名词介绍

- paxos: 应该是分布式领域较早出现的数据一致性协议（本文研究的正是此协议）
- basic paxos: 通常说的 paxos 就是指 basic paxos，或者称为 classical paxos
- multi-paxos: paxos 的改进，迈出了工程实践的步伐（性能改善，工程落地）
- epaxos/fast-paxos: 其他一些 paxos 的改进，特别是 epaxos 最近几年得到较多的讨论和重视
- raft: 从 paxos 而来，类似于 multi-paxos，但是更为简单，容易理解，更为简单
- quorum: 英文翻译：法定人数，可以理解为多数派，大多数，超过半数的一个集合，更为精确的定义是 ”任意两个 quorum 必须有交集)
- state machine replication model: 复制状态机模型
- Crash Fault Tolerance： 故障容错（节点离线，网络延迟等）
- Byzantine Fault Tolerance： 拜占庭容错（节点离线，网络延迟，节点作恶）
- pbft: 实用拜占庭容错算法
- hotstuff: 也是一种拜占庭容错共识算法
- libraBFT: 基于 hotstuff

先认识名词，从整体上有一些概念，战略上藐视。

预先准备

本文重点分析 paxos (basic paxos) 算法。顺带会提及 multi-paxos 以及 raft 算法。

paxos 很难理解？争取听完本次分享，大家能彻底理解 paxos!

先忘记区块链，忘记 pbft，忘记 hotstuff.

单机？分布式？

为什么要有分布式系统？ 单机容易故障，无法保证服务高可用。

于是出现多副本模型，但多副本模型就存在两个问题：

- 如何确保复制是成功的？（高可用）
- 如何确保值是唯一的？（一致性）

复制是否成功

我们首先把整个模型抽象一下，到最简单的模型。为此，我们定义两个操作：

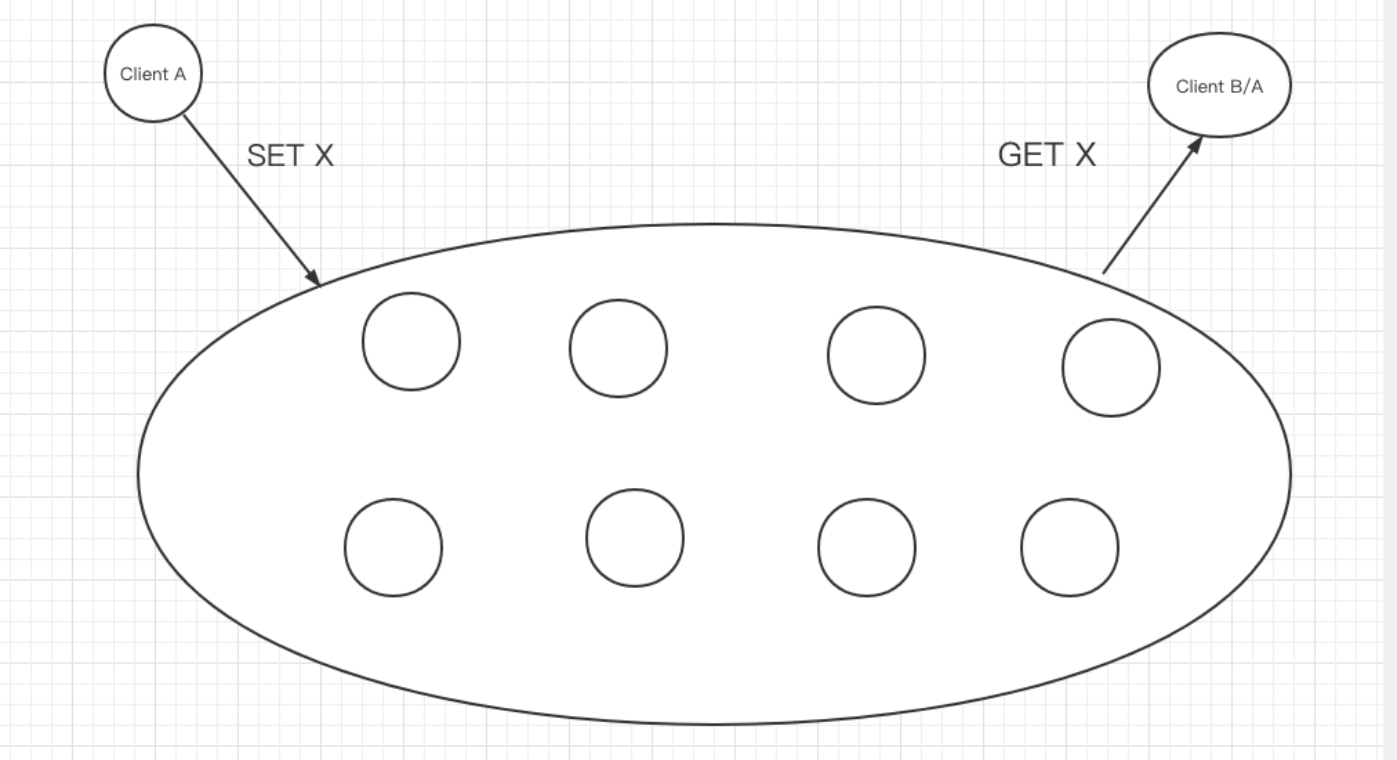
- SET X
- GET X

在这里先不考虑并发，不考虑正确性，不考虑其他操作，也不考虑多个值。

只有这两个操作，而且只操作数据 X，X 初始值为 null.

根据上面的抽象，我们定义复制成功的要求就是：

如果执行了 SET X，那么 GET X 一定能取到值

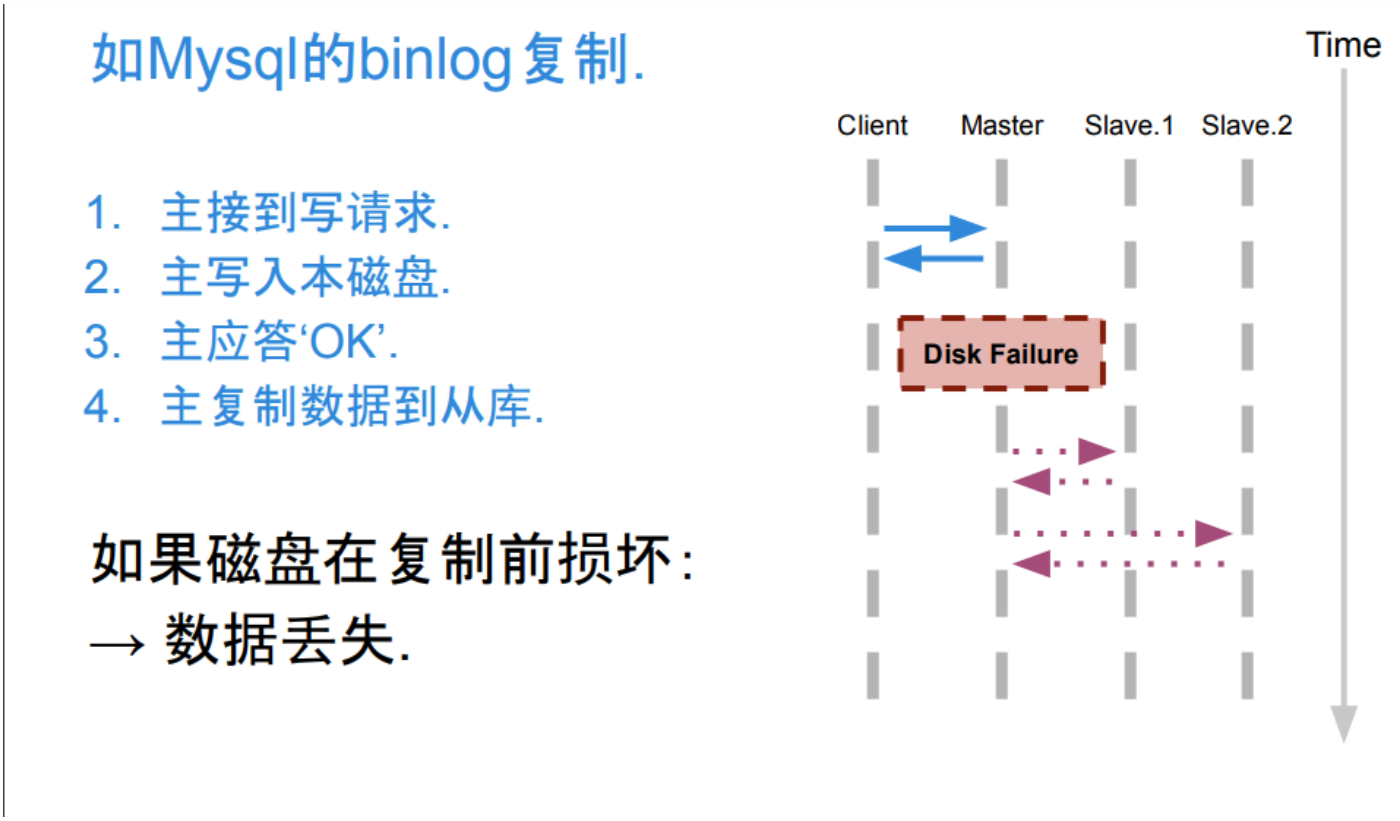


基础的复制策略

- 主从异步复制
- 主从同步复制
- 主从半同步复制
- 多数派写（读）

主从异步复制

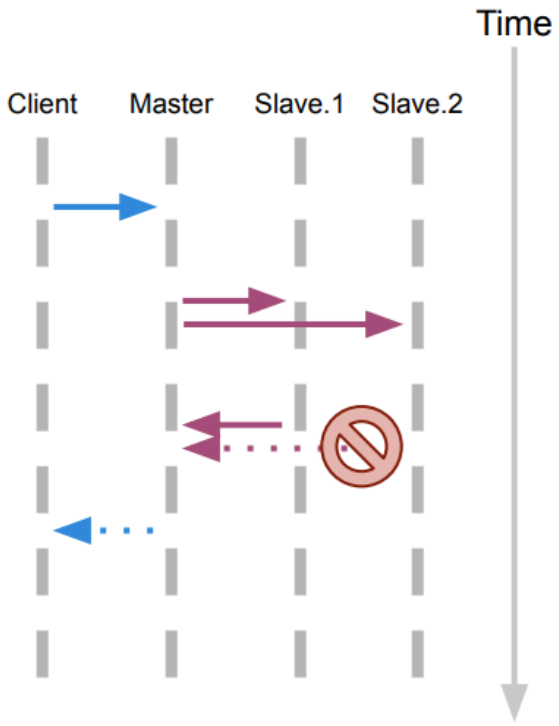
不满足复制成功的要求



主从同步复制

失联节点不确定，写入 slave 个数也不确定，不满足复制成功的条件

1. 主接到写请求.
2. 主复制日志到从库.
3. 从库这时可能阻塞...
4. 客户端一直在等应答'OK', 直到所有从库返回.



一个失联节点造成整个系统不可用.

