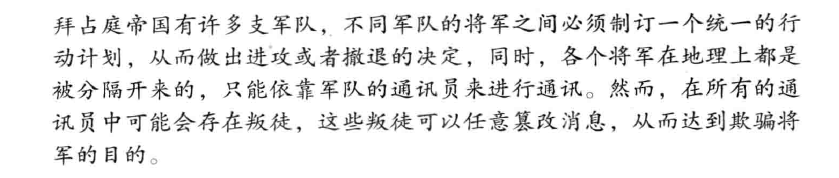
# 概述

## 背景

**拜占庭将军问题**

提及paxos协议的起源，首先得说说这个“拜占庭将军”问题，这个问题实际上是分布式数据一致性问题的一个抽象故事。



故事中的通讯员就是所谓的通信信道，拜占庭故事的结局就是在这种情况下将军之间不可能接收到一个完全正确的决定。也就是说在分布式系统中，试图在异步系统和不可靠通道上达到一致性状态是不可能的。

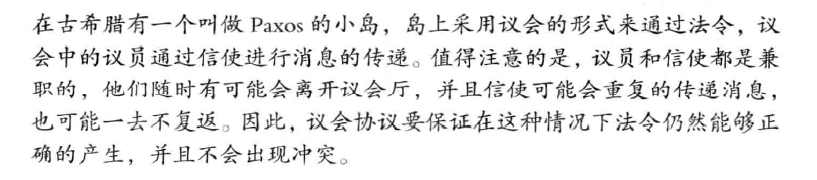
有人就会问，既然是不可能，那研究的分布式一致性协议有啥用呢？

实际工程上，由于信道编码纠错等机制，我们可以假设所有信道传输的消息是完整的，是没有被篡改过的，也就是故事中的通讯员不能篡改消息。考虑到这种实际场景下，我们研究的分布式一致性协议才有用处。

那么当信道不存在篡改的情况，该场景下又是如何解决上述问题的呢？

lamport在1990提出了paxos算法解决上述问题。

lamport将上述的问题设想了这么一个场景：



这个场景就具体代表了分布式系统中，多个节点中，数据如何保持一致性，多个节点最终对于某个数据值能够保证一致，也就是对应场景中法令是唯一有效的，不冲突。

如何解决上述的正确法令的产生，paxos算法。

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。这里的提案可以理解为某个状态值，也可以理解为一条binlog日志，甚至是一条命令（command）。根据应用场景不同，提案可以代表不同的含义，Paxos解决的常见问题有：

领导者选举：多个候选节点就谁将是leader达成一致；

互斥：也可以看成分布式锁，进程就谁先进入临界区达成一致；

原子广播：比如分布式存储系统中主备数据复制，多个主节点将写操作的log同步给备节点用于数据回放，为了让备节点与主节点数据完全一致，必须保证log的顺序。即进程需要就消息传递的顺序达成一致。

可见Paxos更关注如何就数据达成一致，不失一般性，假设有一组可以提出（propose）提案（value）的进程集合，如何保证在这些提案中，只有一个被选定（chosen）?要求：

只有被提出的value才能被选定；

只有一个value被选定，并且如果某个进程认为某个value被选定了，那么这个value必须是真的被选定的那个。

这就是算法的安全性（Safety）要求。为了方便描述，我将系统Paxos角色限定为两种：提议者（Proposer）和接受者（Acceptor），暂时不考虑学习者（Learner）。在一轮分布式选定一个value的共识过程中，可能存在多个提议者和多个接受者，只要这样才能够容忍节点故障和摆脱单点问题。

## 简介

分布式系统中的节点通信存在两种模型：**共享内存（Shared memory）和消息传递（Messages passing）**。

基于消息传递通信模型的分布式系统，不可避免的会发生以下错误：

进程可能会慢、被杀死或者重启；

消息可能会延迟、丢失、重复；

在基础Paxos场景中，先不考虑可能出现消息篡改即拜占庭错误的情况；

……

Paxos算法解决的问题是在一个可能发生上述异常的分布式系统中如何就某个值达成一致，保证不论发生以上任何异常，都不会破坏决议的一致性。

简而言之：Paxos的目的是让整个集群的结点对某个值的变更达成一致。Paxos可以说是一个民主选举的算法——**大多数节点的决定会成个整个集群的统一决定**。任何一个点都可以提出要修改某个数据的提案，是否通过这个提案取决于这个集群中是否有超过半数的节点同意。取值一旦确定将不再更改，并且可以被获取到（不可变性，可读取性）。

