Paxos Made Simple

Leslie Lamport

01 Nov 2001

翻译/批注：hutu92

2018 年 9 月 25 日

摘要

Paxos 算法，当用简单的英语表达时是非常简单的。

一、Introduction

可能是因为之前的描述对大多数读者来说太过 Greek 了,Paxos 作为一种实现容错的分布式系统的算法被认为是难以理解的。但事实上，它可能是最简单，最显而易见的分布式算法了。它的本质其实就是一致性算法——也就是论文“The-Part-Time-Parliament”中提到的“synod”算法。在下一节中我们将展示，该一致性算法基本满足了所有我们想要它满足的特性。最后一节则展示了完整的 Paxos 算法，通过直接应用协商一致的状态虚拟机来构建分布式系统——这种方法应该是广为人知的,因为这可能是分布式系统理论中被引用最多的领域。

二、The Consensus Algorithm

2.1 The Problem

假设有一些进程可以提出 value。一致性算法保证了在所有提出的 value 里只有一个会被选中。如果没有 value 被提出，那么也就没有 value 会被选中。如果一个 value 被选中了，那么其他的进程应该能够获取该 value。协商一致的要求如下：

① **只能选择已经被提出的 value**
② **只能选择一个 value**
③ **进程只能获取那些真正被选中的 value**

我们不会尝试指定确切的要求。但是我们的目标是要确保总有一些被提出的 value 会被选中，如果一个 value 最终被选中了，那么其他进程最终要能够获取该 value。

我们用三类 agent 来代表一致性算法中的三类角色：proposers, acceptors 和 learners。在具体的实现中，一个进程可能扮演不止一类 agent，但是从 agent 到进程的映射我们在这里并不关心。

假设 agent 之间可以通过发送 message 互相通信。我们使用通常的**异步的非拜占庭模型**（*customary asychronous, non-Byzantine model*），其中：

① **Agents 以任意速度执行，可能发生故障，可能重启。因为所有的 agent 都可能在一个 value 被选中之后故障并重启，因此一般的方法是不可行的，除非 agent 能记住一些信息，即使发生了故障或重启。**
② **发送的 message 可以是任意长度的，可能重复，也可能丢失，但是它们不会被损坏，即消息内容不会被篡改（非拜占庭问题）。**

2.2 Choosing a Value

存在一个单一的 acceptor agent 是最简单的选择 value 的方式。proposer 向 acceptor 发送提议，后者会选择他所收到的第一个提议值。虽然很简单，但是这种方法是不能满足要求的，因为 acceptor 的故障就将导致接下来的操作都无法进行。

因此我们需要尝试另一种选择值的方式。这次我们将有多个而不是一个 acceptors。proposer 将会向一个 acceptor 的集合发送 value。acceptor 可能会接受（*accept*）value。但是该 value 只有在足够多的 acceptor 都接受它的情况下才算被选中了。

那么怎样才算足够大呢？为了确保只有一个 value 会被选中，我们可以认为一个足够大的 agent 集合由任意的大多数（*majority*）agent 组成。因为任意两个 majority 都至少

有一个公共的 agent，因此**如果一个 agent 最多只能接受一个 value，那么这种方法是可行的**。

在没有故障和 message 丢失的情况下，我们**想要有一个 value 能被选中，即使只有一个 proposer 提出了一个 value**。这就需要满足以下要求：

> **P1. acceptor 必须接受它收到的第一个 proposal**

但是这个要求会引起这样一个问题。不同的 proposer 可能会在几乎同时提出好几个不同的 value，这会**导致这样一种情况：每个 acceptor 都接受了一个 value，但是没有一个 value 是被一个 majority 接受的**。即使只提出了两个 value，而它们各自被一半的 acceptor 所接受，那么任意单个 acceptor 的故障都将让我们无法获取它选择了哪个 value。

