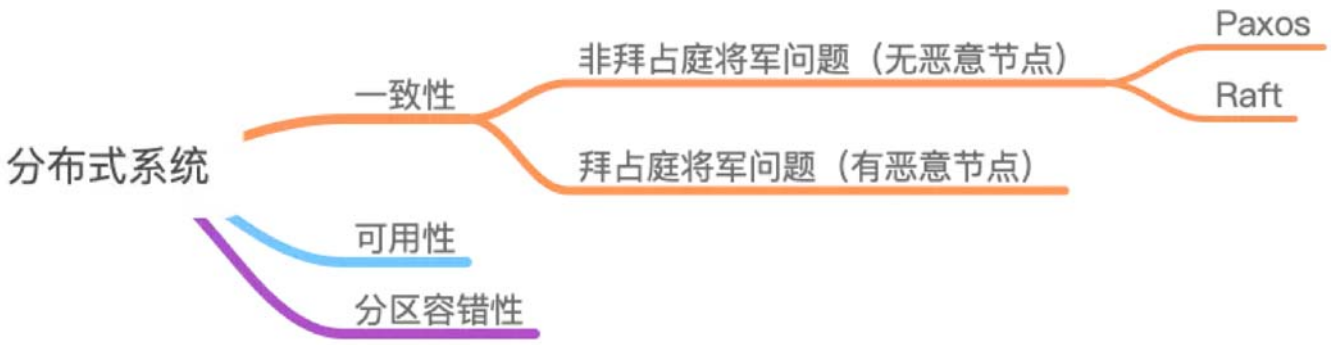
Paxos共识算法详解

在一个分布式系统中，由于节点故障、网络延迟等各种原因，根据CAP理论，我们只能保证**一致性（Consistency）、可用性（Availability）、分区容错性（Partition Tolerance）** 中的两个。

对于一致性要求高的系统，比如银行取款机，就会选择牺牲可用性，故障时拒绝服务。MongoDB、Redis、MapReduce使用这种方案。

对于静态网站、实时性较弱的查询类数据库，会牺牲一致性，允许一段时间内不一致。简单分布式协议Gossip，数据库CouchDB、Cassandra使用这种方案。



> 图1

如图1所示，一致性问题，可以根据是否存在恶意节点分类两类。无恶意节点，是指节点会丢失、重发、不响应消息，但不会篡改消息。而恶意节点可能会篡改消息。有恶意节点的问题称为拜占庭将军问题，不在今天的讨论范围。Paxos很好地解决了无恶意节点的分布式一致性问题。

背景

Paxos类型

Paxos本来是虚构故事中的一个小岛，议会通过表决来达成共识。但是议员可能离开，信使可能走丢，或者重复传递消息。对应到分布式系统的节点故障和网络故障。

> 图2

如图2所示，假设议员要提议中午吃什么。如果有一个或者多个人同时提议，但一次只能通过一个提议，这就是Basic Paxos，是Paxos中最基础的协议。

显然Basic Paxos是不够高效的，如果将Basic Paxos并行起来，同时提出多个提议，比如中午吃什么、吃完去哪里嗨皮、谁请客等提议，议员也可以同时通过多个提议。这就是Multi-Paxos协议。

