Paxos Made Simple 译文

原文名：Paxos Made Simple [PDF下载] Leslie Lamport 2001/11/01
翻译：phylipsbmy
原译文链接： http://duanple.blog.163.com/blog/static/709717672011440267333/
审校：Jerry Lee oldratlee<at>gmail<dot>com

译序

“在PODC2001会议上，我总是听到人们在抱怨Paxos算法是那么的难以理解。人们总是被那些古希腊的名称弄得晕头转向，而使得他们觉得论文难以理解，实际上算法本身是很简单的。于是在会议期间我就找了几个人聚在一起，试着直接向他们口头解释该算法。回家之后，我将这些内容整理了下来，后来又基于Fred Schneider和Butler Lampson的建议做了修改。就形成了现在的这个版本，虽然已经有13页长了，但是其中仍未包含任何一个比 $n_1 > n_2$ 更复杂的公式。”

上面这段摘自Lamport的my writings，my writings是Lamport本人对自己以往发表的论文的一些总结，其中很多文字涉及到这些论文的创作来源。可以看出该论文的产生经历，与拜占庭将军问题有着截然相反的历程，在发表The Byzantine General Problem的时候，作者是用拜占庭将军这一场景引入到原来的算法中，而Paxos则是作者最初就是用古希腊的故事情节来描述，我想当时作者之所以采用一个故事性的背景，也是因为拜占庭将军这一写作方法带来的成功而受到的影响。只是事与愿违，人们觉得那篇《The Part-Time Parliament》 [5]
太难理解了，而且通篇没有数学化的公式证明。

根据Lamport的说法，当时的三个审稿人认为这篇文章虽然重要性不够但还有点意思，只是还应该把所有有关Paxos(Lamport在论文中虚构出的一个岛屿的名称)这一描述的地方全部删掉，但是Lamport觉得这些人太没幽默感了，也就没有按照他们要求的去做。以至于作者虽然在1990年就将它提交给了TOCS，但直到1998年才被发表。但是发表之后，很多人还是觉得原来那篇太难理解了，于是才产生了这一篇。不过现在回头再看，虽然当时Lamport的写作方式令文章被埋没了数年，但是也正因此才产生了如此有趣的一则轶闻，Paxos也成为该算法无可争议的名称，虽然另一篇文章《Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems》在1988年就独立地提出了类Paxos的一致性算法。

如果想及时了
解Spark、Hadoop或者Hbase相关的文章，欢迎关注微信公共帐号：iteblog_hadoop

1 导引

用于实现容错的分布式系统的Paxos算法，一直以来总是被认为很难理解，或许是因为对很多人来说，并不习惯以希腊故事展开的论文形式 [5] 。事实上，它应该是众多分布式算法中最简单易见的一个了。它的核心就是一个一致性算法——论文 [5] 中的“synod”算法。从下一个章节可以看出，它基本上就是根据一个一致性算法所必需满足的条件而顺理成章的呈现出来的。最后一个章节，我们还会通过将Paxos算法作为一个使用状态机的分布式系统构建模式中的一致性实现部分，来完整的描述它。这种使用状态机的方法来构建分布式系统最初是由论文 [4] 引入的，早已为人所熟知，而这篇论文估计也已经是分布式系统理论研究领域被引用地最广泛的了。

2 一致性算法

2.1 问题描述

假设有一组可以提出提案的进程集合。一个一致性算法需要保证：在这些被提出的提案中，只有一个会被选定；如果，没有提案被提出，那么就不会有被选定的提案；当一个提案被选定后，进程应该可以获取被选定的提案信息。

对于一致性来说，安全性(Safety)需求就是这样的：

- 只有被提出的提案才能被选定。
- 只能有一个值被选定(chosen)，同时
- 如果某个进程认为某个提案被选定了，那么这个提案必须是真的被选定的那个。

我们不会精确地描述活性(Liveness)需求。整体上来说，目标就是要保证最终有一个提案会被选定，当提案被选定后，进程最终也能获取到被选定的提案。 [†]译注

在该一致性算法中，有三种参与角色，我们用 Proposers ， Acceptors 和 Learners 来表示。在具体的实现中，一个进程可能充当不止一种角色，在这里我们并不关心进程如何映射到各种角色。

假设不同参与者之间可以通过发送消息来通信，我们使用普通的非拜占庭模式的异步模型：

- 每个参与者以任意的速度执行，可能会出错而停止，也可能会重启。当一个提案被选定后，所有的参与者都有可能失败或重启，因此除非那些失败或重启的参与者可以记录某些信息，否则是不可能存在一个解的。
- 消息在传输中可能花费任意的时间，可能会重复，丢失，但是不会被损坏。 [‡]译注

