Paxos Made Simple

1 Introduction

Paxos算法是莱斯利·兰伯特（Leslie Lamport）于1990年提出的一种基于消息传递且具有高度容错特性的共识（consensus）算法。《The Part-Time Parliament》最早发表于1998年，Paxos岛上有一个议会，这个议会来决定岛上的法律，而法律是由议会通过的一系列的法令定义的。当议会召开时，允许议员缺席，但法令仍然可以达成共识。《Paxos Made Simple》发表于2001年，用更清晰整洁的语言描述了Paxos算法。

本文主要基于《Paxos Made Simple》

2 The Consensus Algorithm

2.1 The Problem

共识问题

> Suppose you have a collection of computers and want them all to agree on something. This is what consensus is about; consensus means agreement.

共识在分布式系统设计中经常出现。我们需要它的原因有很多:同意谁可以访问资源(互斥)，同意谁负责(选举)，或者同意一组计算机之间的事件的共同顺序(例如，下一步采取什么行动，或状态机复制)。

共识问题可以用一种基本的、通用的方式来表述:一个或多个系统可能提出一些值。我们如何让一组计算机恰好同意这些建议值中的一个呢?

共识算法就是为了解决共识问题，那么假设有一组可以提出值的流程。共识算法确保在建议值中选择单个值。如果没有建议值，则不应选择任何值。如果选择了一个值，那么进程应该能够学习所选择的值。协商一致的安全要求如下

> Only a value that has been proposed may be chosen

> Only a single value is chosen, and

> A process never learns that a value has been chosen unless it actually has been

我们的目标是 确保最终选择一些建议的值，如果一个值 被选中，那么进程最终可以学习这个值。

三个角色proposers, acceptors, and learners，实际情况一个server或者说一个process可以扮演多个角色。

proposers：

主动:提出要选择的特定值（提案）

处理客户端的请求

acceptors：

被动:回应proposer的信息

回应代表了形成共识的投票

存储选择的值，决策过程的状态

想知道选择了哪个值

learners：

学习已经选择的值

我们模拟的环境是通过消息发送来实现进程之间的相互通信。且是异步的、**非拜占庭**的。

> Agents operate at arbitrary speed, may fail by stopping, and may restart. Since all agents may fail after a value is chosen and then restart, a solution is impossible unless some information can be remembered by an agent that has failed and restarted.

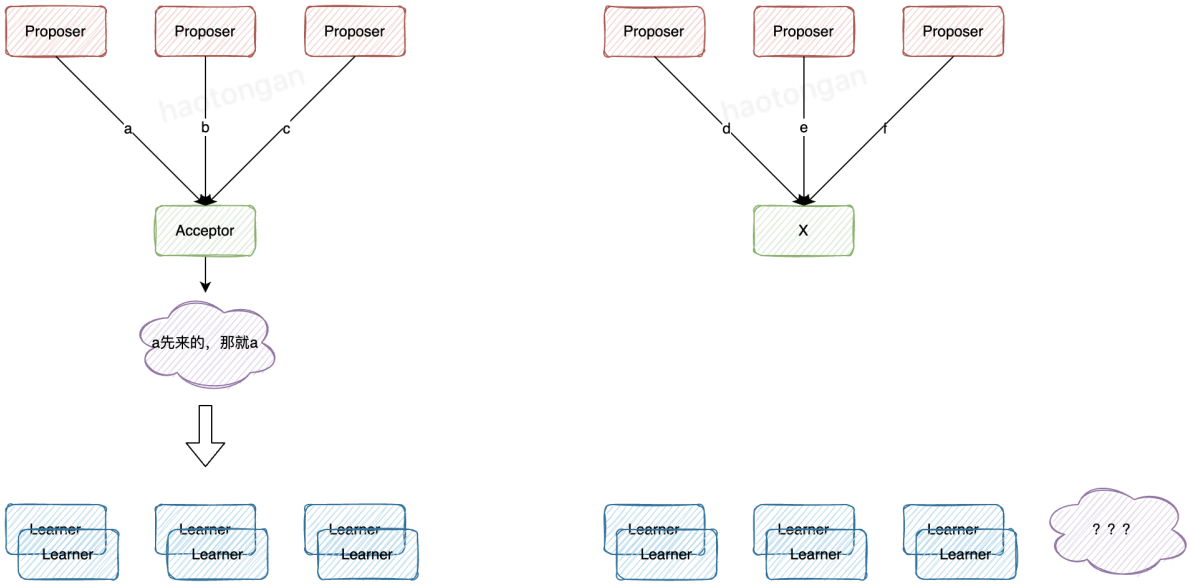
> Messages can take arbitrarily long to be delivered, can be duplicated,and can be lost, but they are not corrupted

