2001 年 11 月 1 日

Paxos 简化

Leslie Lamport

2025 年 10 月 16 日

摘要

Paxos 算法用普通的英语描述时非常简单。

目录

1 引言

实现容错分布式系统的 Paxos 算法被认为难以理解，也许是因为原始描述对许多读者而言晦涩难懂 [5]。事实上，它是最简单且最直观的分布式算法之一。其核心是一个共识算法——即 [5] 中的“议事会”算法。下一节将表明，该共识算法几乎不可避免地由我们希望它满足的性质推导出来。最后一节将解释完整的 Paxos 算法，它是通过将共识机制直接应用于构建分布式系统的状态机方法而得到的——这种方法应当是广为人知的，因为它是关于分布式系统理论中被引用最多的文章的主题 [4]。

2 共识算法

2.1 问题

假设一组可以提出值的过程。共识算法确保从所提出的值中选出一个单一值。如果没有任何值被提出，则不应选择任何值。如果某个值已经被选择，则进程应当能够得知该被选择的值。共识的安全性要求如下：

- 只有被提出的值才能被选择，
- 仅选择一个值，且
- 进程只有在该值实际上已被选择时，才会得知该值已被选择。

我们不试图明确指定活锁要求。然而，目标是确保最终会选出某个提出的值，且如果某个值已被选择，则进程最终能够得知该值。

我们将共识算法中的三个角色由三类代理来执行：提议者（proposers）、接受者（acceptors）和学习者（learners）。在实现中，单个进程可能充当多个代理的角色，但这里我们并不关心代理到进程的映射关系。

假设代理之间可以通过发送消息进行通信。我们采用通常的异步、非拜占庭模型，其中：- 引擎以任意速度运行，可能通过停止而失效，也可能重启。由于在选定一个值之后，所有引擎都可能失效并重启，因此除非某个已失效并重启的引擎能够记住一些信息，否则无法找到解决方案。

- 消息的传递可能耗时极长，可能发生重复，也可能丢失，但不会被破坏。

2.2 选择一个值

选择值的最简单方法是使用一个单一的接受者引擎。提议者将一个提议发送给接受者，接受者选择其接收到的第一个提议值。尽管这种方法简单，但当接受者失效时，将导致任何进一步的进展变得不可能。

因此，我们尝试另一种选择值的方法。不再使用单一的接受者引擎，而是使用多个接受者引擎。提议者将一个提议值发送给一组接受者引擎。每个接受者引擎可以接受该提议值。当足够多的接受者引擎接受了该值时，该值就被选定。那么，“足够多”是多少呢？为了确保仅选择一个值，我们可以让“足够多”表示所有引擎中的多数。因为任意两个多数至少共享一个接受者引擎，只要接受者引擎最多只能接受一个值，这种方案就成立。（在许多论文中已经观察到一种明显的多数推广，似乎最早出现在 [3] 中。）

在没有失效或消息丢失的情况下，我们希望即使只有一个提议者提出一个值，也能选出该值。这提示我们提出如下要求：

P1. 接受者引擎必须接受其接收到的第一个提议。

然而，这一要求会引发一个问题。不同提议者可能在相近时间提出多个值，导致每个接受者引擎都接受了某个值，但没有任何一个值被多数接受者引擎所接受。即使只有两个提议值，如果每个值被大约一半的接受者引擎接受，那么单个接受者引擎的失效可能导致无法确定哪个值被最终选择。

P1 以及值只有在被多数接受者引擎接受时才被选定的要求，意味着接受者引擎必须被允许接受多个提议。我们通过为每个提议分配一个（自然数）编号来跟踪接受者引擎可能接受的不同提议，因此每个提议由一个提议编号和一个值组成。为避免混淆，我们要求不同的提议必须具有不同的编号。如何实现这一要求所实现的值取决于实现方式，因此目前我们仅作假设。当某个具有特定值的提案被多数接受者接受时，该值被选定。此时，我们说该提案（及其值）已被选定。

我们可以允许多个提案被选定，但必须保证所有被选定的提案具有相同的值。通过提案编号的归纳法，只需保证：

P2. 如果一个值为 v 的提案被选定，那么所有被选定的更高编号提案的值均为 v。

由于数值是全序的，条件 P2 保证了关键的安全性属性：仅有一个值被选定。

一个提案要被选定，必须至少被一个接受者接受。因此，我们可以通过满足以下条件来满足 P2：

P2^a。如果一个值为 v 的提案被选定，那么任何接受者接受的更高编号提案的值均为 v。

我们仍然保持 P1 以确保至少有一个提案被选定。由于通信是异步的，一个特定接受者 c 可能从未接收到任何提案，却仍导致某个提案被选定。假设一个新的提案者“醒来”并发出一个更高编号、不同值的提案。P1 要求 c 接受该提案，这将违反 P2^a。为了同时维持 P1 和 P2^a，我们需要将 P2^a 加强为：

