网络。

- Byteball创造性的发明了「主链」概念，也就是经过见证人认定的最短路径MC的Parents优选算法。主链创造了一个全网共识确定的交易时间序列，优雅地避免了双花问题。

8.3 Byteball

- Byteball中“见证人”真正意义就是形成共识机制；12个“见证人”发布的交易单元，在理论上无限宽广的DAG并发交易网络中划出了一道确定性的交易时间序列。
- 基于见证人+主链的共识机制，双重支付等问题得到了轻松解决。
- Byteball取消了区块链和工作量证明挖掘的概念，而是选择了DAG数据存储技术。
  - Byteball中的所有交易都是以加密方式相互关联的。
  - 新产生交易将添加到tips交易单元后面
  - 这样让网络上的所有节点（用户）都参与验证交易，完全的去中心化。

8.3 Byteball

- 在Byteball的Witness节点设计中，witness节点是高度安全的，仅仅能发出见证单元，无法接触交易。
  - Witness并不是矿工，Witness扮演的是WatchMan的角色，帮系统锚定交易发生时间顺序，没有留出作恶的空间。
  - 另外Witenss数量也可以根据需要设置和选择，并不局限于目前的12个见证人。
- Byteball由于每个交易都有发起者的私钥签名，同时每笔交易都验证与引用从前发生的交易，以此编织成一个巨大的网络，对网络的篡改牵一发而动全身。
- Byteball的问题是：由于主链算法和见证人发布频率有关系，交易确认的时间是不确定的；由于Byteball基于关系数据库来存储数据，SQL语言过于紧耦合算法逻辑，在一定程度上限制了Byteball目前的扩展能力和速度。

谢谢！