2.2 提案的选定

选定提案的最简单方式就是只允许一个accpetor存在。Proposer发送提案给accpetor，Acceptor会选择它接收到的第一个提案作为被选定的提案。尽管简单，但是这个解决方式却很难让人满意，因为如果accpetor出错，那么整个系统就无法工作了。

因此，应该选择其他的方式。比如我们用多个accpetor来避免一个accpetor时的单点问题。现在，Proposer向一个Acceptor集合发送提案，某个Acceptor可能会通过(accept)这个提案。当有足够多的Acceptor通过(accept)它的时候，我们就可以认为这个提案被选定了。什么是足够多呢？为了能确保只有一个提案被选定，我们可以让这个集合大的可以包含Acceptor集合中的多数成员。因为任何两个多数集至少有一个公共成员，如果我们再规定一个Acceptor最多只能通过一个提案，那么就能保证只有一个提案被选定(这是对于很多论文都研究过的majority的一个简单的应用 [3])。

在没有失败和消息丢失的情况下，如果我们希望即使在只有一个提案被提出的情况下，仍然可以选出一个提案来，这就暗示了如下这个需求：

P1 . 一个Acceptor必须通过(accept)它收到的第一个提案。

但是这个需求引出了另外一个问题。如果有多个提案被不同的Proposers同时提出，这可能会导致虽然每个Acceptor都通过了一个提案，但是没有一个提案是由多数人都通过的。即使是只有两个提案，如果每个都被差不多一半的Acceptors通过了，此时即使只有一个Acceptor出错都可能使得无法确定该选定哪个提案。 [§]译注

P1 再加上一个提案被选定需要由半数以上的Acceptor通过的需求暗示着一个Acceptor必须能够通过(accept)不止一个提案。我们为每个提案设定一个编号来记录一个Acceptor通过的那些提案。为了避免混淆，需要保证不同的提案具有不同的编号。如何实现这种保证取决于实现，目前我们假设已经提供了这种保证。当一个具有某value值的提案被半数以上的Acceptor通过后，我们就认为该value被选定了。此时我们也认为该提案被选定了。 [¶]译注

我们允许多个提案被选定，但是我们必须要保证所有被选定的提案都具有相同的value值。在提案编号上规约，它需要保证：

P2 . 如果具有value值v的提案被选定(chosen)了，那么所有比它编号更高的被选定的提案

的value值也必须是 v 。

因为编号是全序的，条件 P2 就保证了只有一个value值被选定的这一关键安全性属性。

被选定的提案，首先必须被至少一个Acceptor通过，因此我们可以通过满足如下条件来满足 P2 ：

P2 [a]. 如果具有value值 v 的提案被选定(chosen)了，那么所有比它编号更高的被Acceptor通过的提案的value值也必须是 v。

我们仍然需要 P1
来保证提案会被选定。但是因为通信是异步的，一个提案可能会在某个Acceptor c 还未收到任何提案时就被选定了。假设一个新的Proposer苏醒了，然后产生了一个具有另一个value值的更高编号的提案，根据 P1 ，就需要 c 通过这个提案，但是这与 P2 [a] 矛盾。因此如果要同时满足 P1 和 P2 [a] ，需要对 P2[a] 进行强化：

P2 [b]. 如果具有value值v的提案被选定，那么所有比它编号更高的被Proposer提出的提案的value值也必须是 v 。

一个提案被Acceptor通过之前肯定要由某个Proposer提出，因此 P2 [b] 就隐含了 P2 [a] ，进而隐含了 P2 。

为了发现如何保证 P2 [b] ，我们来看看如何证明它成立。我们假设某个具有编号m和value值v的提案被选定了，需要证明具有编号 n(n > m) 的提案都具有value值 v 。我们可以通过对 n 使用归纳法来简化证明，这样我们就可以在额外的假设下——即编号在 m..(n-1) 之间的提案具有value值 v ，来证明编号为n的提案具有value值 v 。因为编号为m的提案已经被选定了，这意味着肯定存在一个由半数以上的Acceptor组成的集合 C ， C 中的每个Acceptor都通过了这个提案。再结合归纳假设， m 被选定意味着：

C 中的每个Acceptor都通过了一个编号在 m..n-1 之间的提案，每个编号在 m..(n-1) 之间的被Acceptor通过的提案都具有value值 v 。

