**Raft算法论文（中文翻译版）**

1. **摘要**

Raft是一种为了管理复制日志的一致性算法，它提供了和Paxos算法相同的功能和性能，但是它的算法结构和性能和Paxos不同，使得Raft算法更容易理解并且更容易构建实际的系统。为了提升可理解性，Raft将一致性算法分解成了几个关键模块，例如领导者选举、日志复制和安全性。同时它通过实施一个更强的一致性来减少需要考虑的状态的数量，从一个用户研究的结果可以证明，对于学生而言，Raft算法比Paxos算法更加容易学习。Raft算法还包括一个新的机制来允许集群成员的动态改变，它利用重叠的大多数来保证安全性。

1. 内容
   1. 介绍

一致性算法允许一组机器象整体一样工作，即使其中一些机器出现故障也能够继续工作下去。正以为如此，一致性算法在构建大规模软件系统中扮演重要的角色。在过去的10年里，Paxos算法统治着一致性算法这一领域；绝大多数的实现都基于Paxos或者受其影响，同时，Paxos也成为教学领域里讲解一致性问题时的示例。

但不幸的是，虽然有很多工作都在尝试降低它的复杂性，但是Paxos算法仍然十分难以理解，并且，Paxos自身的算法结构需要进行大幅的修改才能应用到实际的系统中，这些都导致了工业界和学术界都对Paxos算法感到十分头疼。

在尝试过Paxos算法之后，我们开始寻找一种新的一致性算法，可以为构建实际的系统和教学提供更好的基础。我们的做法是不同寻常的，我们的首要目标是可理解性；包括算法分解（Raft被分解成了领导者选举，日志复制和安全性），和减少状态机的状态（相比于Paxos，Raft减少了非确定性和服务器相互处于非一致性的方式），一份针对两所大学43个学生的研究表明，Raft算法明显比Paxos算法更加容易理解。在学生同时学习了这两种算法之后，和Paxos比起来，其中33个学生能够回答有关于Raft的问题。

Raft算法在很多方面和现有的一致性算法都很相似（主要是Oki和Liskov的Viewstamped Replication），但是它也有一些独特的特性：

* 强领导者

和其他一致性算法相比，Raft使用一种更强的领导能力形式。比如，日志条目只从领导者发送给其他服务器。这种方式简化了对复制日志的管理并使Raft算法更加易于理解。

* 领导选举

Raft算法使用一个随机计时器来选举领导者。这种方式知识在任何一致性算法都必须实现的心跳机制上增加了一点机制，在解决冲突的时候会更加简单快捷。

* 成员关系调整

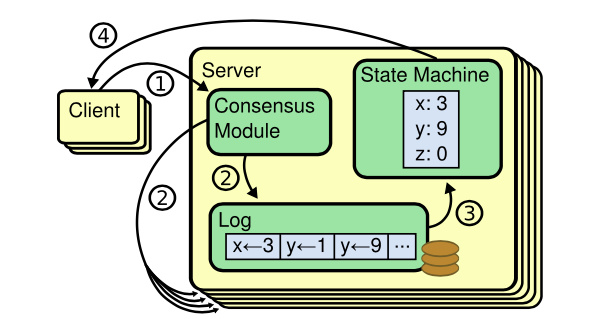
Raft使用一种共同一致的方法来处理集群成员变更的问题。在这种方法下，处于调整过程中的两种不同的配置集群中大多数机器会有重叠，这使得集群在成员变换的情况下，仍然可以继续工作。

我们相信，Raft算法不论处于教学目的，还是作为实践项目的基础，都要比Paxos或者其他一致性算法要更优异。它比其他算法更加简单，更加容易理解；它的算法描述足以实现一个现实的系统；它有很多开源的实现并且在很多公司里使用；它的安全性已经被证明；它的效率和其他的算法比起来也不相上下。

接下来，这篇论文会介绍一下内容：复制状态机问题（第二节），讨论Paxos的优点和缺点（第三节），讨论我们为了可理解性而采取的方法（第四节），阐述Raft一致性算法（5-8节），评价Raft算法（第9节），以及一些相关的工作（第10节）。

* 1. 复制状态机

一致性算法是从复制状态机的背景下提出的（参考英文原文引用37），在这种方法中，一组服务器上的状态机产生相同状态的副本，并且在一些机器宕机的情况下也可以继续运行。复制状态机在分布式系统中被用于解决很多容错的问题。例如，大规模的系统中通常有一个集群领导者，象GFS,HDFS和RAMCloud,典型应用就是一个独立的日志状态机去管理领导者选举和存储配置信息，并且在领导者宕机的情况下，也要存活下来，比如chubby和zookeeper。



一致性算法管理着来自客户端指令的复制日志。状态机从日志中处理相同顺序的相同指令，所以产生的结果也是相同的。

复制状态机通常都是基于复制日志实现的，每一个服务器存储一个包含一系列指令的日志，并且按照日志的顺序进行执行。每一个日志都按照相同的顺序，包含相同的指令，所以每一个服务器都执行相同的指令序列。因为每一个状态机都是确定的，每一次执行操作都产生相同的状态和相同的序列。

保证复制日志相同，就是一致性算法的工作。在一台服务器上，一致性模块接受客户端发送过来的指令，然后增加到自己的日志中去。它和其他服务器上的一致性模块进行通信保证每一个服务器上的日志都以相同的顺序包含相同的指令，尽管有些服务器会宕机。一旦指令被正确的复制，每一个服务器的状态机按照正确的顺序处理它们，然后输出结果返回给客户端。因此，服务器集群看起来像是一个高可靠的状态机。

实际系统中使用的一致性算法，通常包含以下特性：

* 安全性保证（绝对不会返回一个错误的结果），在拜占庭错误下，包括网络延迟，分区，丢包，冗余和乱序等错误，都可以保证正确。
* 可用性：集群中只要大多数机器可以运行并且能够相互通信、和客户端通信，就可以保证可用。因此，一个典型的包含5个节点的集群可以容忍两个节点的失败。服务器被停止就被认为是失败。他们当有稳定的存储的时候，可以从状态中恢复回来并重新加入集群。
* 不依赖时序来保证一致性：物理时钟错误或者极端的消息延迟只有在最坏的情况下，才会导致可用性问题。
* 通常情况下，一条指令可以尽可能快的在集群中大部分节点响应一轮远程过程调用时完成。小部分比较慢的节点不会影响整体的性能。
  1. Paxos算法的问题