这里再简单解释下**拜占庭将军**问题。

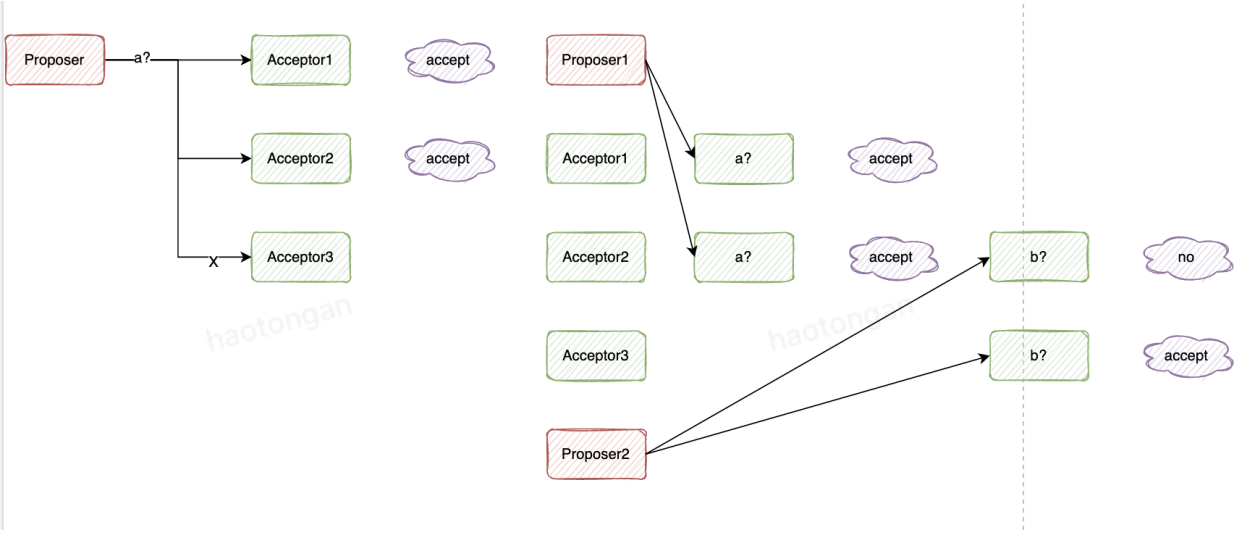
> 在分布式计算中，不同的计算机通过通讯交换信息达成共识而按照同一套协作策略行动。但有时候，系统中的成员计算机可能出错而发送错误的信息，用于传递信息的通讯网络也可能导致信息损坏，使得网络中不同的成员关于全体协作的策略得出不同结论[2]，从而破坏系统一致性[3]。拜占庭将军问题被认为是容错性问题中最难的问题类型之一。

2.2 Choosing a Value

一个很容易想到的方案，如果我们的系统中，只存在一个acceptor，acceptor选择accept收到的第一个提议，那么共识就达成了，但是这个方案很失效，因为acceptor一旦出现任何的故障都会导致后续不会有任何进展，从而无法达成共识。



为了避免上述的情况发生，我们采用多个acceptor，当一个提案产生时，proposer将提议发送给一组acceptor，acceptor可以accept该提议，如果大多数的acceptor接受该提议，那么这个提议是chosen 。 大多数如何定义?可以是2/3或者3/4，最起码要超过半数，这样两个大多数的accptor组中必定是有重复的accptor，如果accptor只能接受至多一个提案，这是一个有效的方案!

