How to Build a Highly Availability System using Consensus

作者：Butler W.Lampson 1996
原文：http://research.microsoft.com/en-us/um/people/blampson/58-consensus/Acrobat.pdf
译者：phylips@bmy 2012-11-18
译文：http://duanple.blog.163.com/blog/static/709717672012102185013508/

[序：关于这篇文章，其实之前在“A brief history of Consensus, 2PC and Transaction Commit”中有过一些介绍，它最重要的意义在于使得 Paxos 算法为理论研究领域的科学家们所重视，并最终使得 Leslie Lamport 那篇“The Part-Time Parliament”从故纸堆里重见天日。

Butler W.Lampson，计算机科学领域的超级大牛，其名气要甚于 Leslie Lamport，1992 年图灵奖以及 2001 年冯诺依曼奖得主，也是微软研究院的，跟 Leslie Lamport 一个单位。关于 Lampson 更详细的介绍可以参考这篇文章“IEEE 约翰·冯诺依曼奖”。这篇文章初读起来感觉可能要比 Leslie Lamport 重写的那篇“Paxos Made Simple”还要艰深一些，当然了这也是可以理解的，因为这篇文章是写在 1996 年的，在此之前基本没人看懂 Leslie Lamport 那篇“The Part-Time Parliament”，即使看懂了也没有意识到其重要性，而 Lamport 在 2001 年重写的那篇“Paxos Made Simple”实际上也听取了很多来自 Lampson 的建议(但是读完又会感觉它很些地方比 Lamport 写的要好，而且这篇文章所讨论的内容并不局限于 Paxos，只是以 Paxos 为实例，讲述了如何描述、解决、理解、证明分布式算法)。Lamport 本人在回顾 Paxos 的整个曲折发表历程时，这样评价到，“在整个悲催的经历中(指论文一开始被拒，没有人重视)，Butler W.Lampson 是一个例外，他立刻意识到这个算法的重要性，并在他的演讲和一篇论文(即本文)中对该算法进行了描述，这引起了 Nancy Lynch(分布式理论研究大牛，Distributed Algorithms 一书作者)的关注”。此后的 1998 年，Lynch 和 Lampson 还合写了一篇文章“Revisiting the Paxos algorithm”，发表在 1999 年的 Theoretical Computer Science 上，从那个时候开始 Paxos 才逐渐引起理论科学家们的关注，而真正为大众熟知应该是在 Google 发表 Chubby 之后了。

通过这篇文章可以看到大牛是如何理解 Paxos 的，当然除了 Paxos 之外，这篇文章还提到了 Paxos 与租约(Leases)机制的结合，并介绍了一种形式化地描述系统行为的方式。阅读本文需要具有一些关于 safety，liveness，guarded command，prophecy variable，backward simulation，abstract function 的基础知识。
]

摘要

Lamport 展示了可以将复制式(replicated，有时也译作多副本)确定性状态机作为一种实现高可用系统的通用方法，通过给定的一致性算法可以让多个副本在所接受的输入上达成一致。他提出的 Paxos 算法，就是一种可以在没有任何实时性保

证的情况下实现一致性的最容错方式。由于一致性是很昂贵的，实际系统通常只是将它用在紧急情况下，然后使用 Leases 来完成绝大部分的计算。这篇论文介绍了用于高效的高可用计算的通用模式，给出了一个用于理解并行容错程序的通用方法，并将 Paxos 作为实例进行说明。

1.导引

如果一个系统可以快速地提供用户所需要的服务，我们就说它是可用的。在不可靠组件的基础上构建高可用系统的唯一方式就是使用冗余，这样系统就可以在即使有某些组件出错的情况下依然可以正常工作。而最简单的冗余方式就是 replication：为每个部分设置多个拷贝或副本。

本文介绍了如何使用副本来构建高可用系统，同时给出了关键算法的一个细致描述和形式化证明。几乎所有的想法都源自 Leslie Lamport：复制式状态机[5]，Paxos 一致性算法[7]，用于描述和分析并行系统的方法[6]。之所以还要写这样一篇文章，是因为即使是在我读完 Lamport 的论文之后，仍然花了很长时间去理解这些方法以及思考如何有效地使用它们。另外让人惊讶的是，似乎很少有人了解这些方法，尽管它们是如此优雅而有力。

在下一节，我们将介绍下，在给定容错的一致性算法的情况下，如何构建一个高效而且高可用的复制式状态机。第 3 节将给出一些关于一致性问题及其应用的背景知识。第 4 节会介绍下我们用于进行形式化描述的方法，并给出关于一致性问题的几种描述形式。第 5 节我们将会给出用于实现一致性的 Paxos 算法背后的一些基本想法，然后在这些想法和描述的基础之上引出了 Paxos 算法。最后会再介绍一些重要的优化，并进行总结。

2.复制式状态机

单纯地进行简单的冗余是不够的，必须进行协调控制才能发挥作用。实现这一点的最简单方式就是让每个正常工作(non-faulty)的副本做同样的事情。这样的话，所有的正常副本都可以提供输出；如果副本们不是 fail-stop 的{!也就是说即使有副本 fail 掉，也不至于会让整个系统 stop}，这样如果 f 个副本都具有相同输出，就可以容忍其中的 f-1 个出错。更复杂的冗余方式(比如纠错码)可能开销会更少，但是它们会对所提供的服务提出特殊要求。

在这一节我们将介绍下如何使用一种完全通用且高度容错的方式，对各个副本进行协调控制。之后我们还会探讨一下 Leases 机制，通过它可以使协调工作几乎在所有情况下都能非常高效地完成。

2.1 副本协调

我们如何安排才能让每个副本都去做同样的事情呢？采用最早由 Lamport 在[5]中提出的一种模式，我们将每个副本作为一个确定性状态机；这意味着状态的改变本质上是一个从[state,input]到[new state,output]的函数。习惯地，我们将每个副本称为一个进程。从相同的状态开始的几个进程，如果看到的也是相同输入序列的话，那么它们就会完成同样的事情，也就是它们会以相同的状态结束并产生同样的输出结果。用术语来说，就是“consensus”(有时也被称为“agreement”或“reliable broadcast”)。通俗地讲，如果几个进程都在某个值上达成一致我们就认为它们达到了“consensus”；后面我们会给出一个形式化的定义。

因此，如果进程都实现了相同的确定性状态机，在输入的 value 值和顺序上又都达到了一致性的话，它们就会完成同样的事情。通过这种方式，就可以去复制任意的计算，并使它高可用。当然，我们也可以通过定义在输入集合上的某个全序关系，将顺序作为输入值的一部分，比如将它们编号为 1,2,3…

在很多应用中，输入是来自多个客户端的服务请求。比如，一个复制式(多副本)的存储服务，可能会接收到 Read(a)，Write(a,d)这样的输入请求，一个飞行控制系统可能会有 ReadInstrument(i)和 RaiseFlaps(d)这样的一些输入。不同的客户端通常是独立地产生自己的请求，因此就要不仅在请求是什么上达成一致，还要在是以什么样的顺序来为请求提供服务上达成一致。最简单的实现方式就是使用从 1 开始的连续整数来为它们编号。这个工作可以在主副本上完成，因为对于一个进程来说可以很容易地完成连续整数的分配。这样存储服务就可以确定输入的编号，比如：Input 1 = Write(x,3),Input 2 = Read(x)。

还有很多其他的不采用连续整数来实现一致性的方式，具体可以参考 Schneider 的文章[11]。这些方法将每个输入用一个来自某全序集合(比如，(client UID,时间戳)对)中的值进行标记，然后设计一种方式来确保已经看到了那些标号小于某给定值的所有可能出现的输入。这通常都是很复杂的，实际系统通常都是通过选择一个 primary 对输入进行排序来完成。