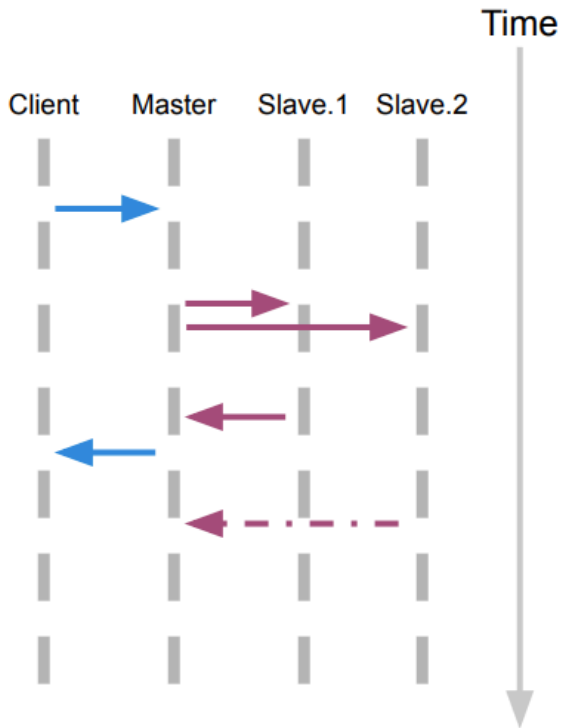
: 没有数据丢失.

: 可用性降低.

主从半同步复制

写入 slave 个数不确定，最少 1 个，最多 n 个，有概率性存在无法获取 X， 不满足复制成功的要求

1. 主接到写请求.
2. 主复制日志到从库.
3. 从库这时可能阻塞...
4. 如果1<=x<=n个从库返回'OK', 则返回客户端'OK'.



: 高可靠性.

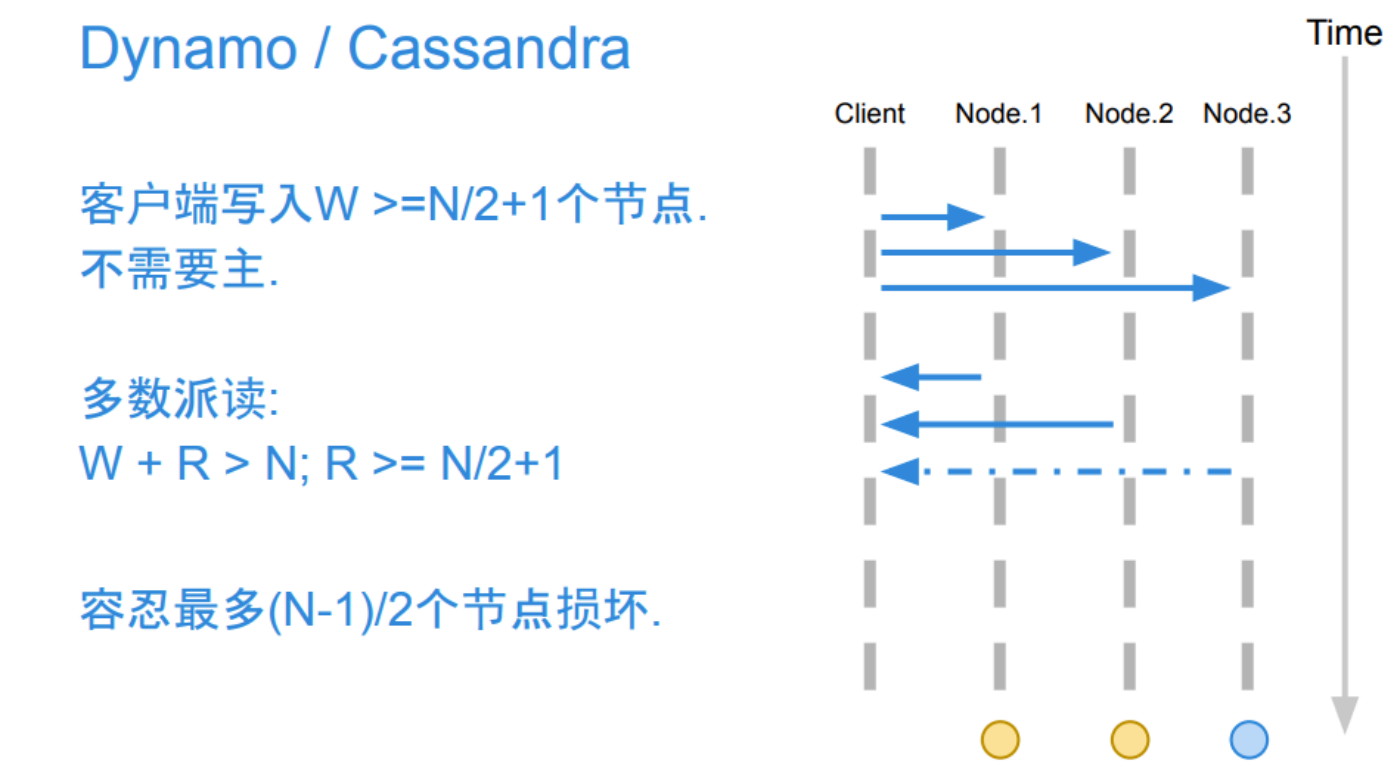
: 高可用性.

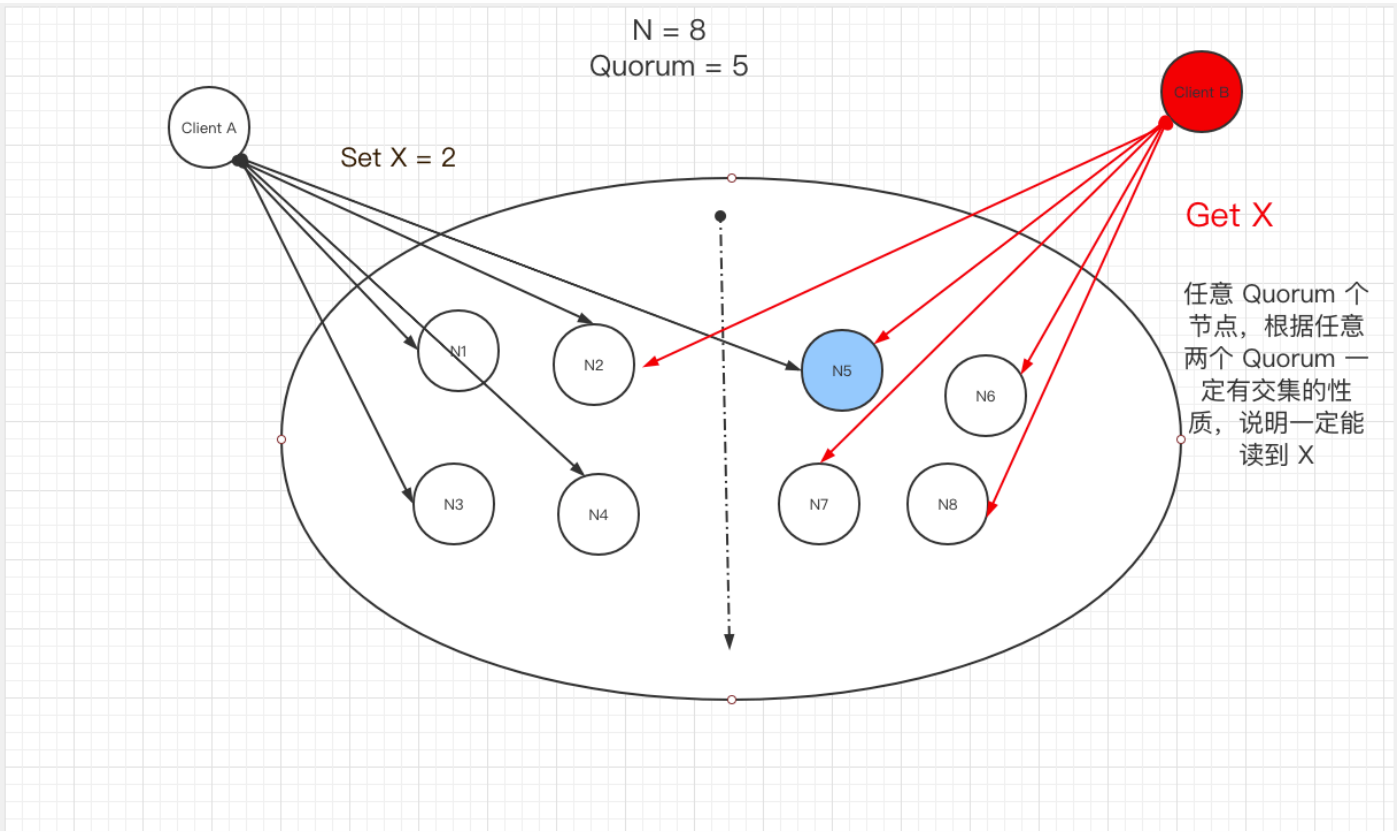
: 可能任何从库都不完整

→ 我们需要 多数派写(读)

多数派写

满足复制成功的要求，先多数派写，再多数派读。





优点：

- 高可靠性
- 高可用性
- 数据完整性有保证

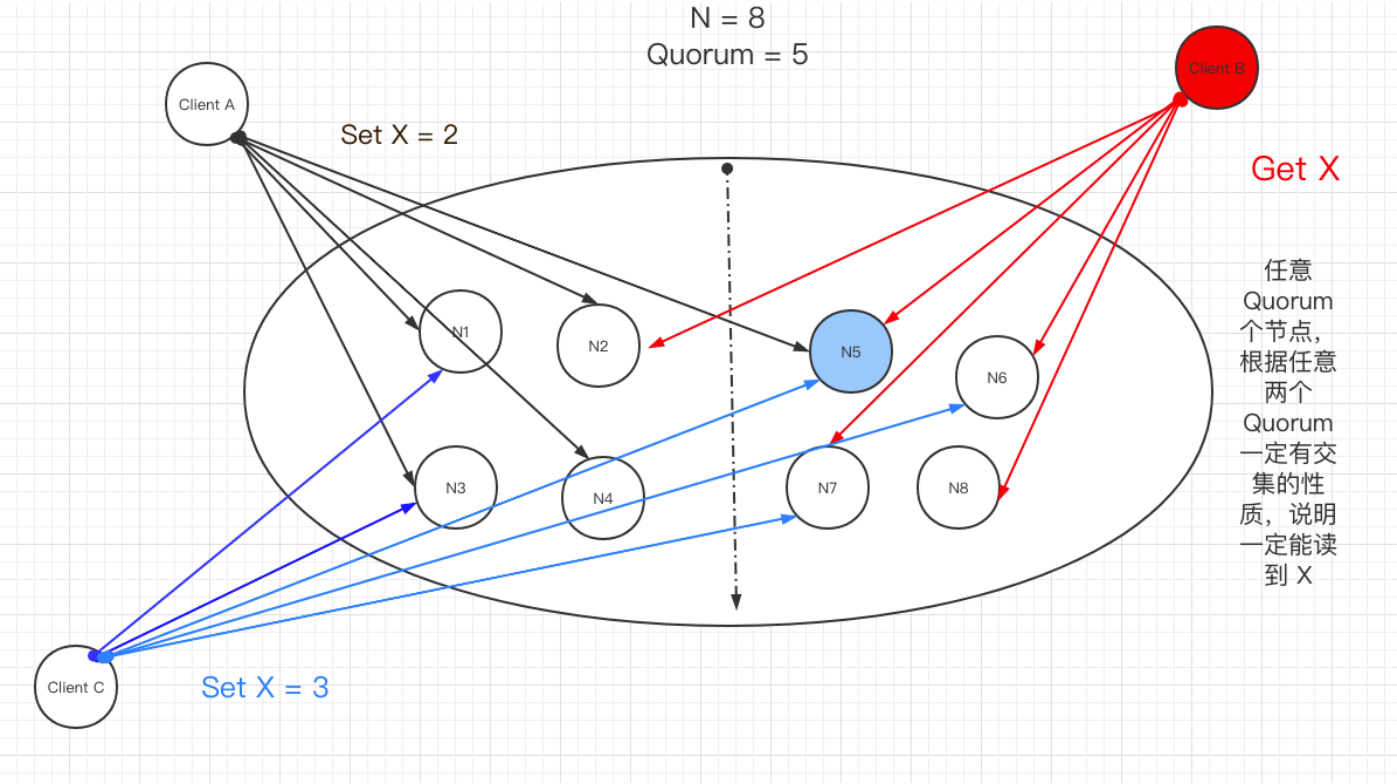
到此，我们解决了成功复制的问题。

值是否唯一

首先，策略我们已经确定，要保证复制成功，需要先多数派写，再多数派读。

扩展一下刚才的抽象，考虑有多个客户端并发的来写（SET X），那么读到的值将不唯一。

这里我们依然不考虑正确性的问题，我们只考虑唯一性的问题。



再重复一遍系统模型的抽象

- 只有两个操作 SET 和 GET
- 只操作一个数据 X，不考虑其他数据
- 不考虑 X 的正确性，只考虑唯一性
- 允许并发 SET X，但最终目标是 X 值唯一

