分布式系统的一致性与共识算法-Paxos

2021-03-11 · 分布式系统 · 7993 words · 16 mins read · 54 times read

> 本文讨论共识算法：Paxos

关于Paxos

Paxos算法是Leslie Lamport于1990年提出的一种基于消息传递共识算法，能保证多副本数据强一致性与分区容错性；现已是当今分布式系统最重要的理论，为后续的如Raft、ZAB等算法、ZooKeeper、Etcd等分布式协调框架奠定了基础。

Google Chubby的作者给了Paxos极高的评价：There is only one consensus protocol, and that's "Paxos" — all other approaches are just broken versions of Paxos（世界上只有一种共识协议，就是Paxos，其他所有共识算法都是Paxos的退化版本）

问题-假设

问题：基于前文对分布式环境下一致性与共识算法的基础理论，在分布式系统中进行节点通信大部分采用基于消息传递通信模型，不可避免的会发生如进程可能会慢、被杀死或者重启等问题，会对分布式系统中各个节点对某一个值达成一致性产生问题；而Paxos就是为了解决这个问题而生的。

场景：在一个分布式数据库系统中，如果各节点的初始状态一致，每个节点都执行相同的操作序列，那么他们最后能得到一个一致的状态。为保证每个节点执行相同的命令序列，需要在每一条指令上执行一个"一致性算法"以保证每个节点看到的指令一致。

假设：Lamport虚构了一个名为Paxos的希腊城邦，这个城邦按照民主制度制定法律，却又不存在一个中心化的专职立法机构，立法靠着"兼职议会"（Part-Time Parliament）来完成，无法保证所有城邦居民都能够及时地了解新的法律提案、也无法保证居民会及时为提案投票。Paxos算法的目标就是让城邦能够在每一位居民都不承诺一定会及时参与的情况下，依然可以按照少数服从多数的原则，最终达成一致意见。注意：Paxos算法并不考虑拜占庭将军问题，即假设信息可能丢失也可能延迟，但不会被错误传递。

Paxos算法运行在允许宕机故障的异步系统中，不要求可靠的消息传递，可容忍消息丢失、延迟、乱序以及重复，保证了2F+1的容错能力，即2F+1个节点的系统最多允许F个节点同时出现故障。

算法推演

首先回看上文在解决分布式环境下复制带来的副本一致性问题时，我们提到可以通过两类基本复制算法解决：Replication methods that prevent divergence (single copy systems) 与Replication methods that risk divergence (multi-master systems) 。

以Replication methods that prevent divergence为例，可以通过如**Master/Slave（主从复制）**、**2-phase commit（两阶段提交）**、**Quorum机制(多数派读写)** 方式实现，但都或多或少存在着问题；本文讨论的Paxos可以看作是对Quorum机制（多数派读写）的进一步升级。

0x01.Paxos算法中对应的角色

- **Proposer**：提出提案 (Proposal)；可以理解为客户端，Proposal信息包括提案编号 (Proposal ID) 和提议的值 (Value)。
- **Acceptor**：参与决策，可以理解为存储节点，回应Proposers的提案。收到Proposal后可以接受提案，若Proposal获得多数派Acceptors的接受，则称该Proposal被批准。
- **Learners**：用于学习被批准的提案

Paxos算法中的角色允许身兼数职，也有了如下的基本约束：

立刻说两句吧！

1. 决策（value）只有在被 proposers 提出后才能被批准（未经批准的称为提案）；
2. 在一次 Paxos 算法的执行实例中，只批准（chosen）一个 value；
3. Learners 只能获得被批准（chosen）的 value。

作者Lamport主要通过不断加强上述3个约束（主要是第二个）获得了 Paxos 算法。

0x02. 系统模型

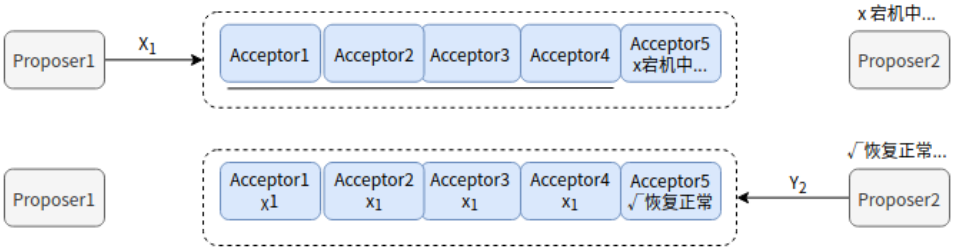
一个系统中，存在多个Proposer节点提出提案，多个Acceptor节点负责决策-接受提案。为了满足只批准一个value 的约束，要求经Quorum（多数派）接受的 value 成为正式的决议。即 一个提案被选定需要被半数以上的Acceptor接受 。