P2^b。如果一个值为 v 的提案被选定，那么任何提案者发出的更高编号提案的值均为 v。

由于一个提案必须由提案者发出后才能被接受者接受，因此 P2^b 蕴含 P2^a，而 P2^a 又蕴含 $P2$。

为了发现如何满足 P2^b，我们考虑如何证明其成立。我们假设某个编号为 m、值为 v 的提案被选定，并证明任何编号大于 m 的提案也具有值 v。为了简化证明，我们对 n 进行归纳，即在额外假设下证明编号为 n 的提案具有值 v：即所有编号在 $m \ldots (n-1)$ 范围内的提案值均为 v，其中 $i \ldots j$ 表示从 i 到 j 的所有编号集合。为了使编号为 m 的提案被选定，必须存在一个由多数接受者组成的集合 C，使得 C 中的每个接受者都接受了该提案。结合归纳假设，m 被选定这一假设意味着：每一个属于 C 的接受器都已接受了一个编号在 $m..(n-1)$ 之间的提案，且每一个编号在 $m..(n-1)$ 之间的提案，如果被任何一个接受器接受，则其值为 v。

由于任何由多数接受器组成的集合 S 都至少包含 C 中的一个成员，因此我们可以通过保持以下不变式来得出编号为 n 的提案具有值 v：

P2^c。对于任意的 v 和 n，如果发出一个值为 v、编号为 n 的提案，则存在一个由多数接受器组成的集合 S，使得要么（a）S 中的任一接受器都没有接受过编号小于 n 的提案，要么（b）v 是 S 中所有接受器所接受的编号小于 n 的提案中编号最高的提案的值。

因此，通过保持 P2^c 的不变性，我们可以满足 P2^b。

为了保持 P2^c 的不变性，若某提案者希望发出一个编号为 n 的提案，则必须了解在某个多数接受器集合中的每个接受器所接受或将来会接受的、编号小于 n 的最高编号提案。了解已经接受的提案并不困难；预测未来的接受行为则很困难。与其尝试预测未来，提案者通过提取一个承诺来控制未来，即承诺不会发生此类接受行为。换句话说，提案者请求接受器们不再接受任何编号小于 n 的提案。这导致了如下发出提案的算法。

1. 提案者选择一个新的提案编号 n，并向某个接受器集合中的每个成员发送一个请求，要求其作出如下响应：

(a) 承诺不再接受任何编号小于 n 的提案；

(b) 如果有，则提供其已接受的编号小于 n 的最高编号提案。

我将此类请求称为编号为 n 的准备请求。

2. 如果提案者从多数接受器收到了所需的响应，则可以发出一个编号为 n、值

为 v 的提案，其中 v 是响应中编号最高的提案的值，或者如果响应者报告没有提案，则由提案者任意选择一个值。一个提议者通过向一组接受者发送请求，要求接受该提议。（这组接受者不必与最初请求所响应的接受者相同。）我们将这种请求称为“接受请求”。

这描述了提议者的算法。那么，接受者呢？它可以从提议者接收到两种类型的请求：准备请求和接受请求。接受者可以忽略任何请求而不会损害安全性。因此，我们只需说明在什么情况下它被允许对请求做出响应。它总是可以响应一个准备请求。它可以在接受一个提议时响应一个接受请求，当且仅当它尚未承诺不接受。换句话说：

P1^a。一个接受者可以接受一个编号为 n 的提议，当且仅当它没有对一个编号大于 n 的准备请求做出响应。

注意到 P1^a 包含了 P1。

现在，我们已经拥有了一个完整的算法，用于选择一个值并满足所需的正确性属性——假设提案编号是唯一的。最终的算法通过一个微小的优化得到。

假设一个接受者收到了一个编号为 n 的准备请求，但它已经对一个编号大于 n 的准备请求做出响应，从而承诺不接受任何编号为 n 的新提议。此时，该接受者没有理由响应这个新的准备请求，因为即使提议者想要发布编号为 n 的提议，它也不会被接受。因此，我们让接受者忽略这样的准备请求。我们还让它忽略那些已经接受过的提议的准备请求。