在过去10年里，Leslie Lamport的Paxos算法几乎成为一致性的代名词，Paxos算法是在课程教学中最经常使用的算法，同时也是大多数一致性算法实现的起点。Paxos首先定义了一个能够达成单一决策一致的协议，比如单条的复制日志项。我们把这一子集叫做单决策Paxos。然后通过组合多个Paxos协议的实例来促成一系列决策的达成。Paxos保证安全性和活性，同时也支持集群成员关系的变更。Paxos的正确性已经被证明，在通常情况下也很高效。

不幸的是，Paxos有两个明显的缺点。第一个缺点是Paxos特别难以理解。完整的解释是，出了名的不透明；通过极大的努力之后，也只有少数人成功的理解了这个算法。因此，有了几次用更简单的术语来解释Paxos的尝试。尽管这些尝试都只关注了单决策的子集问题，但是依然很具有挑战性。在2012年NSDI的会议中的一次调查显示，很少有人对Paxos算法感到满意，甚至在经验老道的研究者中也是如此。我们自己也尝试去理解Paxos，我们一直没能理解Paxos，直到我们读了很多对Paxos的简化解释，并设计了我们自己的算法之后。这一过程花了近一年的时间。

我们假设Paxos的不透明性来自它选择单决策问题作为它的基础。单决策Paxos是晦涩微妙的，它被划分成了两种没有简单直观解释和无法独立理解的场景。因此，很难建立起直观的感受为什么单决策Paxos能够工作。构成多决策Paxos增加了很多错综复杂的规则。我们相信，在多决策上达成一致性的问题（一份日志而不是单一的日志记录）能够被分解成其他方式并且更加直接和明显。

Paxos算法的第二个问题是它没有提供一个足够好的用来构建现实系统的基础。一个原因是还没有一种被广泛认同的多决策问题的算法。Lamport的描述基本上都是关于单决策Paxos的，他简要描述了实施多决策Paxos的方法，但是缺乏很多细节。当然也有很多具体化的多决策Paxos的尝试，但是他们相互不一样，和Paxos的概述也不同。例如chubby这样的系统实现了一个类似Paxos的算法，但是大多数的细节没有被公开。

而且，Paxos算法的结构也不是十分有利于构建实践的系统；但决策分解也会产生其他的结果。例如，独立的选择日志的条目，然后合并成一个序列化的日志并没有带来太多的好处，仅仅增加了不少复杂性。围绕着日志来设计一个系统是更加简单高效的；新日志的条目按照严格的顺序添加到日志中去。另I个问题是，Paxos使用了一种对等的，点对点的方式作为它的核心（尽管它提议了一种弱领导人的方式来优化性能），这在只有一个决策会被制定的世界中是很有意义的，但很少有现实的系统使用这种方式。如果有一系列的决策需要被制定，首先选定一个领导者，然后让他去协调所有的协议，会更加简单快速。

因此，现实的系统很少有和Paxos相似的实践。每一种实现都是从Paxos开始研究，然后发现很多实现上的难题，再然后开发了一种和Paxos完全不同的结构。这样是非常费时和容易出错的，并且理解Paxos的难度使得这个问题更加糟糕。Paxos算法在理论上被证明是正确可行的，但是现实的系统和Paxos的差别是如此的大，以至于这些证明没有什么太大的价值，下面来自chubby的实现非常典型：在Paxos算法描述和实现现实系统中间存在巨大的鸿沟。最终的系统建立在一种没有经过证明的算法之上。

由于以上问题，我们认为Paxos算法既没有提供一个良好的基础给实践的系统，也没有给教学很好的帮助。基于一致性问题在大规模软件系统中的重要性，我们决定看看我们是否能够设计一个拥有更好特性的替代Paxos的一致性算法。Raft算法就是这次实验的结果。

* 1. 为了可理解性的设计

设计Raft算法我们有几个初衷：它必须提供一个完整的实际的系统实现基础，这样才能大大减少开发者的工作；它必须是在任何情况下都是安全的并且在大部分的情况下都是可用的；并且它的大部分操作都是高效的。但是我们最重要也是最大的挑战是可理解性。它必须保证对于普通的人群都可以十分容易的去理解。另外，它必须能够让人形成直观的认识，这样系统的构建者在现实中进行必然的拓展。

在设计Raft算法的时候，有很多的点需要我们在各种备选方案中进行选择。在这种情况下，我们评估备选方案基于可理解性选择：解释各个备选方案有多大的难度（例如，Raft的状态空间有多复杂，是否有微妙的暗示）？对于一个读者而言，完全理解这个方案和暗示是否容易？

我们意识到对这种可理解性分析上具有高度的主观性；尽管如此，我们使用了两种通常适用的技术来解决这个问题：第一个技术就是众所周知的问题分解：只要有可能，我们就将问题分解为几个相互独立的，可被解决的，可解释的和可理解的子问题。例如，Raft算法被我们分解为领导者选举、日志复制、安全性和角色改变几个部分。

我们使用的第二个方法是通过减少状态的数量来简化需要考虑的状态空间，使得系统更加连贯并且在肯能的时候消除不确定性。特别的，所有的日志是不允许有空洞的，并且Raft限制了日志之间变成不一致状态的可能。尽管我们在大多数情况下都试图去消除不确定性，但是也有一些情况下，不确定性可以提升可理解性。尤其是，随机化方法增加了不确定性，但是它们有利于减少状态空间数量，通过处理所有可能选择使用类似的方法，我们使用随机化简化Raft中领导者选举算法。

* 1. Raft一致性算法

Raft算法是一种用来管理章节 2 中描述的复制日志的算法。图 2 为了参考之用，总结这个算法的简略版本。图3列举了这个算法的一些关键特性。图中的这些元素会在接下来的章节中逐一介绍。

