微信自研生产级paxos类库PhxPaxos实现原理介绍

原创 2016-06-22 lynncui 微信后台团队

> 微信重磅开源生产级paxos类库PhxPaxos，本文用科普的口吻向大家介绍PhxPaxos背后的实现原理以及一些有意思的细节。

> 开源地址：
> https://github.com/tencent-wechat/phxpaxos
> 点击阅读原文可自动跳转到github地址

前言

> 本文是一篇无需任何分布式以及paxos算法基础的人可以看懂的。

标题主要有三个关键字，生产级，paxos，实现，涵盖了本文的重点。什么叫生产级，直译就是能用于生产线的，而非实验产品。生产级别拥有超高的稳定性，不错的性能，能真正服务于用户的。paxos就不用说了，而实现，是本文最大的重点，本文将避开paxos算法理论与证明，直入实现细节，告诉大家一个生产级别的paxos库背后的样子。为何要写这篇文章？paxos算法理论与证明不是更重要么？我几年前曾经也读过Paxos论文，虽然大致理解了算法的过程，但是在脑海中却无一个场景去构建这个算法，而后也就慢慢印象淡化以至于最近重读paxos论文的时候，感觉像是第一次读论文的样子。

在真正去实现了paxos之后，我才顿悟了这个问题。人们去理解一个理论或事务的时候，都会往这个理论或事物上套一个场景，然后在脑海里模拟而试图去懂得他。比如经典的Dijkstra算法，事实它并不只用于最短寻路，然而在学习这个算法的时候，最短寻路给我们提供了一个很好的场景，可以让我们更快的去理解它。

第一次读paxos论文，我不知道它的应用场景，不知道它是用来干什么的，也不知道它怎么实现，然而这些往往就是一个基础，大神可能可以自己创造场景，然而更多的人都需要知道这个场景，当你知道后，理解算法将变得更为容易。

本文，将告诉你paxos是什么，用来做什么，怎么使用它，如何工程化，如何做到生产级别，以及在工程上会遇到的问题与解决办法。文中将用尽量讲课方式的口吻，以及尽量避免专业术语，力求把这么一件事能阐述的更为通俗和简单易懂。

什么是paxos

一致性协议

Paxos是一个一致性协议。什么叫一致性？一致性有很多种，也从强到弱分了很多等级，如线性一致性，因果一致性，最终一致性等等。什么是一致？这里举个例子，三台机器，每台机器的磁盘存储为128个字节，如果三台机器这128个字节数据都完全相同，那么可以说这三台机器是磁盘数据是一致的，更为抽象的说，就是多个副本确定同一个值，大家记录下来同一个值，那么就达到了一致性。

Paxos能达到什么样的一致性级别？这是一个较为复杂的问题。因为一致性往往不取决与客观存在的事实，如3台机器虽然拥有相同的数据，但是数据的写入是一个过程，有时间的先后，而更多的一致性取决于观察者，观察者看到的并未是最终的数据。这里就先不展开讲，先暂且认为paxos保证了写入的最终一致性。

为何说是一个协议而不是一个算法，可以这么理解，算法是设计出来服务于这个协议的，如同法律是协议，那么算法就是各种机构的执行者，使得法律的约束能得到保证。

Paxos的协议其实很简单，就三条规定，我认为这三条规定也是paxos最精髓的内容，各个执行者奋力的去保护这个协议，使得这个协议的约束生效，自然就得到了一致性。

分布式环境

为何要设计出这么一套协议，其他协议不行么。如最容易想到的，一个值A，往3台机器都写一次，这样一套简单的协议，能不能达到一致性的效果？这里就涉及到另外一个概念，Paxos一致性协议是在特定的环境下才需要的，这个特定的环境称为异步通信环境。而恰恰，几乎所有的分布式环境都是异步通信环境，在计算机领域面对的问题，非常需要Paxos来解决。

异步通信环境指的是消息在网络传输过程中，可能发生丢失，延迟，乱序现象。在这种环境下，上面提到的三写协议就变得很鸡肋了。消息乱序是一个非常恶劣的问题，这个问题导致大部分协议在分布式环境下都无法保证一致性，而导致这个问题的根本原因是网络包无法控制超时，一个网络包可以在网络的各种设备交换机等停留数天，甚至数周之久，而在这段时间内任意发出的一个包，都会跟之前发出的包产生乱序现象。无法控制超时的原因更多是因为时钟的关系，各种设备以及交换机时钟都有可能错乱，无法判断一个包的真正到达时间。

异步通信环境并非只有paxos能解决一致性问题，经典的两阶段提交也能达到同样的效果，但是分布式环境里面，除了消息网络传输的恶劣环境，还有另外一个让人痛心疾首的，就是机器的当机，甚至永久失联。在这种情况下，两阶段提交将无法完成一个一致性的写入，而paxos，只要多数派机器存活就能完成写入，并保证一致性。

至此，总结一下paxos就是一个在异步通信环境，并容忍在只有多数派机器存活的情况下，仍然能完成一个一致性写入的协议。

提议者

前面讲了这么多都是协议协议，在分布式环境当中，协议作用就是每台机器都要扮演一个角色，这个角色严格遵守这个协议去处理消息。在paxos论文里面这个角色称之为Acceptor，这个很好理解。大家其实更关心另外一个问题，到底谁去发起写入请求，论文里面介绍发起写入请求的角色为提议者，称之为Proposer，Proposer也是严格遵守paxos协议，通过与各个Acceptor的协同工作，去完成一个值的写入。在paxos里面，Proposer和Acceptor是最重要的两个角色。

Paxos是用来干什么的

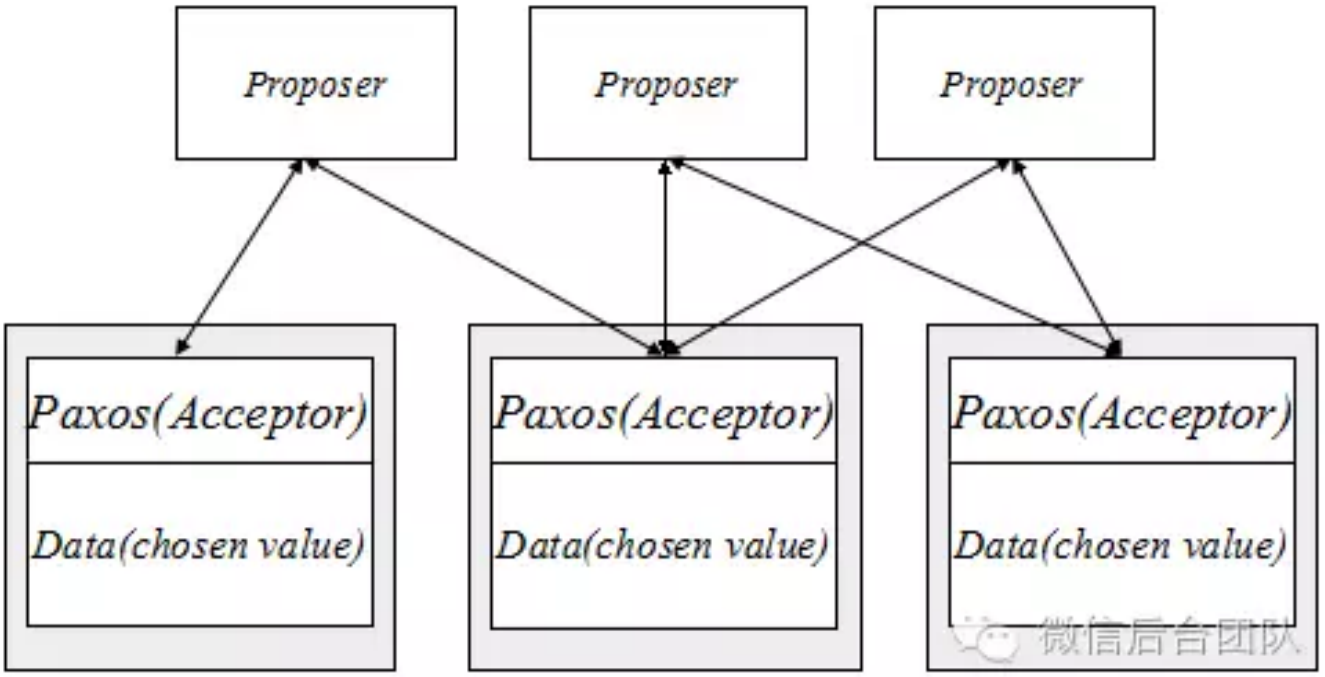
确定一个值

既然说到写入数据，到底怎么去写？写一次还是写多次，还是其他？这也是我一开始苦恼的问题，相信很多人都会很苦恼。

这里先要明确一个问题，paxos到底在为谁服务？更确定来说是到底在为什么数据服务？还是引上面的例子，paxos就是为这128个字节的数据服务，paxos并不关心外面有多少个提议者，写入了多少数据，写入的数据是不是一样的，paxos只会跟你说，我确定了一个值，当这个值被确定之后，也就是这128个字节被确定了之后，无论外面写入什么，这个值都不会改变再改变了，而且三台机确定的值肯定是一样的。