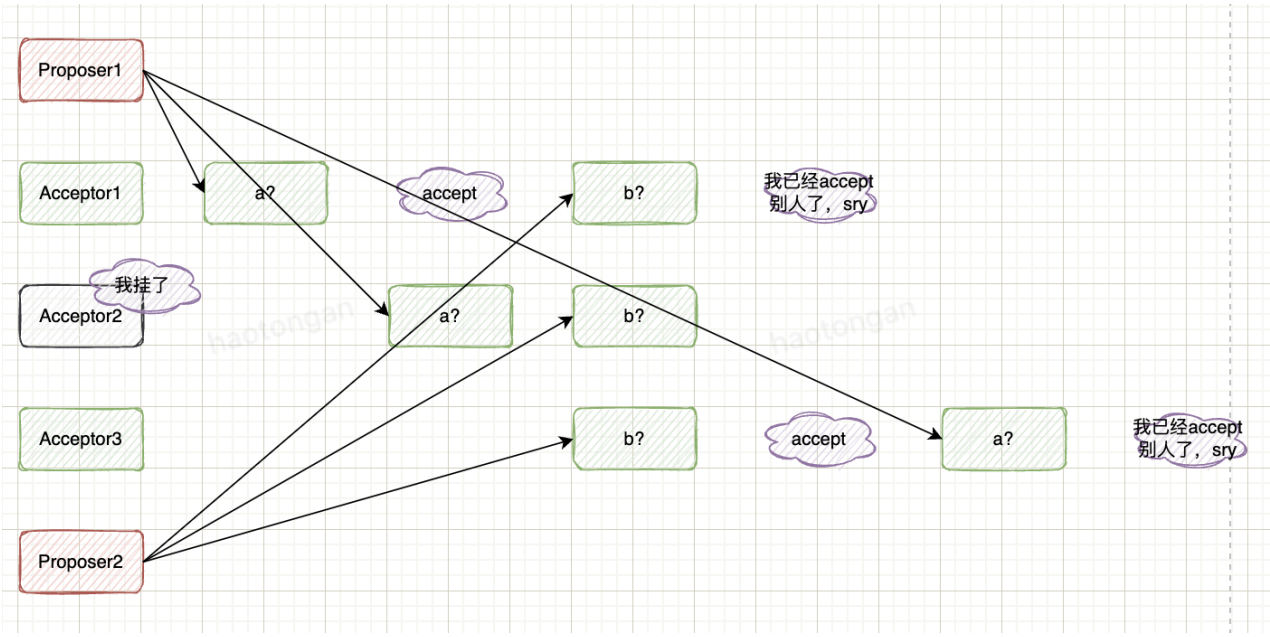


在没有失败或消息丢失的情况下，我们希望选择一个值 ，即使单个proposer只提出了一个值。这就提出了要求（约束）：

> P1. An acceptor must accept the first proposal that it receives.

很明显，当单个的proposer只提出了一个提案，那么acceptor必须要接受它收到的第一个提案，否则很有可能是无法达成共识（该提案没有被大多数acceptor接受）。

但这又会引发一个问题，如果在同一时间范围，多个proposer提出了不同的提案，每个acceptor都接受了一个提案，但没有一个提案被大多数的acceptor所接受。即使只有两个提案，每个提案都被近似一半的acceptor所接受，但只要有一个acceptor出现了故障（网络问题或宕机等），都会使得无法达成共识。



P1和只有当一个提案被大多数acceptor接受时才被认为是chosen的，这意味着我们必须允许一个acceptor接受不止一个提案。

当有多个提案被提出后，我们需要为每个提案分配一个编号（标识）来区分acceptor可能接受的不同提案，因此提案由提案编号和内容（value）组成。为了防止混淆，我们要求不同的提案有不同的编号。同时为了区分提案的先后顺序，这个编号需要做到全局递增的。

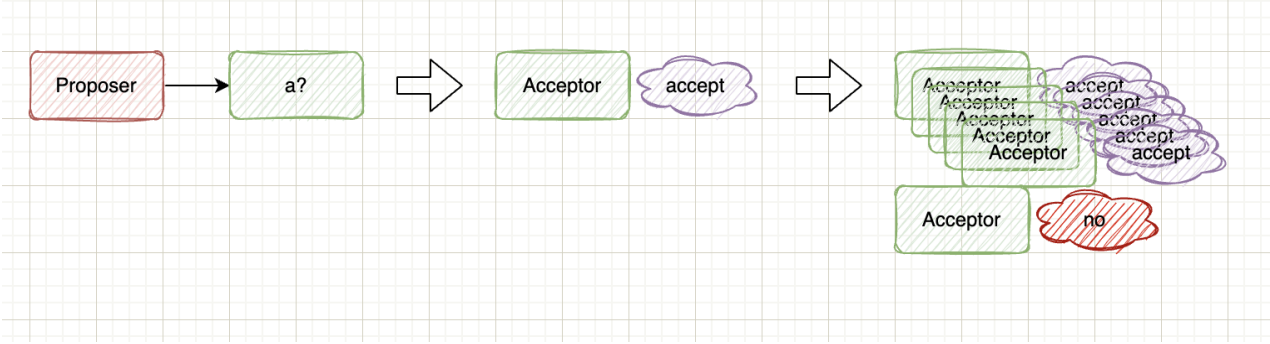
现在我们先假设一下。当具某个提案被大多数acceptor接受时，在这种情况下，这个提案是chosen的，或者说这个value是chosen的。

我们允许多个提案是chosen的，但我们必须保证所有选择的提案具有相同的value。通过对提案号的归纳，这需要:

> P2. If a proposal with value *v* is chosen, then every higher-numbered proposal that is chosen has value *v*.

因为数字是完全有序的，约束P2保证了关键的**safety**属性，即只有一个value是chosen的。

一个提案从提出到被选择，经历了proposer提出提案，acceptor接受提案，到大多数acceptor接受提案后，我们才认为这个提案是chosen的。



当认为一个提案是chosen的之前，一定存在acceptor接受了改提案，基于此，我们可以将约束P2再**强化**一下。

> P2a . If a proposal with value *v* is chosen, then every higher-numbered proposal accepted by any acceptor has value *v*.

考虑一种情况，我们的系统中可能会存在多个proposer来提出提案，也可能会存在acceptor *c*从未接受过提案，如果一个proposer提出了一个和现在大多数acceptor所接受的提案不同的提案，并发送给*c*，由于约束P1的存在，*c*需要接受改提案，但这就违反了约束P2a，同时维护P1和P2a我们将约束P2a强化一下：