通过这个优化，接受者只需记住它曾经接受过的最高编号的提议，以及它已经响应过的最高编号的准备请求。由于 P2^c 必须在任何故障情况下保持不变，即使接受者发生故障并重启，它也必须记住这些信息。注意，提议者可以随时放弃一个提议并完全忘记它——只要它从未尝试使用相同的编号再次发布另一个提议。

将提议者和接受者的操作结合起来，我们可以看到该算法运行在以下两个阶段。

第一阶段。(a) 提议者选择一个提案编号 n，并向多数接受者发送一个编号为 n 的准备请求。

(b) 如果一个接受者接收到一个编号为 n 的准备请求，且该编号大于它之前已响应过的任何准备请求的编号，那么它将对该请求做出响应，承诺不接受任何编号小于 n 的提议，并附带它已接受的最高编号提议（如果有）。第二阶段。(a) 如果提议者接收到编号为 n 的 prepare 请求来自多数接受者，则它会向这些接受者中的每一个发送一个编号为 n、值为 v 的 accept 请求，其中 v 是响应中编号最高的提案的值，如果没有响应报告任何提案，则 v 可以是任意值。

(b) 如果一个接受者接收到一个编号为 n 的 accept 请求，则它会接受该提案，除非它已经对一个编号大于 n 的 prepare 请求作出了响应。

提议者可以发出多个提案，只要它遵循每个提案的算法即可。它可以在协议进行过程中随时放弃一个提案。(即使请求和/或响应在提案被放弃后很久才到达目

的地，正确性仍然得以保持。）如果某个提议者已经开始尝试发出编号更高的提案，那么最好放弃当前的提案。因此，如果一个接受者因为已经接收到一个编号更高的 prepare 请求而忽略了 prepare 或 accept 请求，那么它应该通知提议者，提议者随后应放弃其提案。这是一种性能优化，不会影响正确性。

2.3 学习选定的值

要得知某个值已被选定，学习者必须发现某个提案已被多数接受者接受。一个显而易见的算法是：每当一个接受者接受一个提案时，它应向所有学习者发送响应，告知他们该提案。这使得学习者能够尽快得知选定的值，但要求每个接受者向每个学习者发送响应——响应的总数等于接受者数量与学习者数量的乘积。

非拜占庭故障的假设使得一个学习者可以从另一个学习者处得知某个值已被接受。我们可以让接受者将它们的接受响应发送给一个指定的学习者，该学习者再将值被选定的信息传递给其他学习者。这种方法需要所有学习者多一轮才能发现选定的值，而且可靠性较低，因为指定的学习者可能会失败。但响应的总数仅等于接受者数量与学习者数量之和。

更一般地，接受者可以将它们的接受响应发送给一组指定的学习者，每个指定的学习者在值被选定时即可通知所有学习者。使用更大规模的指定学习者集合可以进一步提高效率。学习者在保证可靠性方面提供了更大的可靠性，但代价是通信复杂性的增加。

由于消息丢失，一个值可能被选中，而没有任何学习者能够发现这一点。学习者可以询问接受者他们已接受的提议，但某个接受者的失败可能使得无法确定是否已有多数接受了一个特定的提议。在这种情况下，学习者只能在新的提议被选中时才知道所选择的值。如果学习者需要知道某个值是否已被选择，它可以要求一个提议者发出一个提议，使用上述算法。

2.4 进展

很容易构造一个场景，其中两个提议者各自持续发出一系列编号递增的提议，而这些提议均不会被选中。提议者 p 完成编号为 n_1 的提议的阶段 1。另一个提议者 q 然后完成编号为 $n_2 > n_1$ 的提议的阶段 1。提议者 p 针对编号为 n_1 的提议发出的阶段 2 接受请求将被忽略，因为接受者们都承诺不会接受任何编号小于 n_2 的新提议。因此，提议者 p 会开始并完成编号为 $n_3 > n_2$ 的新提议的阶段 1，从而导致提议者 q 的第二轮阶段 2 接受请求被忽略。依此类推。

为了保证进展，必须选出一个特殊的提议者作为唯一尝试发出提议的实体。如果这个特殊的提议者能够与多数接受者成功通信，并且使用一个编号大于之前所有已使用编号的提议，则它将成功发出一个被接受的提议。通过在得知某个编号更高的提议请求后放弃当前提议并重新尝试，这个特殊的提议者最终会选中一个