说到这估计肯定会有人蒙逼了，说实话我当时也蒙逼了，我要实现一个存储服务啊，我要写入各种各样的数据啊，你给我确定这么一个值，能有啥用？但先抛开这些疑问，大家先要明确这么一个概念，paxos就是用来确定一个值用的，而且大家这里就先知道这么个事情就可以了，具体paxos协议是怎样的，怎么通过协议里面三条规定来获得这样的效果的，怎么证明的等等理论上的东西，都推荐去大家去看看论文，但是先看完本文再看，会得到另外的效果。

如下图，有三台机器（后面为了简化问题，不做特别说明都是以三台机器作为讲解例子），每台机器上运行这Acceptor来遵守paxos协议，每台机器的Acceptor为自己的一份Data数据服务，可以有任意多个Proposer。当paxos协议宣称一个值被确定（Chosen）后，那么Data数据就会被确定，并且永远不会被改变。



Proposer只需要与多数派的Acceptor交互，即可完成一个值的确定，但一旦这个值被确定下来后，无论Proposer再发起任何值的写入，Data数据都不会再被修改。Chosen value即是被确定的值，永远不会被修改。

确定多个值

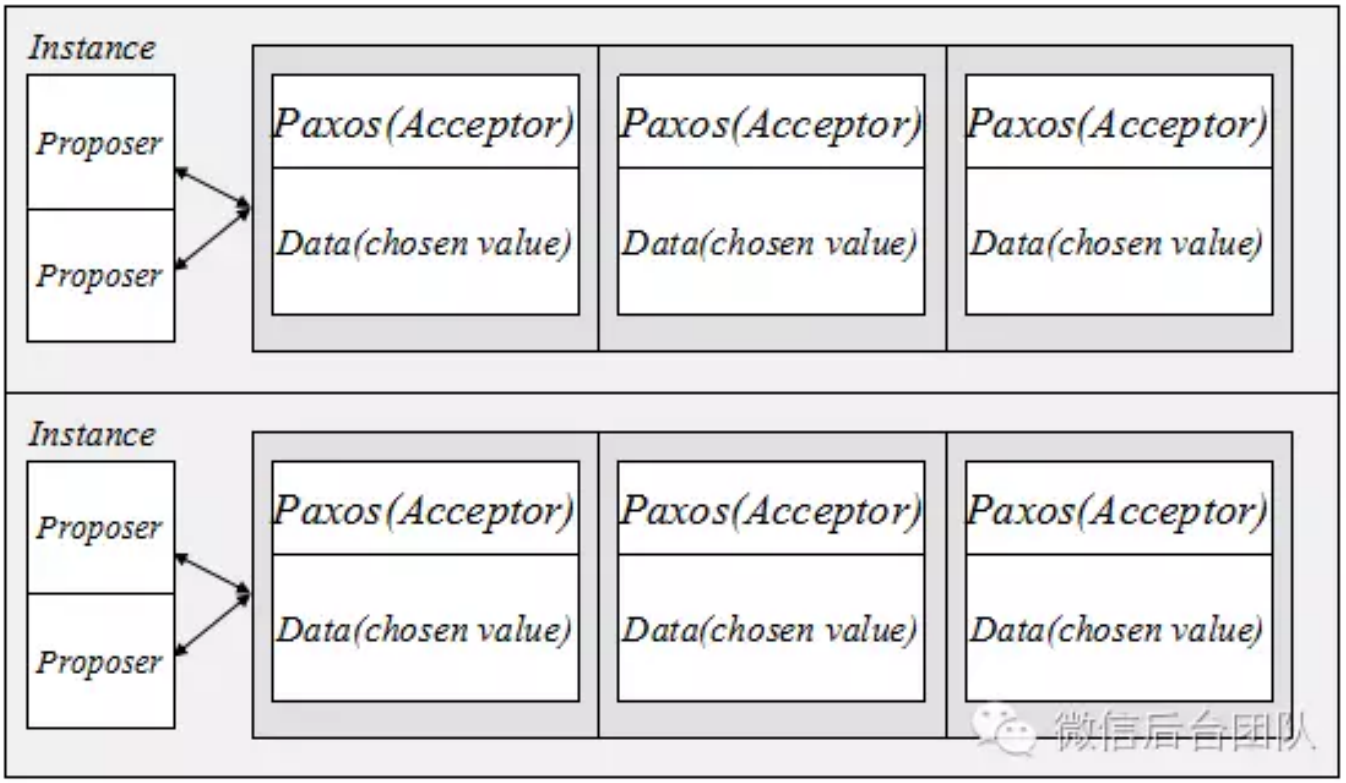
对我们来说，确定一个值，并且当一个值确定后是永远不能被修改的，很明显这个应用价值是很低的。虽然我都甚至还不知道确定一个值能用来干嘛，但如果我们能有办法能确定很多个值，那肯定会比一个值有用得多。我们先来看下怎么去确定多个值。

上文提到一个三个Acceptor和Proposer各自遵守paxos协议，协同工作最终完成一个值的确定。这里先定义一个概念，Proposer，各个Acceptor，所服务的Data共同构成了一个大的集合，这个集合所运行的paxos算法最终目标是确定一个值，我们这里称这个集合为一个paxos instance，即一个paxos实例。

一个实例可以确定一个值，那么多个实例自然可以确定多个值，很简单的模型就可以构建出来，只要我们同时运行着多个实例，那么我们就能完成确定多个值的目标。

这里强调一点，每个实例必须是完全独立，互不干涉的。意思就是说Acceptor不能去修改其他实例的Data数据，Proposer同样也不能跨越实例去与其他实例的Acceptor交互。

如下图，三台机器每台机器运行两个实例，每个实例独立运作，最终会产生两个确定的值。这里两个实际可以扩展成任意多个。



至此，实例(Instance)以成为了我现在介绍paxos的一个基本单元，一个实例确定一个值，多个实例确定多个值，但各个实例独立，互不干涉。

然而比较遗憾的一点，确定多个值，仍然对我们没有太大的帮助，因为里面最可恨的一点是，当一个值被确定后，就永远无法被修改了，这是我们不能接受的。大部分的存储服务可能都需要有一个修改的功能。

有序的确定多个值

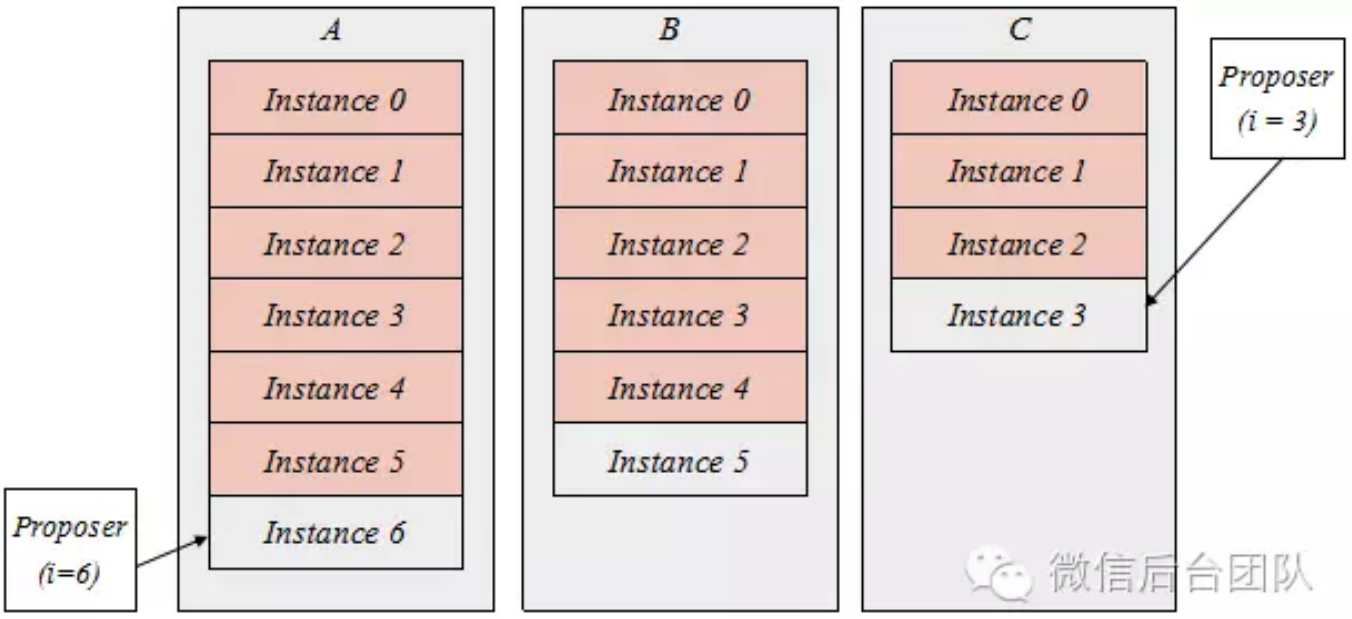
我们需要转换一下切入点，也许我们需要paxos确定的值，并不一定是我们真正看到的数据。我们观察大部分存储系统，如LevelDB，都是以AppendLog的形式，确定一个操作系列，而后需要恢复存储的时候都可以通过这个操作系列来恢复，而这个操作系列，正是确定之后就永远不会被修改的。到这已经很豁然开朗了，只要我们通过paxos完成一个多机一致的有序的操作系列，那么通过这个操作系列的演进，可实现的东西就很有想象空间了，存储服务必然不是问题。