P1 以及 value 只有被 majority 个 acceptor 接受才算被选中的要求就导致了我们的 **acceptor 必须能接受超过多于一个的 proposal**。我们**通过给每个 proposal 赋予一个编号来追踪不同的 proposal，所以一个 proposal 由一个 proposal number 和一个 value 组成**。为了防止出现歧义，我们要求不同的 proposal 要有不同的 number。这里我们仅仅只是做出这个假设，具体的实现可能有所不同。**当一个 proposal 被 majority 个 acceptor 所接受时，我们就认为该 value 被选中了。这种情况下，我们说这个 proposal(同时也包括它的 value)被选中了**。

我们可以**允许多个 proposal 被选中，但是我们必须保证被选中的 proposal 有相同的 value**。通过归纳 proposal number，足以保证：

> **P2.如果一个 value 为 v 的 proposal 被选中，那么所有被选中的更高编号（high-numbered）的 proposal 都具有 value v**

因为 number 都是有序的，**条件 P2 保证了只有单一的 value 被选中这一重要特性。**为了能够被选中，proposal 必须至少被一个 acceptor 所接受。所以我们可以**通过满足以下条件来满足 P2**：

> **$P2^a$.如果一个 value 为 v 的 proposal 被选中，那么每一个被任何 acceptor 所接受的更高编号（high-numbered） proposal 都有 value v**

我们依然需要满足 P1 从而确保有 proposal 被选择。**因为交互是异步的，一个 proposal 可能被一些特定的，没有接受过任何 proposal 的 acceptor c 选中。假设一个新的 proposer “苏醒”（wake up），并且发送了一个带有不同 value 的 high-numbered proposal。P1 要求 c 接受这个 proposal，但是却违背了 $P2^a$。为了同时满足 P1 和 $P2^a$，需要加强 $P2^a$**：

> **$P2^b$.如果一个 value 为 v 的 proposal 被选中，那么之后每个 proposer 提出的 high-numbered proposal 都有 value v**

因为一个 proposal 在被 acceptor 接受之前都要首先由 proposer 提出。因此满足 $P2^b$ 就满足了 $P2^a$，也就满足了 P2。

为了找到如何满足 $P2^b$,我们先对它进行证明。我们**假设有一些 number 为 m，value 为 v 的 proposal 已经被选中了**，接下来我们**证明任何的有 number n > m 的 proposal 的 value 都为 v**。

我们可以**通过归纳到 n 来简化证明**，如此，**在每一个提出的 number 在 m...(n-1)的 proposal 都有 value 为 v（其中 i...j 代表范围为 i 到 j 的所有 number）这一附加的假设下，我们能够证明 number 为 n 的 proposal 的 value 为 v**。

既然有 number 为 m 的 proposal 被选中了，那么必然有这样一个集合 C，它由 majority 个 acceptor 组成，而 C 中的每个 majority 都接受它。**结合归纳假设，从 m 被选中的假设可以推出：**

> **C 中的每一个 acceptor 都接受了 number 在 m...(n-1)中的一个 proposal，而每个被任意 acceptor 接受的 number 在 m...(n-1)的 proposal 都有 value 为 v。**

因为任何由 majority 个 acceptor 组成的集合 S 必然和 C 存在公共的元素，我们可以通过满足以下条件来确保标号为 n 的 proposal 有 value 为 v：

> **$P2^c$.对于任意的 v 和 n，如果有一个 value 为 v，number 为 n 的 proposal 被提出了，那么就存在一个由 majority 个 acceptor 组成的集合 S。要么(a)S 中没有一个 acceptor 接受过 number 小于 n 的 proposal，要么(b)v 是 S 中的 acceptor 接受过的 number 小于 n 的最大 number 的 proposal 的 value**

因此我们可以通过满足 $P2^c$ 来满足 $P2^b$。为了满足 $P2^c$，如果 proposer 想要发出一个 number 为 n 的 proposal，那么它必须要学习已经或者将要被一个 majority 接受的最高 number 不大于 n 的 proposal 的 value。学习那些已经被接受的 proposal 是非常简单的，但是对未来的接受情况进行预测就非常困难了。为了避免对未来进行预测，**proposer 通过获得不会有这样的接受情况的承诺来进行控制。换句话说，proposer 请求 acceptor 不要接受任何 number 小于 n 的 proposal**。这就导出了以下发送 proposal 的算法：

