一、Paxos算法背景

Paxos算法是Lamport宗师提出的一种基于消息传递的分布式一致性算法，使其获得2013年图灵奖。

Paxos由Lamport于1998年在《The Part-Time Parliament》论文中首次公开，最初的描述使用希腊的一个小岛Paxos作为比喻，描述了Paxos小岛中通过决议的流程，并以此命名这个算法，但是这个描述理解起来比较有挑战性。后来在2001年，Lamport觉得同行不能理解他的幽默感，于是重新发表了朴实的算法描述版本《Paxos Made Simple》。

自Paxos问世以来就持续垄断了分布式一致性算法，Paxos这个名词几乎等同于分布式一致性。Google的很多大型分布式系统都采用了Paxos算法来解决分布式一致性问题，如Chubby、Megastore以及Spanner等。开源的ZooKeeper，以及MySQL 5.7推出的用来取代传统的主从复制的MySQL Group Replication等纷纷采用Paxos算法解决分布式一致性问题。

然而，Paxos的最大特点就是难，不仅难以理解，更难以实现。

二、Paxos算法流程

Paxos算法解决的问题正是分布式一致性问题，即一个分布式系统中的各个进程如何就某个值（决议）达成一致。

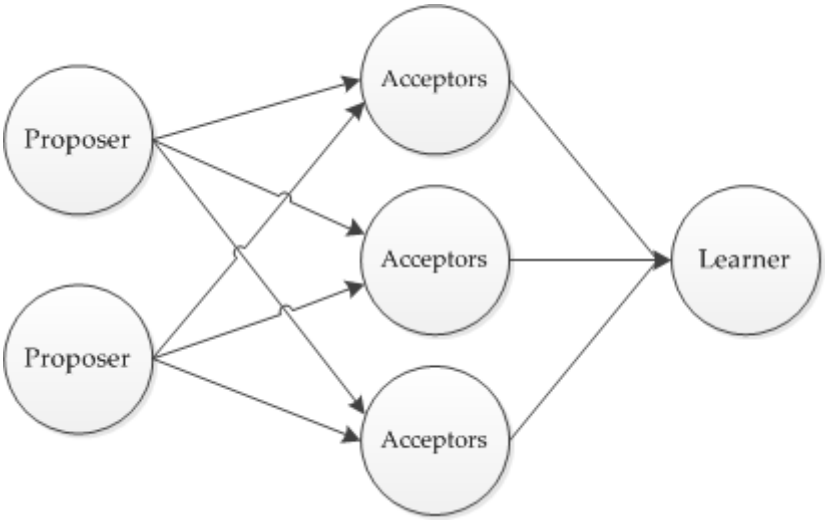
Paxos算法运行在允许宕机故障的异步系统中，不要求可靠的消息传递，可容忍消息丢失、延迟、乱序以及重复。它利用大多数 (Majority) 机制保证了2F+1的容错能力，即2F+1个节点的系统最多允许F个节点同时出现故障。

一个或多个提议进程 (Proposer) 可以发起提案 (Proposal)，Paxos算法使所有提案中的某一个提案，在所有进程中达成一致。系统中的多数派同时认可该提案，即达成了一致。最多只针对一个确定的提案达成一致。

Paxos将系统中的角色分为提议者 (Proposer)，决策者 (Acceptor)，和最终决策学习者 (Learner):

- **Proposer**: 提出提案 (Proposal)。Proposal信息包括提案编号 (Proposal ID) 和提议的值 (Value)。
- **Acceptor**：参与决策，回应Proposers的提案。收到Proposal后可以接受提案，若Proposal获得多数Acceptors的接受，则称该Proposal被批准。
- **Learner**：不参与决策，从Proposers/Acceptors学习最新达成一致的提案（Value）。