Quorum机制下，假设只有一个Proposer提出了一个value，该value也会被决策，要保证约束2，就会产生P1约束 P1：一个Acceptor必须接受第一次收到的提案

P1 是不完备的，不同的Proposer提出不同的value的话，如果遵循P1，就会出现无法形成多数派的情况；因为存在多个提案，这里就需要给每个提案加上一个提案编号以表示顺序，即提案=编号+Value；只要提案的value 是一样的，批准多个提案不违背约束2，我们就可以保证只有一个值被选中，可以得到如下约束 P2：如果某个value为v的提案被批准（chosen），那么之后批准（chosen）的提案必须具有 value v

因为一个提案只有被Acceptor接受才可能被选定，所以我们可以把P2约束改写成对Acceptor接受的提案的约束 P2a：一旦一个具有 value v 的提案被批准（chosen），那么之后任何 Acceptor 再次接受（accept）的提案必须具有 value v

因为通信是异步的，系统是不可靠的，P2a和P1可能会存在冲突，例如一个 value 被批准后，一个Proposer 和一个 Acceptor 从休眠中苏醒，前者提出一个具有新的 value 的提案；这种情况下根据 P1，Acceptor应当接受，根据 P2a，则不应当接受（如下图所示）， P2a 和 P1 有矛盾。于是需要换个思路，转而对 proposer 的行为进行约束得到 P2b：一旦一个具有 value v 的提案被批准（chosen），那么以后任何 Proposer 提出的提案必须具有 value v。



1. Acceptor5认为Y2被选定，Acceptor1~4认为x_1被选定。出现了不一致。
2. x_1被选定了，但Acceptor5因为P1接受了编号更高的提案Y_2，且$X_1 \neq Y2$。这里P1和P2a矛盾了。

由于 acceptor 能接受的提案都必须由 proposer 提出，所以 P2b 蕴涵了 P2a，是一个更强的约束。但是根据 P2b 难以提出实现手段，需要进一步加强 P2b；假设一个编号为 m 的 Value v 已经获得批准，存在一个 Acceptors 的多数派 C，他们都接受（accept）了v，考虑到任何多数派都和 C 具有至少一个公共成员，可以找到一个蕴涵 P2b 的约束 P2c：如果一个编号为 n 的提案具有 value v，该提案被提出，那么存在一个多数派，要么他们中所有人都没有接受（accept）编号小于 n 的任何提案，要么他们已经接受（accept）的所有编号小于 n 的提案中编号最大的那个提案具有 value v。

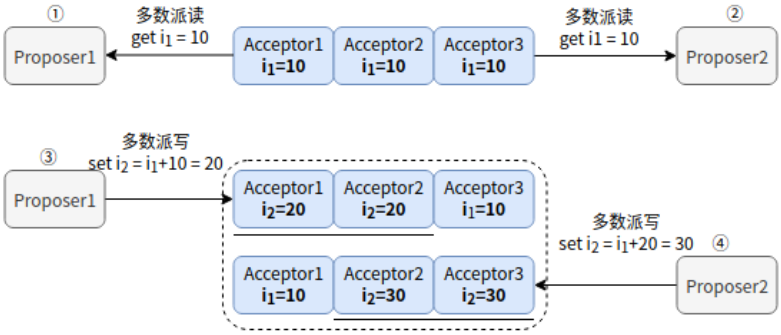
要满足P2c的约束，会涉及两个流程：

- **prepare阶段**：Proposer提出一个提案前，要和足以形成多数派的Acceptors进行通信，获得他们进行的最近一次接受（accept）的提案，根据回收的信息决定这次提案的value，形成提案开始投票。
- **批准阶段**：当获得多数acceptors接受（accept）后，提案获得批准（chosen），由Acceptor将这个消息告知learner（这个过程逐渐细化，就形成了Paxos算法）

立刻说两句吧!