一致性问题普遍存在于计算机的各个领域，小到cpu cache，大到数据库事务，分布式系统。因为现实世界的复杂性，导致一致性问题在分布式领域更加突出。Paxos是当前解决分布式一致性问题最有效、理论最完美的算法，同时也是最难实现和理解的算法。

## 一致性与Paxos

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。这里的提案可以理解为某个状态值，也可以理解为一条binlog日志，甚至是一条命令（command）。根据应用场景不同，提案可以代表不同的含义，Paxos解决的常见问题有：

领导者选举：多个候选节点就谁将是leader达成一致；

互斥：也可以看成分布式锁，进程就谁先进入临界区达成一致；

原子广播：比如分布式存储系统中主备数据复制，多个主节点将写操作的log同步给备节点用于数据回放，为了让备节点与主节点数据完全一致，必须保证log的顺序。即进程需要就消息传递的顺序达成一致。

可见Paxos更关注如何就数据达成一致，不失一般性，假设有一组可以提出（propose）提案（value）的进程集合，如何保证在这些提案中，只有一个被选定（chosen）?要求：

只有被提出的value才能被选定；

只有一个value被选定，并且如果某个进程认为某个value被选定了，那么这个value必须是真的被选定的那个。

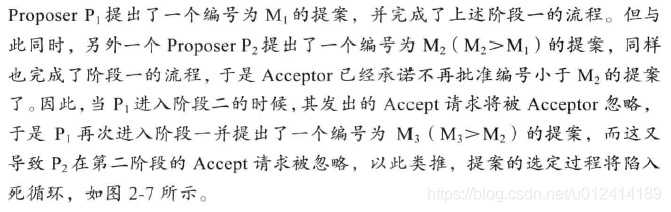
这就是算法的安全性（Safety）要求。为了方便描述，我将系统Paxos角色限定为两种：提议者（Proposer）和接受者（Acceptor），暂时不考虑学习者（Learner）。在一轮分布式选定一个value的共识过程中，可能存在多个提议者和多个接受者，只要这样才能够容忍节点故障和摆脱单点问题。

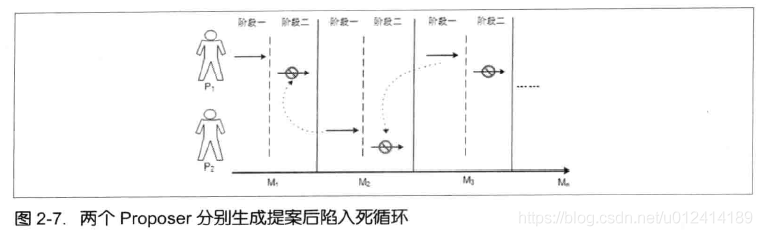
## 特点

### 优点

### 缺点

paxos算法的缺点即**存在活性问题**：考虑到一种极端的情况下，有两个proposer依次提出了一系列编号递增的议案，但是最终paxos无法形成最终的议案。具体场景如下：





在这种情况下，提案只会不断的死循环，提出，被抛弃，再次提出，被抛弃，无法保持活性。

**解决办法：**

**为了保持活性，避免上述的问题，就必须选择一个主Proposer，并规定只能由主Proposer才能提出议案，只有主能提出议案**，那么就算主被抛弃了，下次也会提出更高议案，而其他非主不能再次提出更高的议案，这样就不会陷入死循环中，从而避免了上述的问题。

## 对比

### paxos和raft

Paxos和Raft最大的区别在于**Paxos支持乱序同步，Raft只支持顺序同步**。

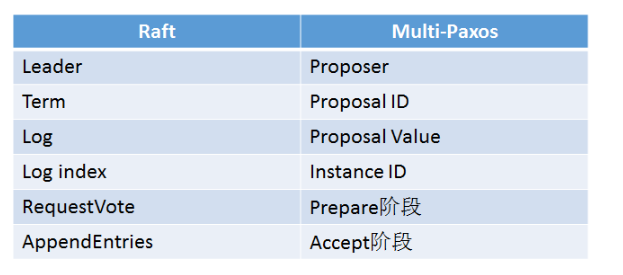
Raft相当于是Paxos协议的一种简化，好处是实现简单，容易理解，坏处是**损失了日志并行同步的性能**。国内和开源界流行Raft，不过对于AWS、微软、Google等大公司，关键系统还是使用Paxos的。