Raft通过选举一个高贵的领导者，然后给予它全部的管理复制日志的责任来实现一致性。领导者从客户端接受日志条目，把日志条目复制到其他服务器上，并且当保证安全性的时候，告诉其他的服务器应用日志条目到他们的状态机上。用有一个领导者大大简化了对日志复制的管理。例如，领导者可以决定新的日志条目需要放在日志的什么位置，而不需要和其他服务器商议。并且数据都从领导者流向其他服务器。一个领导者宕机，可以和其他服务器失去连接，这时一个新的领导者会被选举出来。

通过领导者的方式，Raft将一致性问题分解为3个相对独立的子问题，这些问题会在接下来的章节中进行讨论：

1. 领导者选举：一个新的领导者需要被选举出来，当现存的领导者宕机的时候。
2. 日志复制：领导者必须从客户端接受日志，然后复制到集群中的其他节点，并且强制要求其他节点的日志保持和自己相同。
3. 安全性：在Raft中安全性的关键是状态机的安全：如果有任何的服务器节点已经应用了一个确定的日志条目到它的状态机中，那么其他的服务器节点不能在同一个日志索引位置应用一个不同的指令。章节5.4阐述了Raft算法是如何保持这个特性的，和解决方案涉及到一个额外的选举机制上的限制。

在展示一致性算法之后，这一章会讨论可用性的一些问题和计时在系统的作用。

状态：

|  |  |
| --- | --- |
| 状态 | 所有服务器上持久存在的 |
| CurrentTerm | 服务器当前知道的任期号（初始为0，持续递增） |
| votedFor | 当前获得选票的候选人ID |
| Log[] | 日志条目集，每一个条目包含一个用户状态机执行的指令，和收到的任期号 |

|  |  |
| --- | --- |
| 状态 | 所有服务器上经常变的 |
| CommitIndex | 已知的最大的已经被提交的日志条目的索引值 |
| LastApplied | 最后被应用到状态机的日子条目索引值（初试为90，持续递增） |

|  |  |
| --- | --- |
| 状态 | 在领导者里经常改变的 |
| NextIndex[] | 对于每一个服务器，需要发送给他的下一个日志条目的索引值（初始化为领导者最后索引+1） |
| matchIndex[] | 对于每一个服务器，已经复制给他的日志的最高索引值 |

附加日志RPC

由领导者负责调用來复制日志指令：也会用作heartbeat

|  |  |
| --- | --- |
| 参数 | 解释 |
| Term | 任期 |
| LeaderId | 领导者ID，以便于Follower重定向请求 |
| PrevLogIndex | 新的日志条目之前的日志条目的索引值 |
| PrevLogTerm | prefLogIndex的任期 |
| Entries[] | 准备存储的日志条目 |
| LeaderCommit | 领导者已经提交的日志的索引值 |

返回值

|  |  |
| --- | --- |
| 参数 | 解释 |
| Term | 当前的任期号，用于领导者去更新自己 |
| Success | Follower包含了匹配上PrevLogIndex和prevLogTerm的日志时为真 |

接收者实现：

1. 如果term < currentTerm就返回false
2. 如果日志在prevLogIndex位置处的日志条目的任期号和prevLogTerm不匹配，则返回false（5.3节）
3. 如果已经存在的日志条目和新的产生冲突（索引值相同但任期号不同），删除这一条和之后所有的。（5.3节）
4. 附加日志尚未存在的任何新条目
5. 如果leaderCommit > commitIndex,令commitIndex等于leaderCommit和新日志索引条目中较小的一个。

请求投票RPC:

由候选人负责调用来征集选票：

|  |  |
| --- | --- |
| 参数 | 解释 |
| Term | 候选人的任期号 |
| CandidateId | 请求选票的候选人ID |
| LastLogIndex | 候选人的最后日志条目的索引值 |
| LastLogTerm | 候选人最后日志条目的任期号 |

返回值

|  |  |
| --- | --- |
| 参数 | 解释 |
| Term | 当前任期号，以便候选人去更新自己的任期号 |
| VoteGranted | 候选人赢得了此张选票时返回true |

接收者实现：

1. 如果term < currentTerm，返回false（5.2节）
2. 如果votedFor为空或者为candidateId,并且候选人的日志至少和自己一样新，那么投票给它。

所有服务器都遵守的规则：

所有服务器：

1. 如果commitIndex > lastApplied，那么lastApplied+1，并把Log[lastApplied]应用到状态中。（5.3）
2. 如果接收倒的rpc请求或者响应中，任期号T>currentTerm，那么就令currentTerm = T，并切换为跟随者。

跟随者：

1. 响应来自候选人和领导者的请求
2. 如果超过领导者心跳超时时间,没有收到当前领导者（领导人的任期号与跟随者的当前任期相同），或者给某个候选人投了票，就自己变成候选人。
3. 在转变成候选人之后，立即开始选举过程
   1. 自增当前的任期号
   2. 给自己投票
   3. 重置选举超时计时器
   4. 发送请求投票的RPC给其他服务器
4. 如果接收到大多数服务器的选票，那么，就变成候选人。
5. 如果接受到来自新的领导者的appendlog的请求，转变为跟随者
6. 如果选举过程超时，再次发起一轮选举

领导者：

1. 一旦成为领导人，发送空的append日志RPC（心跳）给其他服务器；在一定的空余时间之后，不停的重复发送，以阻止跟随者超时。（5.2节）
2. 如果接受到来自客户端的请求，附加条目到本地日志中，在条目被应用到状态机后响应客户端。（5.3节）
3. 如果对于一个跟随者，最后日志条目的索引值大于等于nextIndex，那么，发送从nextIndex之后的所有日志条目：
   1. 如果成功，更新响应跟随者的nextIndex和matchIndex
   2. 如果因为日志不一致而失败，减少nextIndex重试
4. 如果存在一个满足n>commitIndex的n，并且大多数的matchIndex[i]>=n，成立，并且log[n].term == currentTerm成立，那么，令commitIndex=n