因为任何包含半数以上的Acceptor的集合S都至少包含 C 中的一个成员，我们可以通过维护如下不变性就可以保证编号为n的提案具有value v ：

P2 [c]. 对于任意的 n 和 v ，如果编号为 n 和value值为 v 的提案被提出，那么肯定存在一个由半数以上的Acceptor组成的集合 S ，可以满足条件 a) 或者 b) 中的一个：

1. S 中不存在任何的Acceptor通过过编号小于 n 的提案
2. v 是 S 中所有Acceptor通过的编号小于 n 的具有最大编号的提案的value值。

通过维护 P2 [c] 我们就可以保证 P2 [b] 了。 [#][译注]

为了维护 P2 [c] 的不变性，一个Proposer在产生编号为 n 的提案时，必须要知道某一个将要或已经被半数以上Acceptor通过的编号小于 n 的最大编号的提案。获取那些已经被通过的提案很简单，但是预测未来会被通过的那些却很困难。在这里，为了避免让Proposer去预测未来，我们通过限定不会有那样的通过情况来控制它。换句话说，Proposer会请求Acceptors不要再通过任何编号小于 n 的提案。这就导致了如下的提案生成算法：

1. Proposer选择一个新的提案编号 n ，然后向某个Acceptors集合的成员发送请求，要求Acceptor做出如下回应：

    1. 保证不再通过任何编号小于 n 的提案
    2. 当前它已经通过的编号小于 n 的最大编号的提案，如果存在的话。

    我们把这样的一个请求称为编号为 n 的prepare请求。

2. 如果Proposer收到了来自半数以上的Acceptor的响应结果，那么它就可以产生编号为 n ，value值为 v 的提案，这里 v 是所有响应中编号最大的提案的value值，如果响应中不包含任何的提案那么这个值就可以由Proposer任意选择。

Proposer通过向某个Acceptors集合发送需要被通过的提案请求来产生一个提案（此时的Acceptors集合不一定是响应前一请求的那个Acceptors集合）。我们称此请求为 accept 请求。

目前我们描述了Proposer端的算法，Acceptor端是怎样的呢？它可能会收到来自Proposer端的两种请求：prepare请求和accept请求。Acceptor可以忽略任何请求而不用担心破坏其算法的安全性。因此我们只需要说明它在什么情况下可以对一个请求做出响应。它可以在任何时候响应一个prepare请求，对于一个accept请求，只要在它未违反现有承诺的情况下才能响应一个accept请求(通过对应的提案)。换句话说：

> P1 [a]. 一个Acceptor可以接受一个编号为 n 的提案，只要它还未响应任何编号大于 n 的prepare请求。

可以看出 P1 [a] 蕴含了 P1 。

我们现在就获得一个满足安全性需求的提案选定算法—假设编号唯一的情况下。再做一些小的优化就得到了最终的算法。

假设一个Acceptor收到了一个编号为 n 的prepare请求，但是它已经对编号大于 n 的prepare请求做出了响应，因此它肯定不会再通过任何新的编号为n的提案，那么它就没有必要对这个请求做出响应，因为它肯定不会通过编号为 n 的提案，因此我们会让Acceptor忽略这样的prepare请求。我们也会让它忽略那些它已经通过的提案的prepare请求。

通过这个优化，Acceptor只需要记住它已经通过的最大编号的提案以及它已经做出prepare请求响应的最大编号的提案的编号。因为必须要保证 P1^c 的不变性即使在出错的情况下，Acceptor必须记住这些信息即使是在出错或者重启的情况下。Proposer可以总是可以丢弃提案以及它所有的信息—只要它可以保证不会产生具有相同编号的提案即可。

将Proposer和Acceptor放在一块，我们可以得到算法的如下两阶段执行过程：

Phase 1.

1. Proposer选择一个提案编号 n ，然后向Acceptors的某个majority集合的成员发送编号为 n 的prepare请求。
2. 如果一个Acceptor收到一个编号为 n 的prepare请求，且 n 大于它已经响应的所有prepare请求的编号，那么它就会保证不会再通过(accept)任何编号小于 n 的提案，同时将它已经通过的最大编号的提案(如果存在的话)作为响应。 [♠]译注

Phase 2.