Raft协议比paxos的优点是容易理解，容易实现。它强化了leader的地位，把整个协议可以清楚的分割成两个部分，并利用日志的连续性做了一些简化：

1. Leader在时，由Leader向Follower同步日志。
2. Leader挂掉了，选一个新Leader，Leader选举算法。

#### 相同

**Raft与Multi-Paxos中相似的概念：**



Raft的Leader即Multi-Paxos的Proposer。

Raft的Term与Multi-Paxos的Proposal ID本质上是同一个东西。

Raft的Log Entry即Multi-Paxos的Proposal。

Raft的Log Index即Multi-Paxos的Instance ID。

Raft的Leader选举跟Multi-Paxos的Prepare阶段本质上是相同的。

Raft的日志复制即Multi-Paxos的Accept阶段。

#### 不同

**Raft与Multi-Paxos的不同：**



Raft假设系统在任意时刻最多只有一个Leader，提议只能由Leader发出（强Leader），否则会影响正确性；而Multi-Paxos虽然也选举Leader，但只是为了提高效率，并不限制提议只能由Leader发出（弱Leader）。

强Leader在工程中一般使用Leader Lease和Leader Stickiness来保证：

Leader Lease：上一任Leader的Lease过期后，随机等待一段时间再发起Leader选举，保证新旧Leader的Lease不重叠。

Leader Stickiness：Leader Lease未过期的Follower拒绝新的Leader选举请求。

Raft限制具有最新已提交的日志的节点才有资格成为Leader，Multi-Paxos无此限制。

Raft在确认一条日志之前会检查日志连续性，若检查到日志不连续会拒绝此日志，保证日志连续性，Multi-Paxos不做此检查，允许日志中有空洞。

Raft在AppendEntries中携带Leader的commit index，一旦日志形成多数派，Leader更新本地的commit index即完成提交，下一条AppendEntries会携带新的commit index通知其它节点；Multi-Paxos没有日志连接性假设，需要额外的commit消息通知其它节点。

### zab和raft

开始启动的情况下就必要选主，然后再提供正常服务都有一个异常场景的恢复过程

**共同点：都使用timeout来重新选择leader**

采用quorum来确定整个系统的一致性(也就是对某一个值的认可)，这个quorum一般实现是集群中半数以上的服务器，zookeeper里还提供了带权重的quorum实现。

都由leader来发起写操作，都采用心跳检测存活性。

leader election都采用先到先得的投票方式

**区别：在于选主的方式**

ZAB是广播式互相计数方式，发现别人比自己牛逼的时候要帮助别人扩散消息，根据本机计数决定谁是主raft是个节点发起投票，大家根据规则选择投于不投，然后各自收到别人对自己的投票超过半数时宣布自己成为主。

ZAB协议，只有当过半节点提交了事务，才会给客户端事务提交的回应，是一个类似二阶段提交的方式，重新选主后，特别有一个同步日志的阶段；而Raft协议，leader提交了事务，并且收到过半follower对准备完成事务的ACK后，自身节点提交事务，至于过半数节点提交事务这件事，是在之后通信过程中渐渐完成的，重新选主后，没有单独日志同步的阶段。

这导致了一个问题，Raft中如果给客户端回应完，leader挂掉了，如何保证一致性。保证在集群中处理过的事务，不会被抹去？关于这点，**Raft在选主阶段，提出了和ZAB类似的策略来解决：选择日志更多的服务器做leader，并给了更多选主的限制，以leader的日志为标准，同步日志**。

下面主要从可理解性、效率、可用性和适用场景等几个角度进行对比分析。

### 可理解性

众所周知，Paxos是出了名的晦涩难懂，不仅难以理解，更难以实现。

而Raft则以可理解性和易于实现为目标，Raft的提出大大降低了使用分布式一致性的门槛，将分布式一致性变的大众化、平民化，因此当Raft提出之后，迅速得到青睐，极大地推动了分布式一致性的工程应用。