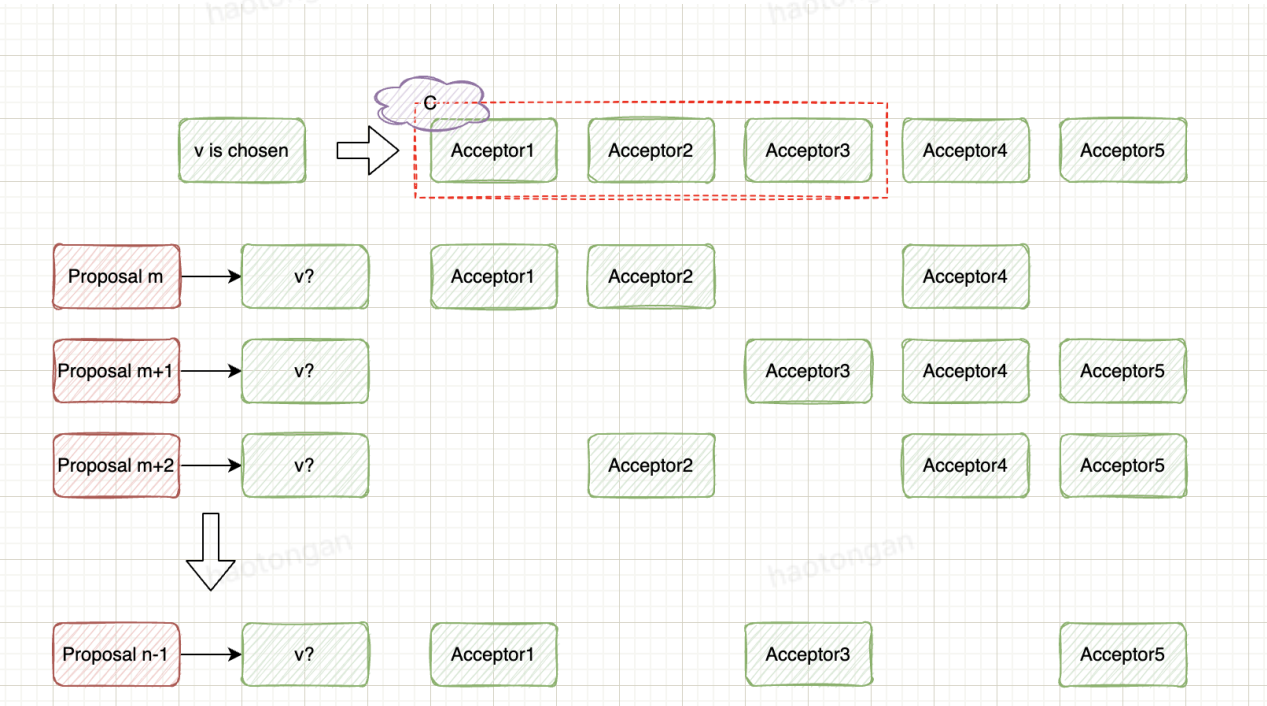
> P2b . If a proposal with value v is chosen, then every higher-numbered proposal issued by any proposer has value v.

与其限制acceptor，不如从源头上来避免这种情况。如果当一个提案value=v是chosen的，那么每一个更高编号的提案都携带value=v，这样acceptor再接受的提案也都是v。不难发现满足P2b的话，一定满足P2a，也一定满足P2。

为了发现如何满足P2b，这里首先让我们思考如何证明P2b。携带value是v的提案m已经是chosen的，那么提案n(n>m)携带的value也是v。

我们假设编号m..(n-1)的提案的value也都是v，而由P2b的if条件已知编号为m的提案其value=v已经是chosen的，那么一定会存在一个大多数acceptor组成的集合C，其中C中的每一个acceptor都接受了该提案。

到目前为止，实际上我们并没有限制acceptor，也就是说acceptor并不会拒绝提案，而当提案m，提案m+1，一直到提案n-1，发送给大多数acceptor时，acceptor会选择接受，那么实际上C中每一个的acceptor都接受过m～n-1中的提案，且任何被接受的提案的value都是v。



> P2c. For any v and n, if a proposal with value v and number n is issued, then there is a set S consisting of a majority of acceptors such that either (a) no acceptor in S has accepted any proposal numbered less than n, or (b) v is the value of the highest-numbered proposal among all proposals numbered less than n accepted by the acceptors in S.

P2c的提出，实际上要求proposer在发布提案n时，需要学习小于n的最高的编号的提案信息，这个提案很有可能已经被大多数acceptor所接受。

> Learning about proposals already accepted is easy enough; predicting future acceptances is hard. Instead

也就是说，当一个提案发布之后，很难去预测这个提案是否会被大多数的acceptor所接受，所以proposer不去预测未来，而是通过获取acceptor的承诺来控制提案的接受与否。因为如果当前系统内不存在被chosen的提案，也就是说不存在大多数acceptor接受某个提案的集合。那么此时有多个proposer提出自己的提案，那么很多提案都是无效的，或者也会出现无法达成共识的情况出现。

那么此时我们也需要对acceptor提出要求。当proposer发送提案给acceptor时，acceptor需要给予一定的承诺，同时将已经接受的提案信息回应发送给proposer以便proposer学习。

当proposer选择一个新的提案号n，并向大多数acceptor发送请求，要求acceptor响应:

> (a) A promise never again to accept a proposal numbered less than n, and
>
> (b) The proposal with the highest number less than n that it has accepted, if any.