2.2 租约机制：高效的高可用计算

实现容错一致性的开销昂贵。进程间的互斥访问(即锁机制)是很廉价的，但是并不容错—如果持有锁的那个进程出错，剩余的进程就会没法访问资源。通过为锁添加一个超时机制我们就得到了容错性的锁或者叫“lease”{!关于 Lease 可以参考这篇文章“Leases 一种解决分布式缓存一致性的高效容错机制”}。即在某个超时时间之前，进程会持有针对某个状态组件或者资源的租约；在该进程持有租约期间，我们称它为该资源的“master”。在租约过期之前，任何其他进程无法触碰该资源。当然为了实现这一点，进程必须具有同步化的时钟。准确地说，如果设两个进程的时钟相差 ε，并且进程 P 的租约是在时间 t 过期，那么 P 就能知道在 P 的时钟为 t- ε 之前时，其他任何进程都不会触碰该资源。

在持有租约期间，“master”可以对资源自由地进行读写。写操作花费的时间需要有个上界，这样就可以保证这些操作要么失败要么在租约失效之前完成；对于某些资源来说这可能是个严重问题，比如对于 SCSI 磁盘来说，它对操作只有很弱的顺序性保证，同时写操作可能会花费相当长的时间。

事务处理系统中的锁通常都采用了租约机制；如果它们过期了，那么事务就会被 abort 掉，这就意味着所有的写操作需要被取消，事务相当于被 skip 掉了。如果进程不是在事务的上下文中使用租约，那么就必须要仔细考量应该提供什么程度的原子性{!因为租约到期时，某些动作可能还未执行完毕，在事务上下文中很简单，直接令事务 abort 即可，但是如果没有事务支持，需要仔细考虑对于这种正在发生还未完成的动作如何处理}，比如，可以保证在每个原子性的写之后资源仍处于正确的状态(也称为 “careful writes”--谨慎写)，或者是采用基于日志的标准 redo 或者 undo 方法[4]。如果 leased 的资源本身也具有多个副本的话，那么后一种方法通常都是必需的。

进程可以通过在租约过期之前进行更新来维持对资源的控制。根据需要，它也可以选择释放持有的租约。如果无法与持有租约的进程进行通信，比如它可能已经出错了，那么在触碰该资源前必须要等待租约过期。因此在租约更新开销和等待租约过期需要的时间之间，存在一个权衡。短租约意味着错误恢复时只需等待比较短的时间，但是意味着需要更大的开销去更新租约。长租约错误意味着恢复时需要等待比较长的时间，但是意味着更新租约的开销要小。

租约通常用来授权某进程去缓存某部分状态，比如缓存或者文件的内容。由于租约实际上就是一种锁，因此实际上也可以通过它来决定它的持有者所能进行的操作。如果租约是互斥的，那么持有它的进程就可以自由地改变被 leased 的状态。这就好像具有了针对该缓存的 owner 访问权限，或者是对于一个多端口磁盘的所有权。

2.3 层次化租约

在一个容错系统中，租约的授权和刷新必须通过运行一个一致性算法来实现。如果这种一致性的大量使用会导致开销太高，那么可以采用一个层次化的租约作为解决方案。每次执行一次一致性选出一个独裁者 C，然后授予 C 一个针对大部分状态的租约。现在 C 会负责关于 x 和 y 的子租约的发放，C 会将它们发放给次级的“master”。每个“master”会控制指定给它的那份资源，然后这些“master”会与独裁者 C 进行沟通来刷新租约。这会产生更多的开销，但是这种情况下只有一个独裁者{!这样一致性开销会减少，因为只有在选举该独裁者时才需要一致性}。而且，独裁者可以很简单，同时不太容易出错，这样就可以使用长租约。

层次化租约被广泛地应用在多副本文件系统以及集群中。

通过综合使用一致性，租约和层次化这些想法，就使得构建一个高效的高可用系统成为可能。

3.一致性

一组进程如果可以在我们称之为“outcome”的某个被通过的 value 值上达成一致，我们就说它们达到了一致性(如果允许它们在任意 value 值上达成一致，那么很明显存在一个平凡解：让它们都通过 0)。因此关于一致性的接口需要完成两个动作：通过(allow)一个值，读取输出结果(outcome)。当所有正常工作的进程都获知了输出结果时，一致性算法就结束了。

除了通用的复制式状态机之外，还有其他大量基于一致性的特殊应用。比较有名的比如：

分布式事务，所有进程都需要在事务是提交还是 abort 上达成一致。每个事务都依赖于一个独立的一致性过程的输出结果。

成员管理，一组通过相互协作来提供高可用服务的进程需要对当前哪些进程属于组内成员达成一致。每当进程出错或启动时，都需要一个新的一致性过程。

在不知道成员组成的情况下进行的 leader 选举。

在没有失败发生的情况下，一致性非常简单。比如最简单的实现就是，选择一个固定的 leader 进程，由它获取所有的 allow 动作，选择一个作为最终输出结果，然后将结果告诉每个进程。但是如果这个进程失败，就有问题了。标准的两阶段提交正是以这种方式进行工作的：如果所有的参与者都处于 prepared 状态，结果就是 commit，只要有一个失败就是 abort。如果 leader 本身失败了，输出结果将是未知的。

另一种简单实现是：弄一组进程，每个都可以选一个值。如果出现半数以上的进程选择了同一个值，那么就可以将它作为输出。如果无法出现一个被半数以上的进程选择的值，那么就没有结果。如果在那个半数以上进程组成的集合中，某些进程失败了，那么结果也会变成未知的。

在存在失败的情况下，一致性会变得非常具有挑战性。在一个具有完美链接的异步系统(在这样的系统中，正常工作的进程可能会花费任意的时间来完成一次状态改变)中，即使是只有一个进程出错，也是不存在一个可以保证终止的一致性算法的[3,FLP 不可能性结论]。在同步系统中，即使在进程可能发生任意或恶意错误的情况下(Byzantine agreement)，仍然是可以达到一致性的，只是消息传输和时间开销会很大[9]。

4.Specifications

我们正在研究这样的一个系统，该系统具有状态(某个可能具有无限个状态的状

态空间中的一个元素)，以及可以将该系统从一个状态转换到另一个状态的一个动作(不一定是确定性的)集合。数据抽象、并行程序、分布式系统、容错系统都可以通过这种方式来建模。通常我们采用多个变量的笛卡尔乘积来描述整个状态空间{!比如有变量 x,y,z，每个变量都有一个取值集合，那么整个的状态空间，就是这些变量所有可能取值的组合}。

4.1 如何来描述一个具有状态的系统

在描述这样的一个系统时，我们会指定某些动作和变量为外部的，剩余的一些就是内部的。我们所需要关心的就是那些外部动作组成的序列(或者也可以说是外部变量的 value 值组成的序列)，因为我们假设系统外部的用户是无法观察到系统内部的动作和变量的。我们将这样的一个序列，称为系统的一个“运行轨迹(trace)”。一个 specification 就是一系列运行轨迹组成的集合，或者也可以看成是针对运行轨迹的一个断言。这样的一个集合也称为一个“属性(property)”{! property 实际上是定义了一系列运行轨迹的集合，这些运行轨迹都具有某个相同的特点}。

我们可以定义两种特殊类型的属性：安全性和活性。通俗地说，“安全性(safety)”属性是用来保证坏的事情永远都不会发生；实际上是对串行程序局部正确性的一个推广。“活性(liveness)”则是用来说明最终一定要发生的某些事情；实际上是对可终止性(termination){!某个条件最终一定会满足，不会无限等待下去}的推广。通过查看运行轨迹的有限的一部分前缀我们就可能可以判断出它不具备安全性，但是对于活性来说却无法这样下判断。所有的属性都可以看做是安全性和活性的一个交集[2]{!参考附录 10.1，要证明 Y 实现了 X，实际上是要证明 Y 具有 X 的所有属性，由于所有的属性都可以看做是安全性和活性的交集，那么只需要从活性和安全性下手即可}。