### 效率

我们主要从负载均衡、消息复杂度、Pipeline以及并发处理几个方面来对比Multi-Paxos、Raft。

#### 负载均衡

Multi-Paxos和Raft的Leader负载更高，各副本之间负载不均衡，Leader容易成为瓶颈。

#### 消息复杂度

**Multi-Paxos和Raft选举出Leader之后，正常只需要一次网络来回就可以提交一条日志，但Multi-Paxos需要额外的异步Commit消息提交，Raft只需要推进本地的commit index，不使用额外的消息**。

因此消息复杂度，Raft最低，Paxos其次。

#### Pipeline

我们将Pipeline分为顺序Pipeline和乱序Pipeline。

**Multi-Paxos支持乱序Pipeline，Raft因为日志连续性假设，只支持顺序Pipeline。但Raft也可以实现乱序Pipeline，只需要在Leader上给每个Follower维护一个类似于TCP的滑动窗口，对应每个Follower上维护一个接收窗口，允许窗口里面的日志不连续，窗口外面是已经连续的日志，日志一旦连续则向前滑动窗口，窗口里面可乱序Pipeline**。

#### 并发处理

**Multi-Paxos沿用Paxos的策略，一旦发现并发冲突则回退重试，直到成功；Raft则使用强Leader来避免并发冲突，Follwer不与Leader竞争，避免了并发冲突**。

Paxos是冲突回退，Raft是冲突避免。Paxos和Raft的日志都是线性的。

### 可用性

Multi-Paxos和Raft均依赖Leader，Leader不可用了需要重新选举Leader，在新Leader未选举出来之前服务不可用。

**Raft是强Leader，Follower必须等旧Leader的Lease到期后才能发起选举，Multi-Paxos是弱Leader，Follwer可以随时竞选Leader，虽然会对效率造成一定影响，但在Leader失效的时候能更快的恢复服务，因此Multi-Paxos比Raft可用性更好**。

# 分类

## Basic Paxos

## Multi Paxos

## Fast Paxos

# Basic Paxos

Basic Paxos可以简单的这么描述：

1、一个或多个服务器可以同时发起提议（propose）。

2、系统必须同意一个被选中的值。

3、只有一个值被选中。

## 角色

Paxos各角色的职能：

Client：产生议题者，发起新的请求（系统外部角色，像民众）；

Proposer：**提议者**，接受Client请求，向集群提出提议propose（同时存在一个或者多个，他们各自发出提案，像议员，替民众提出议案）；

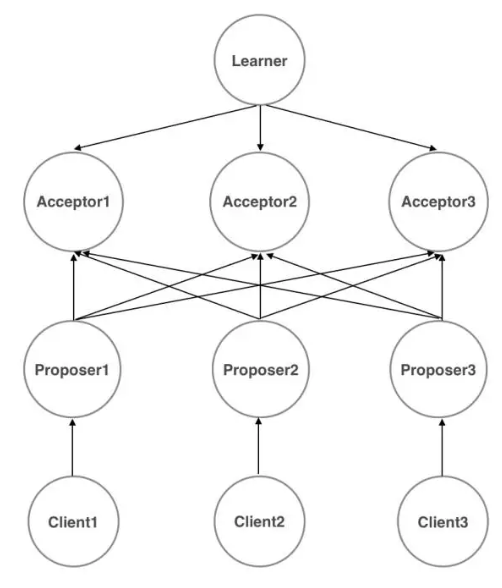
Acceptor（Voter）：**提议投票和接受者**，收到议案后选择是否接受（只有在形成法定人数quorum，一般即为多数派majority时，提议才会最终被接受，像国会）；

Learner：**提议接受者**，最终决策学习者，只学习正确的决议（backup备份，对集群一致性没有什么影响，像记录员）。

**注：**互相发短信其实就是发消息进行通信，短信的时间戳就是“epoch”。

上面4种角色中最主要的是Proposer和Acceptor。Proposer就像Client的使者，由Proposer使者拿着Client的议题去向Acceptor提议，让Acceptor来决策。主要的交互过程在Proposer和Acceptor之间。