这就是一个prepare阶段的请求。如果proposer收到了大多数acceptor的回复，那么proposer可以发布一个编号为n的提案，该提案携带value为v，其中v是acceptor回复中提案最大的value内容，如果acceptor没有回复提案信息，那么v可以是一个任意的内容。

proposer通过向一组acceptor发送请求来发布提案，请求该提案被接受。(这不需要是响应prepare请求的同一组acceptor)我们称其为accept请求。

上面的prepare请求和prepare请求主要是针对proposer进行描述的。那么当请求发送到acceptor时，acceptor也需要做出相应的响应。acceptor是允许不去响应任何请求的，那么根据上面的描述，我们可以得到P1的强化版本，即：

> P1a . An acceptor can accept a proposal numbered n iff it has not responded to a prepare request having a number greater than n.

假设一个acceptor收到一个编号为n的prepare请求，但它已经响应了一个编号大于n的prepare请求，因此 承诺不接受任何编号为n的新提案。那么acceptor就没有理由去响应新的prepare请求，因为它将不会去接受编号为n的提案，或者说任何小于它响应的prepare请求的编号的提案，它都不会去接受，这是acceptor给出的承诺。那么我们可以让acceptor忽略掉这样的prepare请求，或者说当它已经接受了一个编号大于n的提案，任何小于该编号的prepare请求，也都是可以忽略的。

基于此，我们要求acceptor只需要记住它曾经接受过的编号最高的提案和它响应过的编号最高的prepare请求的编号即可。

对于proposer和acceptor我们都进行了描述，下面我们整理一下，看一下算法如何在两个阶段运行的。

prepare阶段：

> Phase 1. (a) A proposer selects a proposal number n and sends a prepare request with number n to a majority of acceptors.
>
> (b) If an acceptor receives a prepare request with number n greater than that of any prepare request to which it has already responded, then it responds to the request with a promise not to accept any more proposals numbered less than n and with the highest-numbered proposal (if any) that it has accepted.

accept阶段：

> Phase 2. (a) If the proposer receives a response to its prepare requests (numbered n) from a majority of acceptors, then it sends an accept request to each of those acceptors for a proposal numbered n with a value v, where v is the value of the highest-numbered proposal among the responses, or is any value if the responses reported no proposals.
>
> (b) If an acceptor receives an accept request for a proposal numbered n, it accepts the proposal unless it has already responded to a prepare request having a number greater than n.

proposer可以发布多个提案，同时也可以在任何时刻放弃某个提案。如果某些proposer已经尝试发布更高编号的提案，那么放弃提案显然是一个好的主意。因此，如果一个acceptor因为已经收到一个编号更高的prepare请求而忽略了编号较低的prepare或accept请求，那么它可能应该通知proposer，然后proposer应该放弃它的提议。这是一个性能优化，并不影响算法的正确性。

2.3 Learning a Chosen Value

当一个提案已经被选择时，learner必须发现一个提案已经被大多数acceptor所接受。

1）一个简单方法，对于每一个acceptor，无论何时只要接受了一个提案，就将该提案发送给全部的learner。这种做法可以使learner尽快的学习到已被chosen的提案，但是这要求每一个acceptor与每一个learner都建立通信，通信的次数至少是acceptor的数量*learner的数量。

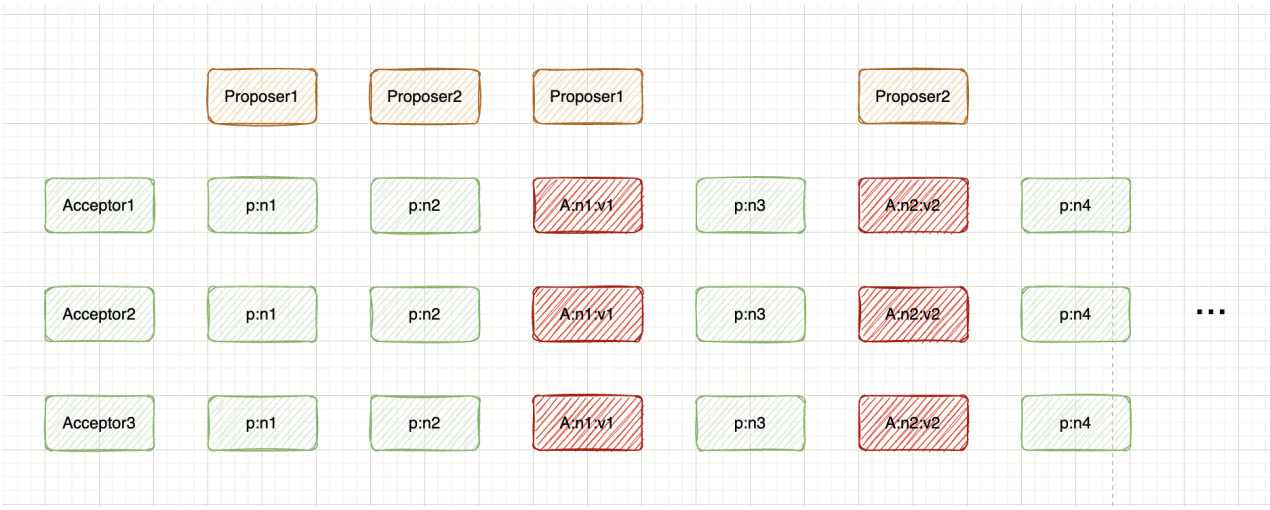
2）考虑到我们的环境是非拜占庭式的，那么当一个learner学习到了已经被chosen的提案，其他的learner是很容易发现这个信息的（消息是不会被篡改的），我们可以让acceptor用他们的接受提案发送给一个distinguished learner（leader）， 当一个提案被chosen时，由distinguished learner通知其他的learner。