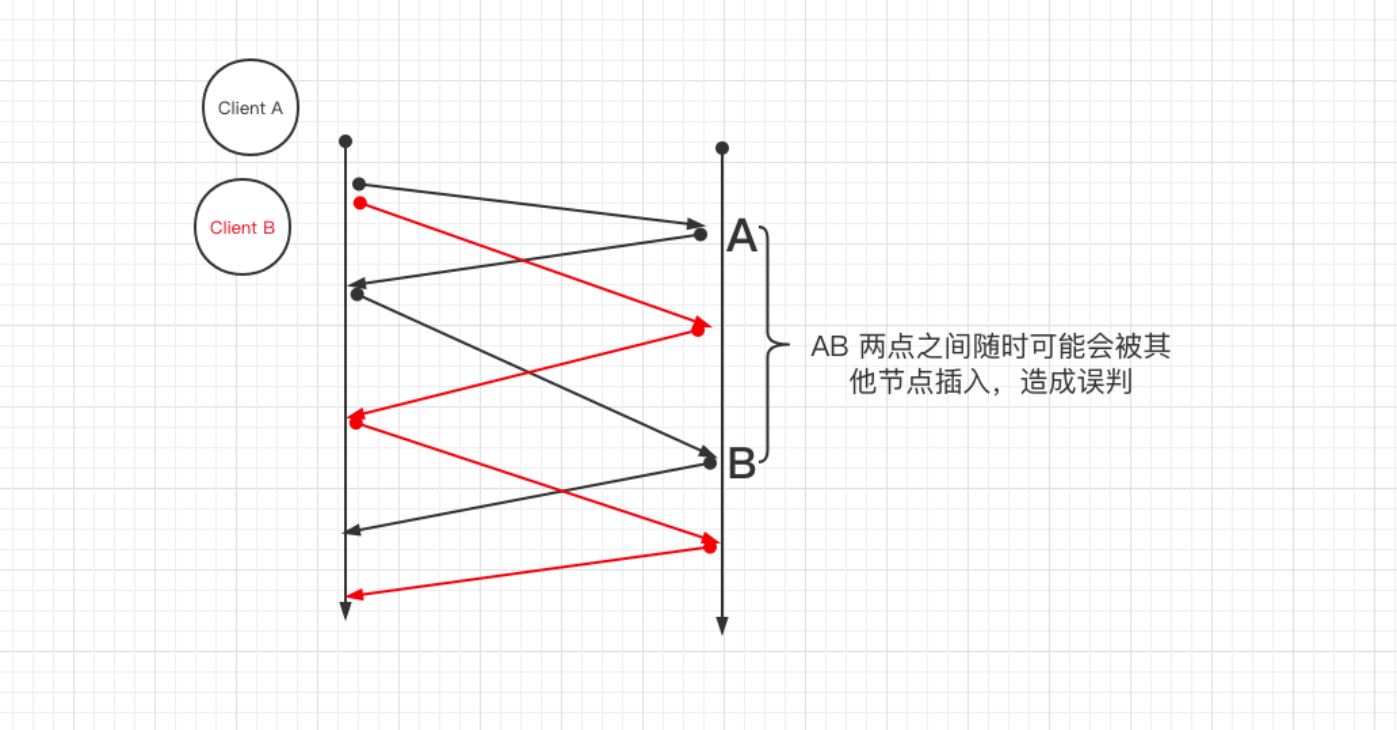
抽象是很重要的，能够帮我们简化模型，思考本质的问题。

如何解决并发的问题，或者说多个 client SET X 的问题?

如何解决并发 SET X

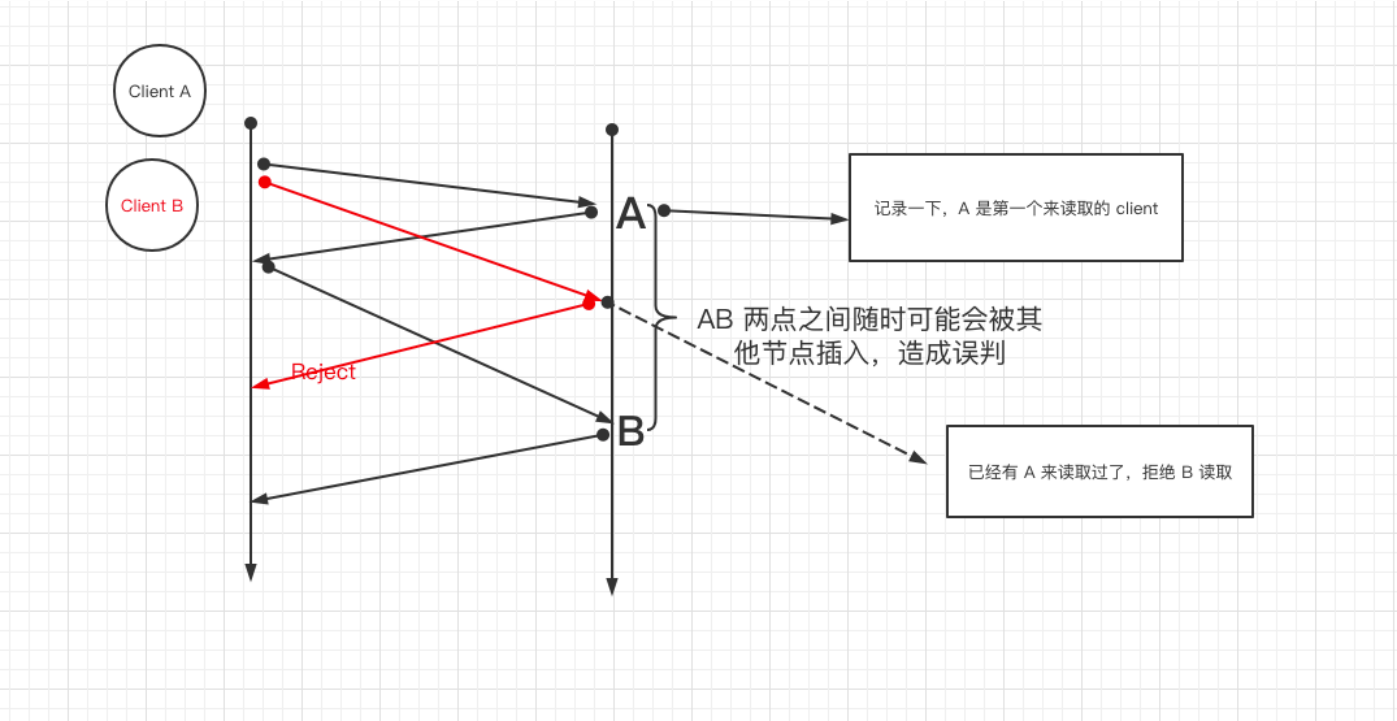
如果 SET X 前运行一次多数派读，知道 X 的值可能已经有别人写入了，那么就不写，如果还没有人写入，那么就写入。

道理很简单，但问题是这涉及到两次 rtt，所以其他 client 就有可能插入进来。



怎么解决呢?

拒绝其他写前读取



这里依然有多数派读的要求，Client A 必须收到 a quorum (>n/2) 个响应才认为本次写前读取 true，后续可以放心大胆的写入；否则判定为 false,认为已经有其他节点优先读取了，放弃后续的写。

有了这个保证，那么 Client A 在第二个 rtt 就可以放心的写了。（参考上面的图）

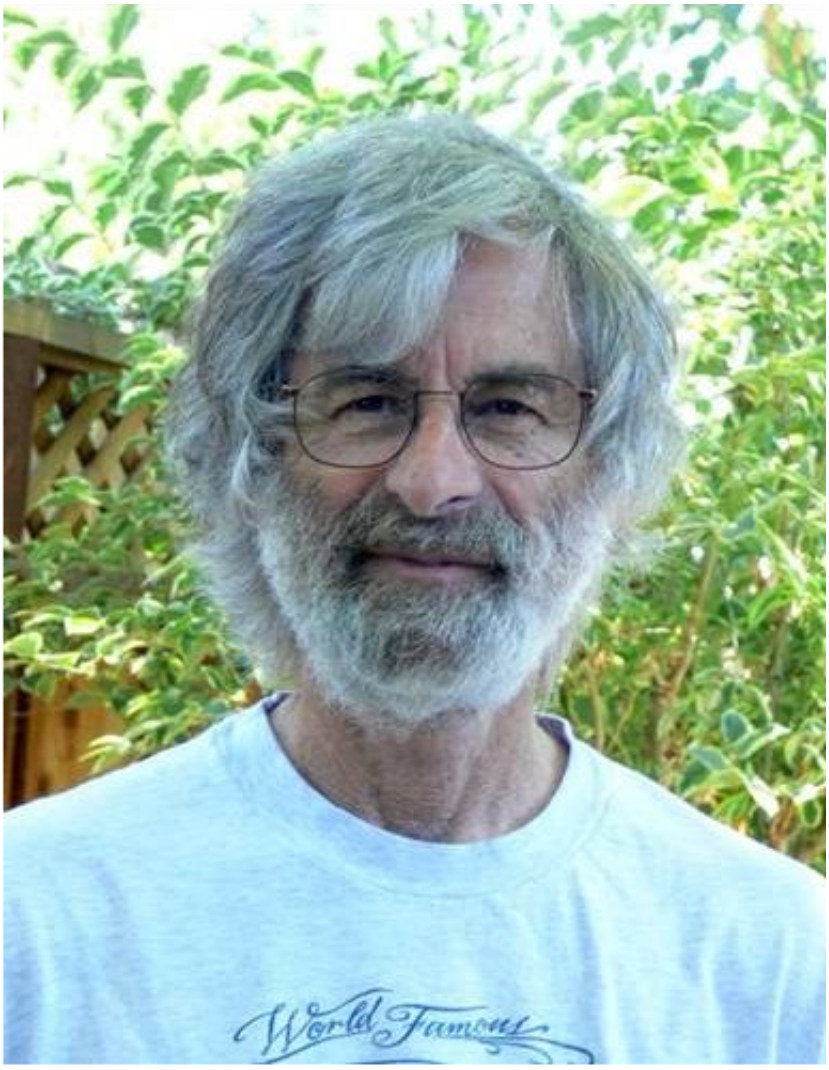
到这里其实就已经讲到 paxos 算法的核心了。(终于讲到 paxos 了）。

也许，我们还可以问两个问题:

