mp.weixin.qq.com

Paxos算法和Uniform Consensus

导语：

> 昨天我们介绍了拜占庭将军问题和FLP Impossibility, 我们知道了在异步网络中的total correct的consensus算法是不存在的, 但是如果我们放松

> liveness的要求，我们在工程和实际应用当中，我们是否可以解决一致性问题呢？今天我们会给大家介绍一个工程中最常用的一个consensus算法 – Paxos算法.

大名鼎鼎的Paxos算法

Consensus主要目的是屏蔽掉故障节点的噪音让整个系统正常运行下去，比如选举过程和状态机复制. 所以

Consensus问题对于agreement条件做了放松，它接受不一致是常态的事实，既然我无法知道某些节点是挂了还是暂时联系不到，那我只要关心正确响应的节点，只要

表决能过半即可，过半表决意味着虽然没有完全一致，但是”投票结果”被过半成员继承下来了，这是因为任何两个quorum一定会存在交集(想象一下有A/B/C三个节点，

两个quorum比如AB和AC一定会有A是交集), 所以不管有多少个quorum存在, 我们能确保他们一定会有交集, 所以他们一定能信息互通而最终达成一致, 其他没有达

成一致的成员将来在网络恢复后也可以和这部分交集内的节点传播出去的”真理”达成一致.

大名鼎鼎的Paxos算法诞生了，Paxos是第一个正

确实现适用于分布式系统的Consensus算法。1990年Lamport提出了这个算法，但有趣的是ACM TOCS的评审委员们没看懂他的论文，主编建议他不要拿

古希腊神话什么长老被砸健忘了的故事写论文，要他用数学语言写简洁点，Lamport也是个人才，他拒绝修改论文，并在一次会议上公开质疑”为什么搞基础理论的人—

点幽默感都没有呢?”

6年光阴过去, 另外一个超级牛人, 图灵奖获得者Butler Lampson看到这篇论文, 而且..... 他看懂了! Lampson

觉得这个算法很重要并呼吁大家重新审视这篇重要论文，后来提出FLP理论的三人组其中的Nacy Lynch重写了篇文章阐述这篇论文，后来终于大家都看懂了，最终

1998年ACM TOCS终于发表了这篇论文 [The Part-Time Parliament. ACM Transactions on Computer Systems 16, 2 (May 1998),

133-169], 至此将近9年了. 最爆笑得是1998发表的时候, 负责编辑的Keith配合Lamport的幽默写的注解, 这里我给八卦一下省的你去翻论文了:

本文最近刚被从一个文件柜里发现，尽管这篇论文是很久之前提交的但是主编认为还是值得发表的(不是我们*ACM TOCS*过去没看懂，是忘记发表了)。但是由于作者

目前在希腊小岛上考古，*ACM TOCS*联系不上这位考古学家，所以任命我来发表这篇论文(其实是*Lamport*拒绝修改论文，不鸟*ACM TOCS*了)．作者貌似是个对计算

机科学稍微有点兴趣的考古学家(赤裸裸的吐槽),尽管他描述的*Paxon*岛上民主制度的故事对计算机科学家没啥兴趣,但是但是这套制度对于在异步网络中实现分布式

系统到时很好的模型(这句总算客观了). 建议阅读的时候直接看第四节(跳过前三节神话故事), 或者最好先别看(你可能会看不懂), 最好先去看看*Lampson*或者*De*

*Prisco*对这篇论文的解释.

看到这你笑喷了没?

我们的”考古学家”Lamport在Paxos算法定义了三个角

色, 其中proposer是提出建议值的进程, acceptor是决定是否接受建议的进程, learner是不会提议但是也要了解结果的进程, 在一个系统中一个进程经常同时扮演这三

个角色. 算法分两个阶段 (以下是最基础的算法, 其中没有learners):

第一阶段: 一个proposer选择一个全局唯一的序号n发给至少过半的

acceptors. 如果一个acceptor收到的请求中的序号n大于之前收到的建议，那么这个acceptor返回proposer一个确认消息表示它不会再接受任何小于这个n的建议.