如何利用paxos有序的确定多个值？上文我们知道可以通过运行多个实例来完成确定多个值，但为了达到顺序的效果，需要加强一下约束。

首先给实例一个编号，定义为i，i从0开始，只增不减，由本机器生成，不依赖网络。其次，我们保证一台机器任一时刻只能有一个实例在工作，这时候Proposer往该机器的写请求都会被当前工作的实例受理。最后，当编号为i的实例获知已经确定好一个值之后，这个实例将会被销毁，进而产生一个编号为i+1的实例。

基于这三个约束，每台机器的多个实例都是一个连续递增编号的有序系列，而基于paxos的保证，同一个编号的实例，确定的值都是一致的，那么三台机都获得了一个有序的多个值。

下面结合一个图示来详细说明一下这个运作过程以及存在什么异常情况以及异常情况下的处理方式。



图中A,B,C代表三个机器，红色代表已经被销毁的实例，根据上文约束，最大的实例就是当前正在工作的实例。A机器当前工作的实例编号是6，B机是5，而C机是3。为何会出现这种工作实例不一样的情况？首先解释一下C机的情况，由于paxos只要求多数派存活即可完成一个值的确定，所以假设C出现当机或者消息丢失延迟等，都会使得自己不知道3-5编号的实例已经被确定好值了。而B机比A机落后一个实例，是因为B机刚刚参与完成实例5的值的确定，但是他并不知道这个值被确定了。上面的情况与其说是异常情况，也可以说是正常的情况，因为在分布式环境，发生这种事情是很正常的。

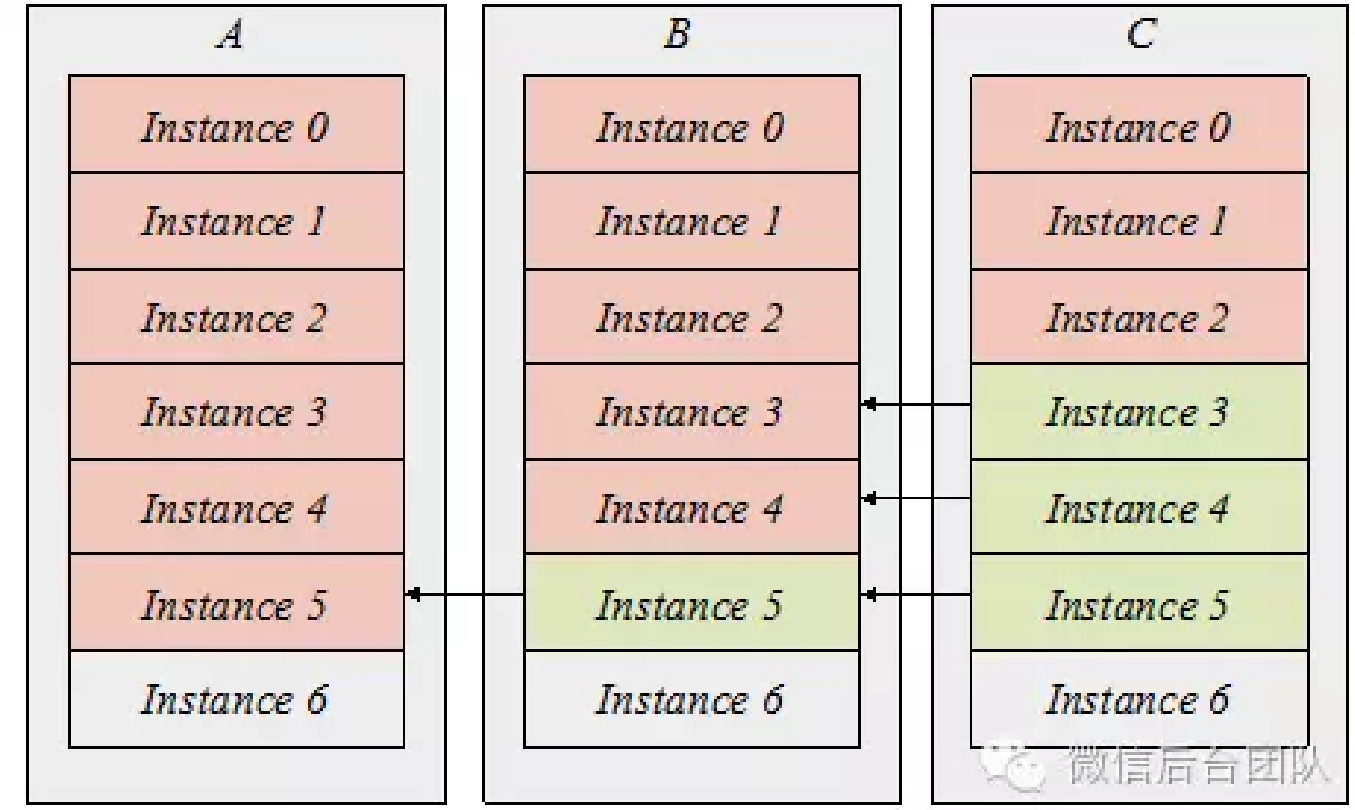
下面分析一下基于图示状态的对于C机的写入是如何工作的。C机实例3处理一个新的写入，根据paxos协议的保证，由于实例3已经确定好一个值了，所以无论写入什么值，都不会改变原来的值，所以这时候C机实例3发起一轮paxos算法的时候就可以获知实例3真正确定的值，从而跳到实例4。但在工程实现上这个事情可以更为简化，上文提到，各个实例是独立，互不干涉的，也就是A机的实例6，B机的实例5都不会去理会C机实例3发出的消息，那么C机实例3这个写入是无法得到多数派响应的，自然无法写入成功。

再分析一下A机的写入，同样实例6无法获得多数派的响应，同样无法写入成功。同样假如B机实例5有写入，也是写入失败的结果，那如何使得能继续写入，实例编号能继续增长呢？这里引出下一个章节。

实例的对齐(Learn)

上文说到每个实例里面都有一个Acceptor的角色，这里再增加一个角色称之为Learner，顾名思义就是找别人学习，她回去询问别的机器的相同编号的实例，如果这个实例已经被销毁了，那说明值已经确定好了，直接把这个值拉回来写到当前实例里面，然后编号增长跳到下一个实例再继续询问，如此反复，直到当前实例编号增长到与其他机器一致。

由于约束里面保证仅当一个实例获知到一个确定的值之后，才能编号增长开始新的实例，那么换句话说，只要编号比当前工作实例小的实例（已销毁的），他的值都是已经确定好的。所以这些值并不需要再通过paxos来确定了，而是直接由Learner直接学习得到即可。



如上图，B机的实例5是直接由Learner从A机学到的，而C机的实例3-5都是从B机学到的，这样大家就全部走到了实例6，这时候实例6接受的写请求就能继续工作下去。