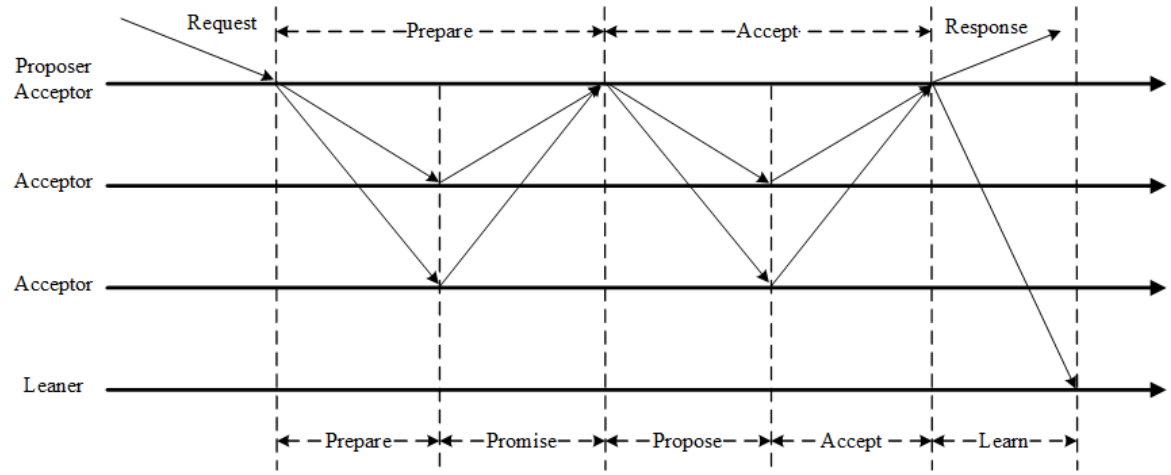
在多副本状态机中，每个副本同时具有Proposer、Acceptor、Learner三种角色。



Paxos算法中的角色

Paxos算法通过一个决议分为两个阶段（Learn阶段之前决议已经形成）：

1. 第一阶段：Prepare阶段。Proposer向Acceptors发出Prepare请求，Acceptors针对收到的Prepare请求进行Promise承诺。
2. 第二阶段：Accept阶段。Proposer收到多数Acceptors承诺的Promise后，向Acceptors发出Propose请求，Acceptors针对收到的Propose请求进行Accept处理。
3. 第三阶段：Learn阶段。Proposer在收到多数Acceptors的Accept之后，标志着本次Accept成功，决议形成，将形成的决议发送给所有Learners。



Paxos算法流程

Paxos算法流程中的每条消息描述如下：

- **Prepare**: Proposer生成全局唯一且递增的Proposal ID (可使用时间戳加Server ID)，向所有Acceptors发送Prepare请求，这里无需携带提案内容，只携带Proposal ID即可。
- **Promise**: Acceptors收到Prepare请求后，做出“两个承诺，一个应答”。