在本文中，我们只考虑安全性。这也是比较合理的，因为根据 FLP 结论我们已经知道对于异步环境的一致性来说，不存在一个可以保证能够终止的算法。这也算是比较幸运的了，因为活性的处理要比安全性困难的多。

通过状态机(该状态机的动作可以被分成外部的和内部的两种)可以很方便的定义出安全性属性。在该状态机上执行的所有外部动作序列就定义了一个安全性属性。注意不要将这里的状态机跟我们用来实现一致性的复制式状态机混淆了。

我们将“系统 Y 实现了另一个系统 X”定义如下：

- Y 的每个运行轨迹也都是 X 的一个运行轨迹；也就是说， X 的安全性属性暗含了 Y 的安全性属性
- Y 的活性暗含了 X 的活性

第一个要求意味着你无法通过观察 Y 来判断出它不是 X；Y 不会做任何 X 不会做的坏事情。第二个意味着 Y 会做任何 X 会做的那些好事情。后面我们不会再讨论活性相关的问题。

如果采用这种方法描述一个具有状态的系统，我们必须首先定义好状态空间，然后描述那些可以执行的动作。状态空间的选择旨在让描述更清晰，而不是为了反映状态的实现。对于每个动作，我们会说明它所作用的状态，以及它是内部的还是外部的。我们会使用参数和结果来描述一个动作，比如 Read(x)代表的是一系列动作而不是一个动作，**返回 3 的 Read(x)**就是其中的一个；该动作发生在客户端读取 x 的时候，并且结果是 3。

下面是一些关于书写这些描述的一些有益建议：

- 记号很重要，因为它可以帮助你了解正在发生的是什么。选择一个合适的词汇。
- 少即是多(Less is more)。动作越少越好。
- 非确定性越多越好，这就允许可以有更多种实现。

I'm sorry I wrote you such a long letter;I didn't have time to write a short one.

Pascal

4.2 Specifying Consensus

现在我们已经可以给出关于一致性的描述了。首先有一个初始化值为 nil 的 outcome 变量，以及一个可以被调用任意次数的动作 Allow(v)。同时还有一个动作 Outcome 用来读取 outcome 变量；它的返回值必须是 nil 或者是某个 Allow 动作的参数值 v，同时它必须总是返回相同的 v 值。

更准确的讲，我们有两个要求：

- 一致性(Agreement)：每个非 nil 的 Outcome 结果必须是相同的
- 合法性(Validity)：一个非 nil 的 outcome 必须是某个被通过(allow)的值

合法性意味着 outcome 值不能是任意值，必须是一个被通过的值。一致性是通过选择某个被通过(allowed)的值，然后将它赋给 outcome 来完成的。根据描述，选择是在被通过的值到达时即时作出的。

下面是关于该描述的一个精确版本，在这里我们称之为 C。在该描述中给出了状态机的状态以及动作。状态如下：

outcome:Value∪{nil} 初始化为 nil

动作如下：

Name	**Guard**	**Effect**
*Allow(v)		**Choose if** outcome=nil **then** outcome :=v **Or skip**
*Outcome		**Choose return** outcome **Or return** nil

在这里，外部动作前面都会用*进行标记。Guard 是一个预先需要判断的条件，对于将要进行动作的当前状态来说它必须是 true 的；对于上面两个动作来说它都是 true 的(直接用空白进行表示)。**Choose or** 表示这是一个非确定性选择，就

像 Dijkstra{!1972 年图灵奖得主，提出过 goto 有害论，信号量和 PV 原语，哲学家就餐问题，最短路径算法和银行家算法，Algo 60 和 THE 操作系统的设计和实现者}提出的“guarded command”{!首先“guarded command”是指命令的执行是被防护的，也就是说在命令之前会有一个需要预先判断的条件，只有在满足该条件的情况下才会执行命令，如果不满足条件会根据上下文来判断做什么，但是都不会执行被防护的命令。具体参考 Guarded_Command_Language，Guarded commands. non-determinacy and a calculus for the derivation of programs。这里的 choose or 即是指用户可以选择其中的一个去执行，选择哪个都是可以的，不需要依据什么条件。Dijkstra 提出 guarded command 的目的就是为了描述非确定性的程序，由于在满足条件的情况下，中间的语句都有可能被执行，这样就可以用相同的程序文本内容来描述不同的程序过程，详细解释参见附录 10.1。}。

需要注意即使是在选择已经做出的情况下，Outcome 也是可以返回 nil 值的。这实际上也反映了实际实现中在具有多个副本情况下，Outcome 通常是通过与其中一个副本进行通信来实现的，而该副本可能还不知道已经做出了选择。

下面我们会给出一个具有终止属性的一致性描述 T。内部的 Terminate 动作发生时，outcome 需要保证不能为 nil。用户可以通过调用 Done 来判断算法是否已经执行完毕。我们使用方框来标记与 C 的不同之处。

State： outcome:Value∪{nil} 初始化为 nil

done:Bool 初始化为 false

Name	**Guard**	**Effect**
*Allow(v)		**Choose if** outcome=nil **then** outcome :=v **Or skip**
*Outcome		**Choose return** outcome **Or if not** done **then return** nil
*Done		Return done
Terminate	outcome != nil	done:=true

需要注意的是，描述 T 并没有陈述任何有关终止是否会真正发生的内容{!也就是说该描述只是说明了终止发生的前提及影响，但是对终止是否发生以及何时发生没有做出任何要求}。对于一个实现来说，如果 Outcome 总是返回 nil 值，那么也是符合该描述 T 的。这看起来不太符合期望，但是这是对于一个异步的实现来说，我们所能达到的最好程度了。换句话说，更强的描述会将异步的实现排除掉{!如果继续对描述进行加强，会导致描述无法将异步的实现囊括在内，上面的描述已经是所能提供的最好的一个了}。

最后，下面我们会给出一个具有“延迟的一致性(deferred consensus)”的更复杂描述 D。它会记录已经被通过的 value 值，同时之后会通过内部动作 Agree 来选

定其中的一个。

State：outcome:Value∪{nil} 初始化为 nil
done:Bool 初始化为 false

allowed：**set** Value 初始化为{}

Name	**Guard**	**Effect**
*Allow(v)		allowed:=allowed∪{v}
*Outcome		**Choose return** outcome **Or if not** done **then return** nil
*Done		Return done
Agree(v)	Outcome=nil **and** v **in** allowed	outcome:=v
Terminate	outcome != nil	done:=true

很明显，D 实现了 T。为了证明这一点，需要使用将在下一节描述的抽象函数方法，但是，还需要采用一个预言变量(prophecy variable)或者是进行逆向模拟(backward simulation)[1,10]，因为 C 和 T 都是只要看到了被通过的那个 value 值就会立即将它作为输出结果(outcome)，但是一个实际实现可能会很晚才做出选择。这里我们给出描述 D 主要有两方面原因。一是它比 T 要更容易让人理解，尽管它具有更多的状态和动作。二是它将对于预言变量(prophecy variable)的需求移到了“D 实现了 T”的证明过程中，因此就大大简化了关于“Paxos 实现了 D”的证明。

5.实现

在本节中，我们首先会解释一下用来说明“某实现满足某描述”的抽象函数方法。该方法很通用而且也很实用。之后，我们会再讨论一下关于进行设计和理解实现的一些秘诀(hints)，同时通过前面给出过的一个简单实现来展示下该方法和这些小窍门的应用。下一节将会展示一下如何用该方法和这些秘诀来引出 Lamport 的 Paxos 一致性算法。