下面用一幅图来标识角色之间的关系。

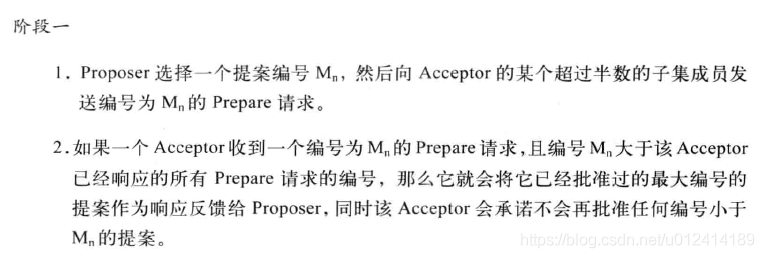


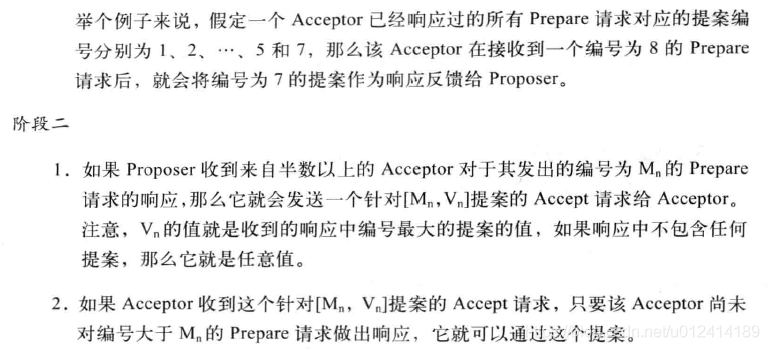
上图中是画了很多节点的，每个节点需要一台机器么？答案是不需要的。上面的图是逻辑图，物理中，可以将Acceptor和Proposer以及Client放在一台机器上，Acceptor启动端口进行TCP监听，Proposer来主动连接即可。所以完全可以将Client、Proposer、Acceptor、Learner合并到一个程序里面。

## 算法

paxos算法有着严格的数据公式的证明，但太过于复杂，能以理清，这里我们不在讨论具体的证明过程，只给出paxos算法是如何从众多提案中最终选择一个统一提案的过程。

首先和上述的故事场景类似，存在一个提出者（Proposer）和接收者（Acceptor），同样，整个选举的过程也分为两个阶段：





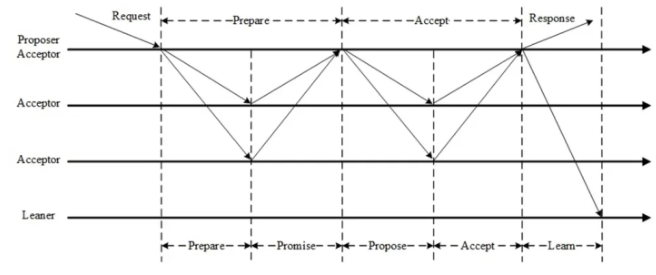
通过上述的两个阶段，即可保证最终可以得到一个统一的提案。

有人会问，现在知道了这个paxos算法，但是到底怎么用，是怎么一回事呢？

在设计分布式系统的时候，我们可以将每个节点向管理节点选举时，按照上述提交者的步骤，将每个消息就加上对应的编号，同样，管理节点接收到信息，也按照上述的步骤进行返回，最后即可按照此算法就多个节点同一个数据值达到一致。

