**问题：**

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。

**作用：**

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。用于解决分布式系统中一致性问题。  
　　“与其预测未来，不如限制未来”，这应该是Paxos协议的核心思想

**基本术语：**

**角色：**

Proposer(提议发起者)；Acceptor(提议接受者)；Learner(提议学习者)；

**操作：**

Proposal(提案)：由proposer提出，被aceeptor批准或否决，包含｛编号，提案值｝  
Accept(批准): 表示提案被acceptor批准  
Choose(选择): 表示提案“被选择”，也就是被多数acceptor批准

**内容：**

Value（提案值）: 提案的值，议案的组成之一  
　　Proposal Number（提案编号）：提案编号不能冲突，提案的组成之一

**Paxos约束条件：**

**P1：一个acceptor必须接受（accept）第一次收到的提案。**

注意P1是不完备的。如果恰好一半acceptor接受的提案具有value A，另一半接受的提案具有value B，那么就无法形成多数派，无法批准任何一个value。  
　　约束2并不要求只批准一个提案，暗示可能存在多个提案。只要提案的value是一样的，批准多个提案不违背约束2。于是可以产生约束P2：

**P2：一旦一个具有value v的提案被批准（chosen），那么之后批准（chosen）的提案必须具有value v。**

注：通过某种方法可以为每个提案分配一个编号，在提案之间建立一个全序关系，所谓“之后”都是指所有编号更大的提案。  
　　如果P1和P2都能够保证，那么约束2就能够保证。  
　　批准一个value意味着多个acceptor接受（accept）了该value.因此，可以对P2进行加强：

**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**

由于通信是异步的，P2a和P1会发生冲突。如果一个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。根据P1，后者应当接受，根据P2a，则不应当接受，这中场景下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：

**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。**

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。因此需要进一步加强P2b。  
　　假设一个编号为m的value v已经获得批准（chosen），来看看在什么情况下对任何编号为n（n>m）的提案都含有value v。因为m已经获得批准（chosen），显然存在一个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：

**P2c：如果一个编号为n的提案具有value v，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。**

**Paxos过程：**

**通过一个提案分为两个阶段：**

**1. prepare阶段：**

proposer(提案发起者)选择一个提案编号n并将提案编号n请求发送给acceptors中的一个多数派（acceptors的一个子集）；  
　　acceptor收到提案编号消息后，如果提案的编号n大于之前收到的提案编号，则承诺不再回复小于n的提案；如果该acceptor已经批准过提案，则将自己批准的提案回复给proposer；

**2. 批准阶段：**

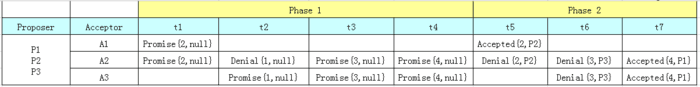
当一个proposer收到了多数acceptors对提案的承若（回复）后，就进入批准阶段。它要向回复提案请求的 acceptors发送accept请求，如果没有发现有一个acceptor接受过一个提案，那么向所有的acceptor发起自己的提案（自己的值和提案编号n），否则，从所有acceptors的回包中选择提案编号最大的提案，将该提案的值作为此次提案的值，提案编号仍然为n。  
　　在不违背自己向其他proposer的承诺的前提下，acceptor收到accept请求后即接受这个请求。

**简单示例：**

Proposer：P1,P2, P3  
　　Acceptor: A1, A2, A3

**场景一：Acceptor都未接受过提案**

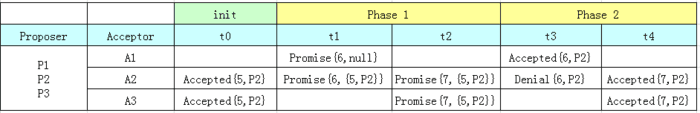
1. P1 向A2，A3发送提案编号为1,v=p1
2. P2向A1，A2发送提案编号为2,v=p2
3. P3向A2，A3发送提案编号为3,v=p3
4. t1时刻，A1，A2接收到P2的提案，并做出承若（编号小于2的拒绝）
5. t2时刻，A2，A3接收到P1的提案，由于A2已经做过承若，如拒绝该提案，A3则做出承若（编号小于1的拒绝）
6. t3时刻，A2，A3收到P3的提案，由于编号3大于编号1，2；故A2，A3做出承若（编号小于3的拒绝）
7. t4时刻，由于P1未能收到大部分的承若，再次发起提案，编号为4；A2,A3收到提案后，由于编号4为最大的编号，故A2,A3做出承若（编号小于4的拒绝）
8. t5时刻，P2由于收到大部分的承若（A1，A2），向A1，A2发送批准请求；A1批准通过，而A2最后的承若是编号不能小于4，故拒绝该批准请求；
9. t6时刻，P3收到大部分的承若（A2，A3），向A2，A3发送批准请求；由于A2，A3最后的承若是编号不能小于4，故都拒绝该批准请求；
10. t7时刻，P1收到大部分的承若（A2，A3），向A2，A3发送批准请求，A2，A3都接受该批准请求；至此此次提案被选出来及{4,P1}（多数派的选择）