两个承诺：

\1. 不再接受Proposal ID小于等于（注意：这里是<= ）当前请求的Prepare请求。

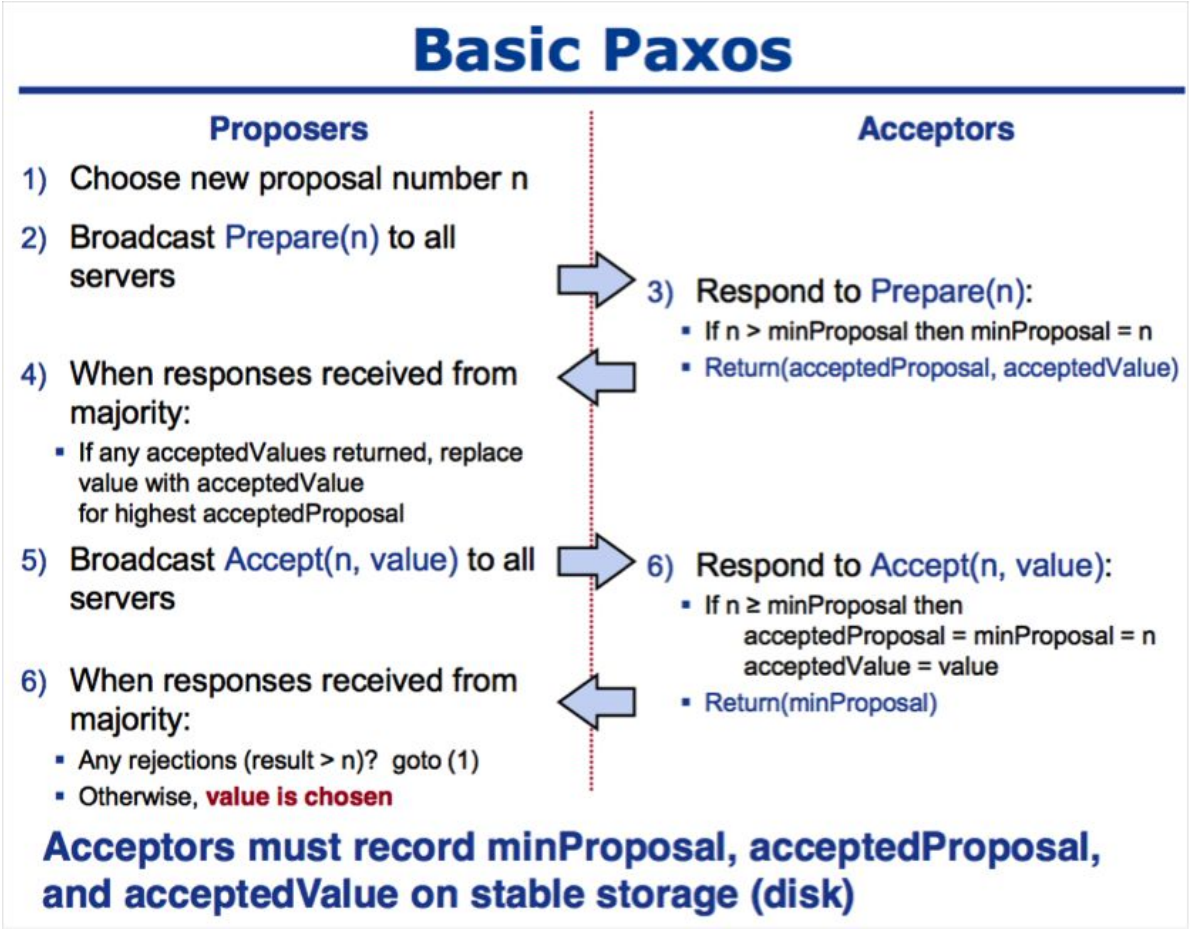
\2. 不再接受Proposal ID小于（注意：这里是< ）当前请求的Propose请求。

一个应答：

不违背以前作出的承诺下，回复已经Accept过的提案中Proposal ID最大的那个提案的Value和Proposal ID，没有则返回空值。

- **Propose**: Proposer 收到多数Acceptors的Promise应答后，从应答中选择Proposal ID最大的提案的Value，作为本次要发起的提案。如果所有应答的提案Value均为空值，则可以自己随意决定提案Value。然后携带当前Proposal ID，向所有Acceptors发送Propose请求。
- **Accept**: Acceptor收到Propose请求后，在不违背自己之前作出的承诺下，接受并持久化当前Proposal ID和提案Value。
- **Learn**: Proposer收到多数Acceptors的Accept后，决议形成，将形成的决议发送给所有Learners。