- 如果第一个 rtt 失败了呢？（思考）
- 如果第二个 rtt 失败了呢？（思考）

初识 paxos

图灵奖大牛 Leslie Lamport （莱斯利·兰伯特）



来自知乎：

Lamport在分布式系统理论方面有非常多的成就，比如Lamport时钟、拜占庭将军问题、Paxos算法等等。除了计算机领域之外，其他领域的无数科研工作者也要成天和Lamport开发的一套软件打交道：著名的LaTeX。这是目前科研行业应用最广泛的论文排版系统，名字中的"La"就是指Lamport。

实际上Paxos在1990年就被提出了。当时Lamport写了一篇名为《The Part-time Parliament》的论文，在这篇文章中作者讲了一个虚构的故事。这个故事发生在希腊的神话中的一个名叫Paxos的岛屿（也就是算法名称的来由），作者将分布式一致性的问题比喻为岛上的立法机构如何对一项决议达成一致的问题。Lamport本来是觉得用故事加以描述更易理解；但其结果完全相反。这篇文章当时的评审几乎没有人看懂，只有一位名叫Butler Lampson的计算机科学家读懂了故事，并意识到这是一篇解决分布式一致性问题的论文。当然这篇论文就被埋没了多年，原文1998年才得以发表，后来Lamport也又重新“正儿八经”地写了一篇《Paxos Made Simple》。

https://lamport.azurewebsites.net/pubs/paxos-simple.pdf

Abstract

The Paxos algorithm, when presented in plain English, is very simple.

liveness and safety

from wiki:

In order to guarantee safety (also called "consistency"), Paxos defines three properties and ensures the first two are always held, regardless of the pattern of failures:

Validity (or non-triviality)
Only proposed values can be chosen and learned.[15]
Agreement (or consistency, or safety)
No two distinct learners can learn different values (or there can't be more than one decided value)[15][16]
Termination (or liveness)
If value C has been proposed, then eventually learner L will learn some value (if sufficient processors remain non-faulty).[16]
Note that Paxos is not guaranteed to terminate, and thus does not have the liveness property. This is supported by the Fischer Lynch Paterson impossibility result (FLP)[6] which states that a consistency protocol can only have two of safety, liveness, and fault tolerance. **As Paxos's point is to ensure fault tolerance and it guarantees safety, it cannot also guarantee liveness**.

(Basic/Classical) Paxos

开始之前，记住刚才讲到的系统抽象（实际上 paxos 也仅仅只是解决了这个问题，想清楚这个系统抽象，理解起来会容易得多，避免走弯路）

- 只有两个操作 SET 和 GET
- 只操作一个数据 X，不考虑其他数据
- 不考虑 X 的正确性，只考虑唯一性
- 允许并发 SET X，但最终目标是 X 值唯一

paxos 定义了 3 种角色：

- Proposer: 发起提案 value（一个提案可以理解为一个操作，或者一个值）
- Acceptor: 接受一个 value 或者驳回
- Learners: 当某个提案被大多数 acceptor 接受后，则认为 value 被选定，然后复制 value

> Leaners 暂时不重要，可以不用关心

Tips: 这里有一些概念可能比较难以理解，由于中英文的差异，阅读论文的时候可能会比较苦恼

举个例子：
如果你阅读 paxos 的一些文档，可能会经常看到 ”value be choosen" 等一些概念，这里的 choosen 就可以理解为值已经确定，唯一。

主要过程分为两个阶段（如同上述讨论的两个 rtt）

paxos phase1

phase 1a: Prepare

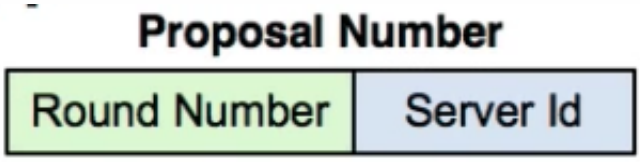
先来看几个概念：

- proposal number n: 全局唯一且递增，用来给一个提案编号
- proposal v: 一个提案，通常用 value(v) 表示

proposer 发起 "prepare" 请求，请求里携带一个 proposal number n，这个 n 要比它之前的 prepare 消息使用的值大。然后它发送（广播）这个 "prepare" 消息给 a Quorum of Acceptors（大多数的 acceptors)。

值得注意的是，这个 prepare 消息通常不包含具体的提案 v。（正如上面的写前读取，不需要携带将要写的值 x)

- proposer 需要持久化它看到的最大的 proposal number



phase 1b: Promise

acceptor 收到从某个 proposer 发过来的 prepare 消息，拿到消息里的 proposal number n.

先看几个概念：

- **minProposal: 之前已经接收到的 prepare 消息里的最大 proposal number**
- acceptedProposal: 之前已经接受过的 proposal number

- acceptedValue: 之前已经接受过的 proposal

1. 如果 n > minProposal

- 那么 acceptor 必须返回一个响应，同时做出两个 promise
  - promise1: **它将拒绝后续所有 proposal number <= n 的其他 prepare 请求**
  - promise2: **它将拒绝后续所有 proposal number < n 的其他 accept 请求（第二阶段会讲到）**
- 如果 acceptor 之前已经接受过一个 proposal，那么返回的响应里还会携带 (acceptedProposal, acceptedValue)，否则返回(nil, nil)
- 同时 acceptor 记录 minProposal = n

2. 如果 n <= minProposal

- 拒绝这个 Prepare 消息（**然后这个 proposer 会以更大的 proposal number 发起 prepare**）

这里要注意:

- acceptor 需要持久化 (minProposal, acceptedProposal, acceptedValue)，以防止崩溃或者重启

第一阶段对比之前的讨论，就相当于之前的写前读取，用来判断谁准备写入。但是区别是： 上面的例子中只接受第一个 client 的写前读取，后续其他的 client 的写前读取全部拒绝；而这里的 acceptor 其实是允许接收多个 prepare(写前读取) 的，**想想看为什么？** 相同的地方是都需要对写前读取做记录 (minProposal = n)

paxos phase2

phase 2a: Accept

如果 proposer 收到了 a Quorum of Acceptors返回的 promise，那么说明他可以开始提案了。

如果 promise 里含有 (acceptedProposal, acceptedValue)，那么**就放弃自己原本的提案**，从返回的这些 promise 里挑出 acceptedProposal 最大的 acceptedValue 作为本次的提案 value。

（很重要，我自己初看的时候对这里放弃自己的提案很是想不通，还是那句话，把系统抽象到最简模型，提案的具体值不重要，重要的是达成一致）

这里的理解： accepted 并不表示某个提案 v 被确定了（be choosen)，之所以放弃自己的提案，其实是相当于继续了未完成的 paxos 过程。后面会讲到。

如果返回的 promise 里都没有 acceptedValue 值，那么才使用自己的提案 value （说明这是第一次提案）

然后 proposer 发起 "Accept" 请求 (n, v)，广播给大多数 acceptor。

phase 2b: Accepted

acceptor 收到 "Accept" 消息后(n, v)，取出里面的 proposal number n.

1. 如果 n >= minProposal

- acceptor 更新 acceptedProposal = minProposal = n
- acceptor 更新 acceptedValue = v
- acceptor 响应 "Accepted" 消息，消息里携带 minProposal

2. 如果 n < minProposal:

- 拒绝这个 "Accept" 请求，同样也会回复一个响应，消息里携带 minProposal

注意，acceptor 是可以接受多次 acceptedValue 的，只要满足上面的条件。

当 proposer 收到了大多数的 acceptor 返回来的 "Accepted" 消息，则认为这个提案已经确定(be choosen)。

如果返回的消息有任何 minProposal >n，则重新以更大的 proposal number n执行 paxos.

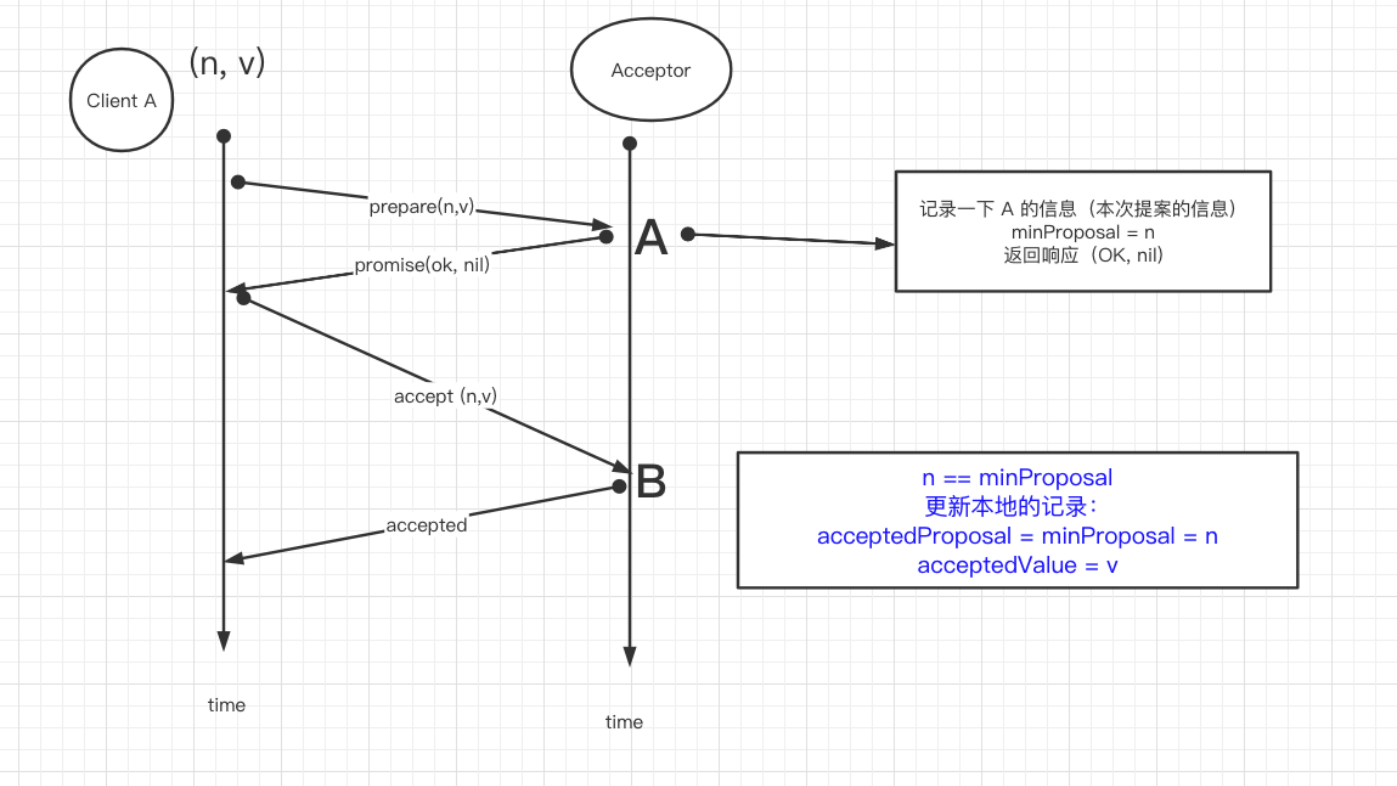
接下来，就是 **Leaner 发挥作用的时候了，Leaner 就会复制这个提案**。实际的做法可能有多种，比如 Leaner 和 proposer 集成到一个角色，再通知其他的 Leaner 这个被 choosen 的提案v; 或者 Leaner 自己执行一遍 paxo是就能知道被 choosen 的提案。