如何应用paxos

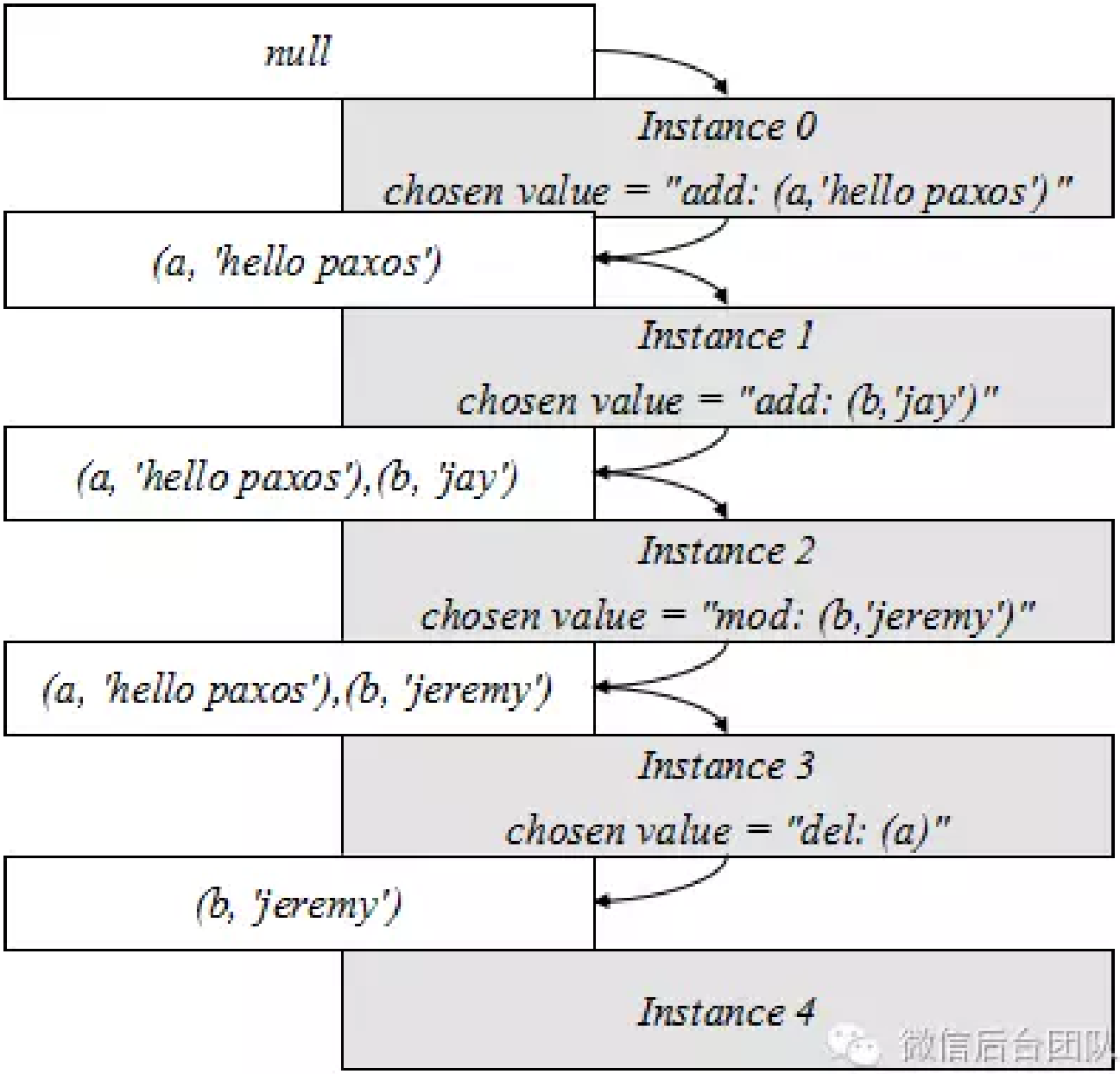
状态机

一个有序的确定的值，也就是日志，可以通过定义日志的语义进行重放的操作，那么这个日志是怎么跟paxos结合起来的呢？我们利用paxos确定有序的多个值这个特点，再加上这里引入的一个状态机的概念，结合起来实现一个真正有工程意义的系统。

状态机这个名词大家都不陌生，一个状态机必然涉及到一个状态转移，而paxos的每个实

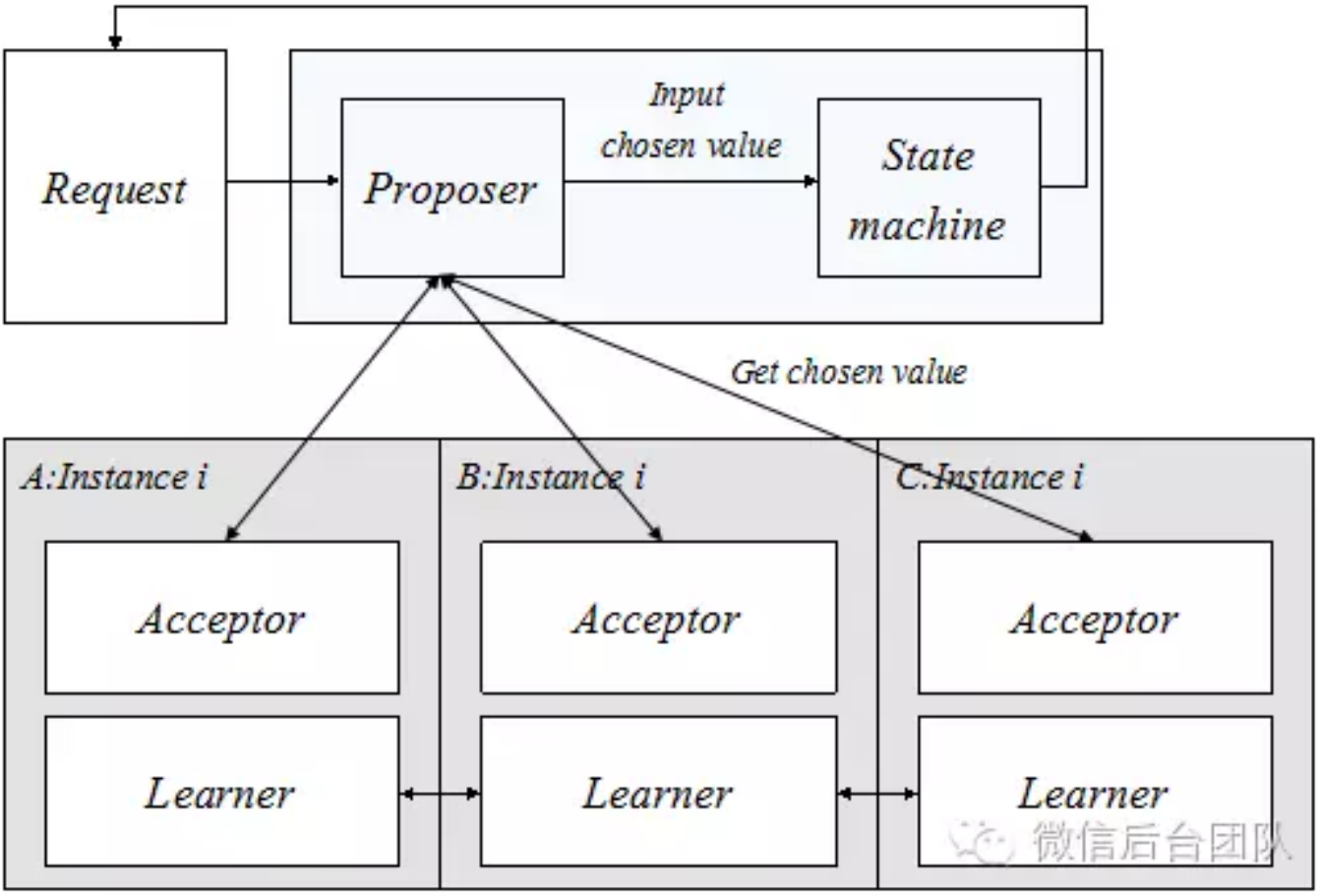
例，就是状态转移的输入，由于每台机器的实例编号都是连续有序增长的，而每个实例确定的值是一样的，那么可以保证的是，各台机器的状态机输入是完全一致的。根据状态机的理论，只要初始状态一致，输入一致，那么引出的最终状态也是一致的。而这个状态，是有无限的想象空间，你可以用来实现非常多的东西。

如下图这个例子是一个状态机结合paxos实现了一个具有多机一致的KV系统。



实例0-3的值都已经被确定，通过这4个值最终引出(b, ‘jeremy’)这个状态，而各台机器实例系列都是一致的，所以大家的状态都一样，虽然引出状态的时间有先后，但确定的实例系列确定的值引出确定的状态。