一个关于Raft算法的浓缩总结

|  |  |
| --- | --- |
| 特性 | 解释 |
| 选举安全特性 | 对于一个给定的任期号，最多只有一个领导者会被选举出来 |
| 领导者只append原则 | 领导者绝对不会删除和覆盖自己的日志，只会append |
| 日志匹配原则 | 如果两个日志在相同的索引位置的日志条目的任期号相同，我们我们认为这两个日志从头开始到这个位置的之前的日志，完全相同 |
| 领导者完全特性 | 如果某个日志条目在某个任期号中已经被提交，那么，这个日志条目必然出现在更大任期号的领导者的日志中。 |
| 状态机完全特性 | 如果一个领导者已经将给定位置的日志条目应用到状态机中，那么，其他任何服务器的找个索引位置不会应用一个不同的日志。 |

**Raft在任何时候都保证以上特性。**

* + 1. Raft基础

一个Raft集群包含若干个服务器节点：通常是5个，这允许整个系统容忍2个节点的失败。在任何时刻，每个服务器节点都处于三种状态之一：领导者，候选人和跟随者。在通常情况下，系统只有一个领导人，并且其他节点都是跟随者。跟随者都是被动的：他们不会法任何请求，都是简单的响应来自领导者或者候选人的请求。领导者处理所有的客户端请求（如果一个客户端跟跟随者联系，那么跟随者会把请求重定向给领导者）。第二种状态候选人，是用来在5.2节描述的领导者选举时使用。图4展示了这些状态和他们之间的转换关系，这些转换关系会在接下来进行讨论。

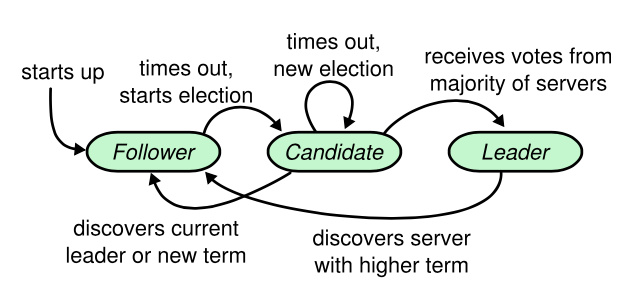


图4

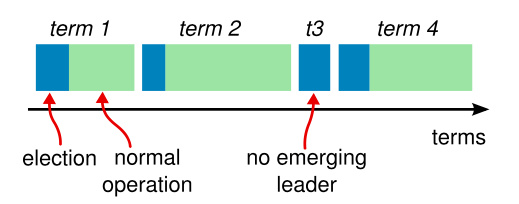


图5

Raft把时间分为任意长度的任期。如图5。任期用连续的整数标记。每一段任期从一次选举开始，就像章节5.2描述的那样，一个或多个候选人尝试成为领导者。如果一个候选人赢得选举，那么它在接下来的任期内充当领导者的职责。在某些情况下，一次选举会造成选票的瓜分。在这种情况下，这次选举会以没有领导人结束；一个新的任期（和一次新的选举）会很快重新开始，Raft保证了在一个给定的任期内，最多只有一个领导者当选。

不同的服务器节点可能多次观察到任期之间的转换，但在某些情况下，一个节点也可能观察不到任何一次选举或者整个任期全程。任期在Raft算法中充当逻辑时钟的作用，这会允许服务器节点查明一些过期的信息比如陈旧的领导者。每一个节点存储一个当前任期号，这一编号在整个时期内单调的增长。当服务器之间通信的时候会交换当前任期号；如果一个服务器的当前任期号比其他服务器小，那么它会更新自己的任期号到较大的编号值。如果一个候选人或者领导者发现自己的任期号过期了，那么，它会立即恢复成跟随者的状态。如果一个节点接收到一个包含过期任期号的请求，那么它会直接拒绝这个请求。

Raft算法中服务器节点之间的通信采用远程过程调用（RPCs），并且基本的一致性算法只需要两种类型的RPCs，

1. RequestVote，请求投票，由候选人在选举期间发起。
2. AppendEntries，附加日志，由领导者发起，用户复制日志和维持心跳。

第7节为了在服务器之间传输快照，增加了第三种RPCs。当服务器没有收到RPC的响应时，会进行重试，并且能够并行的发起RPCs来获得最佳的性能。

* + 1. 领导者选举

Raft使用一种心跳机制来触发领导者选举。当服务器启动的时候，它们的身份都是跟随者。一个服务器节点继续保持着跟随者状态，只要它从领导者或者候选人处接收到有效的RPCs，领导者周期性的向所有跟随者发送心跳包（即不包含日志项的附加日志）来维护自己的权威，如果一个跟随者在一段时间内没有接收到任何消息，也就是选举超时，那么它会认为集群中没有可用的领导者，随即发起选举以选出新的领导者。

要开始一个选举过程，跟随者要先增加自己的任期编号并且切换到候选人状态。然后它会并行的向集群中的其他节点发送请求投票的RPCs来给自己投票。候选人会继续保持着当前的状态直到以下三件事之一发生：

1. 它赢得了这次选举
2. 其他的服务器成为领导者
3. 一段时间之内没有任何一个获胜的人。

这些结果会分别在下面的段落里讨论。

当一个候选人在集群里的大多数服务器获得了针对同一个任期号的选票，那么它就赢得了这次选举并成为领导者。每一个服务器最多会对一个任期号投出一张选票，按照先来先服务的原则（注：5.4节在投票上增加了一点额外的限制），要求大多数选票的规则确保了最多只会有一个候选人赢得此次选举（选举安全性）。一旦候选人赢得选举，它就立刻成为领导人。然后它会向其他服务器发送心跳来建立自己的权威并阻止新的领导人的产生。

在等待投票的时候，候选人可能会从其他的服务器收到它是领导人的appendlog的RPC，如果这个领导人的任期号（包含在此次RPC中）不小于候选人当前的任期号，那么候选人承认领导人合法并恢复回跟随者的状态。如果此次RPC中的任期号比自己小，那么，候选人就会拒绝此次RPC并且保持候选人的状态。

第三种能的结果是，候选人既没有赢得选举，也没有输，如果多个跟随者同时成为候选人，那么，选票有可能被瓜分，以至于没有人获得多数的选票。当这种情况发生的时候，每一个候选人都会超时，然后通过增加当前任期号来进行新一轮的选举，然而，没有其他机制的话，选票可能会被无限期的重复瓜分。

