用paxos实现多副本日志系统--basic paxos部分

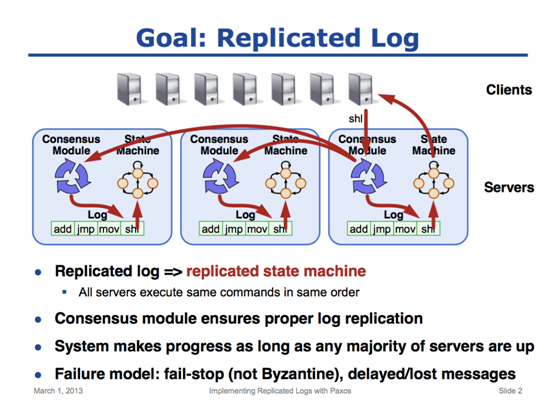
| 导语 为了理解paxos协议，开始时看了好些资料，但始终没有理解透，直到我看了这个视频：http://v.youku.com/v\_show/id\_XNjgyODc3ODU2.html。看懂之后，我就想按照自己的理解重新写一遍paxos的说明。结果写到一半，我发现越讲越不明白，反而不如之前我看到但没懂的资料了。这时我才意识到，或许这个斯坦福教授的讲解思路已经是最好的了，至少我跟着这个思路，把paxos协议理解清楚了。所以我把自己写了一半的paxos说明丢掉了，改为尝试逐页翻译（意译）这个视频的讲解。 希望这篇文章能够帮助到希望了解paxos协议的同学。

手机屏幕截图

描述已自动生成

Hello, 我是John Ousterhout。

在这个视频里，我会给大家讲解paxos一致性协议。paxos在80年代被莱斯利·兰伯特发明后，几乎成为了一致性协议的代名词。大学里讲一致性协议时，基本上都是讲paxos算法，大部分实现出来的一致性系统也都是基于paxos协议的。paxos是最重要的，也可能是唯一的一致性协议。

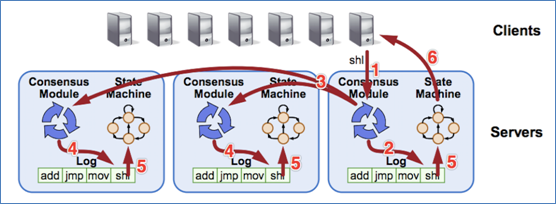


我将以多副本log系统的实现为背景，来解释paxos协议。这个系统使用一系列的log作为输入，来驱动自己的状态机的改变。这里状态机的概念是指一个程序，接收输入，改变自己的状态，产生输出。从这个角度来说，所有程序都是一种状态机。

不同的备份状态机，如果能保证初始状态一致，输入的内容和顺序（即log）完全一致，那么使用这些log最终跑出来的状态机状态，也将是一致的。

所以这几个备份对外提供服务时，如果一个副本挂掉了，其他副本能够完全替代他接着服务。所以多副本log系统可以实现多副本状态机。

具体的做法如下：



1. 一个client希望状态机（state machine）执行某个动作, 则发起请求到server端。server端的一致性模块（consensus module）接收此请求。

2. 一致性模块将此请求暂时记录到本地log。

3. 一致性模块也同时将此请求同步给其他server。

4.其他server也将此请求记录入log。在一致性模块确定这个log已经被其他server确认后（即已经达到了一致性了）。

5. 各server再把此请求提交给状态机进行执行。

6. 状态机执行完请求后，返回结果给客户端。

状态机的初始状态是一致的，只要这个一致性模块能保证log的一致性，那么就可以保证状态机最终达到一样的状态。而一致性模块正是使用paxos来保证log的一致性的。这是paxos协议起作用的地方。

这个系统最重要的一个特性是，只要server里边的过半数(majority)是正常工作的，就能保证这个系统能正常对外提供服务。假设我们有5台server，那么允许有两台机器当机，而整个系统仍能正常工作。所以典型的server数会选择3，5，或7。

让我们再来看下这个系统的失效模型(failure module)。这个系统的失效模型是fail-stop模型，也就是说一台机器可能crash并停机，但一旦一台server恢复并上线，那他就是正常的表现。不存在一台机器故障时，我们认为他在线，但却对外表现出诡异的状态。诡异的状态就是指反馈一些错误的信息给其他机器（比如被黑客攻破）。这种存在欺骗行为的故障，叫做拜占庭(Byzantine)问题，paxos是不解决这种问题的。paxos认为消息在网络上被延时或丢失或重放都是ok的；但消息被篡改的问题要通过其他方法去解决。