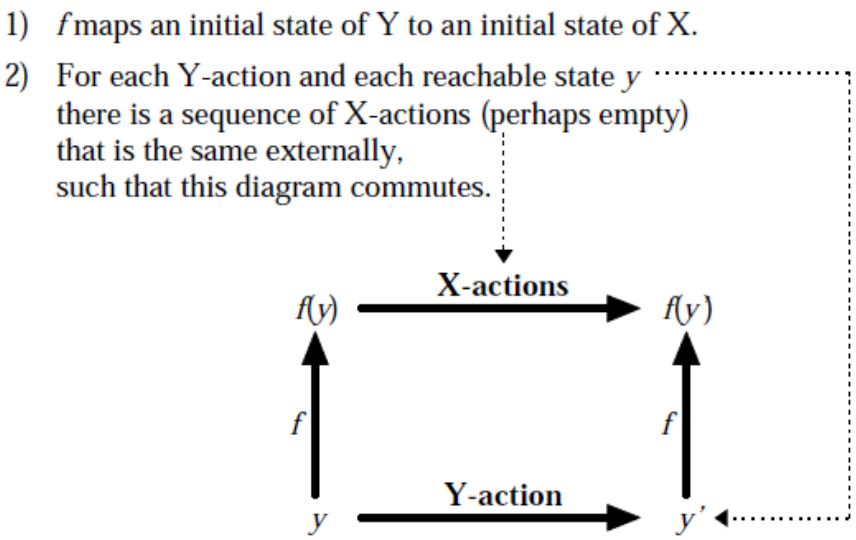
5.1 证明“Y 实现了(implemets)X”

关于“实现了(implemets)”的定义已经明确告诉了我们需要做哪些事情(不考虑 liveness)：证明 Y 的每个运行轨迹也都是属于 X 的。如果通过在纸上简单地写写画画来完成这件事基本是不可能的，因为通常每个运行轨迹是无限长的，而且 Y 有无限多个运行轨迹。因此证明中需要使用归纳法，同时我们希望可以有一种一劳永逸的归纳。幸运地是，恰好存在一种不需要显式地枚举所有的运行轨迹就可

以证明“Y 实现了 X”的通用方法。该方法由 Hoare{! C. A. R. Hoare，著名的快速排序算法的发明者，因其对 Algol60、程序设计语言理论、交互式系统及 APL 的贡献，获得 1980 年图灵奖，也是 2011 年冯诺依曼奖得主}最早发明并用于证明数据抽象的正确性的。之后被 Lamport[6]和其他一些人推广到了任意并行系统中。

该方法的工作过程如下。首先，定义一个从 Y 的状态到 X 的状态的抽象函数 f。然后证明 Y 是对 X 的一个模拟：

1) f 会将 Y 的某个初始状态映射到 X 的某个初始状态
2) 对于每个 Y-action 及每个可达状态 y，都有一系列具有相同外部性的 X-actions(也可能是个空集)与之对应，如图所示：



如果将所有的内部动作排除之后，X-actions 和 Y-action 是一样的那么我们就认为 X-actions 和 Y-action 是外部一致的。也就是说，如果 Y-action 是内部的，那么所有的 X-actions 也必须都是内部的，如果 Y-action 是外部的，那么除了其中那个与 Y-action 相同的之外，其余所有的 X-actions 都必须是内部的。

通过简单的归纳就可以证明“Y 实现了 X”：对于 Y 中的每个行为，我们都可以构造出一个与之具有相同外部性的 X 中的行为，通过使用(2)就可以将每个 Y-action 映射为一系列具有相同外部性的 X-actions。那么 X-actions 序列将会与原始的 Y-actions 具有相同的外部性。

如果“Y 实现了 X”，那么是否总是可以通过这种方法来证明呢。答案是“几乎都可以”。有时可能需要根据一些特定规则在确保修改后的 Y 具有与 Y 完全一致的运行轨迹的前提下，为 Y 添加一些辅助的“history”和“prophecy”变量。通过合理使用 history”和“prophecy”变量，总是可能找到一个合适的抽象函数[1]。与之等价的另外一种方式是，使用抽象关系而不是抽象函数，同时除了进行我们在[10]中描述的正向模拟(forward simulation)外，还要进行逆向模拟(backward simulation)。描述这些内容主要是为了完整性，但是不会在本文对它们展开描述。

为了证明 Y 模拟了 X，我们通常都需要知道 Y 的所有可达状态，并不是每个从 Y 的任意状态开始的动作都模拟了一系列 X 的动作序列；事实上，甚至抽象函数都

不会覆盖 Y 的每个状态。最简单的方式就是使用不变性(invariant)来刻画 Y 的可达状态，所谓不变性就是一个对所有可达状态都是 true 的断言。通常都需要使用一个比模拟需求更强的不变性；强出来的那部分通常都是一个可以保证满足模拟需求的更强的归纳假设。整个的证明通常具有如下类似结构：

- 定义一个抽象函数
- 确立一个用来刻画可达状态的不变性，需要证明每个动作都维持了该不变性
- 建立模拟过程，需要证明每个 Y 动作都是对一系列具有相同外部性的 X 动作序列的一个模拟

该方法只与动作相关，不会涉及与运行轨迹相关的事情。更确切的说，它对每个动作的处理都是独立的。只有不变性会将动作关联起来。因此如果我们修改(或添加)了 Y 里的某个动作，那么我们只需要验证这个新动作维持了不变性，同时也是对一系列具有相同外部性的 X 动作序列的一个模拟就可以了。

下面是一些关于如何利用该方法更好进行实现的产生，理解以及正确性证明的建议：

- 首先写出描述。
- 提出关于实现的想法，这是最关键最具创造性的一步。通常可以将关键的想法体现在抽象函数之中。
- 确保实现中的每个动作都是对描述中的一系列动作的模拟。通过增加不变性来简化这一过程。证明每个动作都维持了不变性。为实现这些目标，这个过程中可能需要不断对实现或描述进行修改。
- 首先保证实现的正确性，然后想办法进行优化。更高效意味着更复杂的不变性。这时也可能需要对描述进行修改以得到更高效的实现。

An efficient program is an exercise in logical brinksmanship.

Dijkstra

下面我们就给出每个实现所对应的抽象函数，以及关于 Paxos 算法的不变性。关于该不变性的证明以及每个 Y 动作都是对一系列 X 动作的模拟，过程很简单，这里我们就不再给出。

5.2 针对那些简单实现的抽象函数

回忆下我们之前提过的那两个简单，非容错的实现。在第一个实现中，有一个 leader 进程，具有与描述完全一致的状态，它会将 outcome 告诉其余所有人(这就是两阶段提交的工作方式)。它针对 C 的抽象函数就是：

outcome=该协调者的 outcome
done =每个节点都拿到了 outcome

它不是容错的，如果 leader 失败了那么整个过程就失败了。

在第二个实现中，具有一组进程集合，每个都会选择一个 value 值。如果半数以上的选择了同一个 value 值，那么这个 value 值就是 outcome。它针对 C 的抽象函数就是：

outcome=半数以上节点选择的那个或者是 nil(不存在一个多数者集合)

done =每个节点都拿到了 outcome

这个也不是容错的，如果大多数都无法达成一致或者达成一致的多数者集合中某个成员出错，整个过程就失败了。

6.Paxos 算法

在这一节，我们会描述下 Lamport 的用于实现一致性的 Paxos 算法[7]。此外 Liskov 和 Oki 独立发明了该算法，在他们的那篇文章中[8]是将它作为多副本数据存储系统中的一部分。作为一个最著名的异步一致性算法，它具有如下重要属性：