下图例子告诉大家Proposer，Acceptor，Learner，State machine是如何协同工作的。



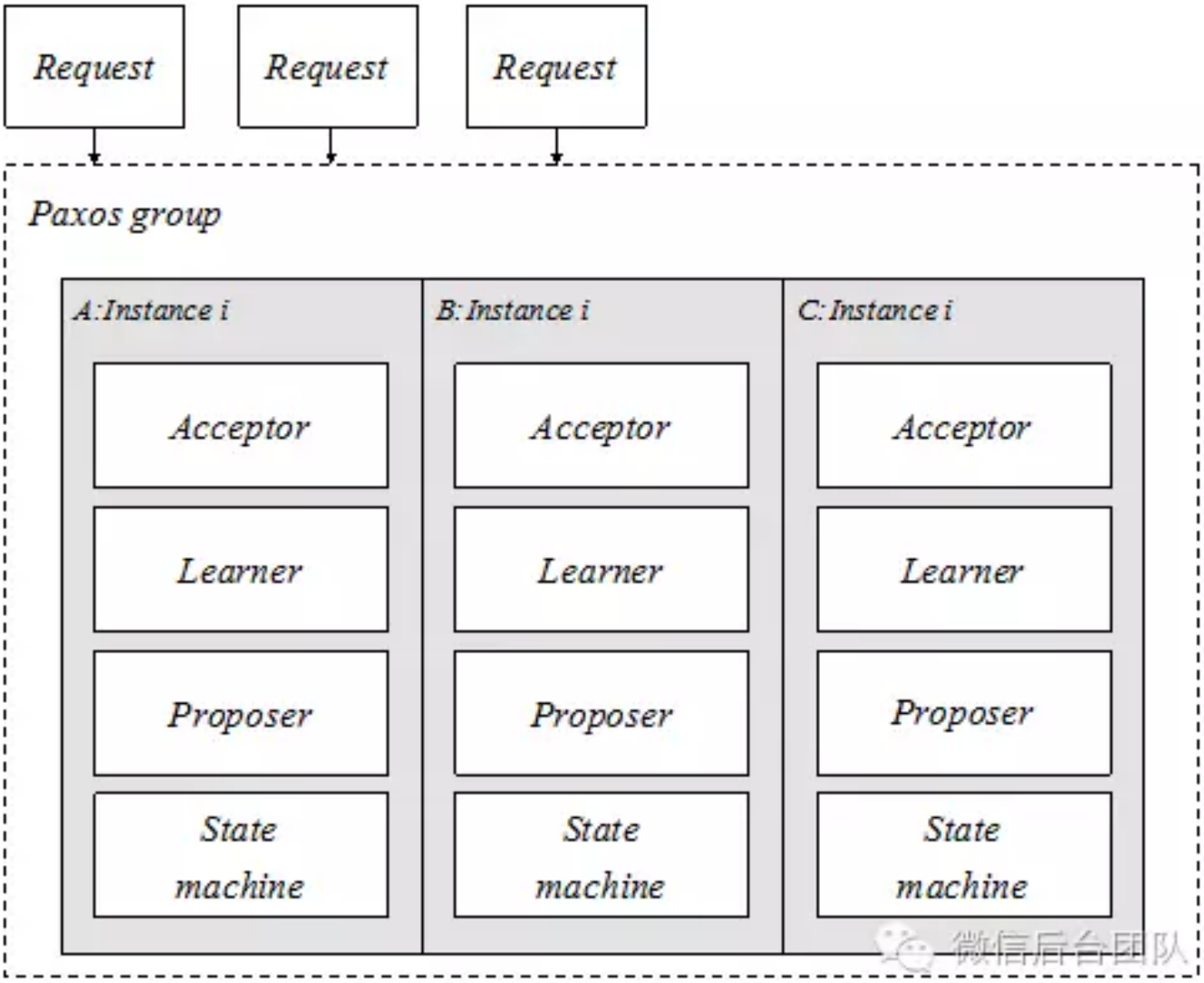
一个请求发给Proposer，Proposer与相同实例编号为x的Acceptor协同工作，共同完成一值的确定，之后将这个值作为状态机的输入，产生状态转移，最终返回状态转移结果给发起请求者。

工程化

我们需要多个角色尽量在一起

上文提到一个实例，需要有Proposer和Acceptor两个角色协同工作，另外还要加以Learner进行辅助，到了应用方面又加入了State machine，这里面势必会有很多状态需要共享。如一个Proposer必须于Acceptor处于相同的实例才能工作，那么Proposer也就必须知道当前工作的实例是什么，又如State machine必须知道实例的chosen value是啥，而chosen value是存储于Acceptor管理的Data数据中的。在概念上，这些角色可以通过任意的通信方式进行状态共享，但真正去实现，我们都会尽量基于简单，高性能出发，一般我们都会将这些角色同时融合在一个机器，一个进程里面。

下图例子是一个工程上比较常规的实现方式。



这里提出一个新的概念，这里三台机器，每台机器运行着相同的实例i，实例里整合了Acceptor，Proposer，Learner，State machine四个角色，三台机器的相同编号实例共同构成了一个paxos group的概念，一个请求只需要灌进paxos group里面就可以了，跟根据paxos的特点，paxos group可以将这个请求可以随意写往任意一个Proposer，由Proposer来进行提交。Paxos group是一个虚设的概念，只是为了方便解释，事实上是请求随意丢到三台机任意一个Proposer就可以了。

那么具体这四个角色是如何工作的呢。首先，由于Acceptor和Proposer在同一个进程里面，那么保证他们处于同一个实例是很简单的事情，其次，当一个值被确认之后，也可以很方便的传送给State machine去进行状态的转移，最后当出现异常状态，实例落后或者收不到其他机器的回应，剩下的事情就交给Learner去解决，就这样一整合，一下事情就变得简单了。

我们需要严格的落盘

Paxos协议的运作工程需要做出很多保证(Promise)，这个意思是我保证了在相同的条件下我一定会做出相同的处理，如何能完成这些保证？众所周知，在计算机里面，一个线程，进

程，甚至机器都可能随时挂掉，而当他再次启动的时候，磁盘是他恢复记忆的方法，在paxos协议运作里面也一样，磁盘是她记录下这些保证条目的介质。

而一般的磁盘写入是有缓冲区的，当机器当机，这些缓冲区仍然未刷到磁盘，那么就会丢失部分数据，导致保证失效，所以在paxos做出这些保证的时候，落盘一定要非常严格，严格的意思是当操作系统告诉我写盘成功，那么无论任何情况都不会丢失。这个我们一般使用fsync来解决问题，也就是每次进行写盘都要附加一个fsync进行保证。

Fsync是一个非常重的操作，也因为这个，paxos最大的瓶颈也是在写盘上，在工程上，我们需要尽量通过各种手段，去减少paxos算法所需要的写盘次数。
万一磁盘fsync之后，仍然丢失或者数据错乱怎么办？这个称之为拜占庭问题，工程上需要一系列的措施检测出这些拜占庭错误，然后选择性的进行数据回滚或者直接丢弃。

我们需要一个Leader

由于看这篇文章的读者未必知道paxos理论上是如何去确定一个值的，这里简单说明一下，paxos一个实例，支持任意多个Proposer同时进行写入，但是最终确定出来一个相同的值，里面是运用了一些类似锁的方法来解决冲突的，而越多的Proposer进行同时写入，冲突的剧烈程度会更高，虽然完全不妨碍最终会确定一个值，但是性能上是比较差的。所以这里需要引入一个Leader的概念。

Leader就是领导者的意思，顾名思义我们希望有一个Proposer的领导者，优先由他来进行写入，那么当在只有一个Proposer在进行写入的情况下，冲突的概率是极小的，这样性能会得到一个飞跃。这里再次重申一下，Leader的引入，不是为了解决一致性问题，而是为了解决性能问题。

由于Leader解决的是性能问题而非一致性问题，即使Leader出错也不会妨碍正确性，所以我们只需要保证大部分情况下只有一个Proposer在工作就行了，而不用去保证绝对的不允许出现两个Proposer或以上同时工作，那么这个通过一些简单的心跳以及租约就可以做到，实现也是非常简单，这里就不展开解释。

我们需要状态机记录下来输入过的最大实例编号

状态机可以是任何东西，可以是kv，可以是mysql的binlog，在paxos实例运行时，我们可以保证时刻与状态机同步，这里同步的意思是指状态机输入到的实例的最大编号和paxos运行当中认为已经确认好值的实例最大编号是一样的，因为当一个实例已经完成值的确认之后，我们必须确保已经输入到状态机并且进行了状态转移，之后我们才能开启新的实例。
但，当机器重启或者进程重启之后，状态机的数据可能会由于自身实现问题，或者磁盘数据丢失而导致回滚，这个我们没办法像上文提到的fsync一样进行这么强的约束，所以提出了