足够高的提议编号。

如果系统中足够多的部分（包括提议者、接受者和通信网络）正常工作，那么通过选举一个单一的特殊提议者，就可以实现活性。Fischer、Lynch 和 Patterson [1] 的著名结果表明，一个可靠的选举提议者算法必须使用随机性或真实时间——例如，通过使用超时。然而，无论选举是否成功，安全性都可以得到保证。

2.5 实现

Paxos 算法 [5] 假设存在一组进程组成的网络。在其共识算法中，每个进程同时扮演提议者、接受者和学习者的角色。该算法选择一个领导者，该领导者承担特殊提议者的角色。提案者和杰出的学习者。Paxos 一致性算法正是上述描述的算法，其中请求和响应作为普通消息发送。（响应消息会附带相应的提案编号，以防止混淆。）稳定的存储在故障期间得以保持，用于保存接受者必须记住的信息。接受者在实际发送响应之前，会将其预期响应记录在稳定的存储中。

剩下的就是描述如何保证永远不会有两个提案使用相同的编号。不同的提案者从不相交的编号集合中选择编号，因此两个不同的提案者永远不会发出编号相同的提案。每个提案者在稳定的存储中记住它已尝试发出的最高编号提案，并在第一阶段使用比之前所有编号更高的提案编号。

3 实现状态机

实现分布式系统的一种简单方法是将系统设计为一组客户端向中央服务器发送命令。服务器可以被描述为一个确定性状态机，按照某种顺序执行客户端命令。该状态机具有当前状态；执行一步时，它以命令为输入，产生输出和新的状态。例如，在分布式银行系统中，客户端可能是柜员，状态机的状态可能包括所有用户的账户余额。当执行取款操作时，会执行一个状态机命令，只有当余额大于取款金额时才减少账户余额，输出为旧余额和新余额。

使用单一中央服务器的实现方式在该服务器发生故障时会失效。因此，我们改用一组服务器，每个服务器独立实现状态机。由于状态机是确定性的，如果所有服务器执行相同的命令序列，它们将产生相同的若干状态和输出序列。客户端发出命令后，可以使用任意一个服务器为其生成的输出。

为确保所有服务器执行相同的状态机命令序列，我们实现一系列独立的 Paxos 一致性算法实例，第 i 个实例选择的值即为该序列中的第 i 个状态机命令。每个服务器在算法的每个实例中都扮演提案者、接受者和学习者三个角色。目前，我假设服务器集合是固定的，因此所有一致性算法的实例都使用相同的代理集合。

在正常运行时，会选举一个单一的服务器作为领导者，该领导者在共识算法的所有实例中充当唯一的提议者（唯一尝试发出提议的实体）。客户端将命令发送

给领导者，领导者决定每个命令在序列中的位置。如果领导者决定某个客户端命令应为第 135 个命令，它就会尝试让该命令被选为共识算法第 135 次实例的值。通常这种情况会成功。它可能会因为故障或因为另一个服务器也认为自己是领导者且对第 135 个命令的值有不同看法而失败。但共识算法确保最多只有一个命令会被选为第 135 个命令。

该方法效率的关键在于，在 Paxos 共识算法中，提议的值直到第 2 阶段才被确定。回想一下，在提议者算法完成第 1 阶段后，要么提议的值已经被确定，要么提议者可以自由地提议任何值。

现在我将描述 Paxos 状态机在正常运行时的工作方式。之后，我将讨论可能出现的问题。我考虑的是当上一个领导者刚刚失败并且新领导者被选中时的情况。（系统启动是一个特殊情况，此时尚未提出任何命令。）

新领导者在共识算法的所有实例中都是一个学习者，因此应该知道大多数已经选择的命令。假设它知道命令 1-134、138 和 139——即共识算法实例 1-134、138 和 139 中选择的值。（我们稍后会看到，命令序列中出现这种间隙是如何产生的。）然后它执行实例 135-137 以及所有大于 139 的实例的第 1 阶段。（我将在下面描述这是如何实现的。）假设这些执行的结果决定了实例 135 和 140 中提议的值，但其他所有实例的提议值则未被约束。接着，领导者执行实例 135 和 140 的第 2 阶段，从而选择命令 135 和 140。