- 它运行在多个进程之上，在这些进程里，由一组 leader 进程负责指导一组 agent 进程达到一致性
- 无论同时具有多少个 leader，无论 leader 或 agent 进程发生故障或恢复有多频繁，无论它们有多慢，也无论有多少个消息会被丢失，延迟或重复，它都是正确的
- 如果具有一个存活时间足够长的单个 leader 进程，同时在此期间该 leader 可以与半数以上的 agent 进程完成两次通信，那么算法就可以结束
- 如果总是存在太多的 leader 进程(幸运的是，我们已经知道完全保证它结束是不可能的)，那么它可能不会结束

为了得到一个完整的一致性算法，我们需要将它与一个用来选择单个 leader 的基于超时的算法相结合。如果这个基于超时的 leader 选举算法选不出 leader 或者选出了多个 leader，算法可能不会结束但依旧是安全的。只要该 leader 选举算法曾经在足够长的时间内产生过单一的 leader，那么算法就是可以结束的。

我们首先会介绍下 Paxos 的最简单版本，不需要考虑需要存储的数据量以及需要传递的消息个数，之后再描述一些可以使它更高效的优化方案。为了实现一个真正高效的实际系统，通常都需要配合使用租约机制。

6.1 The Idea

首先我们回顾一下上面描述的框架。有一组 agent 进程，我们用集合 I 表示。每个 agent 的行为是确定性的；每个 agent 会完成它所被告知需要做的事情。每个 agent 都具有应对 crash 的持久化存储。在算法的每次运行过程中，agent 集合是固定的(尽管它可以通过使用 Paxos 算法本身进行改变)。此外还有一组 leader 进程，我们用一个全序集合 L 表示，它们负责通知 agent 需要做什么事情。Leader 可以自由的加入或退出，它们不是确定性的，同时也没有持久化的存储。

Paxos 的核心思想源自前面描述的那个非容错的基于 majority 的一致性算法。如果 agent 无法在一个多数集上达成一致，或者是多数集中的某些成员发生故障，这样剩余的成员就无法确定是否已经达到了一致性，这样这个算法就会有问题。为了解决这个问题，Paxos 会运行多轮(rounds)，每一轮采用集合 N 的一个数字来表示。在第 n 轮，会有一个 leader 来尝试为 value 值 vn 来获取一个多数集。如果某轮出了问题，另一轮会重新开始。如果在第 n 轮里，找到了一个针对 vn

的多数集，那么 vn 就是 outcome。

很明显如果这样可以工作，还需要保证那些成功获取到多数集的任意两轮必须具有相同的 value 值。Paxos 算法的关键就在于保证这个属性。

在每一轮，leader 会：
- 询问所有 agent，以了解它们过去所有轮的状态
- 选择一个 value 值，并通知 agent，尽量让大多数都通过它
- 如果成功的让大多数通过了它，那么就将这个 value 值发布给每个节点

对于一个成功的 round 来说，这需要总共 2.5 个来回。如果 leader 重复地失败，或者同时有几个 leader，那么可能会花很多轮才能达到一致。{!注意此处的角色划分还没有像 Paxos Mode Simple 里那样清晰，基本上这里的 agent 就相当于 Paxos Mode Simple 里的 acceptor，leader 则相当于 proposer ，但是与该文相比，Paxos Mode Simple 却去掉了 round 这个概念，而是以编号代之，但是显然 round 更具体，更有助于算法的理解}

6.2 状态和抽象函数

Agent 的状态就是一个针对每一轮的持久化变量“status”；持久化意味着它不会受 agent 故障的影响。状态要么是一个 Value 值，要么是特殊符号 no 和 neutral 中的某个。Agent 的动作定义保证只有在它的状态为 neutral 的情况下才会改变状态。因此 agent 的状态可以通过如下数组 s 来定义：

State：si,n:Value∪{no,neutral} 初始化为 neutral

如果半数以上的 agent 状态都是 no{!如果没有理解轮的概念，而且又受 Paxos Made Simple 影响比较深的话，可能会有这样的疑问：“如果都变成了 no，那么以后如何再变成 value 值？因为根据上述，只有在状态为 neutral 情况下才可以改变”。实际上是这样的：观察 si,n 可以看到该值是与轮数相关的，也就是说 agent 在每一轮都会有一个状态值 si,n，比如它在第 n 轮状态为 no，那么只是说第 n 轮的状态 si,n=no 且不会再改变，但是它在第 n+1 轮的值是 si,n+1 其初始化值为 neutral，然后就又可以重新被赋值了}，那么这一轮就是 dead 的，如果半数以上的 agent 状态都是一个 Value 值，那么这一轮就是成功的。

Leader 的状态有三个：它当前工作在哪一轮(如果它没有工作在任何轮上，那么就是 nil)，该轮的 value 值(如果还没有选择就是 nil)，以及 leader 本身所已知的通过的 value 值。

State：nl:N∪{nil}初始化为 nil
u1: Value∪{nil}初始化为 nil
allowedl:**set** Value 初始化为{}

关于 allowed 的抽象函数实际上就是所有 leader 的 allowedl 的并集：

AF：allowed=∪allowedl

我们将第 n 轮的 value 值定义如下：

vn:如果某个 agent i 的 si,n 具有某个 Value 值，那么 vn 的值就是 si,n，否则就

是 nil。
为保证这都是定义良好的，我们必须保证如下不变性：
Invariant 1： 每轮最多具有一个 value
也就是说，在给定的某轮里，所有具有某个 value 的 agent 的 value 都是相同的。

现在我们可以给出 outcome 的抽象函数了：
AF： outcome=对于每个成功的轮 n 来说，就是 vn；否则如果没有成功的轮，就是 nil
为保证这是定义良好的，我们必须保证如下不变性：
Invariant 2： 任意成功的两轮具有相同的 value 值

我们通过保证每个 leader 每次只会工作在一个 round 上并且永远都不会重用一个 round 编号，以及每轮最多具有一个 leader，来维持不变性 1(一轮最多具有一个 value 值)。为了保证后面这个条件成立，我们需要使用 leader 的标识符作为 round 编号的一部分，比如使用(sequence number,leader identity)作为 round 编号。因此 N={J,L}，这里 J 是某个全序集合，通常都是整数，Leader l 选择(j,l)作为 nl 的值，j 是集合 J 中还未被 leader l 使用过的一个元素。比如，j 可能是逻辑时钟的当前值。后面，我们会展示一下如何选择 j 并避免在 leader 上使用任何可靠性存储。

6.3 不变性

我们引入针对状态的稳定性断言的概念{!注意理解稳定性断言的概念，这对于理解算法也非常重要，不幸的是该概念在 Paxos Made Simple 也被省略了}，这是一种一旦为 true，之后就永远为 true 的断言。这很重要，因为通过稳定性断言可以保证安全性。除此之外，任何东西都可能因为并发或 crash 而改变。

由于 si,n 的一个非 neutral 的 value 值不能被改变，那么如下断言就是稳定的：

vi,n=no
vi,n=v
vn=v
n 是 dead 的
n 是成功的

{!vi,n 就是 agent 关于第 n 轮的状态，no 代表不通过这一轮的 value 值，v 代表通过这一轮的 value 值，vn 就是 leader 在第 n 轮上提出的值，如果对于第 n 轮，有半数以上的 agent 都给出了 no，那么这一轮就是 dead 的，如果半数以上的 agent 都接受了该 value 值，那么这一轮就是成功的，而这一切都因为“si,n 的一个非 neutral 的 value 值不能被改变”，而导致它们都是稳定性断言，即一旦成立就永远成立}
下面是一个更复杂的稳定性断言：

n 是 anchored(anchored 是指对于所有的 m<=n，要么 m 是 dead 的要么 vn=vm)