一种方法，状态机必须严格的记得自己输入过的最大实例编号。

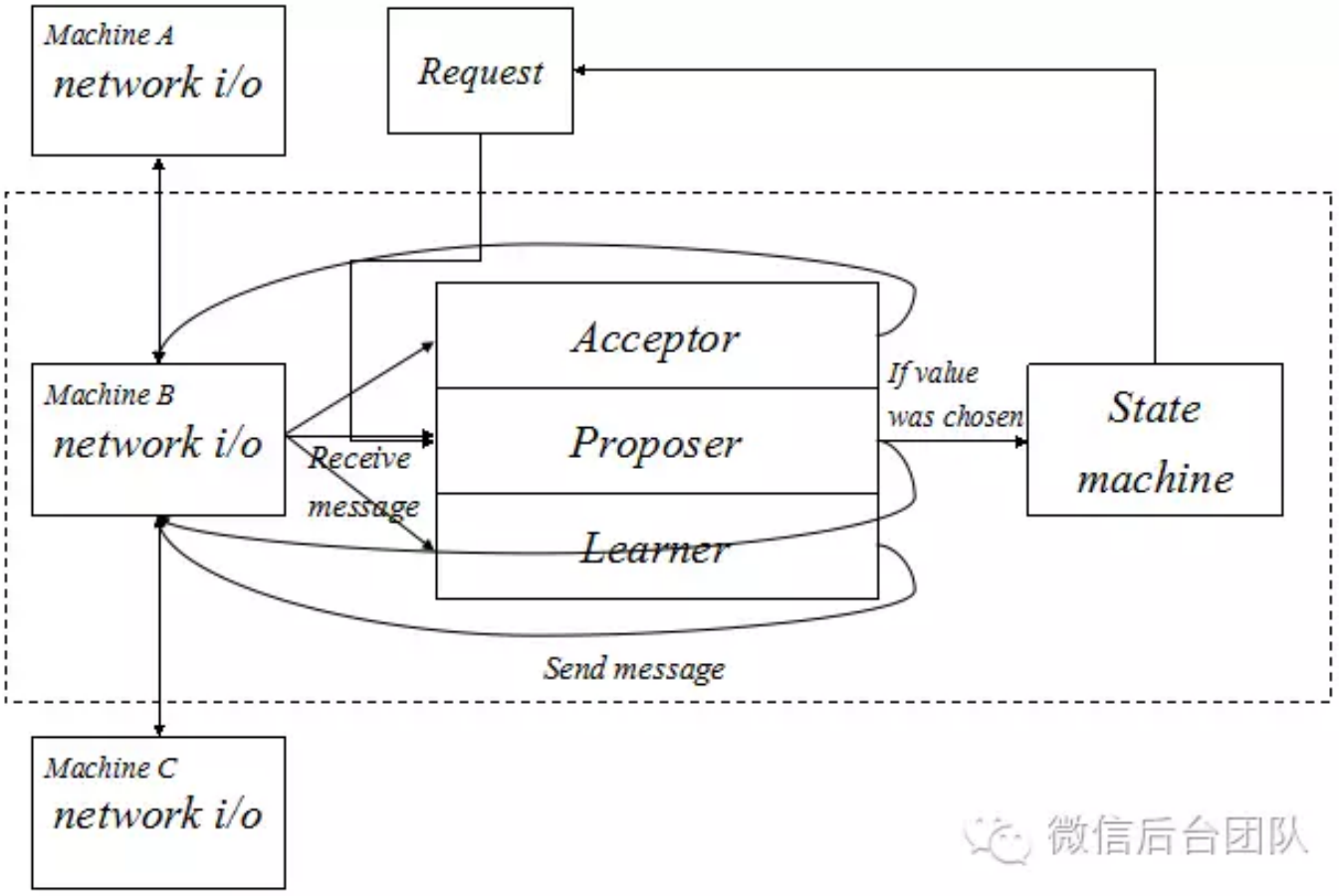
这个记录有什么用？在每次启动的时候，状态机告诉paxos最大的实例编号x，而paxos发现自己最大的已确定值的实例编号是y，而x < y. 那这时候怎么办，只要我们有(x, y]的chosen value，我们重新把这些value一个一个输入到状态机，那么状态机的状态就会更新到y了，这个我们称之为启动重放。

这样对状态机的要求将尽量简单，只需要严格的记录好这么一个编号就可以了。当然不记录，每次从0开始也可以，但这样paxos需要从0开始重放，是一个蠢方法。

异步消息处理模型

上文说到分布式环境是一个异步通信环境，而paxos解决了基于这种环境下的一致性问题，那么一个显而易见的特点就是我们不知道也不确定消息何时到达，是否有序到达，是否到达，我们只需要去遵守paxos协议，严格的处理每一条到达的消息即可，这跟RPC模型比较不一样，paxos的特点是有去无回。

这里先定义一个名词叫paxos消息，这里指的是paxos为了去确定一个值，算法运行过程中需要的通信产生的消息。下图通过一个异步消息处理模型去构建一个响应paxos消息系统，从而完成paxos系统的搭建。



这里分为四个部分：

1. Request，即外部请求，这个请求直接输入到Proposer里面，由Proposer尝试完成一个值的确定。
2. Network i/o，网络i/o处理，负责paxos内部产生的消息的发送与接收，并且只处理paxos消息，采用私有端口，纯异步，各台机器之前的network i/o模块互相通信。
3. Acceptor，Proposer，Learner。用于响应并处理paxos消息。
4. State machine，状态机，实例确定的值(chosen value)的应用者。

工作流程：

1. 收到Request，由Proposer处理，如需要发送paxos消息，则通过network i/o发送。
2. Net work i/o收到paxos消息，根据消息类型选择Acceptor，Proposer，或Leaner处理，如处理后需要发送paxos消息，则通过network i/o发送。
3. Proposer通过paxos消息获知chosen value，则输入value到State machine完成状态转移，最终通知Request转移结果，完成一个请求的处理。
4. 当paxos完成一个值的确认之后，所有当前实例相关角色状态进行清空并初始化进行下一个编号的实例。

生产级的paxos库

RTT与写盘次数的优化

虽然经过我们在工程化上做的诸多要求，我们可以实现出一个基于paxos搭建的，可挂载任意状态机，并且能稳定运行的系统，但性能远远不够。在性能方面需要进行优化，方能上岗。由于上文并未对paxos理论做介绍，这里大概说明一下朴素的paxos算法，确定一个值，在无冲突的情况下，需要两个RTT，以及每台机器的三次写盘。这个性能想象一下在我们在线服务是非常惨烈的。为了达到生产级，最终我们将这个优化成了一个RTT以及每台机器的一次写盘。(2,3)优化到(1,1)，使得我们能真正在线上站稳脚跟。但由于本文的重点仍然不在理论，这里具体优化手段就暂不多做解释。