## 步骤

**Paxos达成一个决议至少需要两个阶段（Prepare阶段和Accept阶段）**。



Prepare阶段的作用：

1. 争取提议权，争取到了提议权才能在Accept阶段发起提议，否则需要重新争取。
2. 学习之前已经提议的值。

Accept阶段使提议形成多数派，提议一旦形成多数派则决议达成，可以开始学习达成的决议。

Accept阶段若被拒绝需要重新走Prepare阶段。

### 描述一

可以将Basic Paxos分为两个大阶段四个小阶段：

1. Phase 1.1：Prepare

Propose提出一个议案，编号为N，此N大于这个proposer之前提出的提案编号，请求acceptors的quorum接受。

1. Phase1.2：Promise

如果N大于此acceptor之前接受的任何提案编号则接受，否则拒绝。

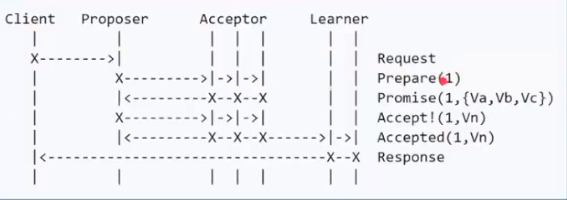
1. Phase2.1

如果达到多数派，proposer会发出accept请求，此请求包含提案编号N，以及提案内容。

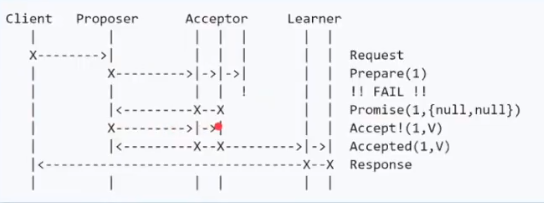
1. Phase2.2：Accepted

如果此acceptor在此期间没有收到任何编号大于N的提案，则接受此提案内容，否则忽略。

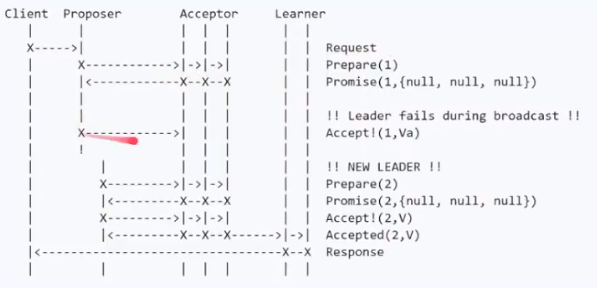
基本流程：



部分节点失败，但是达到了Quorums：

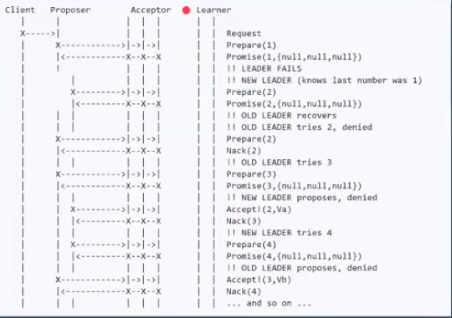


**Proposer失败：**



注：如果一个Proposer宕机，则会另起一个Proposer，Client连接到这个新的Proposer上，然后另起一个编号为2的Proposer执行任务。

**潜在问题：活锁（liveness）或dueling**



### 描述二

Paxos在原作者的《Paxos Made Simple》中的描述：

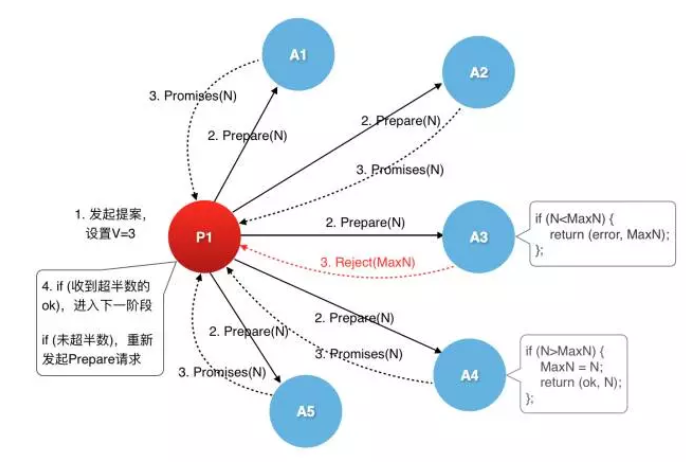
**决议的提出与批准**

通过一个决议分为两个阶段：

**1）prepare阶段**

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；**acceptor收到prepare消息后，如果提案的编号大于它已经回复的所有prepare消息，则acceptor将自己上次接受的提案回复给proposer，并承诺不再回复小于n的提案**；