{!可以将此处的“n 是 anchored”与 Paxos Made Simple 里的 P2b“如果具有 value 值 v 的提案被选定，那么所有比它编号更高的被 proposer 提出的提案的 value 值也必须是 v”进行对比，P2b 类似于下面的 **Invariant 3**}
换句话说，当且仅当你回头看在 n 之前的所有轮，如果跳过那些 dead 的轮，看

到的都是与 n 相同的 value 值时，我们才认为“n 是 anchored”。如果之前的所有轮都是 dead 的，那么无论第 n 轮的 value 是多少，n 都是 anchored。为了保证这是定义良好的，我们需要定义一个在 N 上的全序关系，这里我们采用字典序。

现在，我们所需要做的就是在维持 **Invariant 2** 的同时，保证会向着一个成功的 round 推进。为了说明如何来维持该不变性，我们会对它逐步加强直到得到一个可以简单地通过分布式算法就能维护的形式为止，也就是说，对于所有的动作来说，每个都只需要使用进程的本地状态即可。{!观察这个过程，可以发现 Lamport 在 Paxos Made Simple 里的讲述思路跟这里如出一辙，只是由于在这里引入了状态机、抽象函数等，显得更加形式化一些，但是实际上这里却是要更具体一些，因为 Paxos Made Simple 里给出的已经是一个优化过的版本了，而本文是先从最原始的无任何优化的版本讲起，然后再指出哪些地方可以优化}
Invariant 2： 任意成功的两个 round 具有相同的 value 值
该不变性可以通过如下不变性来保证：
Invariant 3： 对于所有的 n 和 m<=n，如果 m 是成功的，那么 vn=nil 或者 vn=vm
该不变性又可以通过如下不变性来保证：
Invariant 4：对于所有的 n 和 m<=n，如果 m 是非 dead 的，那么 vn=nil 或者 vn=vm
然后我们可以通过谓词演算对不变性 4 做如下推导：
=对于所有的 n 和 m<=n，m 是 dead 的，或者 vn=vil 或者 vn=vm
=对于所有的 n，vn=nil 或者(对于所有的 m<=n，m 是 dead 的或者 vn=vm)
=对于所有的 n，vn=nil 或者 n 是 anchored
这样最终我们只需要保证所选择的每个非 nil 的 vn，满足如下条件即可：
只有唯一的一个，并且 n 是 anchored
这样算法的剩余部分就显而易见了。

{!关于上面的 **Invariant 3** 和 **Invariant 4**，需要知道这样一个事实，dead 状态是指半数以上的 agent 的状态为 no，而成功状态则是指半数以上的 agent 的状态为某 Value。但是非 dead，只是说没有满足半数以上的 agent 的状态为 no 这一条件，并不意味着就一定是成功状态，所以二者是有区别的，但是成功属于非 dead，所以如果 3 成立就可以保证 2 成立}

6.4 算法

Leader 在为某一轮选举 value 值时，需要保证该轮是 anchored 的。为实现这一点，leader l 会选择一个新的 nl 值，同时询问所有的 agent 获取它们所有编号小于 nl 的 round 的状态(以及这些 round 的 value，即 round 编号)。Agent 在返回响应消息之前，会将它内部那些早于 nl 的且处于 neutral 状态的 round 改为 no，这样 leader 就有足够的信息 anchor 该 round。来自半数以上的 agent 的关于该查询的响应会提供给 leader 足够信息，使得 leader 可以让 nl 轮处于 anchored 状态，如下：

Leader 会从 nl 开始往回查看，并跳过那些报告中没有 Value 值的轮，因为这些轮肯定是 dead 的(因为 l 以及收到了半数以上的 agent 的响应，报告值要么是 Value 值要么是 no，如果没有收到 Value 值的报告，说明这半数以上的 agent 报告的都是 no，因此该轮肯定是 dead 的)。当 l 碰到一个具有 Value 值状态的

agent 的轮 n 时，它会选择该轮的 value 值 vn 作为 ul。因为根据 **Invariant 4**，n 是 anchored，并且所有从 n 到 nl 之间的轮都是 dead 的，那么只要将 ul 作为它的 value 值，就可以保证 nl 也是 anchored{!这里其实有用归纳法，先假设对于 n 该不变性成立，然后看如何做才能让 nl 维持该不变性}。

如果之前的所有轮都是 dead 的，那么 leader 可以为 ul 选择任意被允许的值。在这种情况下，nl 也是 anchored 的。{!观察上面的过程，读者可能会想，根据 anchored 定义，管它 m 是 dead，还是非 dead 的，只要都让它保证 vn=vm 就可以确保 n 是 anchored，干嘛还要费那么大劲去判断 m 是否是 dead 的？这想法实际上对应了 Paxos Made Simple 里所说的那个平凡解，就是让所有轮都提出相同的一个 value 值，那么这个系统就只能通过一个 value 值，而且每一轮有可能是由不同的 leader 发起的，这样也就不允许它们提出不同的 value 值，因此会有很大的局限性。通过判断前面的某些轮是否是 dead 的，就给了后面的那些可以提出不同的 value 值的机会。**总的来说，Paxos 算法的本质就是：为了避免 SPOF(Single Point of Failure)，必须让它基于 majority，但是这样有可能无法一次完成决议，那么就让它可以发起多轮决议，但是为确保安全性，必须要保证任意成功的两轮都具有相同的 value 值。为保证这一点，就在发起新的一轮时查看一下之前那些轮的状态，很明显的做法就是如果发现已经有成功的轮了，那么就让这一轮提出一个与之相同的 value 值。但是这里有一个问题，在查看之前的那些轮的结果时，可能由于消息的延迟等问题，之前这些轮的结果可能目前还未确定，预测未来是很困难的，因此干脆让 agent 直接把之前那些轮的状态强制置为 no，然后为了容错以及确保可以获知已经被选定的 value 值，不要求收到所有 agent 的响应，只要收到半数以上的响应即可，如果通过这些响应可以判断出之前的那些轮肯定都是 dead 的，ok 那么我就可以不管它们，自己任意提出一个新的 value 值，但是如果不确定是否是 dead 的(只要不满足半数以上 agent 返回 no 这个条件)，那么我就保守点，还是提出一个与前面这些轮相同的 value 值。只要每次都是按照这个原则进行选取，那么就可以保证安全性。这中间其实是一个关于 value 值选取的灵活性与安全性上的权衡：如果极端点为了安全，可以让所有轮都提出相同的 value 值，这就是我们前面提到的平凡解；对于 Paxos 来说，它的做法是只有在可以确定换 value 值是安全的情况下才换，如果不确定就坚决不换}**

因为“anchored”和“dead”都是稳定的属性，因此任何状态的改变都不会使得该选择变得不合法。

在接下来的第二个来回中，leader 会命令每个 agent 为轮 nl 接受 ul。那些在 nl 轮仍处于 neutral 状态的 agent(因为它还没有为更新的轮的查询做出响应，如果做出了响应，它会将 neutral 改为 no)会通过将它们的在轮 nl 的状态修改为 ul 来接受 ul；在任何情况下它都会将状态报告给 leader。如果 leader 收到了来自半数以上的 agent 的报告都是 ul，那么它就知道 nl 轮已经是成功的了，同时会将 ul 作为该算法最终通过的 outcome 值，同时在最后的半轮中将该事实发送给所有进程。因此整个过程使用了 5 条消息，或者说是 2.5 个来回。

需要注意的是只要某些 agent 通过了某个 value 值并形成了一个多数集，那么该轮就成功了(描述中的 Agree 动作发生了，并且 outcome 发生了变化)，即使是此时还没有 agent 或者 leader 知道这已经发生了。实际上，某个 round 是有可能在没有 leader 知道下的情况下成功的，比如某个 agent 可能会在通过了某个值后但还未向 leader 报告之前就出错了，或者是 leader 本身出错了。