Basic Paxos

角色

Paxos算法存在3种角色：Proposer、Acceptor、Learner，在实现中一个节点可以担任多个角色。

Learner不参与投票过程，为了简化描述，我们直接忽略掉这个角色。

算法

运行过程分为两个阶段，Prepare阶段和Accept阶段。

Proposer需要发出两次请求，Prepare请求和Accept请求。Acceptor根据其收集的信息，接受或者拒绝提案。

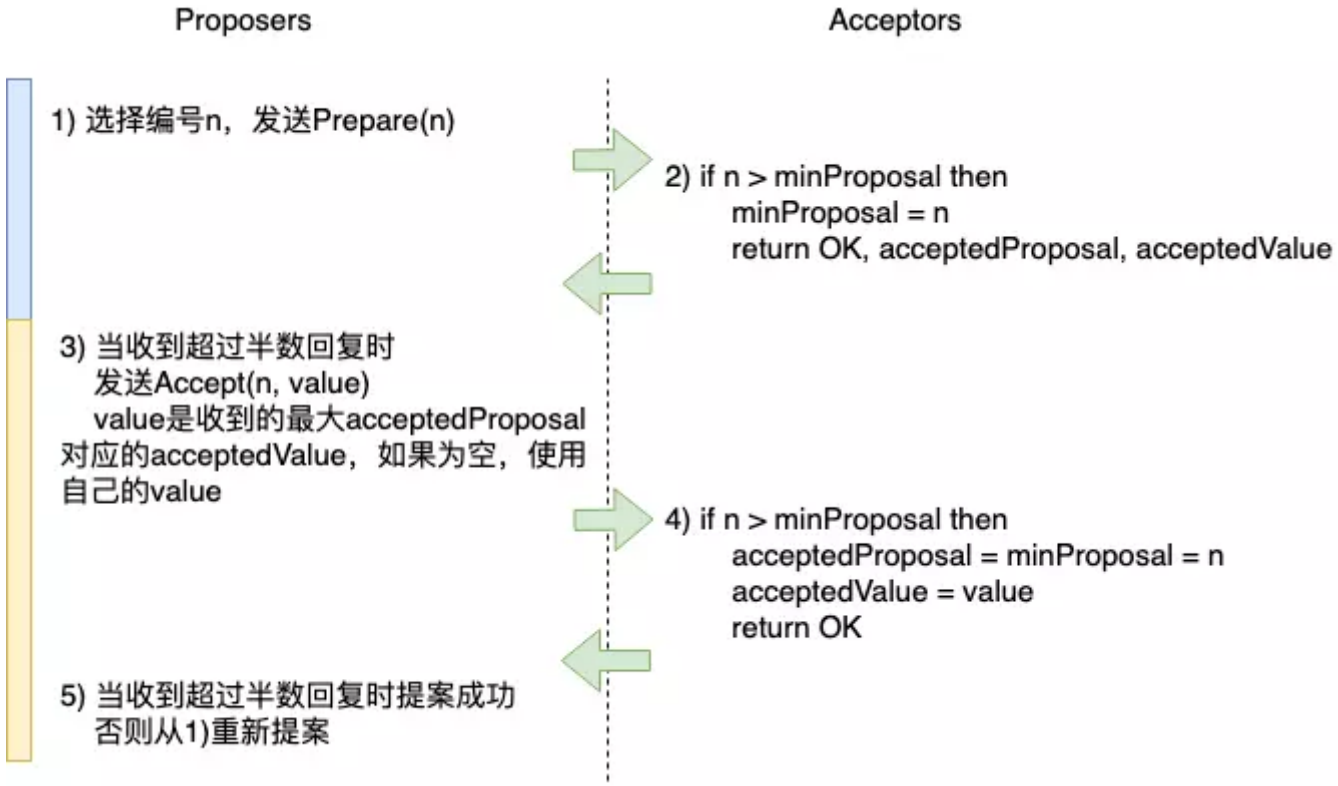
Prepare阶段

- Proposer选择一个提案编号n，发送Prepare(n)请求给超过半数（或更多）的Acceptor。
- Acceptor收到消息后，如果n比它之前见过的编号大，就回复这个消息，而且以后不会接受小于n的提案。另外，如果之前已经接受了小于n的提案，回复那个提案编号和内容给Proposer。

Accept阶段

- 当Proposer收到超过半数的回复时，就可以发送Accept(n, value)请求了。 n就是自己的提案编号，value是Acceptor回复的最大提案编号对应的value，如果Acceptor没有回复任何提案，value就是Proposer自己的提案内容。
- Acceptor收到消息后，如果n大于等于之前见过的最大编号，就记录这个提案编号和内容，回复请求表示接受。
- 当Proposer收到超过半数的回复时，说明自己的提案已经被接受。否则回到第一步重新发起提案。