手机屏幕截图

描述已自动生成

有好几种方法来分解多副本log系统的问题。但用paxos来解决这个问题时，分解出来的第一步是简单且容易想象的，即basic paxos。

basic paxos又称为single decree paxos。这一步需要处理的问题是：

1. 有一台或多台服务会提议（propose）一些值。

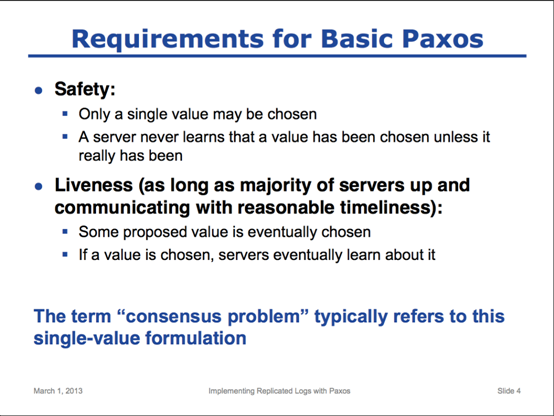
2. 系统必须通过一定机制从propose的值中选定（chose）一个值。

3. 只有一个值能被选定（chosen）。即不允许系统先选定了值A，后面又改选定值B。

这些问题可能是你能想象的最基本的一致性问题。当人们在使用“一致性”这个术语时，他们一般就是在讨论这种最基本的形式。而当人们在讨论paxos协议时，也可能就是在讨论basic paxos。

一旦我们解决了basic paxos这个最简单形式的一致性，我们就能通过单个paxos实例来确定单条log的一致性。

而通过一系列的paxos实例，我们就能确定一系列log的一致性问题，这就是multi paxos。我们先来看basic paxos。



在我们进一步深入讨论basic paxos细节之前，我们先来看下basic paxos需要达到什么样的要求。总的来说，basic paxos需要达到两方面的要求：安全性（safety）和活性（liveness）。

安全性(safety)这个要求就是指这个系统决不能做出不对的事情。对于paxos来说，就是指：

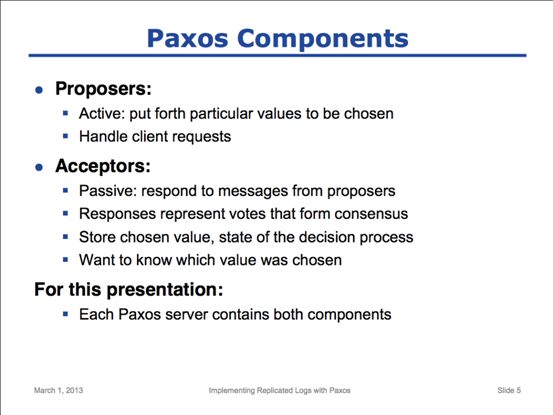
1. 必须保证只有一个值可能被选定，绝对不能出现系统中有两个值被选定的状况。

2. 一台server如果根据算法得出一个值被选定的结论了，那这个值就一定是真的被选定，并且永远不变了。

以上就是安全（safety）特性。而活性（liveness）特性指的是：只要server中的过半数（majority）能正常工作的，并且消息也能够在合理的延时范围内到达，那这个系统就总能正常工作，得出正确的结果。正常工作包括两方面：

1. 总是能最终选定一个值，而不是一直处在没有值被选定的状态。

2. 如果一个值被选定（chosen），那其他server最终必然能够得知这个值。



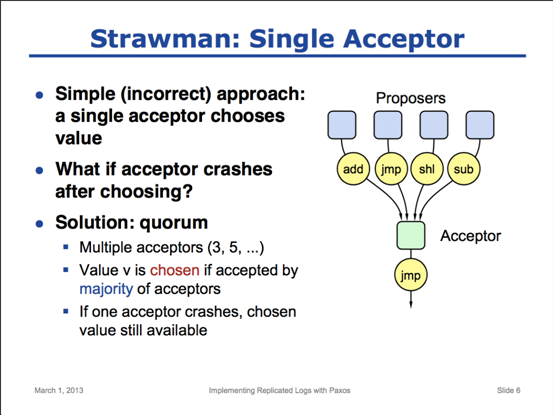
basic paxos需要两个组件(component)来配合工作：提议者（proposers）和接受者（acceptors）。

提议者（proposers）是主动部分，会主动尝试做一些事情。提议者（proposers）会接受客户端的请求，并根据请求在系统中向其他server提议特定的值，尝试让其他server同意并最终选定此值。