并发情况下：如果一个没有chosen过任何proposer提案的Acceptor在prepare过程中回答了一个proposer针对提案n的问题，但是在开始对n进行投票前，又接受（accept）了编号小于n的另一个提案（例如n-1），如果n-1和n具有不同的value，这个投票就会违背P2c。因此需要对P1进行加强，在prepare过程中，acceptor进行的回答后不会再接受（accept）编号小于n的提案， `P1a：当且仅当Acceptor没有回应过编号大于n的prepare请求时，Acceptor接受（accept）编号为n的提案。`

0x03.prepare阶段：为什么需要获取多数派Acceptor最近接受的提案



如果两个Proposer进程并发进行读写操作, 在多数派读写的实现中,会产生一个Proposer覆盖另一个Proposer的问题. 从而产生了数据更新点丢失的情况

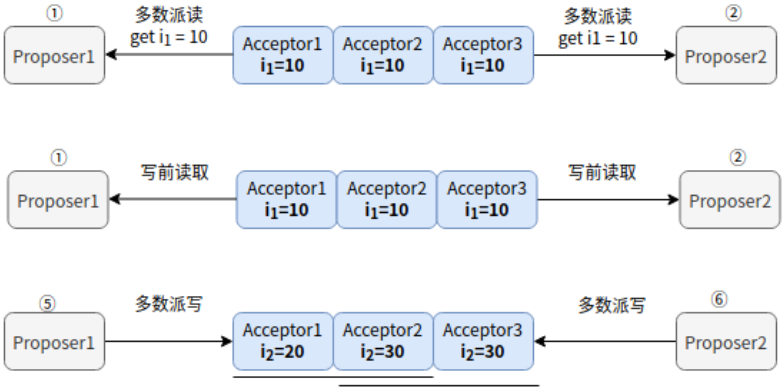
已上图为例子:

问题：Proposer2步骤④进行多数派写的时候，因为并发问题覆盖了Proposer1的多数派写操作，导致Proposer1写入的值失效。

如何解决：Proposer2更新的时候不能直接更新i2版本，而是应该检测到i2的存在, 进而将自己的结果保存在下一个版本i3中,再进行多数派写。问题可推广：系统对于i的某个版本，只能有一次写入—> 一个值（变量的一个版本）被确定了之后，不允许进行修改

解决方案：每次Proposer写之前进行一次多数派读，以便确认是否存在其他Proposer在写；如果存在，则放弃写入；这种操作称为写前读取操作。

0x04.并发进行写前读取操作（确定一个值）导致数据不一致问题



问题：可能出现两个Proposer同时进行写前读取操作，获取到的结果都是没有其他Proposer在写入;这时候同时进行写操作，还是会造成数据不一致的情况。

如何解决：Acceptor节点需要记录最后一个做过写前读取操作的Proposer；进行限制，只允许最后一个完成写前读取的Proposer可以进行后续写入,拒绝之前做过写前读取的Proposer写入的权限。

已上便是Paxos的核心原理，通过2次多数派读写, 实现了强一致性的共识算法。

0x05.提案的提出与批准

1. prepare阶段
    1. Proposer选择一个提案编号n并将prepare请求发送给Acceptors中的一个多数派；
    2. Acceptor收到一个编号为N的prepare消息后，只N大于该Acceptor已经响应过的所有Prepare请求

立刻说两句吧!

Acceptor承诺不再接受任何编号小于N的提案。

2. 批准阶段
    1. 如果Proposer收到半数以上Acceptor对其发出的编号为N的Prepare请求的响应，进入批准阶段。Proposer会发送一个针对[N,V]提案的Accept请求给半数以上的Acceptor。（V为prepare阶段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决定）
    2. 如果Acceptor收到一个针对编号为N的提案的Accept请求，只要该Acceptor没有对编号大于N的Prepare请求做出过响应，就接受该提案。

```
Client   Proposer      Acceptor     Learner
   |         |          |  |  |       |  | --- First Request ---
   X-------->|          |  |  |       |  |  Request
   |         X--------->|->|->|       |  |  Prepare(N)
   |         |<---------X--X--X       |  |  Promise(N,I,{Va,Vb,Vc})
   |         X--------->|->|->|       |  |  Accept!(N,I,V)
   |         |<---------X--X--X------>|->|  Accepted(N,I,V)
   |<---------------------------------X--X  Response
   |         |          |  |  |       |  |
```