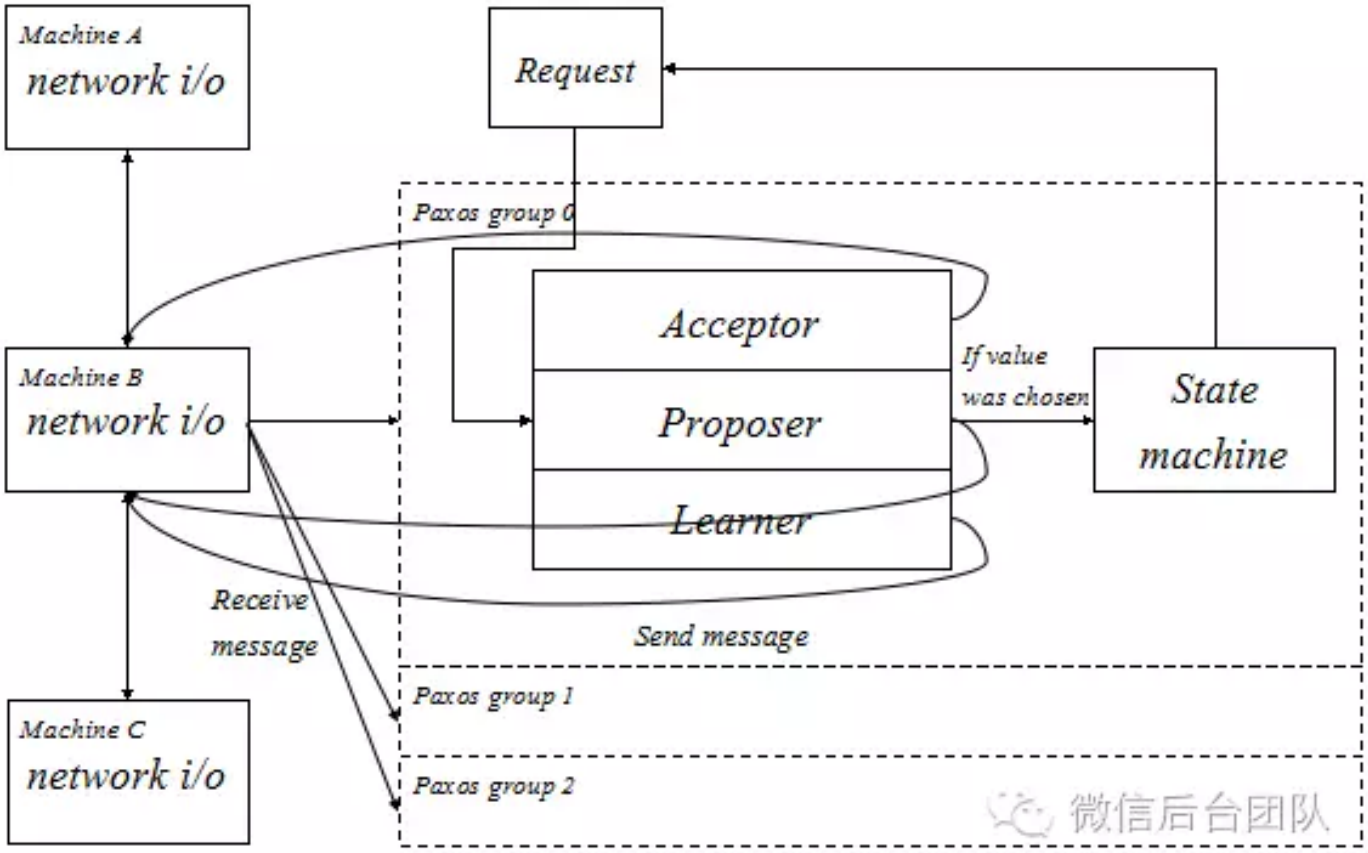
同时运行多个paxos group

由于我们实例运行的方式是确保i实例的销毁才能运行i+1实例，那么这个请求的执行明显是一个串行的过程，这样对cpu的利用是比较低的，我们得想办法将cpu利用率提升上来。

一个paxos group可以完成一个状态机的输入，但如果我们一台机器同时有多个状态机呢？比如我们可以同时利用paxos实现两种业务，每个业务对应一个状态机，互不关联。那么一个paxos group分配一个端口，我们即可在一台机器上运行多个paxos group，各自端口不同，互相独立。那么cpu利用率将能大幅提升。

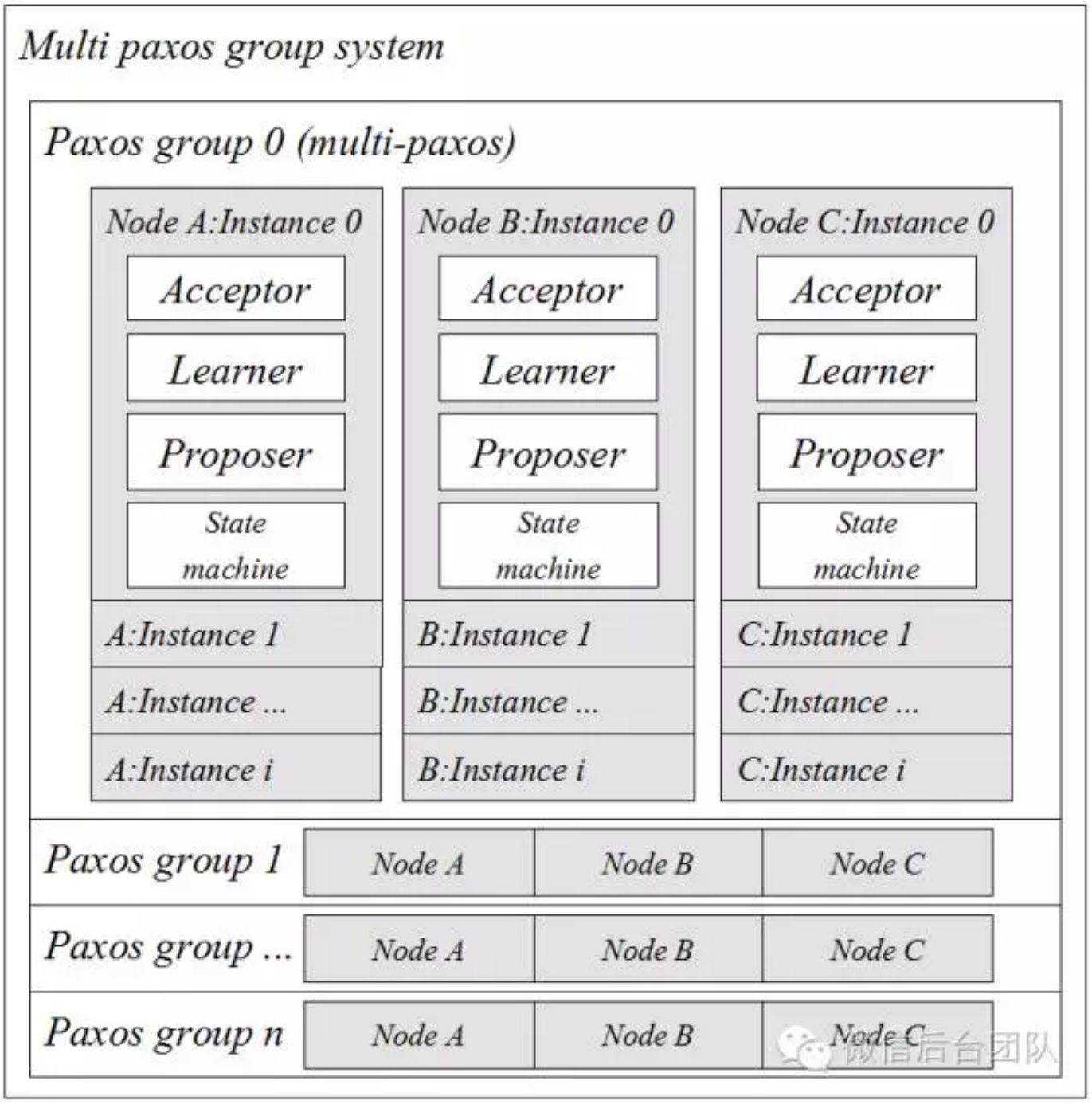
比如我们想实现一个分布式的kv，那么对于一台机器服务的key段，我们可以再在里面分割成多个key段，那每个小key段就是一个独立的状态机，每个状态机搭配一个独立paxos group即可完成同时运行。

但一台机器搞几十个，几百个端口也是比较龌龊的手法，所以我们在生产级的paxos库上，实现了基于一个network i/o搭配多组paxos group的结构。



如上图，每个group里面都有完整的paxos逻辑，只需要给paxos消息增加一个group的标识，通过network i/o的处理，将不同group的消息输送到对应的group里面处理。这样我们一台机器只需要一个私有端口，即可完成多个状态机的并行处理。

至此我们可以获得一个多个paxos group的系统，完整结构如下：



更快的对齐数据

上文说到当各台机器的当前运行实例编号不一致的时候，就需要Learner介入工作来对齐数据了。Learner通过其他机器拉取到当前实例的chosen value，从而跳转到下一编号的实例，如此反复最终将自己的实例编号更新到与其他机器一致。那么这里学习一个实例的网络延时代价是一个RTT。可能这个延迟看起来还不错，但是当新的数据仍然通过一个RTT的代价不断写入的时候，而落后的机器仍然以一个RTT来进行学习，这样会出现很难追上的情况。

这里需要改进，我们可以提前获取差距，批量打包进行学习，比如A机器Learner记录当前实例编号是x，B机器是y，而x < y，那么B机器通过通信获取这个差距，将(x,y]的

chosen value一起打包发送给A机器，A机器进行批量的学习。这是一个很不错的方法。

但仍然不够快，当落后的数据极大，B机器发送数据需要的网络耗时也将变大，那么发送数据的过程中，A机器处于一种空闲状态，由于paxos另外一个瓶颈在于写盘，如果不能利用这段时间来进行写盘，那性能仍然堪忧。我们参考流式传输，采用类似的方法实现Learner的边发边学，B机器源源不断的往A机器输送数据，而A机器只需要收到一个实例最小单元的包体，即可立即解开进行学习并完成写盘。