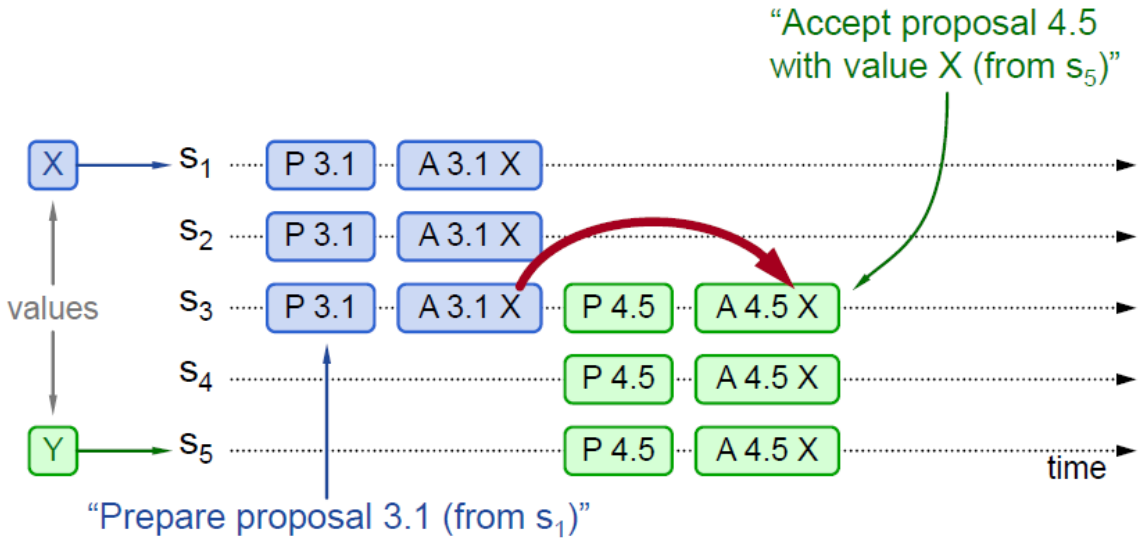
Paxos算法伪代码描述如下：



Paxos算法伪代码

1. 获取一个Proposal ID n，为了保证Proposal ID唯一，可采用时间戳+Server ID生成；
2. Proposer向所有Acceptors广播Prepare(n)请求；
3. Acceptor比较n和minProposal，如果n>minProposal，minProposal=n，并且将 acceptedProposal 和 acceptedValue 返回；
4. Proposer接收到过半数回复后，如果发现有acceptedValue返回，将所有回复中acceptedProposal最大的acceptedValue作为本次提案的value，否则可以任意决定本次提案的value；
5. 到这里可以进入第二阶段，广播Accept (n,value) 到所有节点；
6. Acceptor比较n和minProposal，如果n>=minProposal，则 acceptedProposal=minProposal=n，acceptedValue=value，本地持久化后，返回；否则，返回minProposal。
7. 提议者接收到过半数请求后，如果发现有返回值result >n，表示有更新的提议，跳转到1；否则value达成一致。