第二阶段: 如果proposer收到过半的acceptor的确认消息表示他们不会接受n以下的建议, 那么这个proposer给这些acceptor返回附带他的建议值v的确认消息. 如果

一个acceptor收到这样的确认消息{n, v}, acceptor就会接受它. 除非在此期间acceptor又收到了一个更高的n的建议.

对证明有兴趣的读

者可以去看看Paxos Made Simple. 简单来讲，算法的正确性有两个方面。

提议的safety是由sequence number N决定的，如果N是

全序集, 唯一而且
一定有先后, 并且
在每个proposer上
都单调递增, 那么
acceptor选择的结
果就是安全的. 实
际应用中, 这个N经
常是timestamp, 进
程id, 还有进程的本

体counter的组合,
比如: timestamp +
node id + counter,
或者像twitter的
snowflake基于
timestamp和网卡
mac地址的算法.
这样可以保证事件
顺序尽量接近物理

时间的顺序，同时保证事件number的唯一性.

过半表决可以保证网络出现多个分区的时候，任何两个能够过半的分区必然存在交集，而交

集内的进程就可以保证正确性被继承，以后被传播出去.

Paxos是一个非常基础的算法，更多的时候你需要在Paxos的基础之上实现你的算法.

Paxos的过半表决有一定局限性. 这牵扯到分区可用性. 如果网络分区的时候, 没有形成多数派, 比如一个网络内被均匀的分成了三个小区, 那么整个系统都不能正常

工作了. 如果分成一个大区, 一个小区, 那么小区是无法工作的, 如果大区和小区非常接近, 比如是501 vs 499, 这意味着系统的处理能力可能会下降一半. 所有这些都

影响到了Paxos的可用性. 举个例子, Zookeeper的ZAB协议的选举和广播部分很类似Paxos, Zookeeper允许读取过期数据来获得更好的性能, 所以一般情况下是

Sequential Consistency. 但是他有一个Sync命令, 当你每次都Sync+Read的时候, 虽然性能大打折扣, 但是Zookeeper就是和Paxos一样能保证

Linearizability了.
当zookeeper在发生网络分区的时候, 如果leader在quorum side (大区), 那么quorum side的读写都正常, 但是non-quorum side因为无法从小

区选出leader, 所有连接到non-quorum side的客户端的所有的读写都会失败! (也可以不发Sync命令降低到SC级别, 通过过期的缓存让读操作能继续下去) 如果把client

也考虑在内，假设如下图所示，99%的节点连同1%的客户端处于一个分区，那么会导致99%的客户端都无法正常工作，尽管这个时候集群是99%的节点都是好

的. (通常我们的网络拓扑结构不会发生这么极端不平衡的情况).

有人会觉得网络分区在同一个数据中

心内发生的概率非常低吧?

其实不然, 举一个例子, github在2012年有一次升级一对聚合交换机的时候, 发现了一些问题决定降级, 降

级过程中其中要关闭一台交换机的agent来收集故障信息，导致90秒钟的网络分区，精彩的开始了。首先，github的文件服务器是基于DRBD的，因为每一对文件服

务器只能有一个活动节点，另外一个作为热备，当主节点出现故障后，热备节点在接替故障的活动节点之前会发一个指令去关闭活动节点. 如果网络没有分区，活动

节点的硬件发生了故障导致响应超时，那么这样可以避免brain-split. 但是在网络分区的情况下，活动节点没有收到关闭指令，热备节点就把自己作为新的活动节点了，当

网络恢复之后，就有了两个活动节点同时服务，导致了数据不一致不说，还会发生互相尝试杀死对方的情况，因为两边都认为对方是有故障的，需要杀死对方，结果

有些文件服务器真的把对方都杀死了.