1. 如果Proposer收到来自半数以上的Acceptor对于它的prepare请求(编号为 n)的响应，那么它就会发送一个针对编号为 n ，value值为 v 的提案的accept请求给Acceptors，在这里 v 是收到的响应中编号最大的提案的值，如果响应中不包含提案，那么它就是任意值。
2. 如果Acceptor收到一个针对编号 n 的提案的accept请求，只要它还未对编号大于 n 的prepare请求作出响应，它就可以通过这个提案。

一个Proposer可能或产生多个提案，只要它是遵循如上的算法即可。它可以在任意时刻丢弃某个提案。(即使针对该提案的请求和响应在提案被丢弃很久后才到达，正确性依然可以保证)。如果某个Proposer已经在试图生成编号更大的提案，那么丢弃未尝不是一个好的选择。因此如果一个Acceptor因为已经收到更大编号的prepare请求而忽略某个prepare或者accept请求时，那么它也应当通知它的Proposer，然后该Proposer应该丢弃该提案。当然，这只是一个不影响正确性的性能优化。

2.3 获取被选定的提案值

为了获取被选定的值，一个Learner必须确定一个提案已经被半数以上的Acceptor通过。最明显的算法是，让每个Acceptor，只要它通过了一个提案，就通知所有的Learners，将它通过的提案告知它们。这可以让Learners尽快的找出被选定的值，但是它需要每个Acceptor都要与每个Learner通信—所需通信的次数等于二者个数的乘积。

在假定非拜占庭错误的情况下，一个Learner可以很容易地通过另一个Learner了解到一个值已经被选定。我们可以让所有的Acceptor将它们的通过信息发送给一个特定的Learner，当一个value

被选定时，再由它通知其他的Learners。这种方法，需要多一个步骤才能通知到所有的Learners。而且也是不可靠的，因为那个特定的Learner可能会失败。但是这种情况下的通信次数，只是Acceptors和Learners的个数的和。

更一般的，Acceptors可以将它们的通过信息发送给一个特定的Learners集合，它们中的每个都可以在一个value被选定后通知所有的Learners。这个集合中的Learners个数越多，可靠性就越好，但是通信复杂度就越高。

由于消息的丢失，一个value被选定后，可能没有Learners会发现。Learner可以询问Acceptors它们通过了哪些提案，但是一个Acceptor出错，都有可能导致无法判断出是否已经有半数以上的Acceptors通过的提案。在这种情况下，只有当一个新的提案被选定时，Learners才能发现被选定的value。因此如果一个Learner想知道是否已经选定一个value，它可以让Proposer利用上面的算法产生一个提案。 [♥]译注

2.4 进展性

很容易构造出一种情况，在该情况下，两个Proposers持续地生成编号递增的一系列提案，但是没有提案会被选定。Proposer p 为一个编号为 n_1 的提案完成了Phase1，然后另一个Proposer q 为编号为 n_2 ($n_2 > n_1$) 的提案完成了Phase1。Proposer p 的针对编号 n_1 的提案的Phase2的所有accept请求被忽略，因为Acceptors已经承诺不再通过任何编号小于 n_2 的提案。这样Proposer p 就会用一个新的编号 n_3 ($n_3 > n_2$) 重新开始并完成Phase1，这又会导致在Phase2里Proposer q 的所有accept请求被忽略，如此循环往复。

为了保证进度，必须选择一个特定的Proposer来作为一个唯一的提案提出者。如果这个Proposer可以同半数以上的Acceptors通信，同时可以使用一个比现有的编号都大的编号的提案的话，那么它就可以成功的产生一个可以被通过的提案。再通过当它知道某些更高编号的请求时，舍弃当前的提案并重做，这个Proposer最终一定会产生一个足够大的提案编号。

如果系统中有足够的组件(Proposer，Acceptors及通信网络)工作良好，通过选择一个特定的Proposer，活性就可以达到。著名的FLP结论 [1] 指出，一个可靠的Proposer选举算法要么利用随机性要么利用实时性来实现—比如使用超时机制。然而，无论选举是否成功，安全性都可以保证。 [♦]译注

2.5 实现

Paxos算法 [5] 假设了一组进程网络。在它的一致性算法中，每个进程扮演着Proposer，Acceptor及Learner的角色，该算法选定一个Leader来扮演那个特定的Proposer和Learner。Paxos一致性算法就是上面描述的那个，请求和响应都是以普通消息的方式发送(响应消息通过对应的提案的编号来标识以防止混淆)。使用可靠性的存储设备来存储Acceptor需要记住的信息来防止出错。Acceptor在真正送出响应之前，会将它记录在可靠性存储设备中。

剩下的就是需要描述保证提案编号唯一性的机制了。不同的Proposers会从不相交的编号集合中选择自己的编号，这样任何两个Proposers就不会有相同编号的提案了。每个Proposer需要将它目前生成的最大编号的提案记录在可靠性存储设备中，然后用一个比已经使用的所有编号都大的