接受者（acceptors）是被动部分，只有在提议者（proposers）触发他时，他才做出响应。而响应的结果你可以看成是一次投票，表示接受者（acceptor）是否接受提议者（proposer）的提议。接受者（acceptor）还要保存一些特定的信息，比如自己选定了什么值，当前自己处于决策中的哪个步骤等。最后，接受者（acceptor）自己也需要知道最终什么值被整个系统选定了（acceptor自己接受的值并不一定是系统选定的值）。接受者（acceptor）知道这个值后才能做进一步的动作，以多副本log系统为例，在确定值后，才能把这条log输入给状态机。

在传统的兰伯特博士的paxos范式里边，还包括第三个角色：监听者（listener）。监听者就是希望知道最终系统选定了什么值的角色，这个角色在这里也被包含在了acceptor的功能里边，所以不需要这个角色了。

还要说明的是，在这个视频里边，每个server都假设同时包含两个角色，上面都部署一个proposer，一个acceptor。但两个角色分离在不同的机器，每台机器都只扮演一个角色（proposer或acceptor），这是完全可能的。



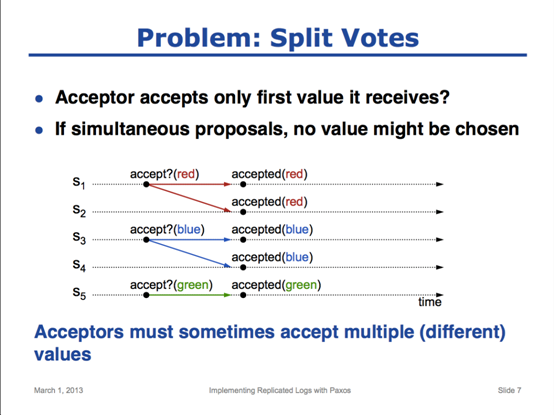
在接下来的几页ppt里面，我们会看一下几种简单的解决方案，以此来说明在达到一致性的路上，我们需要解决哪些问题。

这页ppt展示的是一种非常简单的方案，但并不能正确工作。这个方案就是只设置一个acceptor，让这个acceptor来处理所有的提议。当不同的提议被proposers先后提交之后，acceptor只选择其中的一个值作为选定值（chosen value）。

这很简单，但不幸的是，一旦这个唯一的acceptor 在选好一个值后就crash了，那我们就无法知道到底什么值被选定了，只有等这个acceptor重启恢复后，这个系统才能接着工作。

记住这个系统的目标之一是，当这个系统的过半数(majority)仍在线时，这个系统要能正常工作。而现在一台机器crash就不能工作了。

解决的方法就是我们必须以某种方式使用少数服从多数（quorum）的机制。我们要有一组acceptors，而不是一台，典型的数量是3，5，7等。而一个值必须被过半数accepor接受之后，才能被认为是真的选定了。这样的话，如果一个acceptor在接受某个值后crash了，那剩下的acceptor仍能告知我们被选定的值是什么。



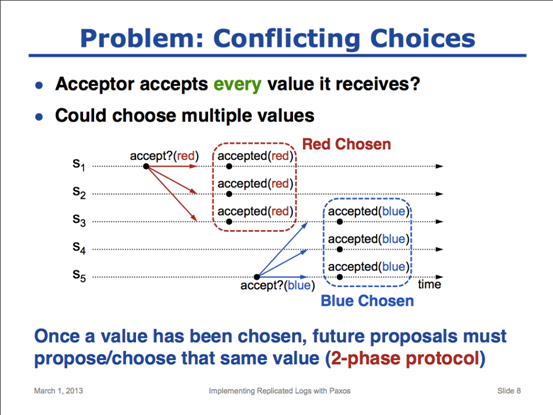
然而，要使少数服从多数（quorum）的机制真的生效，还是很有技巧性的。例如，假设我们只设置一个条件：acceptor只接受他收到的第一个值。然后我们定义被过半数（majority）aceeptor同意的值为选定值。

（在说算法之前，先解释下图示的含义，在后面的说明中会很多次使用这种图来进行说明。图中的S1~S5代表着5台服务器，每台服务器都是同时担任proposer和acceptor两种角色。s1~s5右边那长长的虚线剪头表示时间线。accept?(red) 表示这个server提议一个值red； accepted(red)表示服务器同意接受一个值red）

那有些情况下，我们根本就不能得到大多数。如图所示，s1向s1、s2提议red值，s1、s2同意了；s3向s3、s4提议blue值，s3、s4同意了；s5向s5提议green值，s5也同意了。这么下来，没有任何值是被过半数acceptor接受的，所以根本就没有值被选定。