1. 一个 proposer 选择了一个新的 proposal number n 并且给一些 acceptor 集合的每个成员发送了一个请求，并希望它们的如下回复：

   (1) 承诺再也不接受 number 小于 n 的 proposal

   (2) 如果已经接受了 number 小于 n 的最大 number proposal，返回

   我们将这样的请求称为 number 为 n 的 prepare request

2. 如果 proposer 接受了来自一个 majority 对于请求的回复，那么它就可以发送一个 number 为 n，value 为 v 的 proposal。其中 v 要么是返回的 number 小于 n 的最大 number proposal 的 value，如果没有 proposal 返回，那么 proposer 随意选择一个值。

proposer 将 proposal 发送给一些 acceptor 的集合，请求它们接受。(这里的 acceptor 的集合不一定要和回复初始请求的 acceptor 的集合是同一个集合)我们将这样的请求称之为 accept request。

这里描述的是 proposer 的算法。那么 acceptor 呢？它可以从 proposer 那里接受到两种请求：prepare request 和 accept request。acceptor 可以安全的忽略任何的请求。因此我们只需要讨论它被允许回复请求的情况。它总能对 prepare request 进行回复。它可以对一个 accept request 进行回复，代表接受一个 proposal，如果它没有承诺不那么做的话。换句话说：

> **$P1^a$. acceptor 可以接受 number 为 n 的 proposal，如果它之前没有回复任何 number 大于 n 的 prepare request**

可以发现 $P1^a$ 包含 P1。

现在我们已经为满足安全性要求选择 value 提供了一个完整的算法——假设 proposal number 唯一的情况。最终的算法只是在这基础之上做了一个小小的优化。

假设一个 acceptor 接受了一个 number 为 n 的 prepare request，但是它已经回复了一个 number 大于 n 的 prepare request，因此它承诺不会接受任何 number 为 n 的新的 proposal。该 acceptor 是不会接受该 proposer 想要发出的 number 为 n 的 proposal 的，所以该 acceptor 没有理由回复这个 prepare request。**因此，我们让 acceptor 忽略这样的 prepare request**。同样，我们**对于已经接受的 proposal 对应的 prepare request 也是忽略的**。

经过这样的优化以后，acceptor 只需要记住它接受过的最大 number 的 proposal 以及它回复过的最大 number 的 prepare request 的 number。即使发生了故障也要满足 $P2^c$，因此即使发生了故障或者重启了，acceptor 也要记住这些信息。**需要注意的是 proposer 总是可以放弃一个 proposal 并且将它忘得一干二净**——只要它不再发送另一个具有相同 number 的 proposal。

将 proposer 和 acceptor 的行为结合在一起，我们可以看到算法分两阶段执行。

Phase 1.

(1) proposer 选择一个 proposal number n，然后向一个 majority 发送 number 为 n 的 prepare request。

(2) 如果一个 acceptor 接收到一个 number 为 n 的 prepare request，并且 n 大于任何它已经回复的 prepare request 的 number，那么它将承诺不再接受任何 number 小于 n 的 proposal，并且回复已经接受的最大 number 的 proposal（如果有的话）。

Phase 2.

(1) 如果 proposer 接受了来自 majority 对它的 prepare request 的回复，那么接下来它将给这些 acceptor 发送一个 number 为 n，value 为 v 的 proposal 作为 accept request。其中 v 是收到的回复中最大 number 的 proposal 的 value，或者如果回复中没有 proposal 的话，就可以是它自己选的任意值。

(2) 如果 acceptor 收到一个 number 为 n 的 accept request，如果它没有对 number 大于 n 的 prepare request 进行过回复，那么就接受该 accept request。