提案开始Phase1。

3 实现状态机模型

实现分布式系统的一种简单方式就是，使用一组客户端集合然后向一个中央服务器发送命令。服务器可以看成是一个以某种顺序执行客户端命令的确定性状态机。该状态机有一个当前状态，通过输入一个命令来产生一个输出以及一个新的状态。比如一个分布式银行系统的客户端可能是一些出纳员，状态机状态由所有用户的账户余额组成。一个取款操作，通过执行一个减少账户余额的状态机命令(当且仅当余额大于等于取款数目时)实现，将新旧余额作为输出。

使用中央服务器的系统在该服务器失败的情况下，整个系统就失败了。因此，我们使用一组服务器来代替它，每个服务器都独立了实现了该状态机。因为状态机是确定性的，如果它们都按照相同的命令序列执行，那么就会产生相同的状态机状态和输出。一个产生命令的客户端，就可以使用任意服务器为它产生的输出。

为了保证所有的服务器都执行相同的状态机命令序列，我们需要实现一系列独立的Paxos一致性算法实例，第 i 个实例选定的值作为序列中的第 i 个状态机命令。在算法的每个实例中，每个服务器担任所有的角色(Proposer、Acceptor和Learner)。现在，我们假设服务器集合是固定的，这样所有的一致性算法实例都具有相同的参与者集合。

在正常执行中，一个服务器会被选为Leader，它会在所有的一致性算法实例中被选作特定的Proposer(唯一的提案提出者)。客户端向该Leader发送命令，它来决定每个命令被安排在序列中的何处。如果Leader决定某个客户端命令应该是第135个命令，它会尝试让该命令成为第135个一致性算法实例选定的value值。通常，这都会成功，但是由于出错或者另一个服务器也认为自己是Leader，而它对第135个命令应该持有异议。但是一致性算法可以保证，最多只有一个命令会被选定为第135个命令。

这种策略的关键在于，在Paxos一致性算法中，被提出的value只有在Phase2才会被选定。回忆下，在Proposer的Phase1完成时，要么提案的value已确定，要么Proposer可以自由地提出一个值。

现在我们已经知道在正常运行时，Paxos状态机实现是如何工作的。下面我们看下出错的情况，看下之前的Leader失败以及新Leader被选定后会发生什么。(系统启动是一种特殊情况，此时还没有命令被提出)。

新的Leader选出来后，首先要成为所有一致性算法执行实例的Learner，需要知道目前已经选定的大部分命令。假设它知道命令1-134,138及139—也就是一致性算法实例1-134,138及139选定的值(稍后，我们会看下命令间的缺口是如何形成的)。然后，它需要执行135-137以及所有其他大于139的算法执行实例的Phase1(下面会描述如何来做，即如何为这无限多个实例执行Phase1)。假设执行结果表明，将要在执行实例135和140中被提出的提案值已经确定，但是其他执行实例的提案值是没有限制的 [♣]译注
。那么现在该Leader就可以执行实例135和140的Phase2，进而选定第135和140号命令。

Leader以及其他所有已经获取该Leader的已知命令的服务器，现在可以执行命令1-135。然而它还不能执行138-140，因为目前为止命令136和137还未选定。Leader可以将下两个到来的客户端

请求命令作为命令136和137。但是我们也可以提起一个特殊的“noop”命令作为136和137号命令来填补这个空缺，(通过执行一致性算法实例136和137的Phase2来完成) [**]译注 。一旦该noop命令被选定，命令138-140就可以执行了。

命令1-140目前已被选定了。Leader也已经完成了所有大于140的一致性算法实例的Phase1，而且在这些实例中，它可以自由的提出任何值。它将下一个客户端的请求命令作为第141个命令，并且在Phase2中将它作为一致性算法的第141个实例的value值。它会将下一个客户端的请求命令作为命令142，如此...

Leader可以在它提出的命令141被选定前提出命令142。它发送的关于命令141的消息有可能全部丢失，因此在所有其他服务器在获知Leader选定了谁作为命令141之前，命令142就可能已被选定。当Leader无法收到实例141的Phase2的期望响应之后，它会重传这些信息，但是仍然可能会失败，这样就在被选定的命令序列中，出现了缺口。假设一个Leader可以提前确定 α 个命令，这意味着在i被选定之后，它就可以提出命令 $i + 1$ 到 $i + \alpha$ 的命令了。这样就可能形成一个长达 $\alpha - 1$ 的命令缺口。

一个新选择的Leader需要为无数个一致性算法实例执行Phase1——在上面的情景中，就是135-137以及所有大于139的执行实例。只要向其他的服务器发送一个合适的消息内容，就可以让所有的执行实例使用同一个的提案编号计数器 [††]译注 。在Phase1，只要一个Acceptor已经收到来自某个Proposer的Phase2消息，那么它就可以为不止一个的执行实例做出承诺。（在上面的场景中，就是针对135和140的情况。）因此一个服务器（作为Acceptor角色时）通过选择一个适当的短消息就可以对所有实例做出响应，那么执行这样无限多个实例的Phase1也就不会有问题 [‡‡]译注 。 [§§]译注