这似乎意味着，要允许一个acceptor改变自己的主意，也即允许一个acceptor有时接受多个不同的值。从另一个角度来说，是没有办法保证一轮投票就得出一个大多数的，要允许多轮投票。

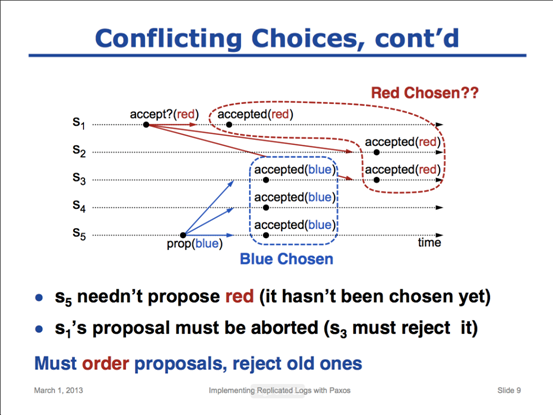
在这里需要给大家说明的是：接受（accept）并不意味着（chosen）。一个值只有被过半数acceptor接受，这个值才是被选定。



现在，我们来尝试下另一种方式：让acceptor接受他收到的所有值。这会导致两个问题，我们分别在这一页和下一页ppt中说明。第一个问题就是这会造成多个值被选定。

例如图中所示，s1向s1、s2、s3提议red值，s1、s2、s3都同意了。这时根据过半数同意即为选定的原则，那么s1就认为red值已经被选定了。但后来s5又向s3、s4、s5提议blue值，s3、s4、s5也同意了，同样，s5认为blue值被选定了。这导致了整个系统有两个值都被选定。这就违反paxos的安全（safety）特性：只有一个值能被选定，并且一旦被选定就不能修改。

这个问题的解决方案是：在s5提议前，如果发现一个值已经被选定，那么他就必须抛弃自己的提议，而只能提议被选定的值。所以在s5提议blue前，s5要先确定是否有其他值被选定，通过某种方式，s5知道red已经被选定，那s5就必须抛弃自己blue的提议，转而提议red值。这时s3、s4、s5收到s5的提议，虽然再次同意了，但因为同意的都是red值，所以最终被选定的仍是red值，并不违反paxos的安全（safety)特性。这种解决方案就是二阶段协议（2-phase protocol）。



不幸的是，二阶段协议（2-phase protocol）本身并不足以解决一致性问题。

假设s1准备向s1、s2、s3提议red，s1在提议前，检查发现当前没有其他值被选定，所以提议被发出。

但几乎同时，在s2、s3同意s1之前，s5也准备向s3、s4、s5提议blue。提议前，s5向s3、s4、s5询问也确认了当前没有其他值被选定，所以提议blue被发出。

因为网络延时等原因，s5的blue建议先被过半数acceptor同意了，s5认为blue已经被系统选定。此后s2、s3才又收到s1迟来的提议red，并且根据当前的规则（acceptor接受所有自己收到的提议），s2、s3也同意了red提议。这样s1就认为red值被选定，这就又造成了有两个值被选定。

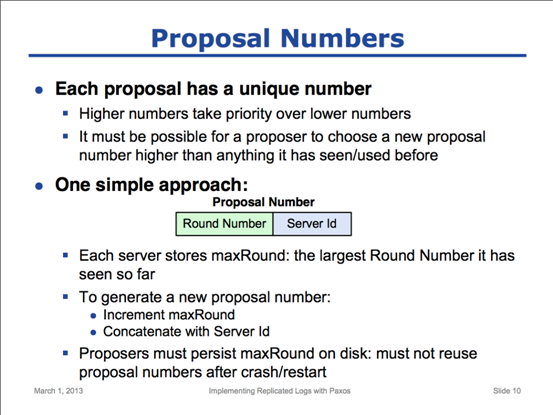
这个问题的解决方案是：一旦我们选定的一个值，其他竞争性的提议应该被acceptor拒绝并最终被整个系统抛弃。在这个例子里边，s3已经先接受了s5的blue值，那s3需要以某种方式拒绝掉后面提交上来的red提议。

为了达到这个目的，我们必须对所有的提议进行排序，如果acceptor已经接受了更新的提议，那他就应该拒绝掉老的提议。所以，在这个例子里边，s3已经接受了较新的提议blue，那s3在收到老的提议red时，s3要有办法中断s1的提议。