Basic Paxos

Basic Paxos为最基本的Paxos实现，上文所述即为Basic Paxos，通过两轮RPC确定某一个提案（Value）

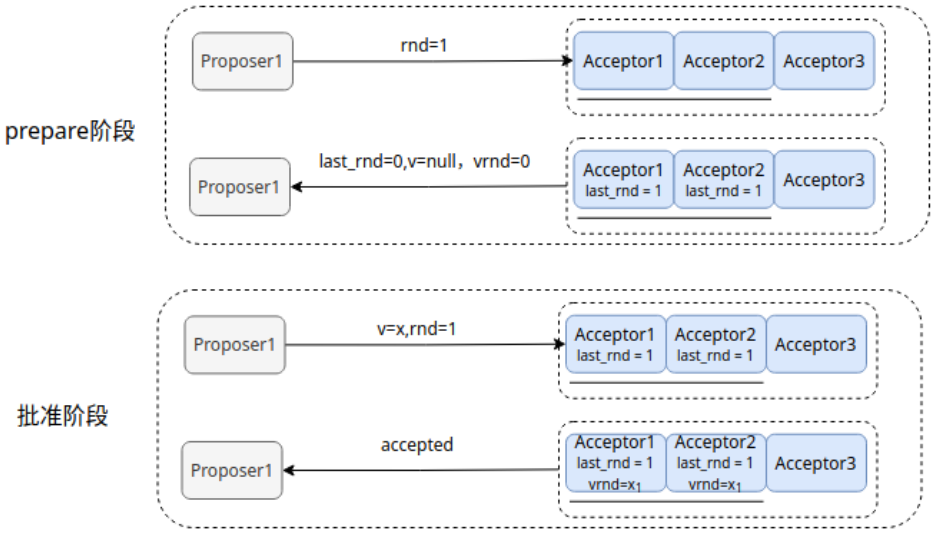
要实现Basic Paxos，算法中的各个角色需要拥有如下功能：

Code

```
Global
        Round：表示一轮Paxos,包含了prepare阶段与批准阶段
        rnd：全局递增，全局唯一的编号，每一个Round对应一个rnd，即提案编号
Acceptor
        last_rnd：Acceptor记住的最后一次发起prepare的Proposer对应的提案编号，以此来批准哪个Proposer的提
        v：Acceptor记住最后被批准的值
        vrnd：一组rnd和v，记录了v在哪个rnd被写入
```

算法实例（prepare阶段、批准阶段）

如下所示为一个不存在冲突的基本流程：



prepare阶段

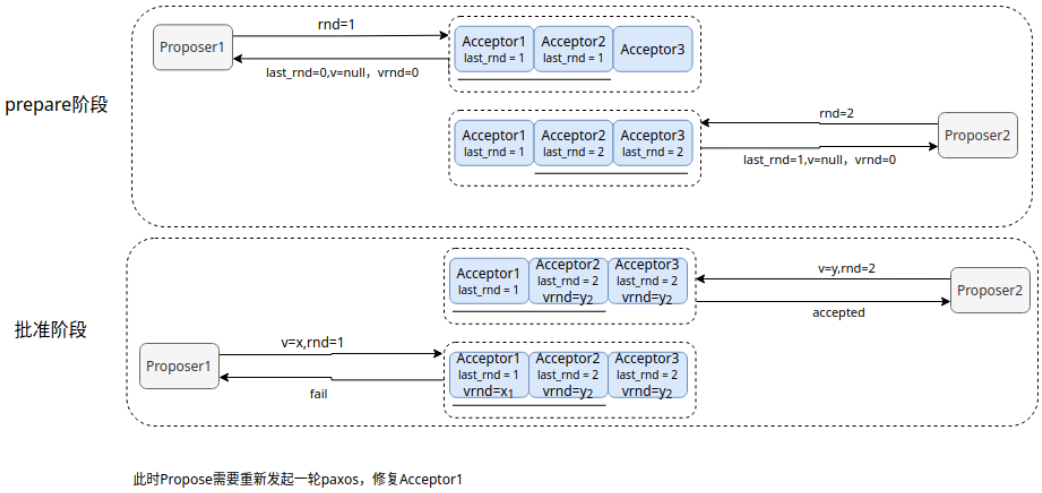
- **Acceptor**
    - Proposer发起的提案的rnd小于Acceptor的last_rnd（并发情况下，获取存在网络延迟），Acceptor会拒绝请求
    - Acceptor会把Proposer请求的rnd保存到本地为last_rnd，后续批准阶段Acceptor只接受带这个rnd为last_rnd的请求
    - 之后Acceptor会返回last_rnd与之前已接受的v、vrnd
- **Propose**