因为Leader的失败和新Leader的选举都是很少见的情况，因此执行一个状态机命令—即在命令值上达成一致性的花费就是执行该一致性算法的Phase2的花费 [¶¶]译注 。可以证明，在允许失效的情况下，在所有的一致性算法中，Paxos一致性算法的Phase2具有最小可能的 时间 复杂度 [2] 。因此Paxos算法基本就是最优的。

在该系统的正常执行情况下，我们假设总是只有一个Leader，只有在当前Leader失效及选举新Leader的较短时期内才会违背这个假设。在特殊情况下，Leader选举可能失败。如果没有服务器担任Leader，那么就没有新命令被提出。如果同时有多个服务器认为自己是Leader，它们在一个一致性算法执行实例中可能提出不同的value，这可能会导致无法选出一个value。但是，安全性一直都可以保证—即不可能会同时有两个命令被选定为第i个状态机命令。Leader的选举只是为了保证progress。

如果服务器集合是变化的，那么必须有某种方式来决定哪些服务器来实现哪些一致性算法实例。最简单的方式就是通过状态机本身来完成。当前的服务器集合可以作为状态的一部分，同时可以通过某些状态机命令来改变。同时通过用执行完第 i 个状态机命令后的状态来描述执行一致性算法实例 $i + \alpha$ 的服务器集合，我们就能让Leader在执行完第 i 个状态机命令后可以提前获取 α 个状态机命令 [##]译注 。这就提供了一种支持任意复杂的重配置算法的简单实现。 [♠♠]译注

参考文献

[1] Michael J. Fischer, Nancy Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. Journal of the ACM, 32(2):374–382, April 1985.
[2] Idit Keidar and Sergio Rajsbaum. On the cost of fault-tolerant consensus when there are no faults—a tutorial. TechnicalReport MIT-LCS-TR-821, Laboratory for Computer Science, Massachusetts Institute Technology, Cambridge, MA, 02139, May 2001. also published in SIGACT News 32(2) (June 2001).
[3] Leslie Lamport. The implementation of reliable distributed multiprocess systems. Computer Networks, 2:95–114, 1978.
[4] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. Communications of the ACM, 21(7):558–565, July 1978.
[5] 1, 2, 3, 4) Leslie Lamport. The part-time parliament. ACM Transactions on Computer Systems, 16(2):133–169, May 1998.

注释

[†] 译注。一个分布式算法，有两个最重要的属性：Safety 和Liveness，简单来说：

- Safety是指那些需要保证永远都不会发生的事情。
- Liveness是指那些最终一定会发生的事情。

[‡] 译注。即其内容不会被篡改，不会发生拜占庭式的问题。
[§] 译注。比如有5个Acceptor，其中2个通过了提案 a ，另外3个通过了提案 b ，此时如果通过 b 的3个中有一个出错了，那么 a 、 b 的通过者都变成了2个，这不清楚该如何决定了。
[¶] 译注。此时的提案已经跟value变成了不同的东西，提案是由编号+value组成的。
[#] 译注。可以看到上面是对一系列条件的逐步加强，如果需要证明它们可以保证一致性，则需要反过来， $P2^c$ => $P2^b$ => $P2^a$ => P2 ， P2 + P1 => 保证了一致性。

我们再看 $P2^c$ ，实际上 $P2^c$ 规定了每个Proposer 如何产生一个提案，对于产生的每个提案 (n, v) 需要满足这个条件“存在一个由超过半数的Acceptor 组成的集合 S ：要么 S 中没有人批准(accept)过编号小于 n 的任何提案，要么 S 的任何Acceptor批准的所有议案（编号小于 n ）中， v 是编号最大的议案的决议”。当Proposer遵守这个规则产生提案时，就可以保证满足 $P2^b$ 。论文中，作者是从如何产生提案进而可以保证 $P2^b$ 来思考，才得到 $P2^c$ 的。下面我们反过来看，证明 $P2^c$ 可以保证 $P2^b$ 。如论文中一样，采用数学归纳法证明。

首先假设提案 (m, v) 被选定了，设比该提案编号大的提案为 (n, v') ,我们需要证明的就是在 $P2^c$ 的前提下，对于所有的 (n, v') ，有 v' = v 。