可以通过一个实例来帮助理解下该算法的工作过程。下表展示了在具有三个元素 a,b,c 的 agent 集合，可选 value 集合为{7,8,9}的情况下，该算法两次不同运行过程中的三个 round。左边的那次运行中，三个 round 都是 dead 的，因此如果 leader 收到所有三个 agent 的响应，那么它就能了解到这个情况，并且可以自由地选择任意的允许值。如果 leader 仅收到了来自 a 和 b 或者是 a 和 c 的响应，那么它最多只知道 round 3 是 dead 的(如果是 a 和 b)，但是它不知道 round 2 是 dead 的，因此必须选择 8.如果它只收到了来自 b 和 c 的响应，那么它就无法知道 round 3 是 dead 的，因此必须选择 9。

在右边的那次运行中，无论 leader 收到来自哪些 agent 的响应，它都不会认为 round 2 是 dead 的。事实上它是成功的，但是除非 leader 会收到来自 a 和 c 的响应，否则它也是不知道这个事实的。但是，它必须选择 9，因为该 value 值是它所能从最新的非 dead 的 round 里看到唯一 value 值。因此一个成功的 round，就像 round 2，实际上扮演了一个可以防止后续的 round 选择另一个不同的 value 值的保护栅。

	Status							
	v_n	$s_{a,n}$	$s_{b,n}$	$s_{c,n}$	v_n	$s_{a,n}$	$s_{b,n}$	$s_{c,n}$
Round 1	7	7	no	no	8	8	no	no
Round 2	8	8	no	no	9	9	no	9
Round 3	9	no	no	9	9	no	no	9
Leader's choices for round 4	7, 8, or 9 if *a, b, c* report 8 if *a, b* or *b, c* report 9 if *b, c* report				9 no matter what majority reports			

由于在这两次运行中都包含了三个 round，这意味着至少有两个不同的 leader 参与了该过程，或者是在每轮被接受前 leader 出错了。否则第一轮就应该成功了。如果多个 leader 相互干扰，并且在一个更早的 round 到达命令阶段前就强制 agent 将它们针对该 round 的状态设为 no，那么该算法就可能无限地运行下去。

下面是针对该算法的细节描述，图中包含了它所涉及的所有动作，同时还包含了与消息发送和接收相关的动作。

Leader l	Message	Agent i
Choose a new n_l		
Query a majority of agents for their status	query(n_l) →	for all $m < n_l$, **if** $s_{i,m} = neutral$ **then** $s_{i,m} := no$
	← report(i, s_i)	
Choose u_l to keep n_l anchored. If all $m < n_l$ are dead, choose any v in $allowed_l$		
Command a majority of agents to accept u_l	command(n_l, u_l) →	**if** $s_{i,n_l} = neutral$ **then** $s_{i,n_l} := u_l$
	← report(i, n_l, s_{i,n_l})	
If a majority accepts, publish the outcome u_l	outcome(u_l) →	

该算法最小化了对网络各种属性的需求：允许消息的丢失，重复和乱序。同时由于节点可能会发生故障以及进行恢复，就算网络状况再好也不会让事情变得简单。从模型上，我们认为网络可以提供从 leader 到所有 agent 的广播通信；实际中，通常可能是通过挨个地给每个 agent 发送消息来实现的。leader 和 agent 都可以根据它们的需要进行任意频度的重传；实际中 agent 只会对它们给 leader 的响应进行重传。

一个完整的证明需要为进程间的通信信道建立一个消息可以丢失，重复的模型，然后证明关于通信信道如下形式的不变性“如果一个消息 report(i,s)已经在信道中了，那么除了那些处于 neutral 状态的轮之外，si 的值会与 s 值保持一致”。{! 首先我们需要知道 si 代表的内容，它实际表示的是 agent i 的状态，根据前面所述及第 7 节优化部分的陈述，它实际包含了 agent 目前关于所有轮的状态信息，即 set(Si,n)，可以看做是一个无限集合，尽管某些 n 目前还未被用到，但是可以认为它对应的 Si,n 值为 neutral。上面的不变性即假设信道中已经存在了一个 report(i,s)消息，然后现在又发送了一个 report(i,si)消息，那么它们关于各轮的状态，除了那些处于 neutral 状态的轮之外，剩余的轮的状态应该是一致的}

6.5 Termination：选择一个 leader

算法什么时候会结束？如果在现有的 round 成功之前不会有一个新的 round 启动，那么只要 leader 可以跟半数以上的 agent 成功完成查询和命令过程，那么该算法最终一定会结束。同时对于查询和命令这两个过程来说，不需要这半数以上的 agent 集合是同一个，与此同时也不需要这些 agent 都必须同时处于正常服务状

态。因此我们希望可以只有一个 leader，并且它每次只运行一个 round。如果同时有几个 leader，那么那个运行最大的 round(round 是全序的)的 leader 最终会胜出，但是如果新的 leader 总是在启动更大的 round，那么可能没有一个最终会成功。在上面的例子中，我们已经观察到过该行为。幸运的是，我们通过 FLP 结论[3]已经知道了，根本不存在一个可以保证结束的算法。

如果一段时间内没有故障发生，而且进程具有时钟，同时消息发送接收以及处理在通常情况下耗费的最大时间是已知的，那么就可以很容易地防止同时存在两个 leader 进程，如下：

每个正常工作的潜在 leader 广播自己的名称。

在进行广播之后的一个来回时间后，如果没有收到比自己更大的名称，那么就让自己成为 leader。

当然如果消息被延迟或者进程对该消息响应地比较晚，该算法可能会出错。如果它出错了，那么某段时间内就可能同时存在两个 leader。那个运行最大 round 的将会成功，除非未来又出现了一些问题导致了另一个 leader 的产生。

7.优化

没有必要将 agent 的全部状态进行保存和传输。实际上，只需要很少的 bit 就可以将所有必要信息进行编码。对于 si 来说，实际上真正重要的部分就是最新的 Value 值以及后续的 no 状态：

Si,lasti=v

Si,m=no，lasti<m<nexti

Si,m=neutral，m>=nexti

我们可以将其编码为(v,lasti,nexti)。这就是一个 agent 所需要保存和报告的所有内容。

类似地，一个 leader 需要从报告中记住所有东西是：有 value 值被报告的最大的 round 以及是由哪个 agent 报告的。简单地对报告进行计数(计数是看是否达到了半数以上)是不够的，因为消息可能会因为重传而重复。

Leader 不一定与 agent 是相同的进程，尽管它们可以是而且通常都是相同进程。Leader 实际上也不需要持久化的存储，尽管在算法中看起来它好像需要一些信息以保证它可以选出一个未被使用过的值 n。实际上它可以在发生故障后，通过向半数以上的 agent 询问它们的 nexti 值，然后就可以选出一个具有更大 j 的 nl 值{!回顾下 6.2 节，nl 值实际上由(j,l)组成的，j 是 leader l 之前从未用过的一个值，l 是 leader l 的标识符}。这样就可以产生一个比所有目前为止出现在该 leader 的命令消息中的 nl 值都大的 nl 值(命令消息即上图中的 Command(nl,ul)，很明显如果 nl 出现在了命令消息中，那么说明它已经被半数以上的 agent 收到过了，那么这半数以上的 agent 的 nexti 值肯定大于等于 nl)，这就是我们所需要的。

如果将 Paxos 用于一个非阻塞提交协议中来实现一致性，那么它的第一个回合

(query/report)实际上是可以合并到 prepare 消息及其响应中的。

最重要的优化实际上是针对一系列一致性问题的场景，通常是一个复制式状态机中的连续步骤。我们会尽量固定一个 leader，通常称之为“primary”，然后运行一系列通过另一个索引 p 进行编号的 Paxos 算法实例。那么，agent 的状态就变成了如下形式：