具体的实现大概是先进行一对一的协商，建立一个Session通道，在Session通道里直接采用直塞的方式无脑发送数据。当然也不是完全的无脑，Session通过心跳机制进行维护，一旦Session断开即停止发送。

如何删除Paxos数据

Paxos数据，即通过paxos确认下来的有序的多个值，后面我们称之这个为paxos log，这些log作为状态机的输入，是一个源源不断的。状态机的状态是有限的，但输入是无限的，但磁盘的空间又是有限的，所以输入必然不能长期保留，我们必须找到方法来把它删除掉。

上文说到我们要求状态机记录下来输入过的最大实例编号，这里定义为Imax，那么每次启动的时候是从这个编号后开始重放paxos log，也就是说小于等于这个编号Imax数据是没用的了，它不会再次使用，可以直接删除掉。但这个想法不够周全，因为paxos是允许少于多数派的机器挂掉的，这个挂掉可能是机器永远离线。而这种情况我们一般是用一台新的机器代替。这台新的机器要干什么？他要从0开始重放paxos log，而这些paxos log从哪里来？肯定是Learner找别的机器拷贝过来的。那别的机器删了怎么办？凉拌。

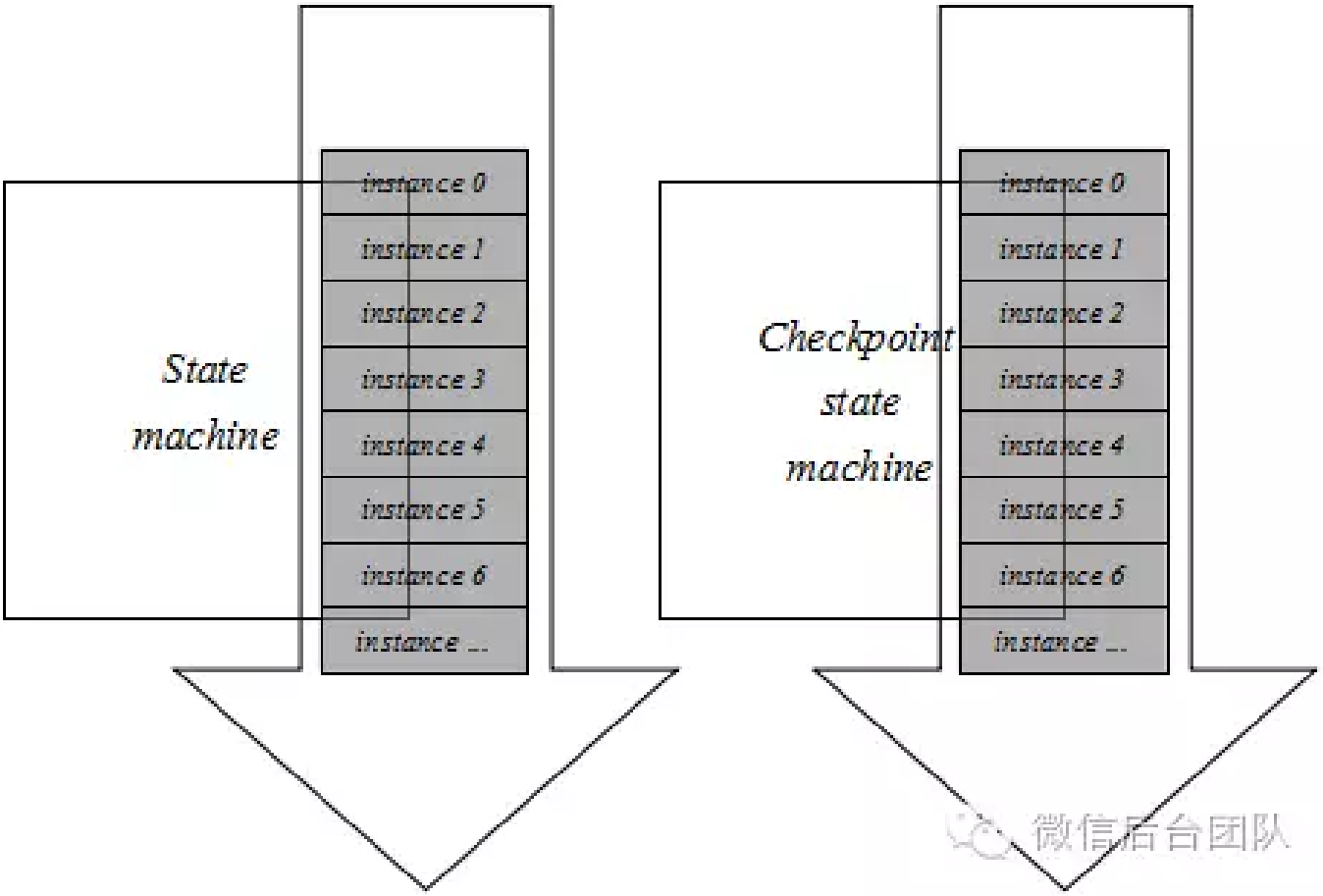
但也并不是没办法了，我可以把这台机状态机相关的数据全部拷贝到新机，然后就可以从Imax来启动了，那么自然就不需要[0,Imax]的paxos log了。但是状态机的数据是无时无刻不在写入的，一个正在写入的数据去拷贝出来，出现什么情况都是不可预期的，所以这个方法并不能简单的实现，什么？停机拷数据？别逗了。但这个思路给了我们一个启示。

我们需要的是一个状态机的镜像数据，这个数据在我们需要去拷贝的时候是可以随时停止写入的，那么只要我们有了这个镜像数据，我们就可以删除paxos log了。

Checkpoint

这个状态机的镜像数据我们就称之为Checkpoint。如何去生成Checkpoint，一个状态机能在不停写的情况下生成一个镜像数据么？答案是不确定的，看你要实现的状态机是什么，有的或许可以并很容易，有的可以但很难，有得可能根本无法实现。那这个问题又抛回给paxos库了，我要想办法去给他生成一个镜像数据，并且由我控制。

一个状态机能构建出一份状态数据，那么搞一个镜像状态机就可以同样构建出一份镜像状态数据了。



如上图，用两个状态转移完全一致的状态机，分别管理不同的状态数据，通过灌入相同的paxos log，最终出来的状态数据是完全一致的。

在真正生产级的paxos库里面，这个特性太为重要了。我们实际实现通过一个异步线程来构建这个镜像数据，而当发现其他机器需要获取这份数据的时候，可以很轻易的停止线程的工作，使得这份数据不再写入。最后发送给别的机器使用。

在目前的实现版本，我们真正做到了删paxos log，新机启动获取checkpoint，数据对齐的完全自动化。也就是说，首先程序会根据磁盘使用情况自动删除paxos log，其次，程序自动的通过镜像状态机生成checkpoint，最后，当一个新机器启动的时候，可以自动的获取到checkpoint，然后通过Learner自动的对齐剩下的数据，从而自动的完成无人工介入的机器更换。

正确性保证

> 分布式算法是很难在工程上去验证他的正确性的，我们只能在工程上利用各种手段去接近正确，这里包括了运行前的测试，运行中的对账，拜占庭问题的细化解决。

模拟异步通信环境

我们对算法内核的构建过程中，使用了内存队列来模拟网络通信，使用一个进程来模拟一个机器。进程通过内存队列来通信。我们对内存队列加以修改，使其支持出队的延迟，丢失，以及乱序，使得整个通信过程能按我们配置的方式来运行。我们通过配置不同的丢失率，延迟时间，以及乱序程度，验证不同参数构造的环境下，paxos的工作效果以及一致性是否得到保证。而我们通过钩子将进程频繁杀掉重启，以及写盘方面的控制，模拟机器当机重启。

运行时的对账

采用crc32算法，对有序的多个值进行累加校验，得到一个当前数据版本的校验值，通过不断的在运行过程中比对每个当前编号实例对应的累加数据校验值，一旦发现机器间校验值不相同，则进行core的处理，防止错误继续扩散。

防止拜占庭问题带来的错误

对于所有磁盘写入的数据，都需要进行二次校验，防止磁盘数据被串改。在发现数据被串改后，能及时的回滚到上一个校验成功的数据，并产生报警。

最后

由于篇幅问题，这里还有更多的有意思的优化，以及更为细节的有趣的问题，就先不做探讨了。相信大家也发现了，这篇文章通篇都在说确定一个值，确定一个值，但就没说到底是怎么去确定一个值的。如果你觉得这篇文章有趣对你有启发，那就去找下论文研究一下paxos到底是怎么确定一个值的吧。

再次提醒大家，点击阅读原文即可到达我们的github开源地址，里面有我们完整的实现源码。

欢迎关注微信后台团队

阅读原文