1. n = m + 1 时，如果有这样编号的提案，首先我们知道 (m, v) 被选定了，这样就不可能存在一个 S 且 S 中没有人批准过小于 n 的提案（ S 与批准 (m, v) 的Acceptor集合肯定有交集），那 v' 只能是多数集 S 中编号小于 n 的最大编号的那个提案的值了，此时 n = m + 1 ，理论上小于n的最大的编号肯定是 m ，同时由于 S 和通过 (m, v) 的Acceptor集合都是多数集，就保证了二者肯定有交集，这样Proposer在确定 v' 取值时，肯定选到就是 v 。

上面实际上就是数学归纳法的第一步，确切的说是使用的是第二数学归纳法。上面是第一步，验证了某个初始值成立。下面，需要假设编号在 [m+1, k-1] 区间内成立，并在此基础上推出 n = k 上也成立。

1. 根据假设编号在 [m+1, k-1] 区间内的所有提案都具有值 v ，需要证明的是编号为k的提案也具有值 v 。根据 P2[c] ，首先同样的不可能存在一个 S 且 S 中没有人批准过小于 n 的提案，那么编号为 k 的value值，只能是一个多数集 S 中编号小于 n 的最大编号的那个提案的值，如果这个最大编号落在 [m+1, k-1] 区间内的，那么值肯定是 v ，如果不是落在 [m+1, k-1] 区间，那么它的编号肯定就是 m 了，不可能比 m 再小了，因为 S 也肯定会与批准 (m, v) 的Acceptor集合肯定有交集，那么它的编号值就不会比 m 小，而编号如果是 m 那么它的值也是 v 。由此得证。

[♠] 译注。 前提（即“如果”半句）中有“ n 大于它已经响应的 所有 prepare请求的编号”，所以可以知道返回这个提案肯定是小于 n 的，即使这个提案是 已经通过的最大编号的提案 。举个例子，假定Acceptor已经响应的请求编号是 1 、 3 、 4 ， n = 6（大于所有编号），则响应 已经通过的最大编号的提案 ，即是 4 。

[♥] 译注。上面这段的意思是，Acceptor发送给Learners的关于提案通过的相关信息可能会丢失，这样learns就无法知道是否有value被选定，此时呢它可以主动去询问Acceptors，但是此时如果被通过的提案刚好是由 n / 2 + 1 个Acceptor通过了，万一其中一个Acceptor出现问题，那么它无法确定被选定的提案，为了确定被选定的value，必须重新发起一次新的提案。

但是这引出一种需要考虑的异常情况，当一个值被半数+1的Acceptor选定后，但是其中一个Acceptor出错而死掉了，那么对于这种情况，Paxos算法能否正确处理呢？因为这种情况下，某个Learner可能会在这个Acceptor还活着的时候获知这个选定的value，但是其他Learner获取信息时该Acceptor可能已经死掉了。对于这种情况虽然Learner可能一时无法判断哪个value被选定了，但是它可以保证此时被选定的value，将一直是被选定的那个value，因为如果Acceptor出错死掉了，但这并不影响保证多数集之间肯定存在一个交，因为该出错的Acceptor对于两个多数集来说，它们都是死掉的那个，根据算法执行过程，我们可以看到多数集都是通过接受响应来体现的，也就是说它们肯定都是还活着的Acceptor，这样不同执行过程中的Phase2的多数集之间肯定存在一个还活着的公共Acceptor。如果一个死掉的Acceptor巧合是两个（ n / 2 + 1 ）多数集唯一的公共元素，那么它应该是无法满足收到多数集的Acceptor的响应的。

[♦] 译注。即使同时有2个或以上的Proposers存在，算法仍然可以保证正确性。

[♣] 译注。根据前面所述经过Phase1，要么提案value值已确定，要么Proposer可以自由提出一个值，那么此处即指135和140的提案value已确定，而其他的则可选任意值，所以下面才能为136和137选一个新来的命令或者是那个特殊的noop命令。

[**] 译注。 通过前面我们已经知道换了新Leader后，Leader已经执行了它们的Phase1，这样就可以直接执行Phase2，同时Phase1的执行结果表明136和137的value值可以任意选择。此处，noop命令不会改变状态机状态，实际上是个虚命令，使用它的意义在于因为命令139和140都确定好了，直接选择一个noop就可以避免额外的命令查找或者等待，就可以尽快填补空缺，从而让139和140尽快执行，降低命令执行的等待时间。