网络分区是真实存在的, 而且在跨多个数据中心的情况下, 网络分区发生的概率更高. 所以使用zookeeper的

时候你一定要理解他的一致性模型在处理网络分区的情况时的局限性. 对于map reduce来讲, 结果是correct or nothing, 宁可牺牲可用性也要保证一致性, safety更重

要. 但是一个分布式系统的服务发现组件就不同了, 对于发现服务而言, having something wrong is better than having nothing, liveness 更重要. 所以Netflix

的黑帮们才自己造了个轮子Eureka，因为如果你的微服务都是无状态的，大多数情况下发现服务只要能达到Eventual Consistency就可以了，而高可用性

是必须的. 后面介绍CAP定理和Eventual Consistency的时候会介绍一下侧重可用性的算法.

除了网络分区, Paxos的另外一个

缺点是延迟比较高. 因为Paxos放松了liveness, 它有时候可能会多个回合才能决定结果, 错误的实现甚至会导致live lock. 比如proposer A提出n1, proposer B提出n2,

如果n1 > n2, acceptor先接受了n1, B收到拒绝之后重新发起n3, 如果n3先于A的确认消息到达acceptor, 而且n3 > n1, 那么acceptor会拒绝A, 接受B, A可能会重

复B的行为，然后无限循环下去.

FLP理论描述了Consensus可能会进入无限循环的情况，但是实际应用中这个概率非常低，大家都知道计算机

科学是应用科学，不是离散数学那样非正即误，如果错误或者偏离的概率非常低，工程中就会采用. 比如费马小定理(Fermat’s Little Theorem) 由于Carmichael

Number的存在并不能用来严格判断大素数, 但是由于Carmichael Number实在是太少, 1024 bit的范围里概率为10^-88, 所以RSA算法还是会用费马小定理.

同样, 根据FLP理论, 异步网络中Paxos可能会进入无限循环. 真实世界中Paxos的如果两个节点不断的互相否定, 那么就会出现无限循环, 但是要永远持续下去

的概率非常非常低，
实际中我们经常让
某个Proposer获取
一定期限的lease，
在此期限之内只有
一个proposer接受
客户端请求并提出
proposal. 或者随机
改变n的增长节奏

和proposer的发送节奏等, 来降低livelock的概率. 当然, 单从纯粹FLP理论来看, 超过一个proposer的时候Paxos是不保证liveness的.

Paxos在实现中，每个进程其实一般都是身兼三职，然后成功提议的那个proposer所在的进程就是distinguished proposer，也是distinguished

learner, 我们称之为 leader.

上面提到的Paxos算法是最原始的形式, 这个过程中有很多可以优化的地方, 比如如果accetpr拒绝了可以

顺便返回当前最高的n, 减少算法重试的回合, 但这不影响这个算法的正确性. 比如Multi-Paxos可以假设leader没有挂掉或者过期之前不用每次都发出prepare

请求, 直接发accept请求, 再比如Fast Paxos等等.

Paxos设计的时候把异步网络的不确定性考虑在内, 放松了liveness的要求, 算法按照crash-

recovery的思想设计, 所以Paxos才可以成为这样实际广泛应用而且成功的算法. 但是Paxos也不能容忍拜占庭式故障节点, 要容忍拜占庭式故障实在是太困难了 (比如,

Paxos中两个quorum中的交集如果都是叛徒怎么办?).

尽管Paxos有很多缺点, 但是Paxos仍然是分布式系统中最重要的一个算法,

比如它的一个重要用途就是 State Machine Replication. 一个 Deterministic State Machine对于固定的输入序列一定会产生固定的结果，不论重复执行多少

次，或者再另外一台一样的机器上执行，结果都是一样的. 分布式系统中最常见的一个需求就是通过某种路由算法让客户端请求去一组状态一致的服务器，数据在服

务器上的分布要有replica来保证高可用性, 但是replica之间要一致. 状态复制就成为了分布式系统的一个核心问题. Paxos的状态复制可以保证整个系统的所有状态机

接受同样的输入序列 (Atomic Broadcast), 如果查询也使用过半表决, 那么你一定会得到正确的结果.