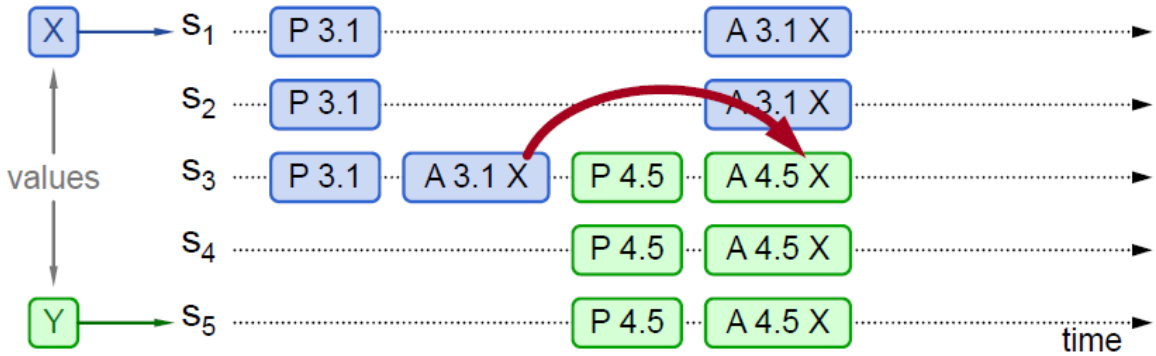
下面举几个例子，实例1如下图：



Paxos算法实例1

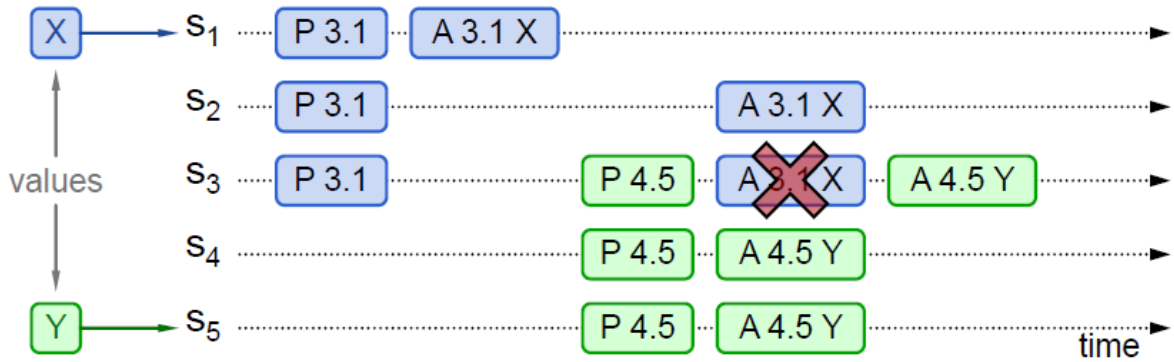
图中P代表Prepare阶段，A代表Accept阶段。3.1代表Proposal ID为3.1，其中3为时间戳，1为Server ID。X和Y代表提议Value。

实例1中P 3.1达成多数派，其Value(X)被Accept，然后P 4.5学习到Value(X)，并Accept。



Paxos算法实例2

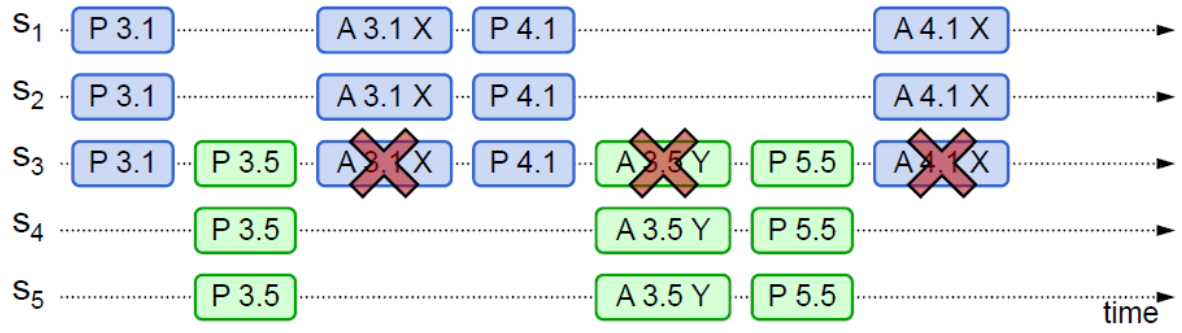
实例2中P 3.1没有被多数派Accept（只有S3 Accept），但是被P 4.5学习到，P 4.5将自己的Value由Y替换为X，Accept（X）。



Paxos算法实例3

实例3中P 3.1没有被多数派Accept（只有S1 Accept），同时也没有被P 4.5学习到。由于P 4.5 Propose的所有应答，均未返回Value，则P 4.5可以Accept自己的Value (Y)。后续P 3.1的Accept (X) 会失败，已经Accept的S1，会被覆盖。

Paxos算法可能形成活锁而永远不会结束，如下图实例所示：



Paxos算法形成活锁

回顾两个承诺之一，Acceptor不再应答Proposal ID小于等于当前请求的Prepare请求。意味着需要应答Proposal ID大于当前请求的Prepare请求。