立刻说两句吧！

- Proposer 收到多数(quorum)Acceptor的应答, 就可以执行后续的批准阶段，此时如果没收到多数(quorum)Acceptor的应答,Proposer就hang住了(paxos只能运行少于半数的节点失效的情况)
- 收到多数(quorum)Acceptor的应答后，如果所有Acceptor的v都为null、vrnd为0，表示所有的Acceptor没有批准过任何提案，Proposer可以选择自己要写入的v进行写入
- 收到多数(quorum)Acceptor的应答后，如果Acceptor的v不为null、vrnd不为0，Proposer选择要写入的v为Acceptor应答的v

批准阶段

- **Acceptor**
  - 在并发提案的情况下，可能已经有其他Proposer又完成了一个rnd更大的prepare, 所以这时不一定能成功运行完prepare阶段；Acceptor会对比请求中的rnd和本地记录的last_rnd,如果请求的rnd=last_rnd，则允许请求；如果请求的rnd≠last_rnd，则拒绝请求
  - 接受请求后，Acceptor会将v写入本地，更新vrnd
- **Propose**
  - 如果Propose在批准阶段被拒绝请求，表示有其他Proposer在进行、完成一个rnd更大的paxos；这时候Propose需要重新发起一轮paxos，修复可能存在已经中断的其他Proposer的运行，如下图所示



此时Propose需要重新发起一轮paxos，修复Acceptor1

算法实例（learner学习阶段）

Learner需要学习最终被选定的Value，一般可以通过以下方式进行通知

1. Acceptor接收提案后，直接发送给所有Learner（通信次数较多，但Learner可以快速获取到被批准提案）
2. Acceptor接收提案后，发给主Learner，主Learner再发送给其他Learner（通信次数较少，但容易出现单点故障）
3. Acceptor接收提案后，发给一个Learner集合，再进行广播（通信次数较少，可靠性高，但网络通信复杂度也高）

算法活性

如果有两个Propose A 和 B，A通过Prepare阶段发送rnd 1，B又通过Prepare阶段发送rnd 2，这个时候 A 因为rnd2 无法通过又执行Prepare阶段发送rnd 3… 依次反复陷入了死循环即活锁，使程序无法取得进展。为了保证进展性，需要选择一个唯一的提议者进行提议，可以通过一些随机性技术实现。

Multi Paxos

Basic Paxos缺陷

Basic Paxos只能对单个值形成决议，并且一次Round的形成需要进行两轮RPC（prepare阶段、决策阶段），网络开销大，效率低，对工业化不友好；Base Paxos任何一个proposer节点都是平等的、可以与其他节点并发地提出提案因而带来的更大的实现复杂度。

也可以理解为Paxos是解决对某一个问题达成一致的一个协议，但是实际生产中大部分的应用场景是对一堆

立刻说两句吧!

Paxos或者Fast Paxos。

Multi Paxos的优化

Multi Paxos对Basic Paxos的核心优化点是增加了Leader节点，只有Leader才能提出提案；再进一步简化节点角色，节点只有主（Leader）和从（Follower）的区别，Follwer节点接收到提案请求后，会转发给Leader节点进行发起提案。

因为只有Leader节点可以发起提案，可以视为对提案的批准处于无并发的全局有序环境中，可以支持批量为多次提案运行prepare阶段，此时对某个提案达成一致只需要进行一轮次的RPC请求，即批准（accept）阶段。

```
Client   Proposer       Acceptor     Learner
   |         |          |  |  |       |  |  --- Following Requests ---
   X-------->|          |  |  |       |  |  Request
   |         X--------->|->|->|       |  |  Accept!(N,I+1,W)
   |         |<---------X--X--X------>|->|  Accepted(N,I+1,W)
   |<-----------------------------------X--X  Response
   |         |          |  |  |       |  |
```