完整算法如图4所示：



> 图4

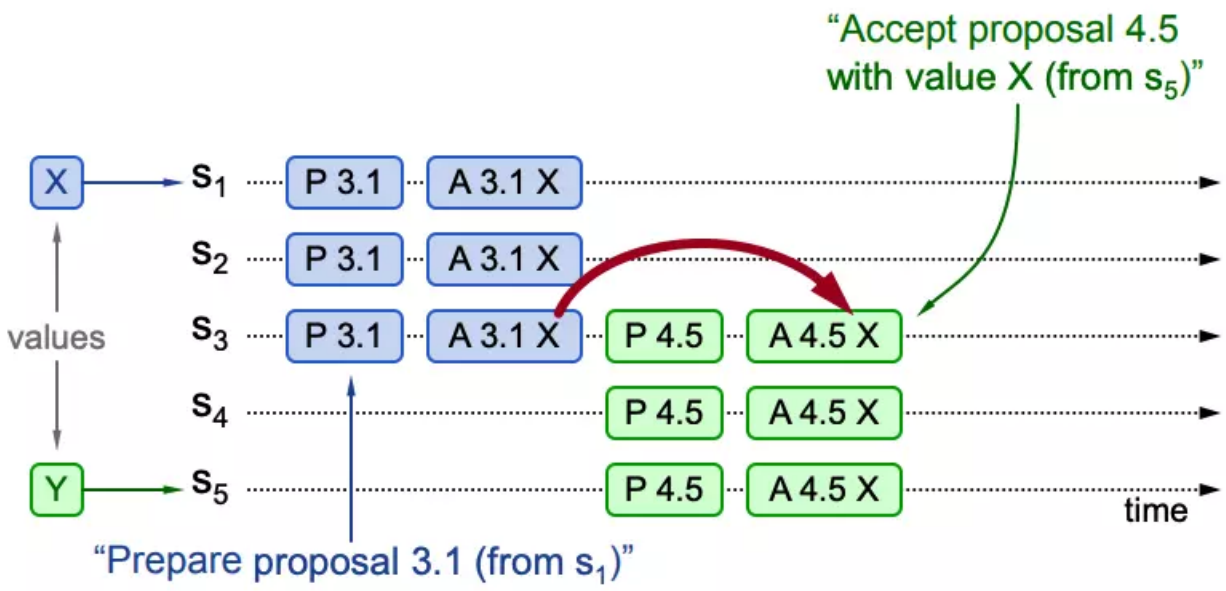
Acceptor需要持久化存储minProposal、acceptedProposal、acceptedValue这3个值。

三种情况

Basic Paxos共识过程一共有三种可能的情况。下面分别进行介绍。

情况1：提案已接受

如图5所示。X、Y代表客户端，S1到S5是服务端，既代表Proposer又代表Acceptor。为了防止重复，



> 图5 以上图片来自Paxos lecture (Raft user study)第13页

这个过程表示，S1收到客户端的提案X，于是S1作为Proposer，给S1-S3发送Prepare(3.1)请求，由于Acceptor S1-S3没有接受过任何提案，所以接受该提案。然后Proposer S1-S3发送Accept(3.1, X)请求，提案X成功被接受。

在提案X被接受后，S5收到客户端的提案Y，S5给S3-S5发送Prepare(4.5)请求。对S3来说，4.5比3.1大，且已经接受了X，它会回复这个提案 (3.1, X)。S5收到S3-S5的回复后，使用X替换自己的Y，于是发送Accept(4.5, X)请求。S3-S5接受提案。最终所有Acceptor达成一致，都拥有相同的值X。