[††] 译注。原文是“Using the same proposal number”，按字面翻译是“使用相同的提案编号”，使用的译文是“使用同一个的提案编号计数器”。根据前面算法可知，编号是会不断增长的，不会使用一个相同不变的编号，所以原文中“相同提案编号”，实际指得是“实现中的编号计数器”。另外，详细完整的多实例Paxos算法（Multi-Paxos算法）在论文《Paxos Made Moderately Complex》中有给出，算法正是所有实例使用同一编号计数器。 [Jerry Lee 注]

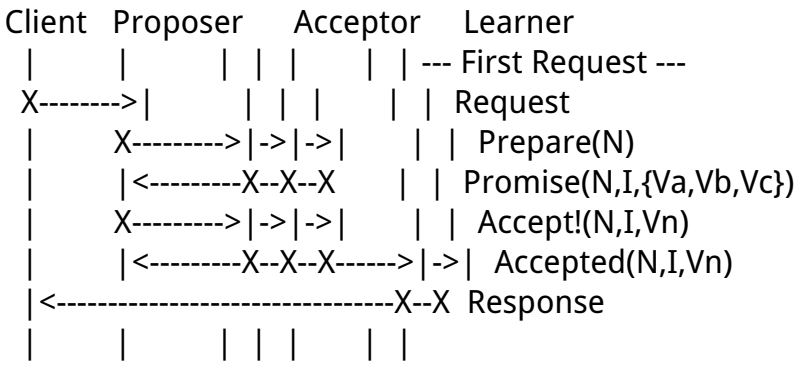
[‡‡] 译注。因为执行实例使用的都是同一个的提案编号计数器，这样它承诺不再通过小于 n 的提案，应该可以应用在所有执行实例上，而不影响正确性。

[§§] 译注。此处应该可以算是对于多个Paxos执行实例同时运行的情况的优化，内容类似于Wiki中提到的 Multi-Paxos 模式。根据wiki上的描述，如果Leader是相对稳定的，那么Phase1可能就是不必要的了，那么对于同一个Leader未来会参与的那些执行实例，是可以直接跳过Phase1的。但是，需要在每个value值中加上执行实例的编号。

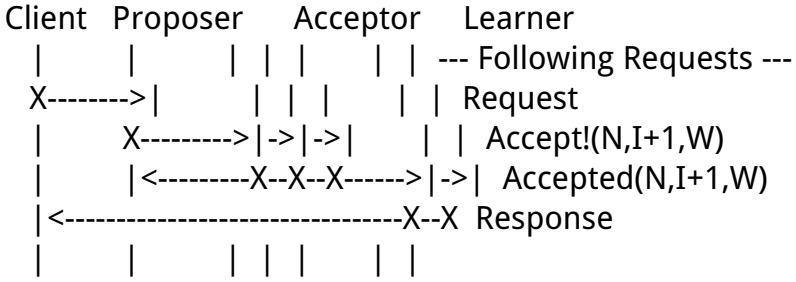
该模式执行过程如下(图中一个竖线应该认为是一个参与者，比如Acceptor下有三个竖线，代表由三个Acceptor)。从图中可以看出，只有第一个执行执行了prepare过程，而在Leader进入稳定状态后，后续的执行实例直接进入了Phase2，同时执行实例的编号(即图中的I)被加入到了消息中：

Message flow: Multi-Paxos, start

(first instance with new leader)



Message flow: Multi-Paxos, steady-state (subsequent instances with same leader)



[¶¶] 译注。那Phase1的花费呢？解释如下：“Leader的失败和新Leader的选举都是很少见的情况”，换句话说，大部分时间里Leader正常。Leader正常时，如果Majority的Proposer在Phase1承诺了编号 n ，由于所有执行实例用同一个的提案编号计数器，即所有实例的Phase1都完成了，之后只提交Phase2消息即可。 [Jerry Lee 补注]

[##] 译注。即在服务器集合可变的情况下，也能预取命令，就需要我们能知道确定该命令的一致性算法执行实例对应的服务器集合，这里提供了一个简单的服务器集合决定方式，也就是说我们既然将服务器集合作为状态机状态的一部分，那么我们就将在执行完第 i 个状态机命令后标识的服务器集合，作为一致性算法执行实例 i + α 的服务器集合。比如我们把第0个状态机命令执行后的服务器集合，作为实现第 i 个的一致性算法实例的服务器集合，第1个状态机命令执行后的服务器集合，作为实现第 i + 1 个的一致性算法实例的服务器集合，依次类推。

[♠ ♠] 译注。实际上呢这就允许我们通过发送一个改变服务器集合的命令来动态的改变执行第 n 个一致性算法的服务器集合，也就是实现了动态重配置的目的。因为该命令会改变直接服务器集合，那么就能影响到后续的执行实例。