Raft使用随机选票超时时间的方式来确保很少会发生选票瓜分的情况，就算发生也能很快解决。为了阻止选票起初就被瓜分，选举超时时间是从一个固定的区间随机选择（比如200-300毫秒）。这样就可以把服务器分散开以至于大多数情况下只有一个服务器会选举超时，然后它赢得选举并在其他服务器超时之前发出心跳包。同样的机制被用在选票被瓜分的情况下。每一个候选人在开始选举的时候会重置一个随机的选举超时时间，然后在超时时间内等待投票的结果；这样减少了在新的选举中另外的选票瓜分的可能性。9.3节展示了这种方案能够快速的选出一个领导人。

领导人选举这个例子，体现了可理解性原则是如何知道我们进行方案设计的。起初我们使用一种排名系统：每一个候选人都被赋予一个唯一的排名，供候选人之间竞争时进行选择。如果一个候选人发现另一个候选人拥有更高的排名，那么，他就会回到跟随者的状态，这样高排名的候选人能够更加容易的赢得下一次选举。但是我们发现这种方法在可用性方面存在一些问题（如果高排名的服务器宕机了，那么低排名的服务器会超时并再次进入候选人状态。而且如果这个行为发生的足够快，则可能导致整个选举过程都被重置掉）。我们针对算法进行了多次调整，但每次调整之后都会有新的问题，最终我们认为随机重试的方法是更加明显和易于理解的。

* + 1. 日志复制

一旦一个领导者被选举出来，他就开始为客户端提供服务。客户端的每一个请求都包含被复制状态机执行的指令。领导人把这条指令作为一个新的日志条目添加到日志中去。然后并行的发起appendLogItem的RPCs给其他的服务器，让他们复制这条日志条目。当这个日志条目被安全的复制（下面会介绍），领导人会应用这个日志条目到它的状态机中，然后把执行的结果返回给客户端。如果跟随者崩溃或者运行缓慢，再或者网络丢包，领导人会不断的重复尝试append log RPCs（尽管已经回复了客户端），直到所有的跟随者都最终存储了所有的日志条目。

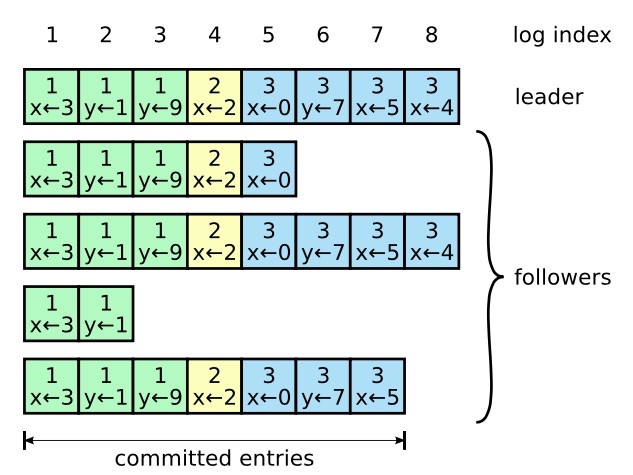


图6：日志由有序的序号标记的条目组成，每个条目都包含创建时的任期号和一个状态机需要执行的指令，一个条目当可以安全的被应用到状态机中去的时候，就认为是可以提交了。

每一个日志条目存储一条状态机指令和从领导人那里收到这条指令时的任期号。日志中的任期号用来检查是否出现不一致的情况。同时也用来保证图3中的某些性质。每一条日志条目也都有一个索引值用来表明它在日志中的位置。

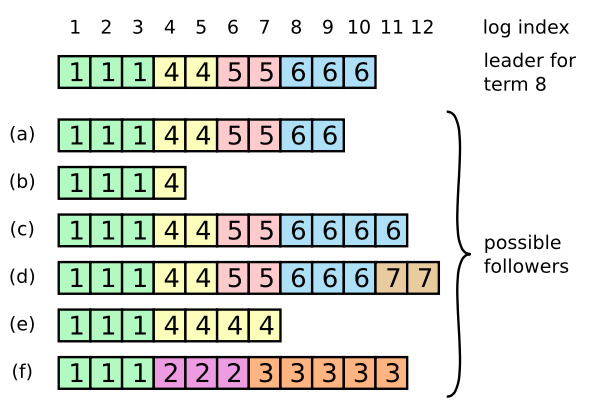
领导人来决定什么时候把这条日志应用到状态机中是安全的；这种日志条目被称为已提交。Raft保证所有已提交的日志条目都是持久化的，并最终会被所有可用的状态机执行。在领导人将创建的日志条目复制到大多数状态机上的时候，日志条目就会被提交。（例如在图6中的条目7）同时，领导人的日志中之前的所有条目也都会被提交，包括由其他领导人创建的条目。5.4节会讨论某些当领导人改变之后应用这条规则的隐晦内容。同时他也展示了这种提交的定义是安全的。领导人跟踪了最大的将会被提交的日志项的索引，并且索引值会被包含在未来的所有append log的RPCs（包括心跳包），这样其他的服务器才能最终知道领导人的提交位置。一旦跟随者知道一个日志条目已经被提交，那么它也会将这个日志条目应用到本地的状态机中（按照日志的顺序）。

我们设计了Raft的日志机制来维护一个不同服务器的日志之间的高层次的一致性。这么做不仅简化了系统的行为，也使得更加可预计，同时它也是安全性保证的一个重要组件。Raft维护着以下特性，这些同时也组成了图3中的日志匹配特性：

1. 如果在不同的日志中的两个条目拥有相同的索引和任期号，那么他们存储了相同的指令。
2. 如果在不同的日志中的两个条目拥有相同的索引和任期号，那么他们之前的所有日志条目也都全部相同。

第一个特性来自这样一个事实，领导人最多在一个任期里在指定的一个日志索引位置创建一条日志条目，同时日志条目在日志中的位置也从来不会发生变化。第二个特性由附加日志RPC的一个简单一致性检查所保证。在发送append log RPC的时候，领导人会把新的日志条目的之前的那条日志的索引位置和任期号包含在里面。如果跟随者在它的日志里找不到这条日志，那么它就会拒绝接受这条日志。