所以总结起来是：

1. 我们需要二阶段协议，在提议前先检查是否有其他值被选定，如果有，就抛弃自己的值，改提议已经选定的值

2. 所有的提议要有序。如果acceptor已经接受了新的提议，就应该拒绝掉老的提议。



让我们看下怎么来确定提议的顺序。我们通过给每个提议附一个唯一的值来表明提议的顺序。这个值必须是从来没有被之前的提议使用过。我们定义大的值总是优先于小的值（即提议号越大越新），所以如果一个proposor要有提议的能力的话，他必须有能力生成一个比他看到过或用到过的所有number都大的值，否则他就不能发出任何提议了。

要做到这一点，一种方法是：

1. 把服务器id作为提议号的低bit部分。这就保证了其他服务器肯定不可能生成一样的号出来。

2. 而提议号的高bit部分是一个round number。每个server都保存了自己至今为止所看到或用到过的最大的round number，设这个值为maxRound。

3. 要生成一个新的提议号时，server用maxRound+1来作为round number，拼接上自己的server id，就得到了一个提议号。（这个提议号不一定是全局最新的，但server很快就会发现这一点）。

4. 为了确保一个proposer在crash后重启，不会碰巧使用了之前用过的提议号，proposer每次更新maxRound时，必须马上把maxRound永久存储在磁盘里。

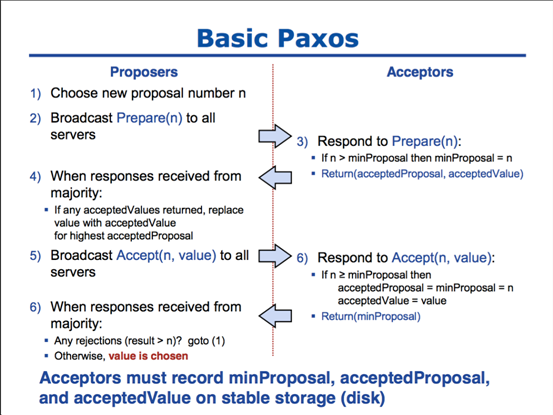
手机屏幕截图

描述已自动生成

这里我们先概述下basic paxos，然后我们再来讲解更多的细节。就像我们前面已经讲过的，我们使用了两阶段(Two-phase)的方法：

阶段1，我们想尝试提议一个值时，先广播一个我们称为prepare 的rpc调用。prepare rpc要达到两个目的：1.找到已经被选定的值（如果有，我们就要在第二阶段时使用这个值）；2. 阻塞掉还没有完成的老提议，以阻止老的提议和我们的新提议竞争。这两个作用就解决了我刚才演示给你们看的两个问题。

阶段2，我们广播另一个被称为accept 的rpc调用，以让系统确认接受一个特定值。一旦过半数acceptor在这个调用中回复"接受"，那我们就可以确定这个提议已经是"被选定（chosen）"了。



这一页ppt，我们演示一个实际的paxos实现的操作细节。让我们沿着basic paxos的生命周期中必须的操作走一遍。

正如我前面提到过的，整个过程都是由proposer作为主动者来驱动的。

proposer想要提议某个值时，就会触发启动这个流程。整个流程propser会至少触发两轮消息广播。第一轮是prepare阶段，第二轮是accept阶段。

在进行两轮广播之前，我们首先要生成一个提议号，n。这个提议号必须是从来没有被使用过的一个值。

然后我们进入prepare阶段，在此阶段，我们向所有的acceptors广播prepare请求（其实只要向过半数acceptor发起就能保证正常工作）。这些prepare请求都必须带着提议号n。当acceptor接收到这个prepare请求后，他做两件事情：

1.首先他承诺，永远也不会同意请求号比这n值小的提议。为了做到这一点，acceptor必须保存一个minProposal值，minProposal表示acceptor在第二阶段能够accept的提议号必须至少大于等于minProposal。但在第一阶段，acceptor只需要比较prepare请求里的提议号n是否大于自己保存的minProposal，如果是，则更新minProposal即可。

2. acceptor如果之前已经accept了一个值，那么acceptor当时要存储记录下已经接受的值acceptedValue和接受的值对应的提议号acceptedProposal，并在此时把acceptedValue和acceptedProposal返回给proposer。如果acceptor没有accept过任何值，那就通过返回特定值（例如NULL）的方式告知proposer，自己没有accept过任何值。