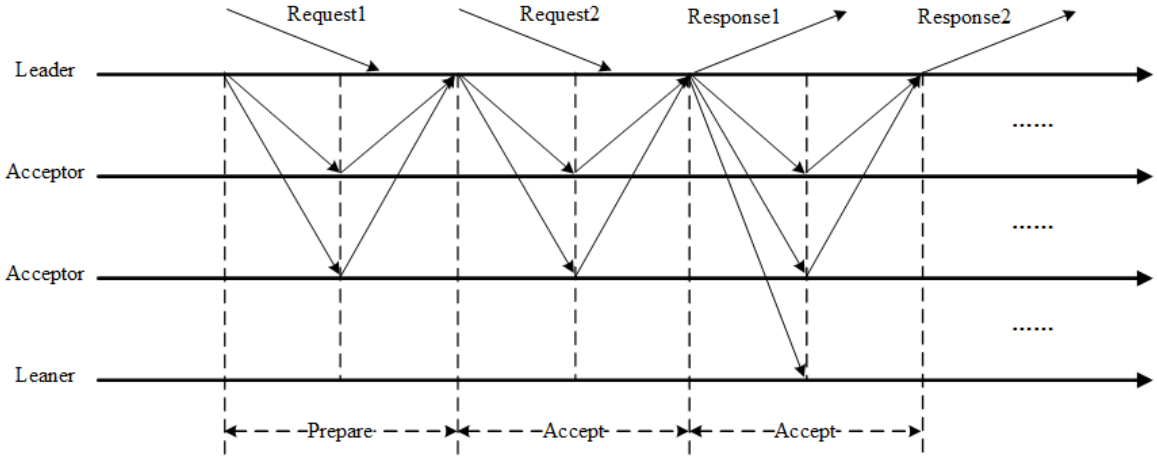
两个Proposers交替Prepare成功，而Accept失败，形成活锁（Livelock）。

三、Multi-Paxos算法

原始的Paxos算法（Basic Paxos）只能对一个值形成决议，决议的形成至少需要两次网络来回，在高并发情况下可能需要更多的网络来回，极端情况下甚至可能形成活锁。如果想连续确定多个值，Basic Paxos搞不定了。因此Basic Paxos几乎只是用来做理论研究，并不直接应用在实际工程中。

实际应用中几乎都需要连续确定多个值，而且希望能有更高的效率。Multi-Paxos正是为解决此问题而提出。Multi-Paxos基于Basic Paxos做了两点改进：

1. 针对每一个要确定的值，运行一次Paxos算法实例（Instance），形成决议。每一个Paxos实例使用唯一的Instance ID标识。
2. 在所有Proposers中选举一个Leader，由Leader唯一地提交Proposal给Acceptors进行表决。这样没有Proposer竞争，解决了活锁问题。在系统中仅有一个Leader进行Value提交的情况下，Prepare阶段就可以跳过，从而将两阶段变为一阶段，提高效率。



Multi-Paxos流程

Multi-Paxos首先需要选举Leader，Leader的确定也是一次决议的形成，所以可执行一次Basic Paxos实例来选举出一个Leader。选出Leader之后只能由Leader提交Proposal，在Leader宕机之后服务临时不可用，需要重新选举Leader继续服务。在系统中仅有一个Leader进行Proposal提交的情况下，Prepare阶段可以跳过。

Multi-Paxos通过改变Prepare阶段的作用范围至后面Leader提交的所有实例，从而使得Leader的连续提交只需要执行一次Prepare阶段，后续只需要执行Accept阶段，将两阶段变为一阶段，提高了效率。为了区分连续提交的多个实例，每个实例使用一个Instance ID标识，Instance ID由Leader本地递增生成即可。

Multi-Paxos允许有多个自认为是Leader的节点并发提交Proposal而不影响其安全性，这样的场景即退化为Basic Paxos。

Chubby和Boxwood均使用Multi-Paxos。ZooKeeper使用的Zab也是Multi-Paxos的变形。

附Paxos算法推导过程

Paxos算法的设计过程就是从正确性开始的，对于分布式一致性问题，很多进程提出（Propose）不同的值，共识算法保证最终只有其中一个值被选定，Safety表述如下：

- 只有被提出（Propose）的值才可能被最终选定（Chosen）。
- 只有**一**个值会被选定（Chosen）。
- 进程只会获知到已经确认被选定（Chosen）的值。

Paxos以这几条约束作为出发点进行设计，只要算法最终满足这几点，正确性就不需要证明了。Paxos算法中共分为三种参与者：Proposer、Acceptor以及Learner，通常实现中每个进程都同时扮演这三个角色。