这种情况的结果是：**新Proposer会使用已接受的提案**

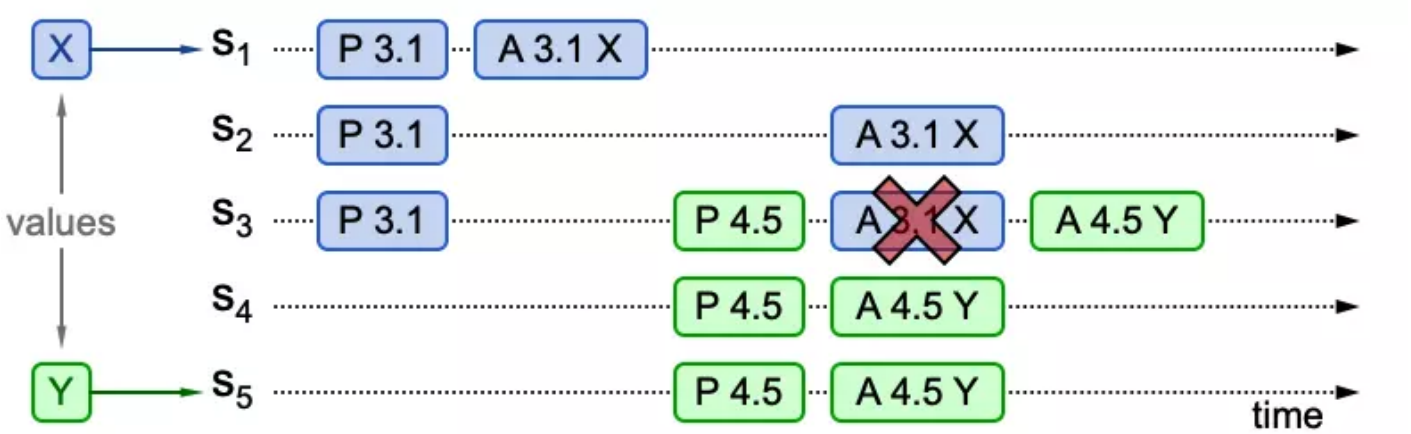
情况2：提案未接受，新Proposer可见

如图6所示，S3接受了提案(3.1, X)，但S1-S2还没有收到请求。此时S3-S5收到Prepare(4.5)，S3会回复已经接受的提案(3.1, X)，S5将提案值Y替换成X，发送Accept(4.5, X)给S3-S5，对S3来说，编号4.5大于3.1，所以会接受这个提案。

然后S1-S2接受Accept(3.1, X)，最终所有Acceptor达成一致。

这种情况的结果是：**新Proposer会使用已提交的值，两个提案都能成功**

情况3：提案未接受，新Proposer不可见

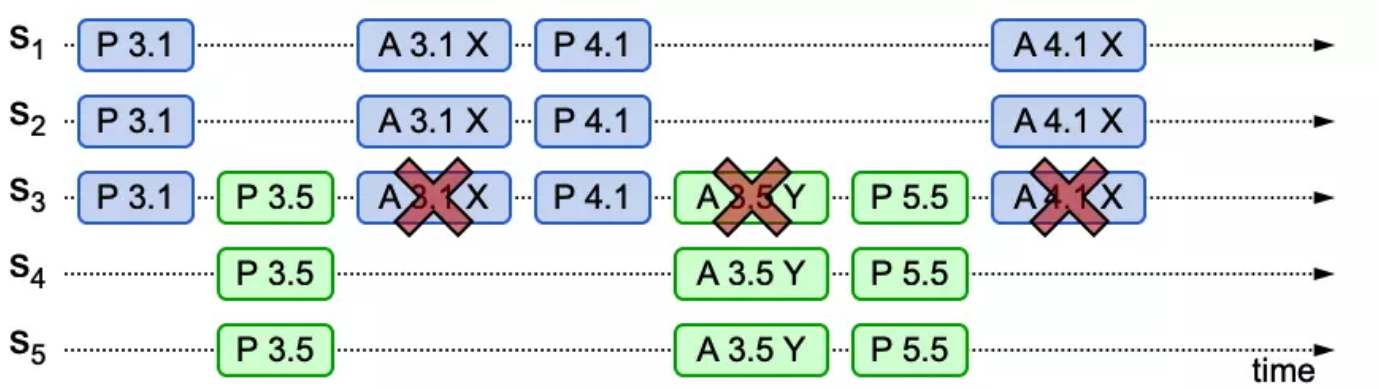


> 图7 以上图片来自Paxos lecture (Raft user study)第15页

如图7所示，S1接受了提案(3.1, X)，S3先收到Prepare(4.5)，后收到Accept(3.1, X)，由于3.1小于4.5，会直接拒绝这个提案。所以提案X无法收到超过半数的回复，这个提案就被阻止了。提案Y可以顺利通过。

这种情况的结果是：**新Proposer使用自己的提案，旧提案被阻止**

活锁 (livelock)



> 图8 以上图片来自Paxos lecture (Raft user study)第16页

解决方案是Proposer失败之后给一个随机的等待时间，这样就减少同时请求的可能。

Multi-Paxos

上一小节提到的活锁，也可以使用Multi-Paxos来解决。它会从Proposer中选出一个Leader，只由Leader提交Proposal，还可以省去Prepare阶段，减少了性能损失。当然，直接把Basic Paxos的多个Proposer的机制搬过来也是可以的，只是性能不够高。