下图是一个proposer和5个acceptor之间的交互，对2种不同的情况做了处理。

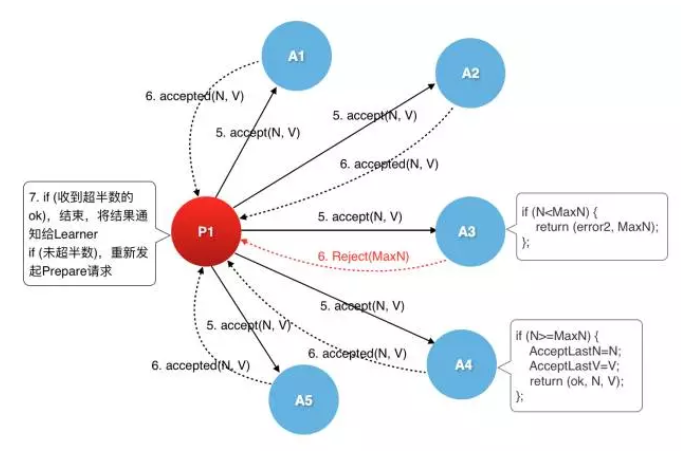


**2）批准阶段**

当一个proposer收到了多数acceptors对prepare的回复后，就进入批准阶段。它要向回复prepare请求的acceptors发送accept请求，包括编号n和value；

在不违背自己向其他proposer的承诺的前提下，acceptor收到accept请求后即接受这个请求。

可以看出，Proposer与Acceptor之间的交互主要有4类消息通信，这4类消息对应于paxos算法的两个阶段4个过程。用2轮RPC来确定一个值。上面的图解都只是一个Proposer，但是实际中肯定是有其他Proposer针对同一件事情发出请求，所以在每个过程中都会有些特殊情况处理，这也是为了达成一致性所做的事情。



如果在整个过程中没有其他Proposer来竞争，那么这个操作的结果就是确定无异议的。但是如果有其他Proposer的话，情况就不一样了。

## 唯一编号

保证Paxos正确运行的一个重要因素就是提案(proposal)编号，编号之间要能比较大小/先后，如果是一个proposer很容易做到，如果是多个proposer同时提案，该如何处理？Lamport 不关心这个问题，只是要求编号必须是全序的，但我们必须关心。这个问题看似简单，实际还稍微有点棘手，因为这本质上是也是一个分布式的问题。

**在Google的Chubby论文中给出了这样一种方法：**

假设有n个proposer，每个编号为ir (0 <= ir < n)，proposol编号的任何值s都应该大于它已知的最大值，并且满足：s % n = ir => s = m \* n + ir

proposer已知的最大值来自两部分：proposer自己对编号自增后的值和接收到acceptor的reject后所得到的值。

我们以3个proposer P1、P2、P3为例。

开始m=0，编号分别为0，1，2。P1提交的时候发现了P2已经提交，P2编号为1 > P1的0，因此P1重新计算编号：new P1 = 1 \\* 3 + 0 = 4。P3以编号2提交，发现小于P1的4，因此P3重新编号：new P3 = 1 \* 3 + 2 = 5

整个paxos算法基本上就是围绕着proposal编号在进行：proposer忙于选择更大的编号提交proposal，acceptor则比较提交的proposal的编号是否已是最大，只要编号确定了，所对应的value也就确定了。所以说，在paxos算法中没有什么比proposal的编号更重要。

# Multi Paxos

参考：

<https://www.cnblogs.com/foxmailed/p/5487533.html>

<https://blog.csdn.net/Will_cruise/article/details/83719255>

## 背景

Paxos对某一个问题达成一致的一个协议。《Paxos Made Simple》花大部分的时间解释的就是这个一个提案的问题，然后在结尾的Implementing a State Machine 章节介绍了我们大部分的应用场景是对一堆连续的问题达成一致。