领导者，以及任何其他学习到领导者所知所有命令的服务器，现在都可以执行命令 1-135。然而，它无法执行命令 138-140，因为它也知晓这些命令，因为命令 136 和 137 尚未被选择。领导者可以将接下来两个客户端请求的命令设为命令 136 和 137。相反，我们让其立即填补这个空缺，通过将特殊“空操作”命令作为命令 136 和 137 提出，以保持状态不变。（它通过执行共识算法实例 136 和 137 的第 2 阶段来实现这一点。）一旦这些命令被选择，无操作命令已被选择，命令 138-140 可以被执行。

命令 $1-140$ 已被选择。领导者还完成了所有大于 140 的共识算法实例的阶段 1，且在这些实例的阶段 2 中可以自由地提出任意值。它将下一个客户端请求的命令编号分配为 141，并在共识算法实例 141 的阶段 2 中将其作为值提出。它将接收到的下一个客户端命令提出为命令 142，依此类推。

领导者可以在得知其提出的命令 141 被选择之前就提出命令 142。在提出命令 141 时发送的所有消息都可能丢失，且命令 142 可能在任何其他服务器尚未得知领导者提出命令 141 的内容之前就被选中。当领导者在实例 141 的阶段 2 消息中未收到预期响应时，它将重新发送这些消息。如果一切顺利，其提出的命令将被选中。然而，它可能首先失败，从而在命令序列中产生一个空缺。一般地，假设一个领导者可以领先 α 个命令——即在命令 1 到 i 被选中之后，它可以提出命令 $i+1$ 到 $i+\alpha$。那么，最多可能出现 $\alpha-1$ 个命令的空缺。

一个新选出的领导者会为共识算法的无限多个实例执行阶段 1——在上述场

景中，为实例 135-137 以及所有大于 139 的实例。通过为所有实例使用相同的提议编号，它只需向其他服务器发送一条合理短的消息即可完成。在阶段 1 中，接受者只有在已接收到某个提议者发送的阶段 2 消息时，才会以超过简单“OK”的响应作出回应。（在该场景中，仅实例 135 和 140 满足此条件。）因此，一个服务器（作为接受者）可以为所有实例通过一条合理短的消息作出响应。因此，执行这些无限多个阶段 1 实例不会带来问题。

由于领导者失败和新领导者选举应是罕见事件，执行状态机命令的有效成本——即就命令/值达成共识的成本——仅为执行共识算法阶段 2 的成本。可以证明，在存在故障的情况下，Paxos 共识算法的阶段 2 具有达成一致的任何算法中最小的可能成本 [2]。因此，Paxos 算法本质上是最优的。

本系统正常运行的讨论假设始终存在一个单一领导者，除了在当前领导者失败和新领导者选举之间短暂的时期外。在异常情况下，领导者选举可能会失败。如果没有任何服务器担任领导者，则不会提出新的命令。如果多个服务器都认为自己是领导者，则他们可以在共识算法的同一实例中提出值，这可能会阻止任何值被选中。然而，安全性得以保持——两个不同的服务器永远不会就作为第 i^{th} 状态机命令所选择的值产生分歧。仅需选举一个领导者，即可确保进度。

如果服务器集合可以发生变化，那么必须有一种方法来确定哪些服务器实现了共识算法的哪些实例。最简单的方法是通过状态机本身来实现。当前的服务器集合可以作为状态的一部分，通过普通的状态机命令进行更改。我们可以允许领导者提前获取 α 条命令，方法是让执行共识算法第 $i+\alpha$ 个实例的服务器集合在执行第 i^{th} 个状态机命令之后由状态指定。这使得能够简单地实现任意复杂的重新配置算法。

参考文献

[1] Michael J. Fischer, Nancy Lynch, 和 Michael S. Paterson. 分布式共识中单个故障过程的不可能性。《ACM 期刊》，32(2):374-382，1985 年 4 月。

[2] Idit Keidar 和 Sergio Rajsbaum. 在无故障情况下容错共识的成本——教程。MIT-LCS-TR-821 技术报告，麻省理工学院计算机科学实验室，剑桥，马萨诸塞州，02139，2001 年 5 月。也发表于《SIGACT 新闻》32(2)（2001 年 6 月）。

[3] Leslie Lamport. 可靠分布式多进程系统的实现。《计算机网络》，2:95-114，1978 年。

[4] Leslie Lamport. 时间、时钟以及分布式系统中事件的排序。《ACM 通讯》，21(7):558-565，1978 年 7 月。

[5] Leslie Lamport. 部分时间议会。《ACM 计算机系统交易》，16(2):133-169，1998 年 5 月。