将Basic Paxos并行之后，就可以同时处理多个提案了，因此要能存储不同的提案，也要保证提案的顺序。

Acceptor的结构如图9所示，每个方块代表一个Entry，用于存储提案值。用递增的Index来区分Entry。

其他Proposer，必须拒绝客户端的请求，或将请求转发给Leader。

当然，还可以使用其他更复杂的选举方法，这里不再详述。

2. 省略Prepare阶段

Prepare的作用是阻止旧的提案，以及检查是否有已接受的提案值。

当只有一个Leader发送提案的时候，Prepare是不会产生冲突的，可以省略Prepare阶段，这样就可以减少一半RPC请求。

Prepare请求的逻辑修改为：

- Acceptor记录一个全局的最大提案编号
- 回复最大提案编号，如果当前entry以及之后的所有entry都没有接受任何提案，回复noMoreAccepted

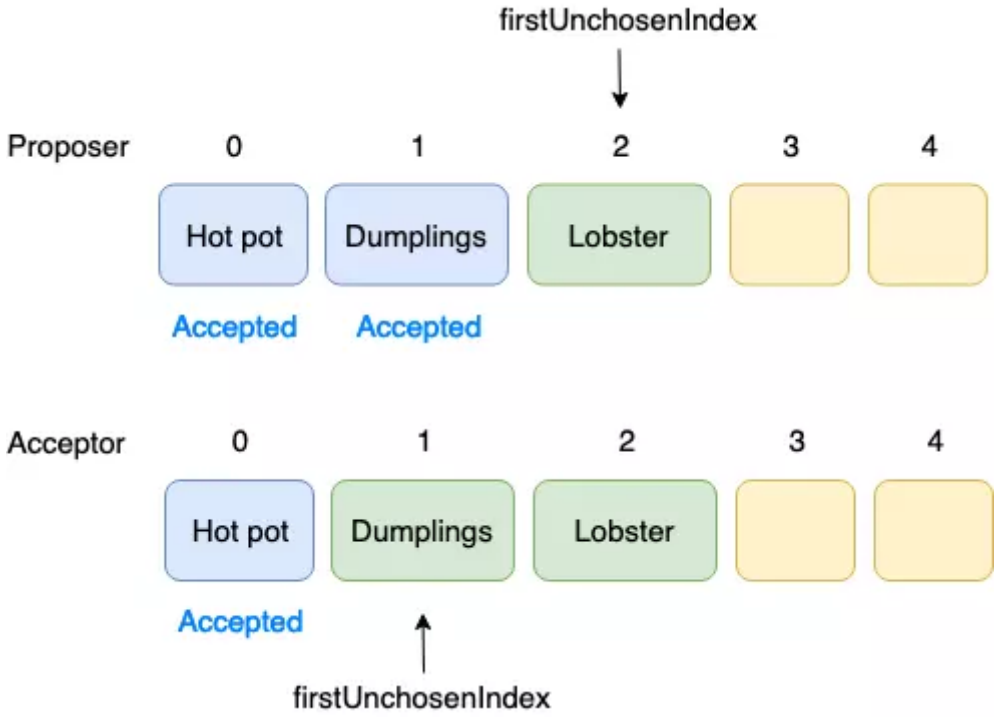
当Leader收到超过半数的noMoreAccepted回复，之后就不需要Prepare阶段了，只需要发送Accept请求。直到Accept被拒绝，就重新需要Prepare阶段。

3. 完整信息流

目前为止信息是不完整的。

- Basic Paxos只需超过半数的节点达成一致。但是在Multi-Paxos中，这种方式可能会使一些节点无法得到完整的entry信息。我们希望每个节点都拥有全部的信息。
- 只有Proposer知道一个提案是否被接受了（根据收到的回复），而Acceptor无法得知此信息。

第1个问题的解决方案很简单，就是Proposer给全部节点发送Accept请求。



> 图10

如图10所示，Proposer正在准备提交Index=2的Accept请求，0和1是已接受的提案，因此firstUnchosenIndex=2。当Acceptor收到请求后，比较Index，就可以将Dumplings提案标记为已接受。

由于之前提到的Leader切换的情况，仍然需要显式请求才能获得完整信息。在Acceptor回复Accept消息时，带上自己的firstUnchosenIndex。如果比Proposer的小，那么就需要发送Success(index, value)，Acceptor将收到的index标记为已接受，再回复新的firstUnchosenIndex，如此往复直到两者的index相等。

总结

Paxos是分布式一致性问题中的重要共识算法。这篇文章分别介绍了最基础的Basic Paxos，和能够并行的