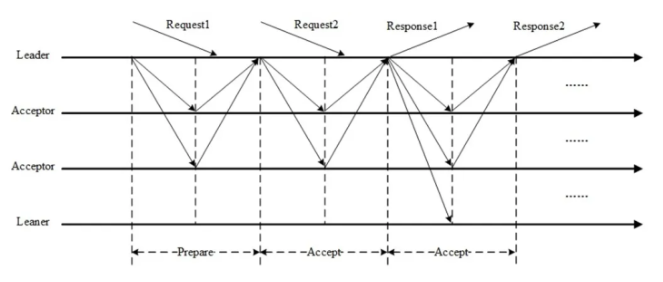
最简单的方法就是实现每一个问题独立运行一个Paxos的过程（instance）。每个过程都是独立的，相互不会干扰，这样可以为一组连续的问题达成一致。但是这样每一个问题都需要Prepare，Accept两个阶段才能够完成。Prepare阶段频繁请求会造成无谓的浪费，我们能不能把这个过程给减少。

这样就引入Proposer Leader的选举，正常的Paxos二阶段从Proposer Group中选举出 Leader后，后续统一由Leader发起提案，只有Leader才能发起提案的话相当于Proposer只有一个，所以可以省略Prepare阶段直接进入到Accpet阶段。直至发生Leader宕机、重新进行选举。

《Paxos Made Live》论文中讲解了如何使用multi paxos实现chubby的过程，以及实现过程中需要解决的问题，比如需要解决磁盘冲突，如何优化读请求，引入了 Epoch number 等。

即：Leader唯一的propser，所有请求都需要经过此Leader。

**Basic Paxos达成一次决议至少需要两次网络来回，并发情况下可能需要更多，极端情况下甚至可能形成活锁，效率低下**，Multi-Paxos正是为解决此问题而提出。

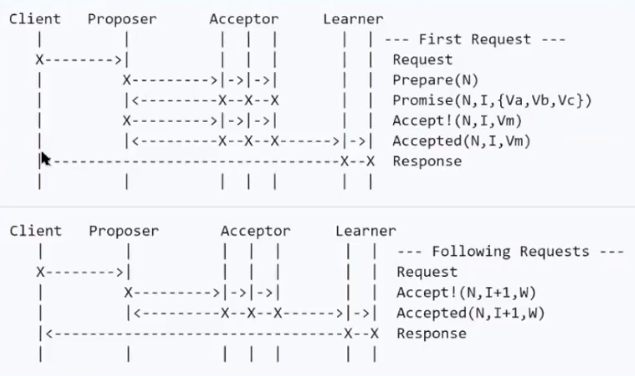


Multi-Paxos选举一个Leader，提议由Leader发起，没有竞争，解决了活锁问题。提议都由Leader发起的情况下，Prepare阶段可以跳过，将两阶段变为一阶段，提高效率。Multi-Paxos并不假设唯一Leader，它允许多Leader并发提议，不影响安全性，极端情况下退化为Basic Paxos。

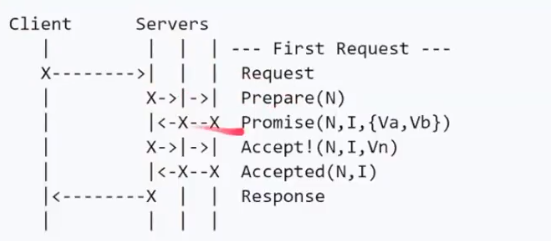
**Multi-Paxos与Basic Paxos的区别并不在于Multi（Basic Paxos也可以Multi），只是在同一Proposer连续提议时可以优化跳过Prepare直接进入Accept阶段，仅此而已**。

## 步骤

基本流程：



减少角色，进一步简化：



## 问题

难以实现，效率低（2轮RPC）、活锁

# 实际应用

由此可知，Paxos在节点宕机恢复、消息无序或丢失、网络分化的场景下能保证决议的一致性。而Paxos的描述侧重于理论，在实际项目应用中，处理了N多实际细节后，可能已经变成了另外一种算法，这时候正确性已经无法得到理论的保证。

**使用Paxos的开源项目包括：腾讯微信PhxPaxos。使用Raft的开源项目：etcd**。

要证明分布式一致性算法的正确性通常比实现算法还困难。所以很多系统实际中使用的都是以Paxos理论为基础而衍生出来的变种和简化版。例如Google的Chubby、MegaStore、Spanner等系统，ZooKeeper的ZAB协议，MySQL MGR，还有更加容易理解的raft协议。大部分系统都是靠在实践中运行很长一段时间才能谨慎的表示，系统已基本运行，没有发现大的问题。