这种方法需要额外的一轮才能让所有的learner学习到提案。同时由于distinguished learner可能发生故障，这个方法可能会失效。但是相比上面的方法，acceptor和learner之间通信的次数大大减少。（acceptor的数量+learner的数量）。

3）方法一和二结合一下，考虑维护一个distinguished learner集合，也就是说当acceptor接受了某个提案后，发送给集合中的所有distinguished learner，再由他们发送给其他的learner。这在一定程度上提升了可靠性，但这样同时也增加了通信的复杂度。

2.4 Progress

上面详细的描述了Paxos中三种角色的作用。通常情况下，一个提案在运行完Paxos实例后是会被选择的。但也容易构建这样一个场景:两个proposer各自不断发出一系列编号不断增加的提案，但没有一个提案被选中。



proposer p发出了提案编号n1，完成了prepare阶段的工作，然后另一个proposer q发出了提案编号n2(n2>n1)，同样完成了prepare阶段的工作。那么proposer p在accept阶段发出的提案n1将会被acceptors忽略，因为acceptors已经承诺不会accept任何编号小于n2的提案了。然后proposer p开始发出提案编号n3，完成prepare阶段的工作，那么那么proposer q在accept阶段发出的提案n2将会被acceptors忽略，这样反复下去没有提案会被大多数acceptor所接受。

为了保证流程正常运作，可以选出一个distinguished proposer作为唯一尝试发出提案的proposer，如果distinguished proposer能够与大多数acceptor成功地通信，并且如果它使用的提案的编号大于任何已经接受的提案，那么它将成功地发出一个被接受的提案。

2.5 The Implementation

Paxos算法假设一个进程网络。在其共识算法中，每个进程扮演proposer、acceptor和learner。算法选择了一个leader扮演distinguished proposer和distinguished learner。上面描述的Paxos算法，其中请求和响应作为普通消息发送(响应消息应该带有相应的提案编号，以防止混淆)。acceptor需要支持持久化存储对应的prepare和accept阶段所响应的提案信息。

剩下要做的就是描述一种机制，以保证不会有两个提案的编号相同。不同的提议者从不相交的数字集合中选择他们的数字，因此两个不同的提议者永远不会发出相同数字的提案。每个提议者都记住 (稳定存储中)它尝试发出的最高编号的提案 ，并以比它已经使用过的更高的提案号开始prepare阶段。

3 Implementing a State Machine

状态机同步

一个确定的状态机，以某种顺序执行命令。状态机具有当前状态，它通过当前状态和收到的命令来决定下一个状态。例如，银行系统可以描述为一个状态机，而状态机状态由所有用户的帐户余额组成。金额提现将通过执行状态机命令来执行，当且仅当余额大于提现金额时，该命令会减少帐户余额，并产生新旧余额作为输出。

如果只有一台服务器提供服务，那么当这台机器故障时，我们无法对外提供有效的服务。因此，我们需要使用一组服务器，每个服务器都独立的实现状态机。因为状态机是确定性的，所有的服务器都执行相同的命令序列，它们将产生相同的状态序列和输出。

如果服务器失败，那么使用单个中心服务器的实现就会失败。因此，我们转而使用一组服务器，每个服务器都独立地 实现状态机。因为状态机是确定性的， 如果所有服务器都执行相同的命令序列，那么它们将产生相同的状态序列和输出。那么，用户或者说客户端可以向任何服务器发出指令来获取输出结果。