一个 proposer 可以生成多个 proposal，只要它能满足算法的要求。它可以在协议的任意时刻放弃 proposal。（正确性依然是得到满足的，即使请求或者回复在对应的 proposal 被放弃了很久之后才到达目的地）当其他的 proposer 已经开始发出更大 number 的 proposal 时，最好放弃当前的 proposal。因此。**如果 acceptor 因为它已经接受了更高 number 的 prepare request 而忽略了其他的 prepare 或者 accept request，那么它应该通知对应的 proposer 放弃该 proposal。这是对性能优化，并不会影响正确性。**

2.3 Learning a Chosen Value

为了获取一个已经被选中的 value，learner 必须要确定已经有一个 proposal 被 majority 接受了。最显而易见的算法就是让每个 acceptor 在接受了一个 proposal 之后向所有的 learner 发送这个 proposal。这能让 learner 尽快地找到被选中的 value，但这需要 acceptor 对每个 learner 进行回复——回复的数量为 acceptor 和 learner 数量的乘积。

没有拜占庭失败（non-Byzantine failures）的假设允许我们让一个 learner 可以从另一个 learner 处获取已经被接受的 value。我们可以让 **acceptor 把它们的接受情况都发送给一个特定的 learner(distinguished learner),这个 learner 再转而通知其他的 learner 有个 value 被接受了**。这种方法需要额外的一个步骤（round）来让所有的 learner 发现被选中的 value。同时这样也是非常不可靠的，因为这个特定的 learner 可能故障。但它需要的回复数仅仅等于 acceptor 和 leaner 数目之和。

更一般地，acceptor 可以将它们的接受情况发送给一个 distinguished learner 的集合，而它们中的任意一个都能在 value 被选中的时候通知所有的 learner。通过提供更大集合的 distinguished learner 在增加通信复杂度的同时提供了更高的可靠性。

因为 message 的丢失，可能没有 learner 知道已经有 value 被选中了。learner 可以直接问 acceptor 它们接受了什么 proposal，但是 acceptor 的故障可能让我们无法知道是否有 majority 接受了一个特定的 proposal。这种情况下，learner 只能在有新的 proposal 被接受的时候才能确定被选中的 value 是什么。如果一个 learner 想要知道一个 value 是否被选中，它可以让一个 proposer 发送一个 proposal，使用上文描述的算法。

2.4 Progress

我们很容易构建这样一个场景，两个 proposer 持续发送比对方的 number 大的 proposal，并且最终它们两者没有一个被选中。proposer p 通过 proposal number n1 完成了 phase 1。另一个 proposer q 接着通过了 proposal number n2 > n1 完成了 phase 1。proposer p 在 phase 2 以 n1 标记的 accept request 会被所有 acceptor 拒绝，因为它们已经承诺不接受任何 number 小于 n2 的 proposal。因此 proposer p 开始用新的 proposal number n3 > n2 来开始并完成 phase 1，而这又导致了 proposer q 在 phase 2 的 accept 被忽略。如此反复进行。

为了保证流程的执行，我们必须选出一个 distinguished proposer，作为唯一的 proposal 发送者。如果 distinguished proposer 能和 majority 进行通信，那么但凡 distinguished proposer 提出一个编号更高的 proposal，该 proposal 终将会被批准。当然，如果 proposer 发现当前算法流程中已经有一个编号更大的 proposal 被提出或正在接受批准，那么它会丢弃当前这个编号较小的 proposal，并最终能够选出一个编号足够大的 proposal。

如果系统足够多的组件都工作正常（proposer, acceptors 以及交互网络），那么通过选出一个单一的 distinguished proposer 就能保持系统的活力。由 Fischer, Lynch, and Patterson 著名的结论可得，选举一个 proposer 的可靠算法必须随机性或实时性（randomness or real time）——比如，使用超时。当然，无论选举成功还是失败，安全性总是可以保证的。

2.5 The Implementation

Paxos 算法假设了一个进程网络。在这个一致性算法中，每个进程扮演着 proposer, acceptor 和 learner 的角色。该算法需要选择一个 leader，来扮演 distinguished proposer 和 distinguished learner 的角色。Paxos 一致性算法正如上文所描述的那样，请求和回复都以普通消息（ordinary message）的形式发送。（Response message 会用相应的 proposal number 标记为了防止混淆）我们需要**使用 stable storage（会在故障时候保存）来维护那些 acceptor 必须保存的信息。acceptor 会在真正发送 response 之前将它记录下来。**