这种做法相对于Master-Slave的的

状态复制一致性好很多. 比如一旦Master节点挂了, 还没来得及复制的结果会导致Slave之间的状态有可能是不一致的, 如果客户端能够访问到延迟的Slave节点,

那么用户展现的数据将会不一致. 如果你可以牺牲性能来换得更高的一致性, 那么你可以通过Paxos表决查询来屏蔽掉延迟或者有故障的节点. 只要系统中存在一个

quorum, 那么状态的一致性就可以保留下去. 比如google的chubby, 只要故障节点不超过一半, 网络没有发生分区, chubby就可以通过Paxos状态机为其他服务

器提供分布式锁.
因为chubby分别部署在每个数据中心,他们没有跨数据中心的通信,所以网络分区的故障频率不像跨数据中心的情况那么多高,chubby牺牲部分性

能和网络分区下的可用性，换来了一致性.

因为Paxos家族的算法写性能都不是很好但是一致性又很重要，所以实现中我们经常做一些

取舍, 让Paxos只处理最关键的信息. 比如Kafaka的每个Partition都有多个replicas, 日志的复制本身没有经过Paxos这样高延迟的算法, 但是为了保证负责接受

producer请求和跟踪ISR的leader只有一个, Kafaka依赖于Zookeeper的ZAB算法来选举leader. 在实际应用中, 状态机复制不太适合很高的吞吐量, 一般都是用于

不太频繁写入的重要信息. Zookeeper不能被当做一个OLTP级的数据库用, 它不是分布式系统协同的万能钥匙。

Uniform

Consensus

对于Consensus问题，我们只关注非故障节点的一致性和整体系统的正常，而Uniform Consensus中要求无论是非故障节点还是故障节点，他

们都要一致，比如在分布式事务的前后，各个节点必须一致，故障节点恢复后也要一致，任何时刻都不应该有两个参与者一个决定提交，一个决定回滚.。

很长一段时间内大多数人没有把Uniform Consensus单独分类, 直到2001年才有人提出Uniform Consensus更难. [Uniform Consensus is

Harder Than Consensus, Charron-Bost, Schiper, Journal of Algorithms 51 2004 15-37]

一般的Consensus问题通常用来解决

状态机复制时容错处理的问题，比如Paxos，而Uniform Consensus所处理的是分布式事务这样的问题，在Uniform Consensus中我们要求所有节点在故

障恢复后都要达成一致. 所以Uniform Consensus的定义在agreement上更加严格:

- termination: 所有进程最终会在有限步数中结束并选取

一个值, 算法不会无尽执行下去.

- agreement: 所有进程(包含故障节点恢复后)必须同意同一个值. (假设系统没有拜占庭故

障)

- validity: 最终达成一致的值必须是V1到Vn其中一个, 如果所有初始值都是vx, 那么最终结果也必须是vx.

总结

至此, 通过对Concensus问题的介绍, 我们对Linearizability和Sequential Consistency的应用应该有了更深入的理解, 在本系列

下一篇文章中我们将会介绍性能更好但是一致性要求更低的Causal Consistency, PRAM以及不需要硬件自动同步的一致性模型，比如Weak

Consistency。

如果您想投稿给我们，或者想转发和采用我们的稿件，请回复“合作”，小编会在2小时内回复您的投稿和合作需求。

本文作者：Daniel，吴强， 现任点融网首席社交平台架构师，前盛大架构师，专注分布式系统和移动应用，有十三年的开发经验，目

前在点融做最爱的两件事情: 写代码和重构代码。

随着新一轮融资，点融网开始了大规模的扩张，需要各种优秀人才的加入，如果您觉得自

已够优秀，欢迎加入我们！