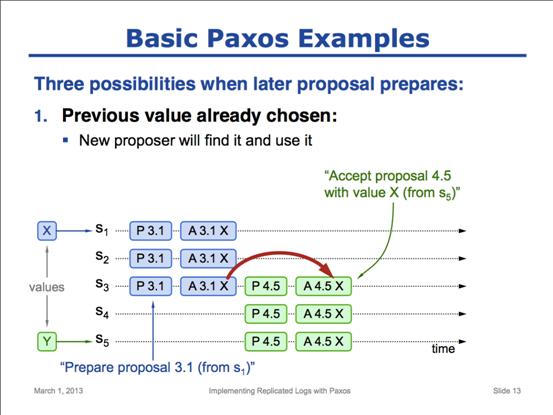
       Proposer在发出prepare请求之后，就等待acceptor响应，必须至少要有过半数的acceptor响应之后，proposer才能进入第二阶段。如果有acceptor告知proposer他已经接受了一个值acceptedValue（可能同时有多个acceptor都返回自己接受了一个值，并且还各不相同），那proposer必须把自己提议的值修改为 所有acceptor返回的提议中，提议号最大的提议对应的acceptedValue。如果所有的acceptor都返回自己未接受过任何值，那proposer就可以仍使用自己的value和提议号，进入accept阶段。

等proposer收到过半数的acceptor响应之后，proposer就进入第二个阶段，即accept阶段。accept rpc输入两个值，一个是提议号n，这个n必须与prepare时的提议号n相等；另一个值是value，value要不就是proposer起始时想要提议的值，要不就是acceptor响应返回的acceptedValue。这个accept请求会发送给所有的acceptor。acceptor收到accept请求时，首先比较请求中的提议号n与自己存储的minProposal值，如果请求中的n小于minProposal，那acceptor就直接拒绝这个请求；如果请求中的n大于等于minProposal，就更新minProposal值为n，并且替换自己的acceptedValue为请求中的value值。

不管acceptor是接受还是拒绝这个提议，acceptor都会返回自己的minProposal值给proposer。

proposer在发送完accept请求后，就等待acceptor的响应。直到过半数的acceptor响应之后，proposer才能决定下一步做什么。如果proposer收到的响应中有拒绝（rejection），那proposer就放弃此轮paxos，回到第一步重新再来：生成新的提议号->prepare阶段->accept阶段。如果proposer收到了过半数的acceptor的接受(acception)，那么proposer就可以确定，自己提交的值被选定了。proposer通过比较acceptor返回的结果值（即acceptor的minProposal值）来确定自己到底是被拒绝还是被接受了：如果结果值大于自己prepare时的提议号，那自己就是被拒绝了；否则，自己就是被接受了。

在这整个过程中，我们可以看出，acceptor必须确保在永久存储（类似磁盘、flash等能够在停机后恢复的存储）中妥善保存了三个值：minProposal，acceptedProposal，acceptedValue。三个值分别代表着：acceptor能够同意的最小提议号，已经接受过的提议号，和已经接受过的值。

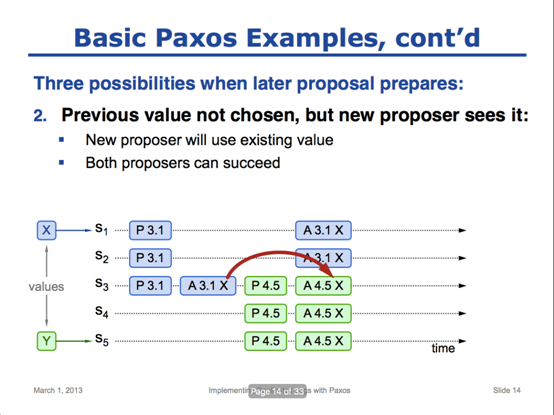


接下来，我们来看一下basic paxos在几种特定的竞争状态下，怎么保证正常工作的。你们需要知道的一点是，如果说basic paxos能因为竞争而导致出问题，那出问题的关键时间点就是某个server对一个议题发起了第二次prepare时。（如果第二次prepare都没有，哪来竞争呢？如果第二次prepare时能顺利被解决，那第三次prepare的发出应该可以类推）。我们知道所有的提议都是被按顺序编号的，所以我们要关注的就是编号较大的那次prepare。

当第二次prepare rpc发出时，basic paxos有可能处于三种状态的任一状态中。三种状态分别在本页、下一页、后一页中讲述。第一种可能的状态就是上一个提议已经走完全部流程并被选定了。这就是这一页ppt所示的情况。