接下来所有的内容都将用于描述如何保证两个 proposal 不会有相同的 number。不同的 proposer 会从不相交的数据集中选择 number，所以不同的 proposer 不会发送具有相同 number 的 proposal。每个 proposer 都会用 stable storage 记住它尝试发送的最大 number 的 proposal 并用一个比所有已经使用过的 number 都大的 number 开始 phase 1。

三、Implementing a State Machine

实现一个分布式系统最简单的方式就是一个 client 的集合向一个 central server 发送命令。central server 可以被看做是一个以一定顺序执行 client 命令的确定性状态机（deterministic state machine）；这个状态机有一个当前状态；它通过将输入作为命令，并产生输出和一个新的状态。比如，分布式银行系统的 client 可以看做是出纳员，而所有用户的帐户余额可以看做状态机的状态。一个取款操作可以通过执行当且仅当这个账户的余额大于提取金额的时候减小账户的余额这一状态机命令完成。

使用单一的 central server 的实现，如果 central server 发生故障，整个系统就会发生故障。因此，我们使用了一个 server 的集合，它们各自独立地实现一个状态机。因为状态机是确定性的，所以在执行完相同的命令序列之后，所有的 server 都会产生相同的状态序列和输出序列。然后，发出命令的 client 可以使用任何 server 为其生成的输出。

为了保证所有的 server 执行相同的状态机命令序列，我们实现了 Paxos 一致性算法的一系列单独实例[1]（ a sequence of separate instances），被第 i 个实例选择的值作为序列中第 i 个状态机命令。每个 server 在算法的每个实例中扮演所有角色（ proposer, acceptor, and learner）。现在，我假设 server 集合是固定的，因此该一致性算法的所有实例都使用相同的 agents 的集合。

在通常的操作中，在一致性算法的所有实例中，只有一个 server 能够被选为 leader 并作为 distinguished proposer（唯一的 proposer 发送者）。client 将命令发送给 leader，leader 决定每个命令应该放在序列的哪个地方。如果 leader 决定一个特定的 client 命令应该作为第 135 个命令，则他尝试去使这个命令被选择为这个一致性算法中的第 135 个实例值。这通常都会成功。但也有可能因为发生故障或者有另一个 server 认为自己是 leader 并且对第 135 条命令是什么有它自己的想法而失败。但是一致性算法确保了第 135 条命令最多只有一个。

Paxos 一致性算法效率的关键在于直到 phase 2 之前都不对提出的 value 进行选择。回忆一下，是在完成了 phase 1 之后才知道要发送的 value，要么可以确定要提议的值，要么 proposer 可以被任意选择。

我现在要讨论的是在正常执行时，Paxos 状态机是怎么工作的。之后，还会描述什么情况下会出错。我考虑当之前的 leader 刚刚失败，并且一个新的 leader 已经被选择时会发生什么（系统刚刚启动时是一个特殊的情况，那时候还没有任何命令被提出）。

新的 leader，也是一致性算法所有实例的 learner，应该知道已经选中的大多数命令。假设它知道命令 1-134,138 和 139——即实例 1-134，138 和 139 选择的值（我们之后将看到在命令序列里的这样的 gap 是如何产生的）。之后，它将执行实例 135-137 以及所有大于 139 的实例的 phase 1（下面我将描述这是如何完成的）。假设这些操作的执行结果确定了实例

[1] 译者注：参见 Lamport 的另一篇论文《The-Part-Time-Parliament》The Multi-Decree Parliament 一节，一个实例即为一个神会协议实例（法令编号）。