一致性检查就像一个归纳步骤，一开始空的日志状态肯定是满足日志匹配特性的，然后当日志拓展的时候的日志匹配特性。因此，每当append log RPC成功的时候，领导人就知道跟随者的日志肯定是和自己的日志完全相同的了。



在正常的操作中，领导人和跟随者的日志保持一致，所以append log RPC的一致性检查从来不会失败，然而，领导者崩溃的情况，会使得日志处于不一致的状态。（老的领导人可能还没有复制所有的日志条目），这种不一致会在领导人和跟随者的一系类崩溃下加剧。图7展示了跟随者的日志可能和新领导人不同的方式。跟随者可能会丢失一些在新领导者中有的日志，它也可能有一些新领导者没有的日志条目，或者两者都会发生。丢失或者多出日志条目可能会持续多个任期。

在Raft算法中，领导者处理不一致是通过强制跟随者直接复制自己的日志来解决。这意味着在跟随者中冲突的日志会被领导者的日志所覆盖。5.4节会阐述如何增加一些限制来使得这样的操作是安全的。

要使得跟随者的日志进入和自己一样的状态，领导者必须找到最后两者达成一致的地方，然后删除从那个点之后的所有日志条目，发送自己的日志给跟随者。所有的操作都在进行append log RPCs的一致性检查时完成。领导人针对每一个跟随者维护了一个nextIndex，这表示下一个需要发送给跟随者的日志的索引地址。当一个领导人刚获得权利的时候，它初始化所有nextIndex的值为自己的最后一条日志的index+1（图7中的11），如果一个跟随者的日志和领导人不一致，那么，在下一次的append log RPC的一致性检查时就会失败，在被跟随者拒绝之后，领导者会减少nextIndex的值并进行重试，这时就会把跟随者冲突的日志条目全部删除并且加上领导者的日志。一旦append log的RPCs成功，那么跟随者的日志就会和领导者保持一致，并且在接下来的任期里继续保持。

如果需要的话，算法可以通过减少被拒绝的附加日志RPCs的次数来优化。例如，当append log RPC被拒绝的时候，跟随者可以返回包含冲突条目的任期号和自己存储的那个任期的最早索引地址。借助这些信息，领导者可以减少nextIndex越过所有个任期冲突的所有日志条目，这样就变成每个任期需要来一次append log RPC,而不是每个日志条目一次。在实践中，我们十分怀疑这种优化是必要的，因为失败是很少发生的，并且也不大可能会有这么多不一致的日志。

通过这种机制，领导人在获得权利的时候，就不需要任何特殊的一致性措施来恢复一致性。它只需要进行正常的操作，然后日志就能自动的在回复append log rpc的一致性的时候，自动趋于一致。领导人从来不会覆盖或者删除自己的日志（图2的领导人append only特性）。

日志复制机制展示了第二节中形容的一致性特性：Raft能够接受、复制并应用新的日志条目，只要大部分机器是工作的；在通常情况下，新的日志条目可以在一次RPC中被复制给集群中的大多数机器；并且单个缓慢的跟随者不会影响整体的性能。

* + 1. 安全性

前面的章节描述了Raft算法是如何选举和复制日志的。然而，到目前为止描述的机制还不能保证每一个状态会按照相同的顺序执行相同的指令。例如：一个跟随者可能会进入不可用状态同时领导者已经提交了若干的日志条目，然后这个跟随者可能被选举为领导人并且覆盖这些日志条目；因此，不同的状态机可能会执行不同的指令序列。

这一节通过在领导者选举的时候增加一些限制来完善raft算法。这一限制保证了任何的领导人对于给定的任期号，都拥有了之前任期的所有被提交的日志条目（图3中的领导人完整特性）。增加这一选举时的限制，我们对于提交的规则也更加清晰。最终，我们将展示对于领导人完整性的简要证明，并说明领导人完整性特性是如何引导复制状态机做出正确行为的。

1. 选举限制

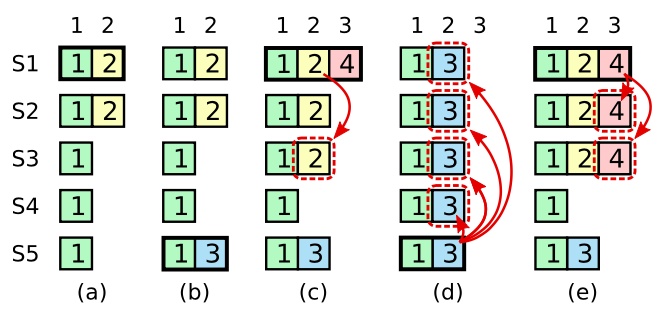
在任何基于领导人的一致性算法中，领导人都必须存储所有已经提交的日志条目。在某些一致性算法中，例如viewstamped replication，某个节点即使一开始没有包含所有已经提交的日志条目，它也能被选举为领导者。这些算法都包含一些额外的机制来识别丢失的日志条目，并把它们传送给新的领导人，要么是在选举阶段，要么在之后很快进行。不幸的是，这种方法会导致相当大的额外的机制和复杂性。Raft使用了一种更加简单的方法，它可以保证所有之前的任期号中已经提交的日志条目，在选举的时候，都会出现在新的领导人中，不需要传送这些日志条目给领导人。这意味着日志条目的传送是单向的，只从领导人传送给跟随者，并且领导人从不会覆盖本地日志中已经存在的条目。

Raft用投票的方式阻止一个候选人赢得选举，除非这个候选人包含了所有已经提交的日志条目。候选人为了赢得选举必须联系集群中的大部分节点，这意味着每一个已经提交的日志条目在这些服务器节点中肯定至少存在于一个节点上。如果候选人的日志至少和大多数的服务器节点一样新（这个新的定义会在下面讨论），那么，它一定持有了所有已经提交的日志条目。请求投票RPC实现了这样的限制：RPC中包含了候选人的日志信息，然后候选人会拒绝掉那些日志没有自己新的投票请求。

Raft通过比较两条日志中最后一条日志的索引值和任期号来定义谁的日志比较新。如果两条日志最后的任期号不同，那么任期号更大的比较新。如果任期号相同，那么索引值比较大的那个比较新。