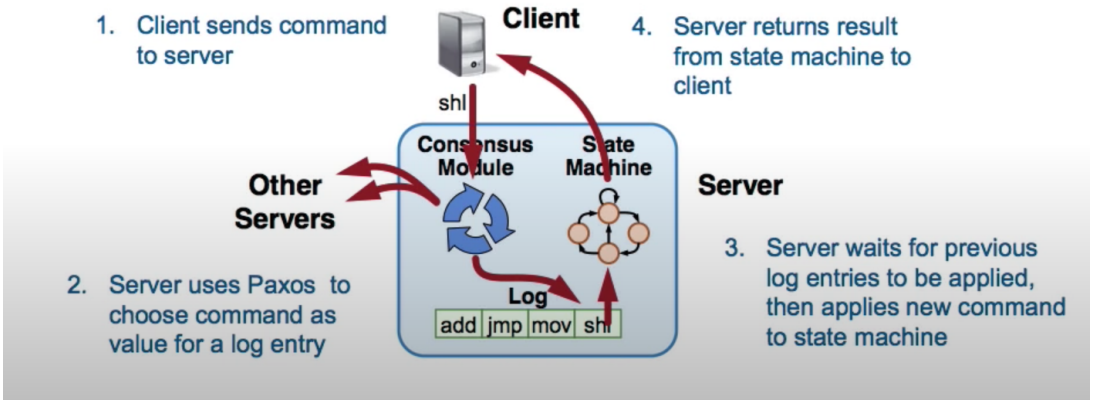
0x01.如何进行Leader选举

Leader选举的过程，可以理解为分布式系统各个节点对“申请Leader”这个提案达成一致。节点启动时，默认处于 Follower 的状态，各个节点会通过心跳的方式定时轮询，确定集群中的Leader节点是否存在；如果不存在Leader节点的情况节点会在心跳超时后基于Basic Paxos的prepare、批准两个阶段向所有其他节点广播竞选Leader的请求，提案被批准后则成为Leader节点。

Leader带有任期属性（一个单调递增的编号），目的是在Leader节点陷入网络分区、宕机后重新恢复，但另外一部分节点仍存在多数派，且已经完成了重新选主的情况，此时必须以任期编号大的主节点为准。

0x02.如何进行数据复制

基本状态机模型，如下图所示：



Multi Paxos在数据复制上采用的是confirm机制，在Basic Paxos协议中，对于决议（value）的读取也是需要执行一轮多数派读取过程的，在实际工程中做数据恢复时（如Leader切换后），对每条日志都执行一轮多数派读代价过大，引入confirm机制的目的在于解决这个问题。

confirm机制

Leader持久化一条日志的时候，客户端向Leader节点发起一个操作请求（如某个值的add command ），进行如下几步操作：

1. Leader节点将command写入自己的变更日志，然后将command信息在下一次心跳包中广播给所有的Follower节点
2. Follower节点接收到信息，将command写入自己的日志，然后给主节点发送确认的消息
3. Leader节点获得多数派Follower节点的确认后，Leader节点提交自己的confirm、响应客户端并且给Follower节点广播该日志可以confirm的消息，Follower节点收到消息后confirm自己的变更

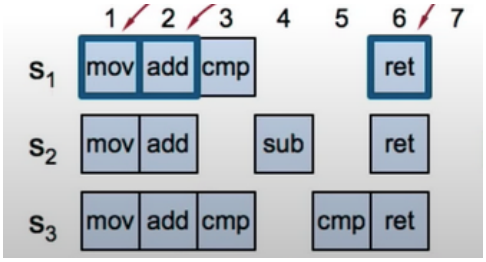
在回放日志时，判断如果一条日志有对应的confirm日志，则可以直接读取本地内容，不需要再执行一轮多数

立刻说两句吧!