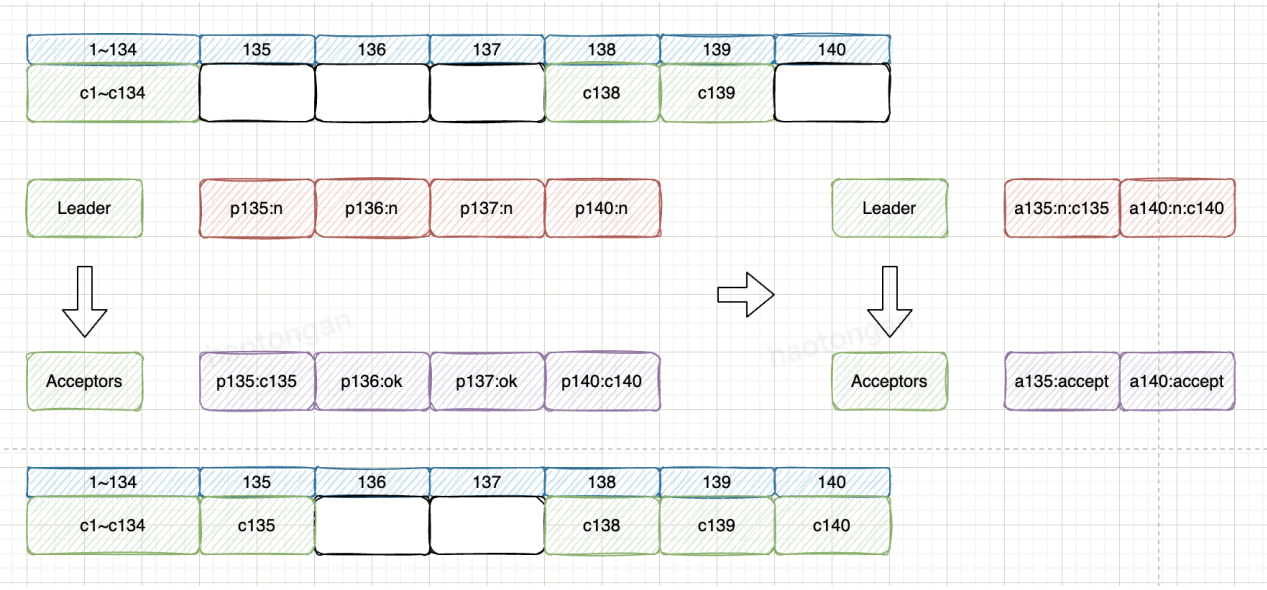
那么为了保证所有服务器都执行相同的状态机命令序列，我们在服务器上实现并且运行Paxos共识算法实例，第i个实例选择的value是状态机命令序列中的第i个命令（这里我们称每一次paxos运行为一次实例）。每个服务器在算法的每个实例中扮演所有角色(proposer，acceptor，and learner)。现在，我们假设服务器集是固定的，因此共识算法的所有实例都使用相同的代理集。

在通常的操作中，一个服务器被选为leader，它在所有算法实例中充当distinguished proposer，也就是唯一尝试发出提议的proposer。客户端将命令发送给leader，然后它来决定每个命令应该出现的顺序。如果leader决定某个客户端命令应该是第135个命令，那么它会尝试将该命令选为共识算法的第135个实例的value。通常情况下它会成功，当然也可能因为某些问题而失败 ，亦或者因为另一个服务器也认为自己是leader，并且对第135条命令是什么有不同的想法。但是共识算法保证最多可以选择一个命令作为第135个命令。

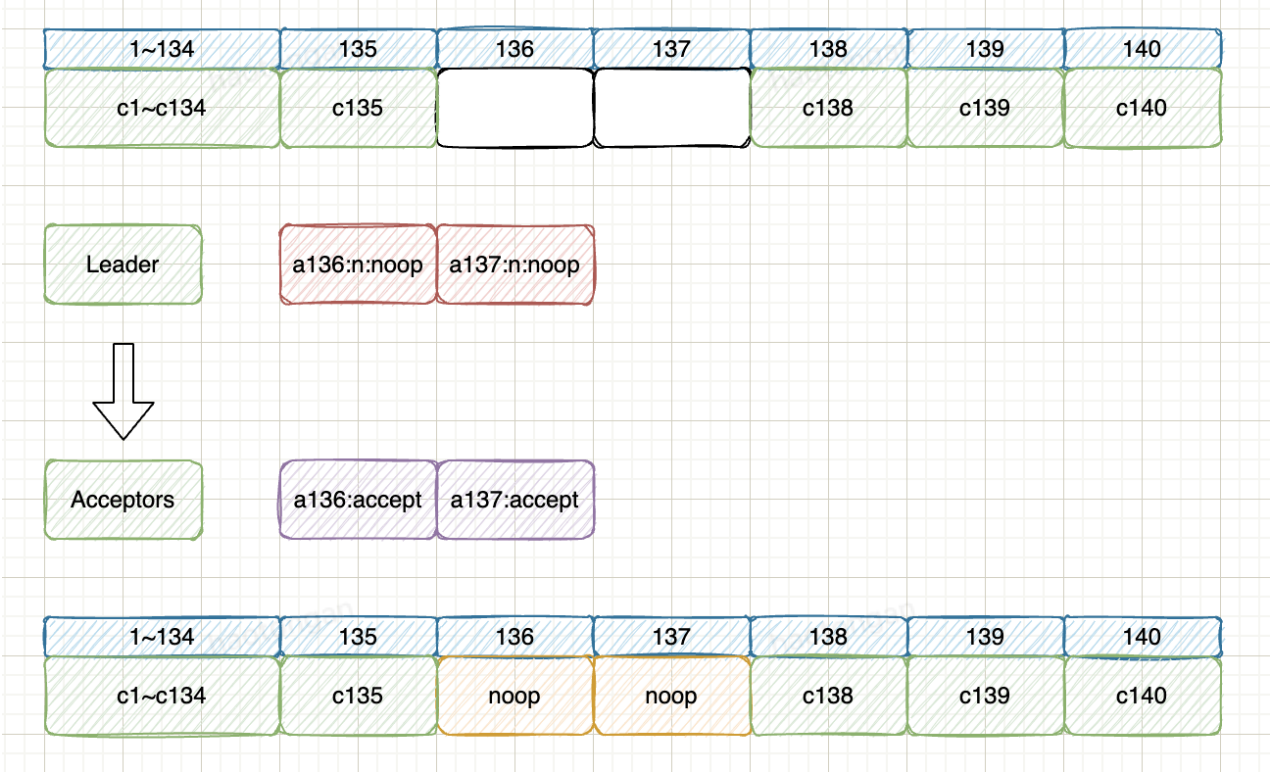
这种方法效率的关键在于，在Paxos共识算法中，要提出的value直到阶段2，也就是accept阶段才被选择。回想一下，当proposer在完成prepare阶段之后，要么确定要提议的value ，要么proposer可以自由地提出任何value。