State： Sp,i,n:Value∪{no,neutral}初始化为 neutral

那么可以用{v,last,next,p}来为 agent i 的状态进行编码：

Sq,i,m=no，for all q<=p and m <next

这样查询就只需要在 leader 发生改变时进行。另外我们还可以将 outcome 消息附在下一个 Paxos 实例的命令消息上捎回来。这样运行一个 round 就只需要两个消息(一个来回)。{!即 Multi-Paxos，Multi-Paxos 本身并无改变或引入新的消息，它只是去掉了那些不必要的消息传输，那么它是如何工作的呢？首先我们假设目前只有一个 leader，它负责提出一系列提案。它首先提出了一个提案，然后该提案也成功地被半数以上的 agent 通过了，那么这些 agent 肯定具有相同的状态值(v,lasti,nexti)，那么后续的提案就可以直接利用这个事实。关于 Multi-Paxos，本文和<<Paxos Made Simple>>都讲得有些简略，更详细的内容可以参考最初的那篇<<The Part-Time Parliament>>。}

8.结论

我们已经展示了如何使用一致性来构建一个高可用系统。思路就是使用复制式状态机，并让每个状态机在输入上达成一致。为使之更高效，需要利用租约来使用单个进程去取代大部分的一致性步骤。

之后，我们又引出了不需要任何实时性保证的最容错的一致性算法。即 Lamport 的 Paxos 算法，该算法会重复运行多轮直到得到一个多数者集合，同时保证在得到多数者集合后的每轮都会得到相同的 value 值。同时，我们也介绍了如何只使用少量消息来实现它，以及在 leader 不发生改变的情况下如何只使用一个消息传输来回完成一系列一致性算法执行实例。

此外，我们还解释了如何来设计以及实现一个并行容错系统。方法就是，首先写出一个简单的状态机描述，找到从实现到描述的抽象函数，建立合适的不变性，然后证明实现是对描述的一个模拟(simulate)。该方法可以用来解决很多非常难的问题。

9.参考文献

1. M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science* **82**, 2, May 1991.

2. B. Alpern and F. Schneider. Defining liveness. *Information Processing Letters* **21**,4, 1985.

3. M. Fischer, N. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM* **32**, 2, April 1985.
4. J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.
5. L. Lamport. The implementation of reliable distributed multiprocess systems.*Computer Networks* **2**, 1978.
6. L. Lamport. A simple approach to specifying concurrent systems. *Comm. ACM*,**32, 1,** Jan. 1989.
7. L. Lamport. The part-time parliament. Technical Report 49, Systems Research Center, Digital Equipment Corp., Palo Alto, Sep. 1989.
8. B. Liskov and B. Oki. Viewstamped replication, *Proc. 7th PODC*, Aug. 1988.
9. N Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.
10. N. Lynch and F. Vaandrager. Forward and backward simulations for timing-based systems. *Lecture Notes in Computer Science* 600, Springer, 1992.
11. F. Schneider. Implementing fault-tolerant services using the state-machine approach: A tutorial. *Computing Surveys* **22** (Dec 1990).

10.附录

10.1 Safety&Liveness

Defining liveness

10.2 Guarded Command

BNF 语法定义如下：

〈 guarded command〉 ::= 〈 guard〉 → 〈 guarded list〉
〈 guard〉 ::= 〈 boolean expression〉
〈 guarded list〉 ::= 〈 statement〉 {; 〈 statement 〉 }
〈 guarded command set〉 ::= 〈 guarded command〉 { 〈 guarded command 〉}
〈 alternative construct〉 ::= if 〈 guarded command set〉 fi
〈 repetitive construct 〉 ::= do 〈 guarded command set〉 od
〈 statement〉 ::= 〈 alternative construct〉 | 〈 repetitive construct〉 |
"other statements" .

其中有两个比较重要的构建块是 if...fi 和 do...od。

对于 if...fi 来说，如果在它内部的那些 guard 的值都不为 true，那么程序会 abort，否则将会选择任意一个 guard 值为 true 的，guarded list 执行。需要说明的是，如果允许 if...fi 中间是一个空的 guarded command set“if fi”，那么它的语义等价于 abort。

比如如下语句就可以用来求 x 和 y 的最大值：

if x ≥ y → m:= x

```
 y ≤ x → m:= y
fi
```

对于 do…od 来说，如果在它内部的那些 guard 的值都不为 true，程序会正常结束而不会 abort，但是反过来如果存在为 true 的 guard，那么它会一直执行直到所有的 guard 都不为 true：在初始化时或者执行完选定的 guarded list 后，如果至少有一个 guard 为 true，那么会重新选择一个 guard 为 true 的 guarded list 执行。这样，在该语句执行完毕后，所有的 guards 都会变成 false。这样如果允许 if…fi 中间是一个空的 guarded command set“if fi”，那么它的语义实际上等价于 skip。比如如下语句，可以保证将 Q1, Q2, Q3 和 Q4 这几个 value 值赋给变量 q1, q2, q3 和 q4，并保证 q1 ≤ q2 ≤ q3 ≤ q4。

```
q1, q2, q3, q4 := Q1, Q2, Q3, Q4;
do q1 > q2 → q1, q2 := q2, q1
 q2 > q3 → q2, q3 := q3, q2
 q3 > q4 → q3, q4 := q4, q3
od .
```

下面再看一段程序，对于该段程序来说，不仅计算过程是不确定性的，最终结果也不是唯一确定的。该程序会确定一个 k 值，该值满足如下条件：
对于一个给定的 n(n>0)和函数 f(i)(0 ≤i<n)， 0 ≤ k < n and (∀ i: 0 ≤ i < n: f(k) ≥ f(i))
(k 实际上是最大值所对应的 index)

```
k:=0; j:=1;
do j ≠ n → if f(j) ≤ f(k) → j:= j + 1
          f(j) ≥ f(k) → k:= j; j:= j + 1
       fi
od
```

只有那些可能的最终状态(可能存在多个 f(i)都具有相同的值，同时它们又都是最大值)才会作为结果，并且每个可能的最终状态都有可能成为结果。

由上述解释及实例可见 guarded command 的核心在于非确定性，通过它可以表达非确定性的程序执行，与常见的程序设计语言非常不同。关于更形式化的一些定义和定理以及更深入的介绍，请参考 Dijkstra 在 1974 年发表的文章“Guarded commands. non-determinacy and a calculus for the derivation of programs”。

10.3 Backward Simulation& Prophecy Variable

Forward and Backward Simulations Part I: Untimed Systems
Forward and Backward Simulations - Part II: Timing-Based Systems
Generic forward and backward simulations

10.4 Abstract Function

1967 年，Floyd 提出用“断言法”证明程序的正确性。1969 年，Hoare 发表了“An

Axiomatic Basis for Computer Programming”，在 Floyd 的基础上，定义了一个小语言和一个逻辑系统。此逻辑系统含有程序公理和推导规则，目的在于证明程序的部分正确性，这就是著名的 Hoare 逻辑。他的工作为公理学语义的研究奠定了基础。

1972 年，Hoare 又发表了“Proof of Correctness of Data Representations”。在这篇论文中，提出了一种将抽象程序转换为具体程序的自动化方法，并提供了关于正确性的证明方法，即证明具体的表示方式具有抽象程序所期望的所有属性。

如果已经证明了数据表示方式是正确的，那么最终具体程序的正确性就只依赖于原始抽象程序的正确性了。因为抽象程序通常非常简单更容易证明正确性，这样通过这样分解一下(即将具体程序的正确性证明分解为具体的数据表示形式和抽象程序的正确性证明)，就会大大简化整个程序的正确性证明。