讨论一下上面关于 n 与 minProposal 的大小关系：

- n == minProposal
- n < minProposal
- n > minProposal

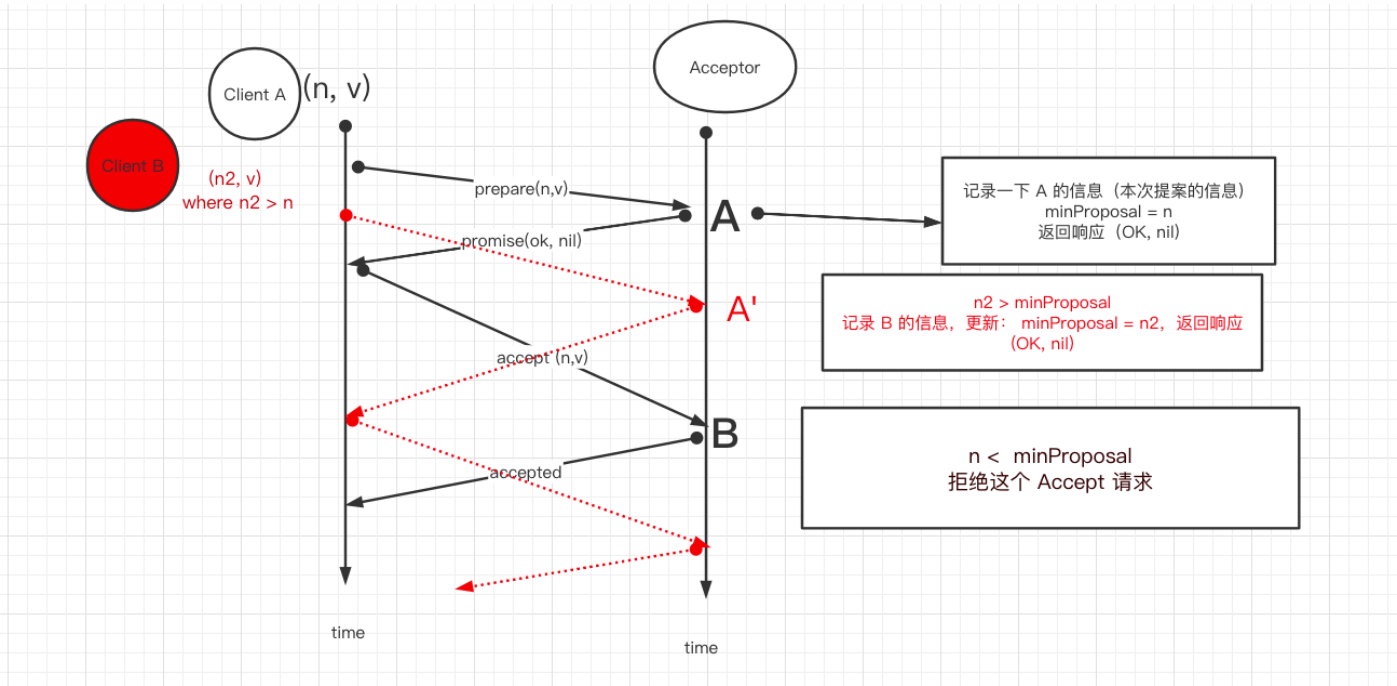
phase 2b: Accepted (n == minProposal)

- **n == minProposal:** 这个很好理解，假设 **paxos** 系统只有一个 **proposer**，那么到了 **accept** 这里，必然 **minProposal** 就是 **prepare** 消息里的 **n**，此次 **accept** 消息自然也是 **n**



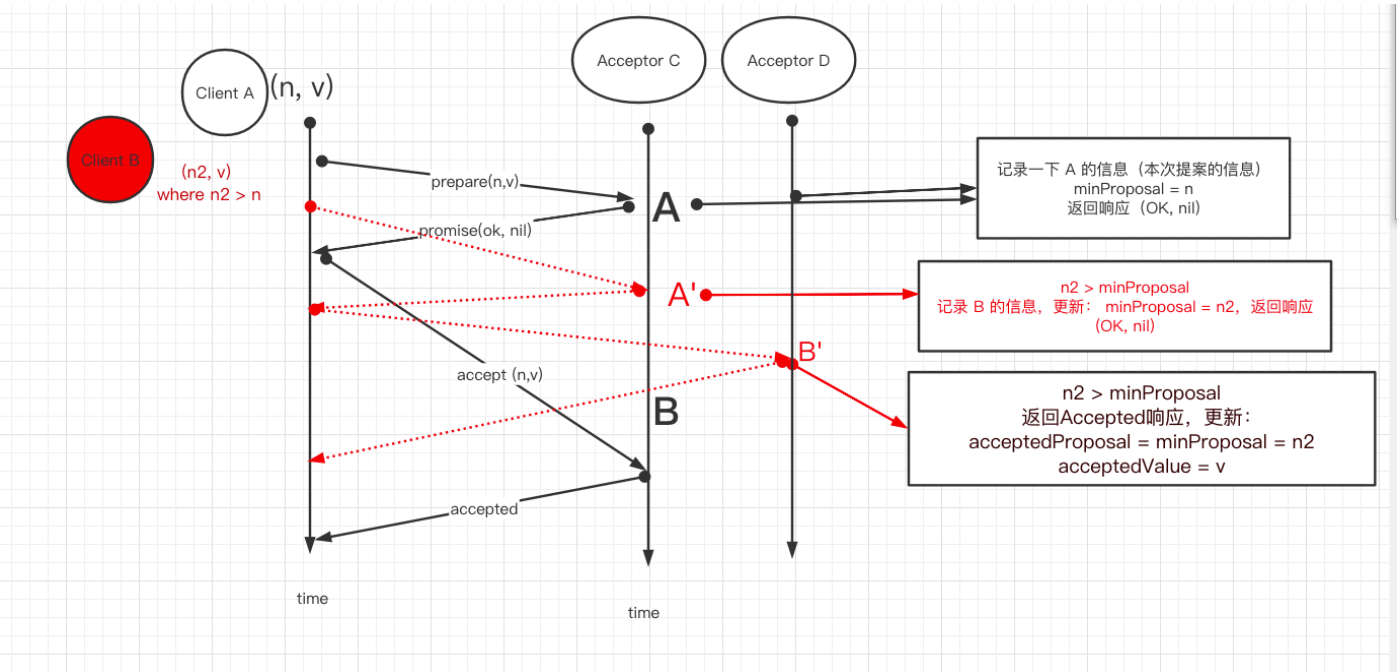
phase 2b: Accepted (n < minProposal)

- **n < minProposal:** 也好理解，如果 **proposerA** 成功执行完 **prepare** 消息后，并确定自己可以开始写入之前，有 **ProposerB** 发起了 **n** 更大的 **Prepare** 消息，那么 **acceptor** 处的 **minProposal** 则会更新到这个值，当 **ProposerA** 的 **Accept** 过来时， **n < minProposal**

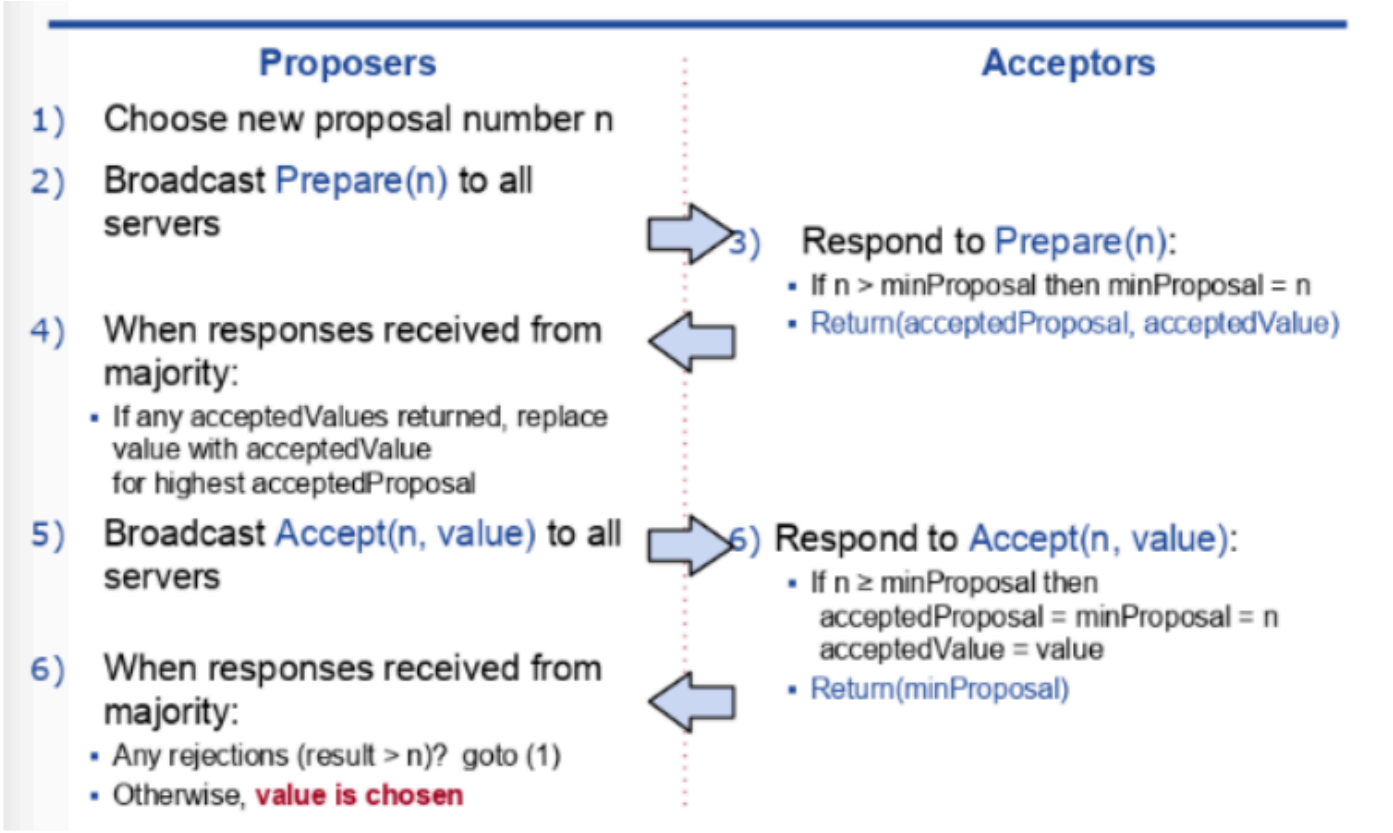


phase 2b: Accepted (n > minProposal)