现在先描述一下在正常操作期间Paxos状态机是如何工作的。然后我们讨论可能出现的问题。比如当前leader刚刚出现了故障，新leader已经被选中时会发生什么。

新的leader，同样作为共识算法中所有实例的learner， 应该知道大多数已经被选择的命令。假设它知道命令1-134、138和139，即在一致性算法的实例1-134、138和139中选择的value。然后，它执行实例135-137和所有大于139的实例的prepare阶段。假设这些执行的结果决定了实例135和140中提议的value（prepare阶段中acceptor响应返回了value），而其他实例中值没有受到约束（对应实例的prepare阶段，acceptor没有响应value）。然后，leader对实例135和140执行accept阶段的算法，从而第135和第140个命令选择了。



leader，以及任何其他学习了所有的命令leader知道，现在可以执行命令1-135。然而它没有办法执行第138-第140号命令，它也知道不能执行，因为第136和第137号命令还没有选择，leader可以将客户端的下两个请求命令作为第136和第137号命令。这里也可以用特殊的命令“no-op”，也就是无操作，来填补目前缺失的第136和第137号命令，维持当前状态机的状态（通过运行实例136，137accept阶段算法来完成，value=“noop”），一旦noop命令被选择了，第138-第140号命令也就可以执行了。



此时命令1-140现在已经被选择。leader还完成了所有大于140的共识算法实例的prepare阶段，并且它可以自由地在这些实例的accept阶段中提出任何值，它将客户端请求的下一个命令作为第141号命令，并将其作为共识算法实例141的accept阶段中的值。然后将接收到的下一个客户机命令作为第142号命令，以此类推。

leader可以在它学习到第141号命令被选择之前就发布第142号命令的提案。第141号命令的提案的消息有可能全部丢失，并且在任何其他服务器知道领导提案的第141号命令之前，第142号命令就已经被选择了。（丢失or网络延迟）。

当leader在实例141中没有收到对其accept阶段消息的预期响应时，它将重传这些消息。如果一切顺利，将选择提案中的命令。但是，它可能会失败，在所选命令的序列中留下空白。一般来说，假设leader可以提前获得a个命令，也就是说，在命令1到i被选中之后，它可以提出命令i+1到i+a。这样就会出现高达a-1个命令的缺口（第i+a被chosen，前面的全部失败）。

新当选的leader可以执行共识算法的无限多个实例的prepare阶段——在上面场景中，也就是实例135-137，以及所有大于139号的实例。对于所有实例使用相同的提议号，它可以通过向其他服务器发送一条合理的短消息来实现这一点。在prepare阶段中，只有当acceptor已经从某个proposer那里收到了accept阶段的消息时，它才会响应不止一个简单的OK（会响应value）。(在这个场景中，只有实例135和140是这种情况。)因此，服务器(作为acceptor)可以响应所有实例，并发送一条合理的短消息。因此，执行阶段1的无限多个实例 不会产生任何问题。

由于leader的失败和新leader的选举应该是较为罕见的事件，因此执行状态机命令的有效成本(即在命令/值上达成共识的成本)是仅执行共识算法的accept阶段的成本。可以看出，在存在故障的情况下， Paxos共识算法的accept阶段在所有算法中达成一致的代价最小。因此，Paxos算法 本质上是最优的。

在系统正常运行时总会存在一个leader，除了当前leader故障和新leader选举间的短暂时间。异常情况下，leader的选举可能会失败，如果没有leader，也就不会有新的命令被提出。如果有多个服务器认为它们是leader，那么它们都可以在共识算法的同一实例中发出提案， 这可能阻止任意值被选择。（上面的case）。但这显然是安全的，也就是说两个不同的服务器不会再第i个命令上产生分歧，选举一个leader来进行显然是最有效的。

如果服务器集合可以改变，那么一定有办法决定哪些服务实现共识算法的哪些实例。实现这个目的最简单的方法就是通过状态机本身。当前的一组服务器可以作为状态的一部分，并可以使用普通的状态机命令进行更改。