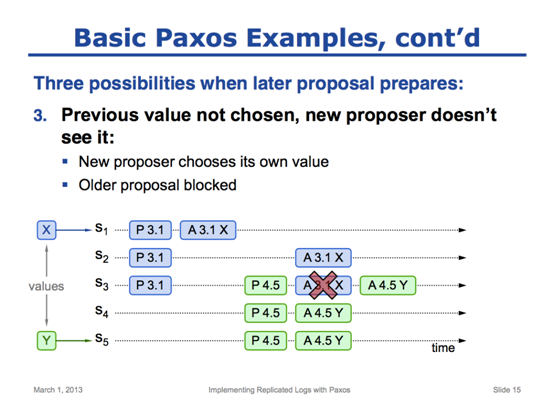
在进一步解释之前，我们先说明下图中各种标识的含义，因为这里使用的标识和前面的已经很不一样了。在图的最左边，你能看到有两个不同的准备被提议的值：s1准备提议的X和s5准备提议的Y。虚线的黑色线是表示一个acceptor的时间线（所以这个图里其实只能看到acceptor上发生的事情）。时间线上里边有以P开头的文字的那些长方形，表示某个acceptor接收到了一个Prepare请求，P后面跟着的数值（3.1，4.5这些），就是提议号。前面我们说过提议号的低bit位表示server id，所以可以知道P3.1是来自s1的prepare请求，而P4.5则是来自s5的prepare请求。所以我们从图中可以看出，s1向s1、s2、s3发起了prepare（3.1）的rpc请求。类似的，三个A4.5 X的长方形则表示三个acceptor接收到了提议号为4.5的提议（我们可以得出这个accept请求来自s5），值为X。

所以在这个图中，我们看到s1提议的X值已经被3个acceptor接受，达到大多数，X值已经被选定。s1也从acceptor的响应中学习到了X值被选定这一点。之后，s5尝试让系统接受Y这个值。如果s5要让Y值被选中，那他必须向过半数的acceptor提交这个值，而在一个集合里边随意选择的两个过半数集合，必然会有交集。在这个例子里边，这个交集就是s3。所以当s5向s3发起prepare请求时，s3就会返回给s5：自己已经选定了值X，提议号为3.1。这时根据paxos协议，s5必须丢弃自己的值Y，改为提议值X。所以s3、s4、s5收到s5提交的A4.5 X。s3、s4、s5都同意这个新的提议，所以X这个值由原来的被3个acceptor接受变成被5个acceptor同意。值并没有改变，反而巩固了。



第二种和第三种可能的状态都是在第二个prepare发出时，还没有值被选定。先提出的值可能因为网络延时，还没有被大多数acceptor接受，只是被部分acceptor接受了。此时新的proposer再发起prepare，只会有两种可能情况：新的proposer碰巧发现了这个值，这就是这一页ppt要描述的情况；另一种情况是新的proposer没有发现这个值，这是下一页ppt要描述的情况。

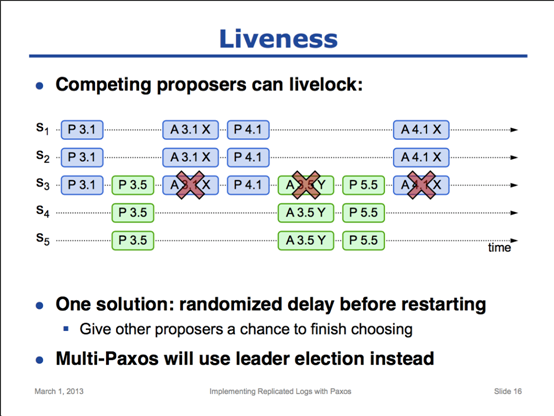
以图中的情况为例，s1尝试让过半数的acceptor接受X值，在和s3交互很顺利，s3很快就达到接受值X的状态了，而s1发往s1、s2的accept请求延时很大，仍没有收到回复。此时s5开始尝试让过半数的acceptor接受Y值，但s3因为之前已经接受了X值，提议号为3.1，所以s5在prepare阶段时就从s3的响应得知了这一点，s5就不再提议自己的值Y，改提议值X。这样，s1和s5的提议都最终会被通过，但值也是确定的X。



第三种情况的部分前提条件和第二种情况是一样的：在第二个prepare发出时，先提议的那个值也是只被部分acceptor接受了，但未达到过半数而没有被选定。和第二种情况不一样的是：新的proposer提出新的值时，并没有看到旧proposer提出的旧值。