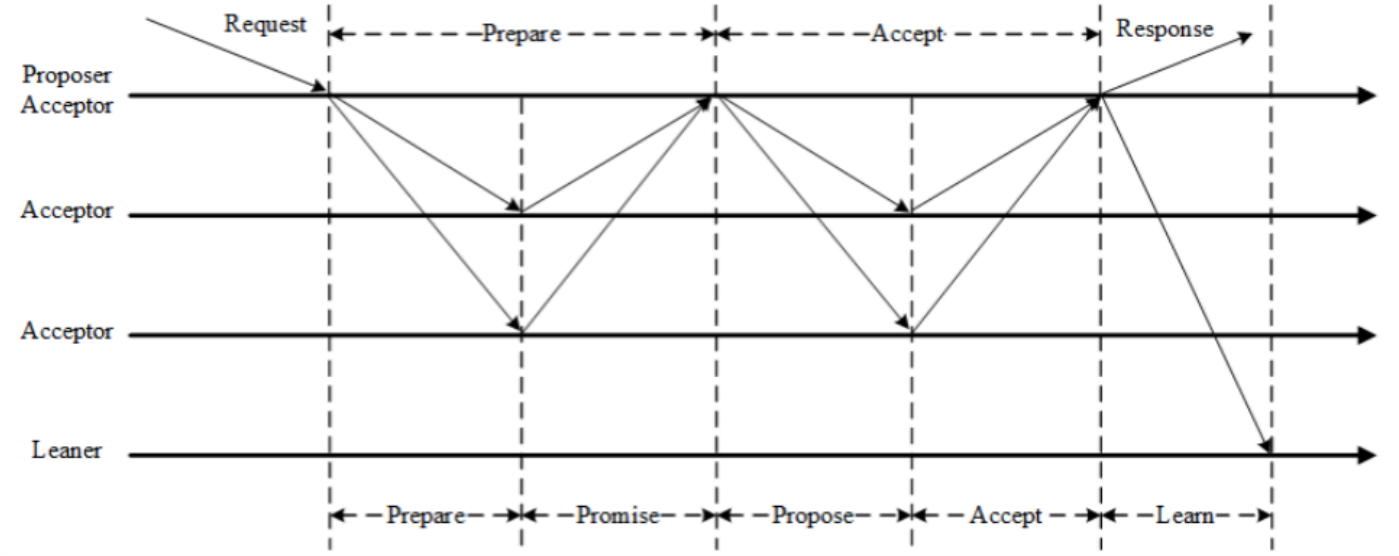
- **n > minProposal:** 假设 **proposerA** 已经成功发起 **prepare** 消息了，但是在发起 **Accept** 请求这段空隙时间，有 **proposerB** 又以更大的 **proposal number n** 成功发起了 **prepare** 消息，那么大多数的 **Acceptor** 的 **minProposal** 必然会更新到这个更大的 **n**，然后又抢在 **proposerA** 前发起了 **Accept** 消息，假设这个 **AcceptB** 消息被上一步没有收到 **prepareB** 的 **acceptor** 收到了，那么这些 **acceptor** 的 **minProposal** 依然记录的是 **proposerA** 的**n**，自然本次的值更大，**n > minProposal**



paxos 完整过程



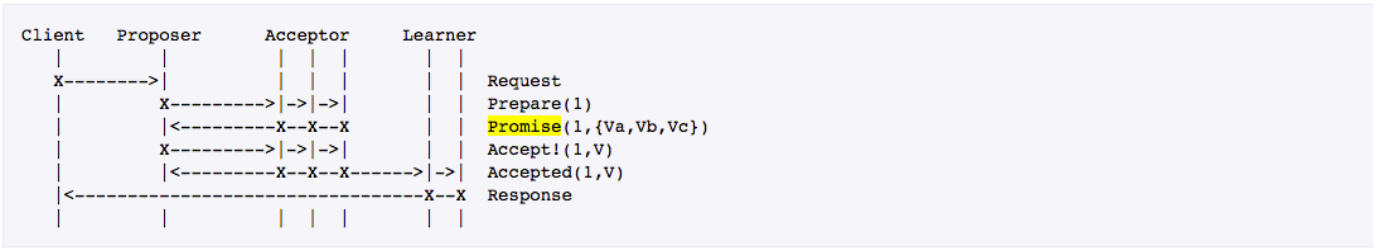
或者类似 pbft 的流程图：



到这里，其实已经把 paxos 的核心算法讲完了。

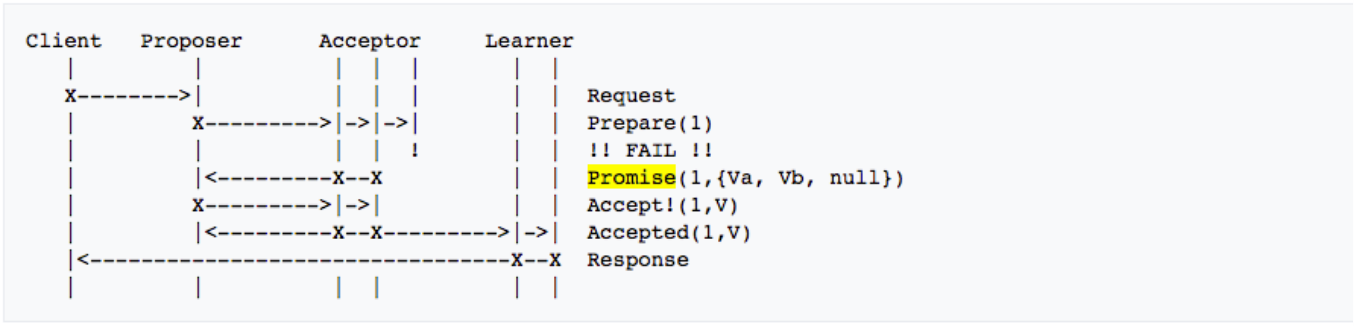
来讨论一下涉及到的几个过程中可能出现的场景。

Basic Paxos without failures

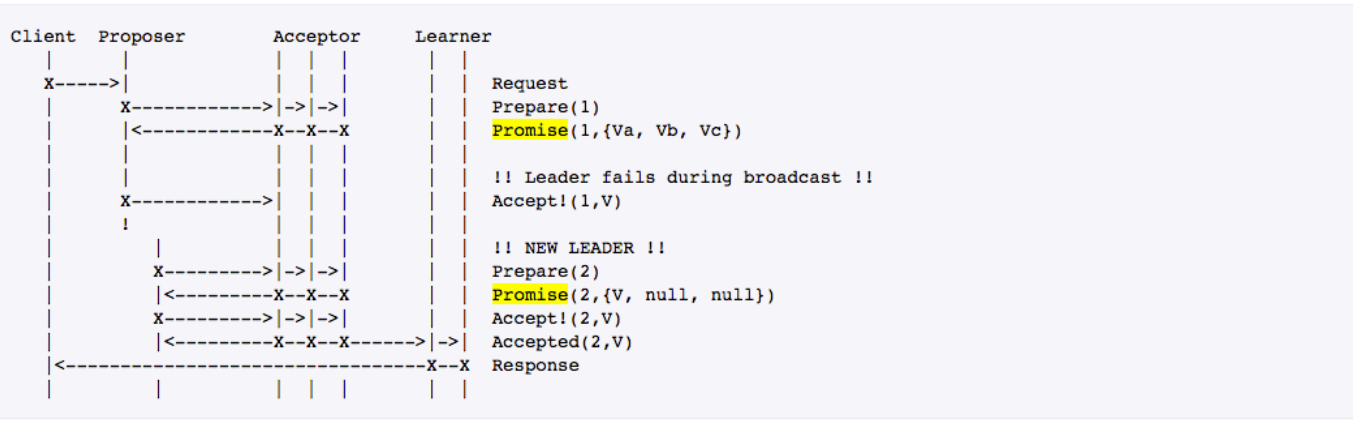


Here, V is the last of (Va, Vb, Vc).

Basic Paxos when an Acceptor fails



Basic Paxos when a Proposer fails



Basic Paxos when multiple Proposers conflict (livelock 活锁)

对比 deadlock，叫法很形象

```
Client   Proposer      Acceptor     Learner
   |        |            |  |  |       |  |
   X-------->|            |  |  |       |  |  Request
   |      X------------>|->|->|       |  |  Prepare(1)
   |      |<------------X--X--X       |  |  Promise(1,{null,null,null})
   |      !            |  |  |       |  |  !! LEADER FAILS
   |         |         |  |  |       |  |  !! NEW LEADER (knows last number was 1)
   |         X--------->|->|->|       |  |  Prepare(2)
   |         |<---------X--X--X       |  |  Promise(2,{null,null,null})
   |      |  |         |  |  |       |  |  !! OLD LEADER recovers
   |      |  |         |  |  |       |  |  !! OLD LEADER tries 2, denied
   |      X------------>|->|->|       |  |  Prepare(2)
   |      |<------------X--X--X       |  |  Nack(2)
   |      |  |         |  |  |       |  |  !! OLD LEADER tries 3
   |      X------------>|->|->|       |  |  Prepare(3)
   |      |<------------X--X--X       |  |  Promise(3,{null,null,null})
   |      |  |         |  |  |       |  |  !! NEW LEADER proposes, denied
   |      |  X--------->|->|->|       |  |  Accept!(2,Va)
   |      |  |<---------X--X--X       |  |  Nack(3)
   |      |  |         |  |  |       |  |  !! NEW LEADER tries 4
   |      |  X--------->|->|->|       |  |  Prepare(4)
   |      |  |<---------X--X--X       |  |  Promise(4,{null,null,null})
   |      |  |         |  |  |       |  |  !! OLD LEADER proposes, denied
   |      X------------>|->|->|       |  |  Accept!(3,Vb)
   |      |<------------X--X--X       |  |  Nack(4)
   |      |  |         |  |  |       |  |  ... and so on ...
```

是否还有很多疑问?

到了这里，你是否还有很多疑问?

我自己看到这的时候，我的疑问就是假设提案v 已经被确定了，即 be choosen, acceptor 保存的 acceptedValue 和 acceptedProposal 难道要一直保存吗? 如果我打算发起第二个提案 v', 按照上面的算法，promise 还是会把 (acceptedValue, acceptedProposal) 返回来，我始终无法发起第二个提案 v'.

更直白的话是，实际的系统肯定是一些连续的不同的提案，比如add, sub, jump, mov, cmp 等等，或者举一个更容易理解的数据库的例子，我们是持续的往数据库里写入数据的：

```
x = 1 ; y = 2; z= 100; x + 100; z - 10 ...
```

当我们成功发起 x=1 这个提案之后，按照 paxos 的流程发起第二个提案 y=2，还是会得到 x=1 这个提案的值。

basic/classical paxos 仅仅只是解决了一个提案（一次操作）的数据一致性问题，这也是纯理论的 basic/classical paxos 解决的核心问题。

你是否跟我一样? 如果你能想到这里，说明你已经理解了上面的算法！（👏👏）

那么应用到实际中，怎么解决呢?

