# 概述

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）。

# 原理

Paxos各角色的职能：

Proposer：提议者，提出议案（同时存在一个或者多个，他们各自发出提案）；

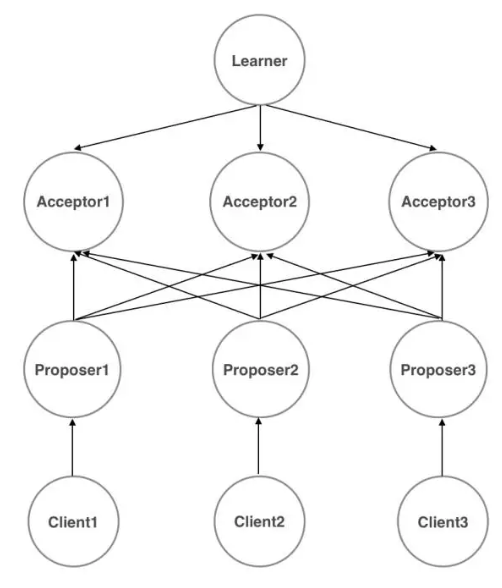
Acceptor：接受者，收到议案后选择是否接受；

Client：产生议题者，发起新的请求；

Learner：最终决策学习者，只学习正确的决议。

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

我们先看看 Paxos 在原作者的《Paxos Made Simple》中的描述。

**决议的提出与批准**

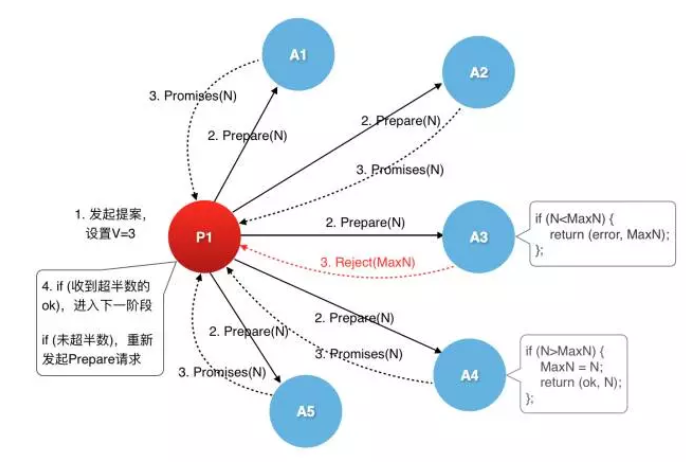
通过一个决议分为两个阶段：

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；

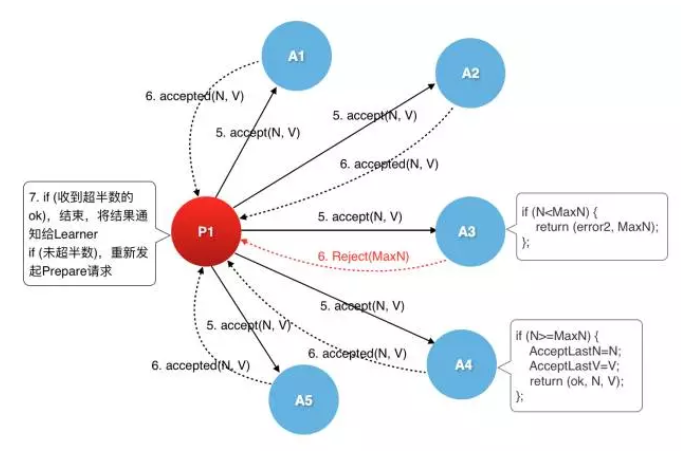
acceptor 收到 prepare 消息后，如果提案的编号大于它已经回复的所有 prepare 消息，则 acceptor 将自己上次接受的提案回复给 proposer，并承诺不再回复小于 n 的提案；

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

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。

# 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。

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，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。