1. 提交之前任期内的日志条目

如同5.3节介绍的那样，领导人知道一条当前任期内的日志记录是可以被提交的，只要它被存储在大部分的服务器上。如果一个领导人在提交日志条目之前崩溃了，那么，未来继任的领导人会尝试复制这条日志条目。然而，一个领导人不能断定一个之前任期里的日志条目被保存到了大多数服务器上的时候，就一定已经提交了。图8展示了一种情况，一条已经被存储到大多数节点上的日志条目，仍然有可能被未来的领导人覆盖掉。

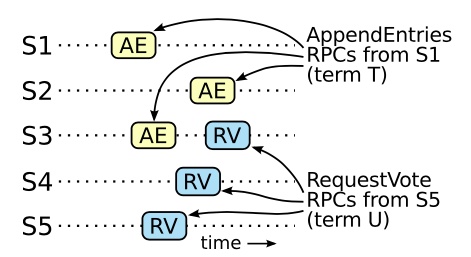


为了消除图8里描述的情况，Raft永远不会通过计算副本数量的方式去提交一个之前任期内的日志条目。只有领导人当前任期里的日志条目通过计算副本数量可以被提交；一旦当前任期的人日条目以这种方式被提交，那么，由于日志匹配特性，之前的日志条目也会被间接的提交。在某些情况下，领导人可以安全的知道一个老的日志条目是否已被提交（例如，该条目是否已经存储到所有服务器上），但是Raft为了简化问题，使用了一种更加保守的方法。

当领导人复制之前任期里的日志时，Raft会为所有日子保留原始任期号，这在提交规则上产生了额外的复杂性。在其他一致性算法中，如果一个新的领导人要重新复制之前的任期的日志时，它必须使用当前的任期号。Raft使用的方法更容易辨别出日志，因为它可以随着时间和日志的变化，对日志维护着同一个任期号。另外，和其他算法相比，Raft中的新领导人只需要发送更少的日志条目（其他算法中必须在他们被提交之前发送更多的冗余日志来为他们重新编号）。

1. 安全性论证

在给定了完整的Raft算法之后，我们现在可以更加精确的讨论领导人完整性特性。（这一讨论基于9.2节的安全性证明）。我们假设领导人完全特性是不成立的，然后我们推出矛盾来。假设任期T的领导人在任期内提交了一个日志条目，但是这个日志条目没有被存储到未来某个任期的领导人的日志中。假设大于T的最小任期U的领导人U没有这条日志条目。



3.1）在领导人U选举的时候，一定没有那条已经被提交的日志条目

3.2）领导人T复制这条日志给集群中的大部分节点，同时，领导人U从集群中的大部分节点赢得了选票。因此，至少有一个节点（投票者，选民）同时接受了来自领导人T的日志条目，并且给领导人U投票了，如图9。找个投票者是产生这个矛盾的关键。

3.3）找个投票者必须在给领导人U投票之前先接受了来自领导人T提交的日志条目，否则它会拒绝T的日志复制请求。（因为此时它的任期号比T的任期号大）。

3.4）投票者在给领导人U投票的时候，仍然保存有这条日志，因为任何中间的领导人都包含该条日志（根据上面的假设），领导人从不会删除条目，并且跟随者只有在跟领导者冲突的情况下才会删除条目。

3.5）投票者在把自己的选票投给U的时候，领导人U的日志必须和自己一样新。这就导致了两者的矛盾之一。

3.6）首先，如果投票者的日志和领导者U的最后一条日志的任期号相同，那么，领导者的日志至少和投票者一样长，所以领导者U的日志一定包含所有投票者的日志。这是另一处矛盾，因为投票者包含了那条已经被提交的日志，但是在上述的假设里，领导者U是不包含的。

3.7）除此之外，领导人U的最后一条日志的任期号就必须投票人大了。另外，它也比T大，因为投票人的最后一个日志条目的任期号至少和T一样大（它包含了来自任期T的已提交日志）。创建了领导人U最后一条日志的之前领导人一定已经包含了那条已被提交的日志（根据上述假设，领导人U是第一个不包含该日志条目的领导人）。所以，根据日志匹配特性，领导人U一定也包含那条被提交的日志，这里产生矛盾。

3.8）这里完成了矛盾，因此，所有比T大的领导人一定包含了所有来自T的已经被提交的日志。

3.9）日志匹配原则保证了未来的领导人也同时会被间接提交的条目，例如图8（d）中的索引2。

通过领导人完全特性，我们就能证明图3中的状态机安全特性，即如果服务器已经在某个给定的索引值应用了日志条目到自己的状态机里，那么，其他服务器不会应用一个不一样的日志到同一个索引值上。在一个服务器应用一条日志条目到它自己的状态机中时，它的日志必须和领导人的日志，在该条目和之前的条目上相同，并且已经被提交。现在我们来考虑在任何一个服务器应用一个指定索引位置的日志的最小任期；日志完全特性保证拥有更高任期号的领导人会存储相同的日志条目，所以之后的任期里应用某个索引位置的日志条目也会是相同的值。因此，状态机安全性是成立的。

最后，Raft要求服务器按照日志中索引位置顺序应用日志条目。和状态机安全性特性结合来看，这就意味着所有的服务器会应用相同的日志序列集到自己的状态机中。并且是按照相同的顺序。

* + 1. 跟随者和候选人崩溃

到目前为止，我们都只关注了领导者崩溃的情况。跟随者和候选人奔溃后的处理方式比领导人要简单的多，并且它们的处理方式是相同的。如果跟随者或者候选人崩溃了，那么，后续发给他们的RPCs都会失败。Raft中处理这种失败就是通过无限重试；如果崩溃的机器重启了，那么，这些RPC就会完整的成功。如果一个服务器在完成了一个RPC，但还没有响应的时候崩溃了，那么，在它重新启动之后就会再次收到同样的请求。Raft的RPCs都是幂等的，所以这样重试不会造成任何问题。例如一个跟随者如果收到append log的请求，但是它已经包含了这一日志，那么他就会直接忽略这个请求。

* + 1. 时间和可用性

Raft的要求之一就是，它的安全性不能依赖时钟。整个系统不能因为某些事件运行的比预期快一点或者慢一点就产生了错误的结果。但是，可用性（系统可以及时的响应的客户端）不可避免的要依赖于时间。例如，如果消息交换比服务器故障的时间长，候选人将没有足够长的时间来赢得选举；没有一个稳定的领导者，Raft将无法工作。