Proposers向Acceptors提出Proposal，为了保证最多只有**一**个值被选定（Chosen），Proposal必须被超过一半的Acceptors所接受（Accept），且每个Acceptor只能接受一个值。

为了保证正常运行（必须有值被接受），所以Paxos算法中：

P1：Acceptor必须接受（Accept）它所收到的第一个Proposal。

先来先服务，合情合理。但这样产生一个问题，如果多个Proposers同时提出Proposal，很可能会导致无法达成一致，因为没有Propopal被超过一半Acceptors的接受，因此，Acceptor必须能够接受多个Proposal，不同的Proposal由不同的编号进行区分，当某个Proposal被超过一半的Acceptors接受后，这个Proposal就被选定了。

既然允许Acceptors接受多个Proposal就有可能出现多个不同值都被最终选定的情况，这违背了Safety要求，为了保证Safety要求，Paxos进一步提出：

P2：如果值为v的Proposal被选定（Chosen），则任何被选定（Chosen）的具有更高编号的Proposal值也一定为v。

只要算法同时满足**P1**和**P2**，就保证了Safety。**P2**是一个比较宽泛的约定，完全没有算法细节，我们对其进一步延伸：

P2a：如果值为v的Proposal被选定（Chosen），则对所有的Acceptors，它们接受（Accept）的任何具有更高编号的Proposal值也一定为v。

如果满足**P2a**则一定满足**P2**，显然，因为只有首先被接受才有可能被最终选定。但是**P2a**依然难以实现，因为acceptor很有可能并不知道之前被选定的Proposal（恰好不在接受它的多数派中），因此进一步延伸：

P2b：如果值为v的Proposal被选定（Chosen），则对所有的Proposer，它们提出的的任何具有更高编号的Proposal值也一定为v。

更进一步的：

P2c：为了提出值为v且编号为n的Proposal，必须存在一个包含超过一半Acceptors的集合S，满足(1)没有任何S中的Acceptors曾经接受（Accept）过编号比n小的Proposal，或者(2) v和S中的Acceptors所接受过(Accept)的编号最大且小于n的Proposal值一致。

满足**P2c**即满足**P2b**即满足**P2a**即满足**P2**。至此Paxos提出了Proposer的执行流程，以满足**P2c**：

1. Proposer选择一个新的编号n，向超过一半的Acceptors发送请求消息，Acceptor回复: (a)承诺不会接受编号比n小的proposal，以及(b)它所接受过的编号比n小的最大Proposal（如果有）。该请求称为Prepare请求。
2. 如果Proposer收到超过一半Acceptors的回复，它就可以提出Proposal，Proposal的值为收到回复中编号最大的Proposal的值，如果没有这样的值，则可以自由提出任何值。
3. 向收到回复的Acceptors发送Accept请求，请求对方接受提出的Proposal。

仔细品味Proposer的执行流程，其完全吻合**P2c**中的要求，但你可能也发现了，当多个Proposer同时运行时，有可能出现没有任何Proposal可以成功被接受的情况（编号递增的交替完成第一步），这就是Paxos算法的Liveness问题，或者叫“活锁”，论文中建议通过对Proposers引入选主算法选出Distinguished Proposer来全权负责提出Proposal来解决这个问题，但是即使在出现多个Proposers同时提出Proposal的情况时，Paxos算法也可以保证Safety。

接下来看看Acceptors的执行过程，和我们对**P2**做的事情一样，我们对**P1**进行延伸：

P1a：Acceptor可以接受（Accept）编号为n的Proposal当且仅当它没有回复过一个具有更大编号的Prepare消息。

易见，**P1a**包含了**P1**，对于Acceptors：

1. 当收到Prepare请求时，如果其编号n大于之前所收到的Prepare消息，则回复。
2. 当收到Accept请求时，仅当它没有回复过一个具有更大编号的Prepare消息，接受该Proposal并回复。

以上涵盖了满足**P1a**和**P2b**的一套完整一致性算法。

注：文中部分图片来自互联网。