从理论到工程实践

根据上面的讨论我们可以知道，执行一个 paxos 流程可以解决一个提案的一致性问题，如果想要发起第二个提案，就势必要处理 (acceptedValue, acceptedProposal) 的问题。

在 Acceptor 上记录了 3 个值 (minProposal, acceptedValue, acceptedProposal)，当第一个提案（比如 x=1) 成功被 choosen 之后，Acceptor 上必须要有一个机制去清除掉这 3 个值，就像最初什么也没发生过时的状态，此时发起第二个提案（y=2） 就和发起第一个提案(x=1) 一样了。

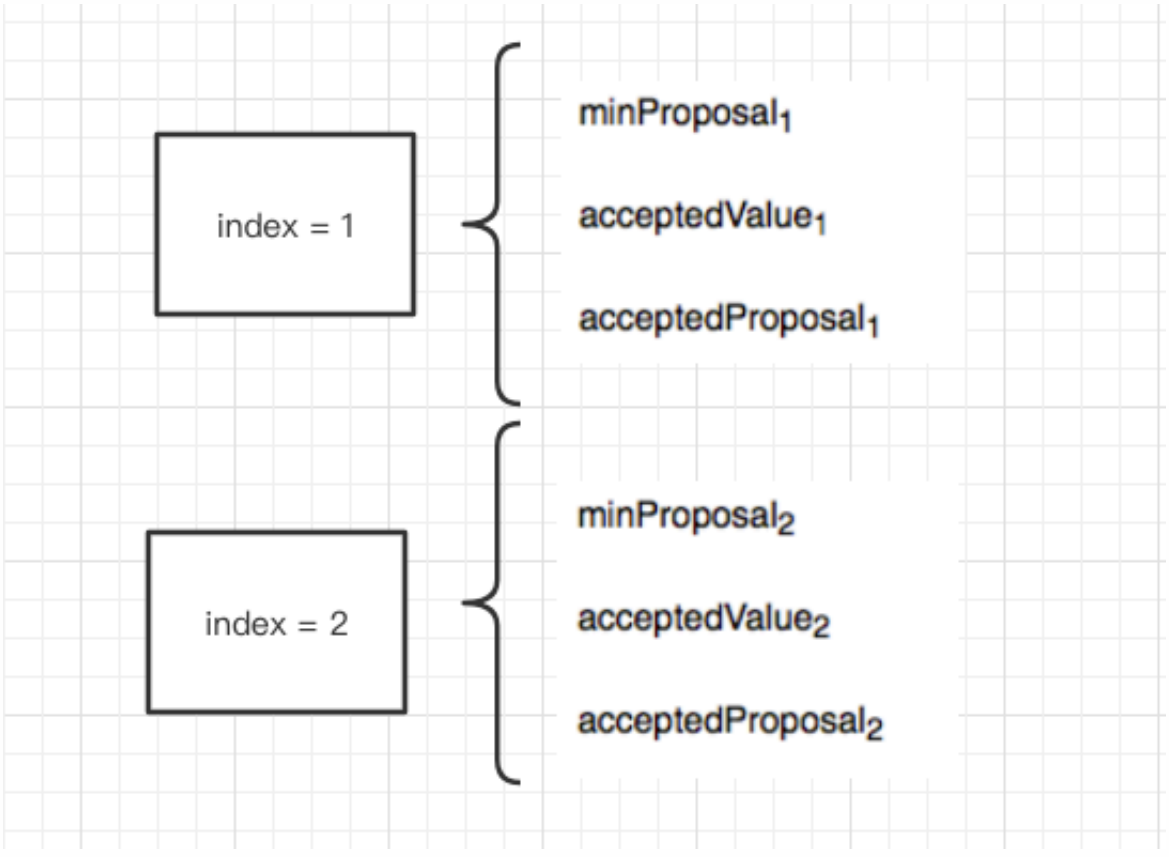
那么问题就来了，Acceptor 什么时候清除 (minProposal, acceptedValue, acceptedProposal) 呢？它必须知道提案已经被 choosen 了，并且要记录下这个已经被 choosen 的提案 (x=1);否则不能清除。

所以当一个提案(x=1) 被 choosen 之后，**发起提案的 proposer 需要广播这个 choosen 结果给 Acceptor，直到大多数 Acceptor 返回成功**。（先提一下：这里这个过程是不是和 raft 里 leader 日志复制的第二个过程类似，即 leader 本地 commit 之后，广播给所有 follower，告诉他们这个 command 已经处于 commited 状态了，然后 follower commited command）

Ok, 到这里 Acceptor 已经知道提案被 choosen(commited) 了，实际做法是记录下这个 choosen 的提案(x=1), 然后它可以放心大胆的清除(minProposal, acceptedValue, acceptedProposal) 这 3 个值了。那么当 proposer 发起第二个提案 y=1后，Acceptor 就能正常处理了。

但， Acceptor 怎么区分这两个不同的提案(x=1 和 y=1） 呢？（靠 proposal number n ?)

我们还得给每一个不同的提案一个编号，或者说 index，即每一个不同的提案具有唯一的序号 index。所以一个 proposer 发起一个提案(prepare 消息）时需要包含 (n,v, index)，那么 Acceptor 这里呢，记录的 3 个值 (minProposal, acceptedValue, acceptedProposal) 和每一个 index 唯一绑定，或者说每一个不同的 index 有自己单独的 3 个值, 即（$\text{minProposal}_{\text{index}}$, $\text{acceptedValue}_{\text{index}}$, $\text{acceptedProposal}_{\text{index}}$）



是不是可以并发了呢?

multi-paxos

经过上面的讨论，我们看到如果给每一个不同的提案 `x = 1 ; y = 2; z= 100; x + 100; z - 10 ...` 做一个编号 index，我们就能够独立的执行 paxos 协议，实现我们实际工程中的连续写操作。

由纯理论到工程实践了！！！！

我们把上面每个运行独立 paxos 协议的实体称为一个 **paxos instance**，每个 paxos instance 独立互不影响。实际的工程中我们通常也会把 proposer/acceptor/leaner 3 个角色放到一个节点上，即一个 paxos 节点具有多重身份，每个身份可以作为一个线程运行，方便数据共享(比如 choosen value)。

性能问题的解决:

- 两阶段的 paxos 性能较差，如何减少 rpc 请求数?
- livelock(活锁) 问题的解决?

据此 multi-paxos 提出一个 Leader 机制，避免 proposer 的冲突，这样一样，所有发起的提案全部由这个 Leader 来发起（至于如何选取 leader，问题不大，有很多做法，就如同 raft 的 leader 选举一样），同时也解决了 livelock 问题。

那么既然有了 leader，第一阶段的 prepare/promise 还有必要吗? 第一阶段的 prepare/promise 到底是干嘛的，上面其实已经讲的很清楚了，就是确认哪一个 proposer 准备写，那么既然有了 leader，自然这个过程就可以省略了。

所以 multi-paxos 就只有 accept/accepted 过程了。

那么我们再重新思考一下：basic paxos 的两个阶段到底解决了什么问题?

- 第一阶段(prepare/promise): 解决了选择唯一 proposer 的问题
- 第二阶段(accept/accepted): 解决了提案被大多数 acceptor 接受的问题(accepted)，结果最初只有发起提案的 proposer 知道

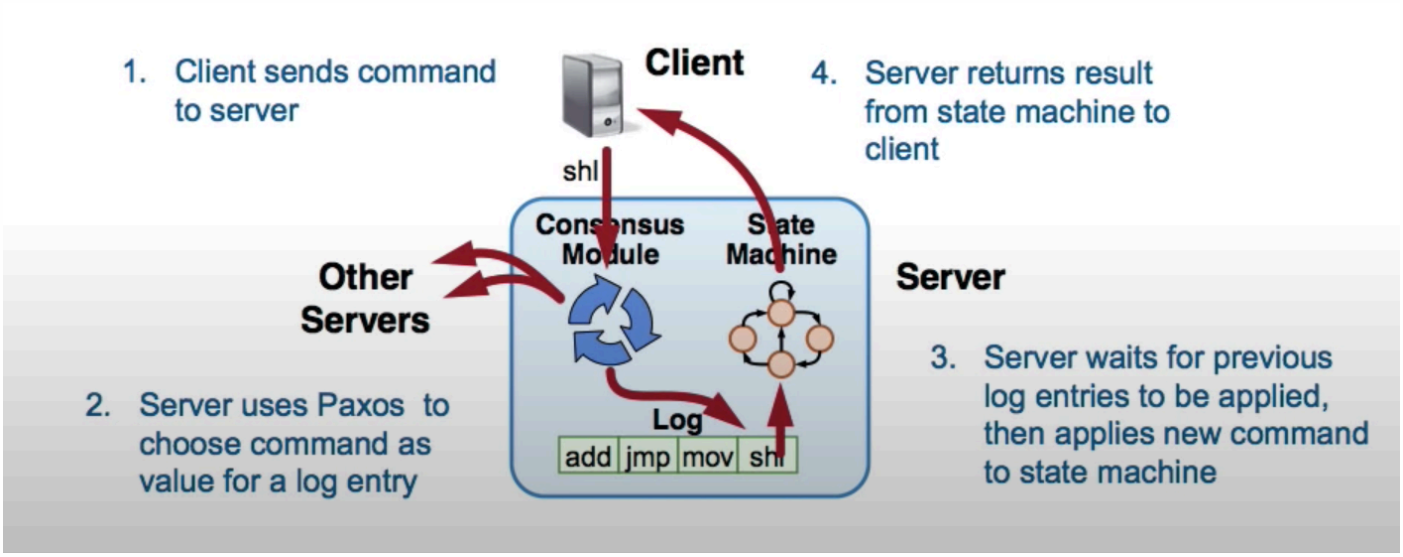
通过 multi-paxos leader 机制，我们简化了第一阶段:

- 直接发起 accep/accepted: 最终大多数 acceptor 会接受这个提案，并且 leader 汇总结果后知道了大多数 acceptor 已经接受了这个提案，于是被 choosen
- 然后广播这个 choosen 结果给所有 acceptor，acceptor 再更新这个提案为 choosen（前面提到过）

这两个过程是不是和 raft 如出一辙，在 raft 里叫 log replication? 😁😁 **multi-paxos ≈ raft**.

状态机

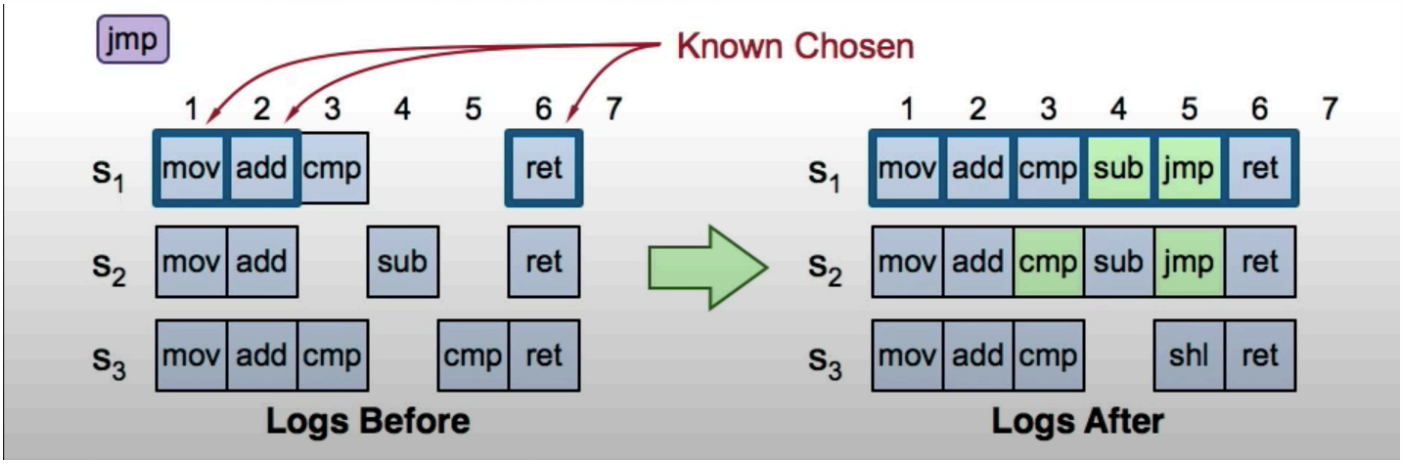
先看一个状态机：



multi-paxos 多提案过程

client 发起一个新的 command jmp，进入共识模块执行 multi-paxos 算法，会去找到最小的已经被 choosen(commited) 的 index， 找到 index = 3（放弃自己的提案 jmp，执行 index=3 的 paxos)， and so on...