p1.png

**场景二：存在接受过提案的Acceptor**

1. 初始：A2，A3已经接受提案{5,P2}
2. P1向A1，A2发送提案编号6
3. P3向A2，A3发送提案编号7
4. t1时刻，A1，A2收到提案，A1做出承若（拒绝编号小于6的提案），A2做出承若（拒绝编号小于6的提案）同 时回给P1的包含已经接受的提案{5,P2}
5. t2时刻，A2，A3收到提案，并做出承若（拒绝编号小于7的提案），同时会给P3的回包包含已经接受的提案{5,P2}
6. t3时刻，P1收到大部分的承若，给A1，A2发送批准请求；A1接受该请求，A2由于承若的编号为7，拒绝该请求；
7. t4时刻，P3收到大部分的承若，给A2，A3发送批准请求；A2，A3都接受该提案；



p2.png

**考虑：**

1. 在收到的承若回包中，包含已经接受的提议，是否可以停止提议操作？  
　　2. 编号如果保证有序，如果每个提议者的编号都是赋予一个较大的值，又该如何处理？

**扩展：**

**二阶段提交（Two-phase Commit）**

二阶段提交也被称为是一种协议(Protocol)。在分布式系统中，每个节点虽然可以知晓自己的操作时成功或者失败，却无法知道其他节点的操作的成功或失败。当一个事务跨越多个节点时，为了保持事务的ACID特性，需要引入一个作为协调者的组件来统一掌控所有节点(称作参与者)的操作结果并最终指示这些节点是否要把操作结果进行真正的提交(比如将更新后的数据写入磁盘等等)。因此，二阶段提交的算法思路可以概括为： 参与者将操作成败通知协调者，再由协调者根据所有参与者的反馈情报决定各参与者是否要提交操作还是中止操作。

**第一阶段(提交请求阶段)**

协调者节点向所有参与者节点询问是否可以执行提交操作，并开始等待各参与者节点的响应。  
　　参与者节点执行询问发起为止的所有事务操作，并将Undo信息和Redo信息写入日志。  
　　各参与者节点响应协调者节点发起的询问。如果参与者节点的事务操作实际执行成功，则它返回一个"同意"消息；如果参与者节点的事务操作实际执行失败，则它返回一个"中止"消息。  
　　有时候，第一阶段也被称作投票阶段，即各参与者投票是否要继续接下来的提交操作。

**第二阶段(提交执行阶段)**

成功  
　　 当协调者节点从所有参与者节点获得的相应消息都为"同意"时：  
　　 协调者节点向所有参与者节点发出"正式提交"的请求。  
　　 参与者节点正式完成操作，并释放在整个事务期间内占用的资源。  
　　 参与者节点向协调者节点发送"完成"消息。  
　　 协调者节点收到所有参与者节点反馈的"完成"消息后，完成事务。  
　　失败  
　　 如果任一参与者节点在第一阶段返回的响应消息为"终止"，或者 协调者节点在第一阶段的询问超时之前无法获取所有参与者节点的响应消息时：  
　　 协调者节点向所有参与者节点发出"回滚操作"的请求。  
　　 参与者节点利用之前写入的Undo信息执行回滚，并释放在整个事务期间内占用的资源。  
　　 参与者节点向协调者节点发送"回滚完成"消息。  
　　 协调者节点收到所有参与者节点反馈的"回滚完成"消息后，取消事务。  
　有时候，第二阶段也被称作完成阶段，因为无论结果怎样，协调者都必须在此阶段结束当前事务。