网络传输与日志空洞

Multi Paxos中的每一个命令都有一个递增的编号即logID，如果顺序的发布提案，效率会非常低下，所以Multi Paxos采用类似TCP滑动窗口的方案，实现基于logID的滑动窗口机制，可以每次发送N个提案对N个提案进行表决，以增加带宽。

TCP协议中如果TCP包在传输中丢失，最坏的情况是会RST这条链接，然后交给应用层逻辑来解决；对于Multi Paxos如果某些提案因为网络或者其他原因没有被表决，那么就会存在 日志空洞(即不连续的日志) 如下图所示，Multi Paxos 是允许日志空洞这种情况存在的。

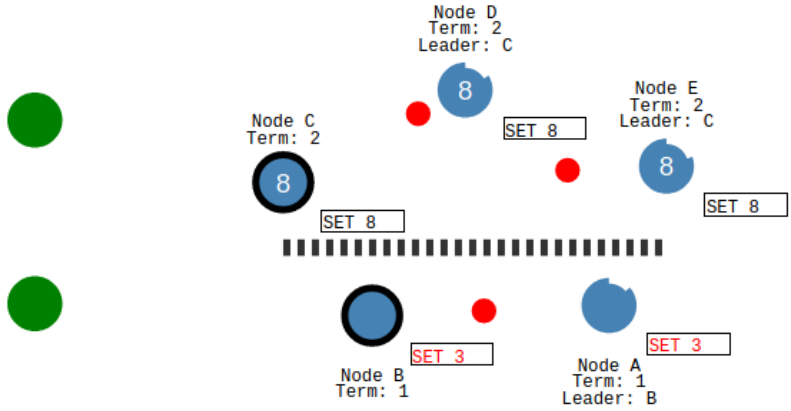


异常情况-Leader切换

如果网络出现了分区，部分节点失联，但只要仍能正常工作的节点的数量能够满足多数派（过半数）的要求，系统就仍可以正常运行。

如下图所示的情况，Node B是任期1的Leader节点，由于网络故障出现了Node A、B和NodeC、D、E两个分区，这时候NodeC、D、E分区的节点通过心跳获知分区内不存在主节点，会进行一次Leader选举，节点数量满足多数派要求，NodeC 当选Leader；此时系统存在NodeC、B两个不同任期的Leader节点。

客户端请求：如果请求到Node A、B分区，不构成多数派的批准，无法进行confirm，如果请求到NodeC、D、E，构成多数派的批准，可以进行confirm，系统可以继续提供服务。



故障恢复：网络问题恢复后，分区会解除，这时候集群内存在两个Leader会以任期编号更大的为主；Node A、B分区的节点会回滚所有未confirm的日志，并进行数据恢复，从Leader节点也就是NodeC发送的心跳包中获得它们失联期间发生的所有变更，并进行confirm操作。

Leader切换后新任Leader对日志重确认

1. 新Leader竞选成功后，拥有对应confirm日志的原始日志，可以直接回放日志；而没有对应confirm日志的原始日志，则需要执行一轮paxos进行重确认操作。
2. 新Leader在上一任Leader的任期内可能存在日志空洞，这些日志空洞也需要进行重确认来补全。
3. 重确认操作的结束位置可以根据中所有Node的最大logID来作为重新确认操作的结束位置。

类Multi Paxos算法

Multi Paxos作为Basic Paxos的改进版本，仅在Paxos的论文中最后Lamport提供了改进的思路，在工程上实现一般都基于原算法的基础上进行一系列的改进，就有了如Raft、ZAB等类Multi Paxos算法；实现思路都是基于Multi Paxos，但是具体实现上都有微小差别，如：Leader竞选的条件、日志是否连续（是否允许存在日志空洞）、Follower节点获取日志confirm操作的方式。

立刻说两句吧!

以Raft为例：

1. Raft仅允许日志最多的节点当选为Leader，而Multi Paxos允许任意节点当选Leader
2. Raft不允许出现日志空洞，便于做Leader切换后的日志重确认，而Multi Paxos允许，实现过程复杂了些

参考

https://zh.wikipedia.org/wiki/Paxos算法

https://blog.openacid.com/algo/paxos

https://icyfenix.cn/distribution/consensus/paxos.html

https://github.com/dappFinance/paxos_raft_protocol

Author : LHR
LastMod : 2021-03-11

#分布式系统 #共识算法 #paxos

Docker容器运行时引擎-runC分析 分布式系统的一致性与共识算法-基础理论