注意，实际可能同时并发发起 3 ~ 20 index 的提案。



multi-paxos 日志空洞

multi-paxos 是允许日志空洞的，也就是不连续的，每次 leader 并发发起 index 任意多个的不同提案，每个提案独立进行，所以会有成功和失败。

multi-paxos 幽灵复现日志问题

”幽灵复现日志“造成的原因就是”日志空洞“和 Leader 切换。

举个例子：

- 用户 A 发起一笔转账，超时没有等到转账结果
- 于是用户 A 再次查询一下是否转账成功，如果再次超时，用户 A 可能重新发起新的转账
- 但是最后结果是 A 成功执行了两次转账

multi-paxos Leader 切换

leader 选举的过程可以简单理解为某个节点 A 发起一轮 basic paxos（提案就是选 A 作为 leader)，最终提案被 choosen，广播给所有 acceptor，于是 leader 产生，并利用 lease 机制保持自己的 leader 身份，避免其他的 proposer 发起竞选 leader.

> lease 机制：即租约机制，声明 leader 有效期，在有效期内，不允许发起竞选 leader；超过 lease，随意进入选举。

leader 切换必然伴随日志不一致的问题，即当前 leader 的日志和前任 leader 的日志不一致。就有可能造成 client 查询的时候返回 false。详细就不展开讲了。

multi-paxos 总结

- 选了一个 leader，避免 proposer 竞争(避免livelock)
- 同时可以并发发起 index = 1,2,3,4... 的提案，每个提案独立运行 paxos 算法
- 每个提案被 accepted 之后记录到本地(例如 mysql binlog)
- 每个提案被 choosen 之后更新这个 index 的状态为 choosen（比如设置这个 index 对应的 $\text{minProposal}_{\text{index}} = \infty$）
- 被 choosen(commited) 的提案放到 state machine 里执行
- 如果同时并发发起 index = 1...10 的 共 10 个提案，中间有一些提案失败了(accepted but not choosen)，下一次会触发继续执行未完成的提案
- multi-paxos 允许提案空洞(不连续）（相反 raft 就必须是连续的，不允许日志空洞）

上面每个 index 构成的本地提案记录，类似于一个列表，raft 里就 log entry，index 称为 logid.

raft

先看一个动画：

http://thesecretlivesofdata.com/raft/

raft 协议和 multi-paxos 很像。

先不讨论 leader 选举的问题，应该很简单。就讨论 log replication(日志复制）的问题。raft 过程如下：

定义了 3 种角色：

- leader: 就是 leader
- follower: 系统中其他节点，接收 leader 消息
- candidate: follower 到 leader 转换的中间状态

过程如下：

- client 发起一个 command (redirect to leader)
- leader 广播这个日志到其余的 follower，称为 AppendEntries rpc (对比 multi-paxos 的 accept/accepted 过程）
- leader 收到大多数 follower 成功响应后，执行 apply entry， 即 commite log，然后广播给其他的 follower（对比 multi-paxos proposer 广播 choosen 的提案）
- leader 回应 client

raft leader election

很简单，每个 follower 持有一个计数器，比如 [100, 300]ms，在这段时间内只要收不到 leader 的 hearbeat，就认为 leader 挂掉（实际有可能是他自己出了问题），然后由 follower 状态转为 candidate 状态，开始竞选 leader.

每个节点只能投一票，如果这个 candidate 收到大多数节点的 vote，则成为 leader，更新任期号 term number.

异常情况：raft 多个节点同时竞选

没关系，只要达不成 quorum vote，就继续下一轮投票；

为了避免出现无休止的重复这个过程（类似 livelock)，每次重新开始竞选时，随机延迟一段时间，避免出现两个 candidate 竞选。

异常情况：脑裂

脑裂就是网络分裂，形成两个或者多个独立的孤岛网络，假设分裂成 A 和 B 网络。

- A 满足多数派条件
- B 不满足多数派条件
- A 和 B 可能会发生各自网络内部的 leader 重新选举，term 增 1

对于 B 来说，由于不满足多数派，故日志始终处于 uncommited 状态，所以是安全的；

对于 A 来说，正常进行 raft 共识；

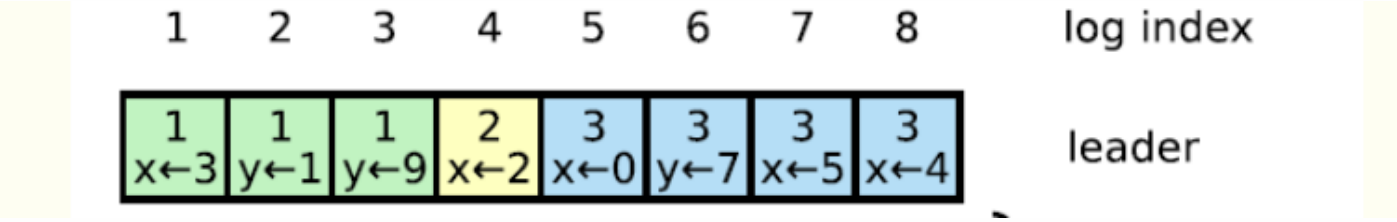
一旦网络恢复：

- A: 继续进行 raft
- B: 退化为 follower，日志回滚

所以也是安全的。

raft log replication

基本和 multi-paxos 一样，但区别是 raft 要求日志是连续的，不允许出现日志空洞。即如果 logid = index 处于 commited 状态，那么 logid <index 的一定都处于 commited 状态。



paxos vs raft

先看一下概念上的一些区别：

- raft leader 基本等同于 multi-paxos leader，但区别是 multi-paxos leader 不是强 leader 性质，实际上两个 leader 也可以（退化成 basic-paxos）
- raft follower 相当于 multi-paxos acceptor
- raft 两阶段的 rpc 分别是 appendEntries 和 applyEntries；分别对应 multi-paxos 中的 accept 和广播 choosen 的消息
- raft 日志 等同于 paxos 提案
- raft log entry 等同于 multi-paxos proposal index.

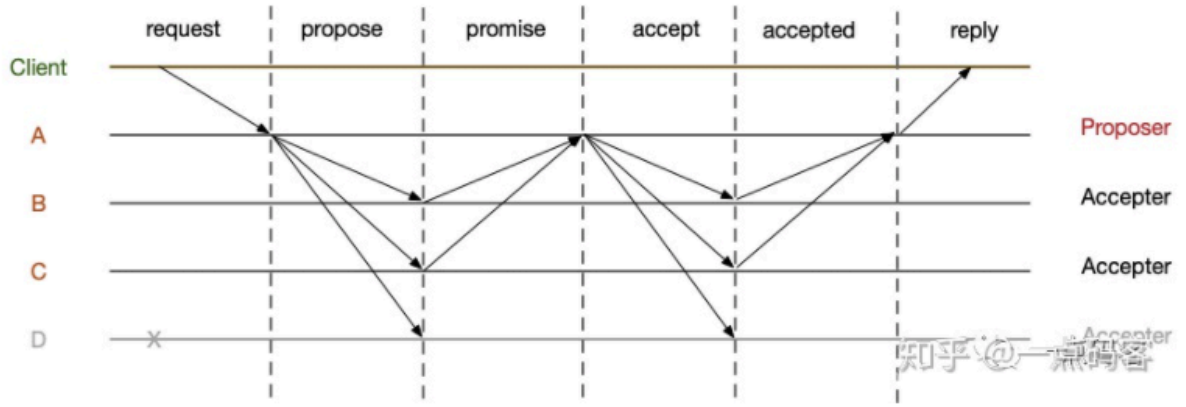
再看一下其他方面：

- multi-paxos 允许日志空洞； raft 不允许日志空洞
- leader election: multi-paxos 弱 leader 性质；raft 强 leader，且要求竞选 leader 的 follower 必须有较大的 logindex

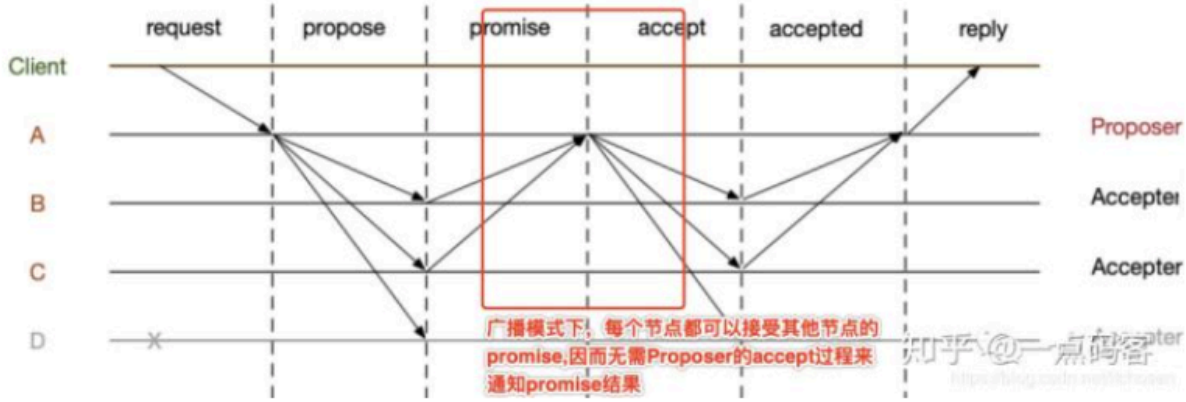
pbft

祭一张图吧

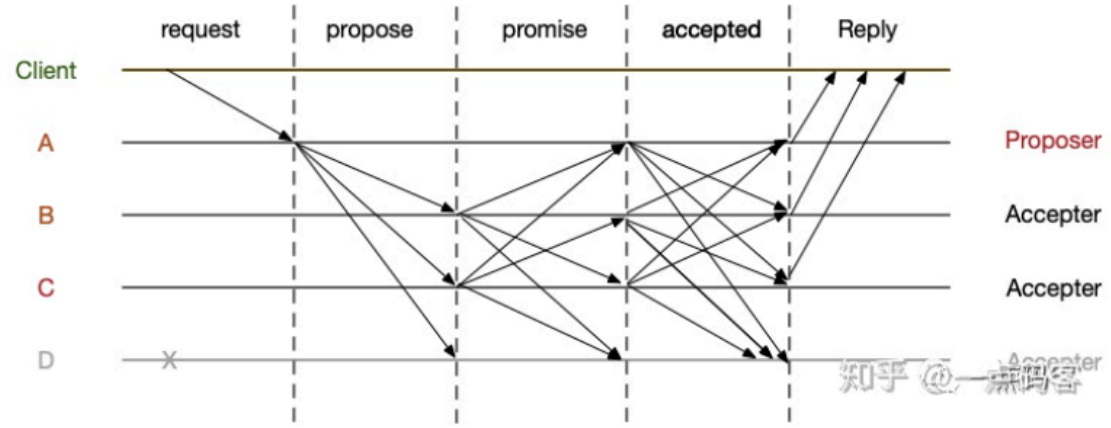
Paxos交互图如下



PBFT是通过广播进行的，因而上述交互可以缩减



进而可以简化为如下交互



上述流程更名后就变为