以图中的情况为例，s1尝试让s1、s2、s3接受值X，prepare阶段没有问题，但accept阶段，只有s1很快同意了值X，而s2、s3因为网络延时，一直未收到s1的accept请求。此时s5尝试让s3、s4、s5接受值Y，发起了prepare请求，因为三个acceptor都没有接受其他值，所以三者都返回了4.5这个值给s5，s5进入accept阶段。这里我们特别说明下s3的情况，s3之前已经收到了s1的prepare请求，回顾之前的讲解，我们知道s3会记录下minProposal为3.1，但s3在还没有收到编号为3.1的accept请求之前，又收到了编号为4.5的prepare请求，这个编号比3.1大，所以s3把minProposal替换成4.5，minProposal的含义是该acceptor所能接受的最小提议号，所以如果s3在收到3.1的accept请求时，就会因为3.1比自己的minProposal小而直接拒绝这个accept请求。s1得知s3拒绝了自己的accept之后，根据协议要求，只能生成新的提议号，重新prepare。但在这个case中，s3在拒绝s1之后，就接受了提议号为4.5的Y值。并且同时s4、s5也已经接受了Y值。Y值已经被大多数acceptor接受而被选定。所以当s1发起第三轮prepare请求时，我们又回到了第一种情况：一个值已经被选定时，有prepare发起的情况。s1在走完第三轮prepare和accept阶段后，系统会把s1、s2的值也更改为Y值。最终结果是所有的acceptor都同意值Y。

这里关键的点在于，一个集合任意选择的过半数，总是有交集的。而处于交集中的acceptor在处理不同proposer提出的请求时，只会处于两种状态：1.acceptor收到了第一个提议的prepare请求，但还没有收到第一个提议的accept请求，此时如果收到第二个提议的prepare请求，那第二个提议的prepare请求中的提议号因为必然比第一个提议的prepare的提议号大，所以让第一个提议的accept请求必然被拒绝掉。2.另一种状态就是收到第二个提议的prepare请求前，acceptor已经收到了第一个提议的accept请求，即acceptor已经接受了一个值，此时acceptor会让第二个提议者得知这一点，第二个提议者也转而提议与acceptor已接受的值一致的值。在这两种情况下，都能消除不一致性，从而使得整个系统达到最终一致。所以这两种情况下，paxos都是安全的。



现在，我们已经知道basic paxos是安全的了。不管是在哪种竞争状态下，我们都能使得只有一个值被选定，并且一旦被选定，这个值就不会再变化。但达到安全性的paxos并不一定总是活着的（live，不一定活着的意思是就算过半数acceptor在线，也不能总是保证系统是可用的）。我们可能达到的一种情况是：两个proposer都在提交提议，但是两个proposer都处于动态活锁中，而一直没有一个值被真正地选定。这一页讲解这种情况。

假设s1、s2、s3收到了s1发起的prepare请求，提议号为3.1。但在收到s1发起的accept请求前，s3、s4、s5又收到了s5发起的prepare请求，提议号为3.5。因为3.5比3.1大，所以s3在收到accept请求时会拒绝接受3.1这个提议。s1得知s3拒绝了3.1提议后（并且同时得知s3当前的minProposal值是3.5），所以马上生成了4.1这个提议号，再次向s1、s2、s3提交了prepare请求。此时s3的minProposal是3.5，并且并没有接受任何值，所以将自己的minProposal改为4.5。之后，s3才又收到了s5提交的编号为3.5的accept请求，但此时s3的minProposal值已经是4.5了，所以s3会拒绝掉3.5这个accept请求。s5得知编号为3.5的accept请求别拒绝后，也重新生成了更大的提议号，5.5，重新提交了prepare请求，这又会导致s3拒绝掉s1的编号为4.1的accept请求。如此反复，则整个系统都始终处于一种活锁状态中，而不能决定出什么值被选定。

要解决这种活锁问题，一种简单的办法是，server如果发现自己的提议被拒绝，那表明有其他server也在想设定这个值，这时server随机地等待一段时间，让另一个server可以有更大的机会来完成整个流程，最终把这个值确定下来。但我们将会看到，在mutil-paxos中我们用另一种方法来解决这个问题：使用leader选举机制来确保在一个时间段内，只有一个proposer在工作。

手机屏幕截图

描述已自动生成

basic paxos还有一个缺点我们前面是没有提到的。那就是只有发起了提议的那些proposer知道什么值被选定了。比如acceptor就不知道到底什么值被选定了。如果其他server想知道到底什么值被选定，唯一的方法就是自己发起一次paxos的提议过程：生成一个提议号，然后进入prepare阶段，你可能会得到一个acceptedValue值（如果不能得到，那说明当前还没有值），然后你进入accept阶段，如果你能让过半数acceptor接受这个acceptedValue，那么你就知道这个值就是被选定的值了。

到此，我们就把所有basic pasxos相关的讲解都讲解完了。