135 和 140 的 value，但是其他实例的 value 还是未确定的。之后，leader 将会执行实例 135 和 140 的 phase 2，从而选择了命令 135 和 140。

leader 以及那些获取了 leader 已知的所有 command 的 server 现在可以执行命令 1-135。然而，它仍然不能执行命令 138-140，即使它已经知道它的内容了，因为命令 136 和 137 并没有被选择。leader 可以将接下来 client 请求的两个命令作为命令 136 和 137。然而，我们通过发送特殊的让状态不发生改变的"noop"命令来马上填充 gap（通过执行一致性算法的实例 136 和 137 的 phase 2 来实现）。一旦这些 noop 命令被选中，命令 138-140 就可以执行了。

命令 1-140 已经被选中了。leader 也完成了一致性算法所有大于 140 的实例的 phase 1，并且可以在这些实例的 phase 2 中自由提出任何 value。它为 client 发送的下一个请求分配命令编号 141，并且将它作为一致性算法实例 141 的 value。之后再将用户的下一个请求作为命令 142，如此往复。

leader 可以在它知道已经发送的命令 141 被选择之前就发送命令 142。可能发送命令 141 的所有数据都会丢失，并且也可能命令 142 在所有 server 知道 leader 发送的命令 141 的任何内容之前被选择。当 leader 没有收到它希望得到的关于实例 141 的 phase 2 信息的回复时，它会对这些信息进行重发。如果所有运行正常的话，发送的命令将会被选中。然而，在一开始可能会发生故障，从而在已选中的命令序列中留下一个 gap。一般来说，假设一个 leader 可以提前获取 α 个命令——这意味着在命令 1 到 i 被选中的前提下，它可以发送命令 $i+1$ 到 $i+\alpha$ 之间的命令。因此，一个至多为 $\alpha-1$ 条命令大的 gap 可能会出现。

一个新的被选中的 leader 可以执行一致性算法的无数多个实例的 phase 1——在上面的场景中，即为实例 135-137 以及所有大于 139 的实例。**对所有实例使用同一个 proposal number，它可以用给其他 server 发送一条相当短的消息来实现。在 phase 1，如果一个 acceptor 已经从一些 proposer 收到 phase 2 信息的时候，它就会不仅仅回复一个简单的 OK。（在例子中，就是对于实例 135 和 140）因此，server（作为 acceptor）可以用一条相当短的消息来回复所有的实例。**在无数个实例的 phase 1 这样执行不会产生任何问题。

因为 leader 的故障和新的 leader 的选举都是小概率事件，因此执行状态机命令的有效开销——即 command/value 达成一致——主要是一致性算法 phase 2 的执行开销。可以看出 Paxos 一致性算法的 phase 2 在所有会出现故障的情况能达到一致的所有算法里有着最小的可能花费。因此，Paxos 算法基本上是最优的。

关于系统的正常运行的讨论假设除了当前 leader 发生故障，新的 leader 还未选出的短暂时间外，总是存在一个单一的 leader。在一些意外的情况下，leader 的选举可能发送故障。如果没有 server 作为 leader 执行，那么就不能有新的命令被发送。**如果有多个 server 认为它们是 leader，那么它们可以对一致性算法的同一个实例发送 value，而这会防止任何 value 被选择[2]。**但是，安全性得以保留——两个不同的 server 永远不会对已经被选为第 i 个状态机命令的 value 存在分歧。单一 leader 的选举仅仅只是为了保证流程的执行。

如果 server 的集合是可以改变的，我们必须要有办法确定哪些 server 实现了一致性算法

[2] 译者注：这句话应该理解为神会协议的多轮表决，两个 proposer 持续发送比对方的 number 大的 proposal，并且最终它们两者没有一个被选中。

的哪些实例。实现这个最简单的方法就是通过状态机它自己。当前的 server 的集合可以作为状态的一部分并且可以随着普通的状态机命令而改变。在执行完第 i 个状态机命令后，我们可以允许 leader 提前获取 α 个命令，通过用状态指定一组执行一致性算法的第 i + α 个实例的 servers。这允许简单的实现任意复杂的重新配置算法。

四、References

[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] Leslie Lamport. The part-time parliament. ACM Transactions on Computer Systems, 16(2):133–169, May 1998.

五、End

关注公众号

欢迎大家公众号留言反馈。

邮箱：631521383@qq.com