领导者选举是Raft对时间要求最为关键的方面。Raft可以选举并维持一个稳定的领导人，只要系统满足以下的时间要求：

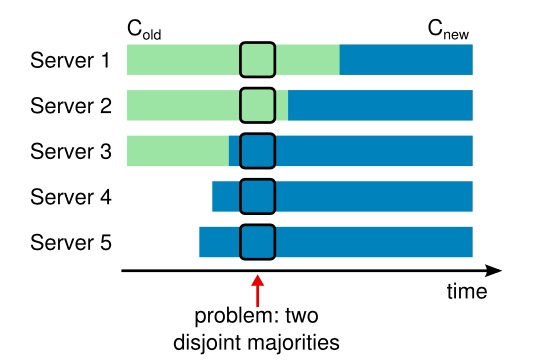
广播时间（broadcastTime） << 选举超时时间（electionTimeout）<< 平均故障间隔时间（MTBF）

在这个不等式中，广播时间指的是一个服务器并行的发送RPCs给集群中的其他服务器并接收响应的时间；选举超时时间就是在5.2节中介绍的选举的超时时间限制。平均故障间隔时间就是对于一台服务器而言，两次故障之间的平均时间。广播时间必须比选举超时时间小一个数量级，这样领导者才能够稳定的发送心跳来阻止跟随者开始进入选举状态；通过随机化选举超时时间的方法，这个不等式也使得瓜分选票变得不可能。选举超时时间应该要比平均故障时间小上几个数量级，这样整个系统才能稳定的运行。当领导人崩溃后，整个系统会大约选举超时的时间里不可用。因此，选举超时时间可能需要在10ms到500ms之间。大多数的服务器的平均故障时间都在几个月甚至更长，很容易满足时间的需求。

* 1. 集群成员变化

到目前为止，我们都假设集群的配置（加入到一致性算法的服务器集合）是固定不变的。但是在实践中，是会偶尔改变集群的配置的，例如替换那些复制的机器或者替换复制级别。尽管可以通过暂停整个集群，更新所有配置，然后重启整个集群的方式来实现，但是在更改的时候，集群会不可用。另外如果存在手动操作步骤，那么就会有操作失误的风险。为了避免这样的问题，我们决定自动化配置改变并且将其纳入到一致性算法中来。

未来让配置修改机制能够安全，那么在转换过程中不能够存在任何时间点使得两个领导人同时被选举成功在同一个任期里。不幸的是，任何服务器直接从旧的配置直接转换到新的配置的方案都是不安全的，一次性原子的转换所有的服务器是不可能的，所以整个转换期间整个集群存在划分为两个独立的大多数群体的可能性。（见图10）



直接从一种配置转换到新的配置是十分不安全的，因为各个机器可能在任何的时候进行转换，在这个例子中，集群的配置从3台变成了5台，不幸的是，存在这样一个时间点，两个不同的领导人在同一个任期里可以被选举成功。一个是通过旧的配置。一个是通过新的配置。

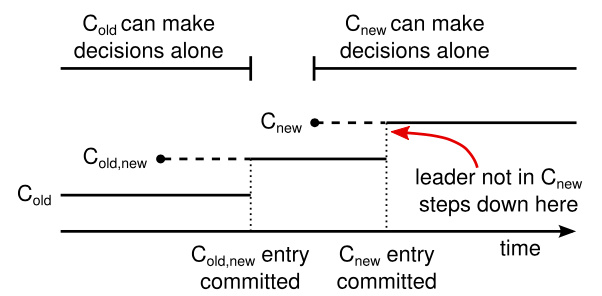
为了保证安全性，配置更改必须使用两阶段方法，目前有很多种两阶段的实现。例如，有些系统在第一阶段停掉旧配置所以集群不能处理客户端请求；然后在第二阶段启用新的配置。在Raft中，集群先阶段到一个过渡的配置，我们称之为共同一致；一旦共同一致已经被提交了，那么，系统就切换到新的配置上。共同一致是新配置和老配置的结合：

1. 日志条目被复制给集群中新、老配置的所有服务器
2. 新旧配置的服务器都可以成为领导人。
3. 达成一致（针对选举和提交）需要分别在两种配置上获得大多数的支持

共同一致允许独立的服务器在不影响安全性的情况下，在不同的时间进行配置转换过程。此外，共同一致可以让集群转换过程中依然响应客户端的请求。

集群配置在复制日志中以特殊的日志条目来存储和通信；图11展示了配置转换的过程，当一个领导人收到一个改变配置从c-old到c-new的请求，它会为了共同一致存储配置（c-old-new），以前面描述的日志条目和副本的方式。一旦一个服务器将新的配置日志条目添加到它的日志中，它就会用这个配置来做出未来所有的决定（服务器总是使用最新的配置，无论它是否已被提交）。这意味着领导者要使用c-old-new规则来决定日志条目c-old,new什么时候需要被提交。如果领导者崩溃了，被选出来的新领导人的配置可能是c-old,也可能是c-old-new，这取决于赢得选举的候选人是否收到了c-old-new的配置，在任何情况下，c-new这一配置都不会单独做出决定。

一旦c-old-new被提交，无论是c-old还是c-new，在没有其他人统一的情况下都不能做出决定，并且领导人完全特性保证了只有拥有c-old-new日志条目的服务器才有可能被选举为领导人。找个时候，领导人创建一条c-new配置的日志条目并复制给集群就是安全的了。再者，每个服务器在见到新的配置的时候会立即生效。当新的配置在c-new的规则下被提交，旧的配置就变得无关紧要，同时不使用新的配置的服务器就可以被关闭了。如图-11，c-old,和c-new没有任何机会同时做出单方面的决定，这保证了安全性。



关于重新配置还有三个问题需要提出。

第一个问题是，新的服务器可能初始化没有存储任何信息的日志条目。当这些服务器以这种状态加入集群中时，那么，他们需要一段时间来更新追赶，这时还不能提交新的日志条目。为了避免这种可用性的间隔时间，Raft在更新配置之前使用了一个额外的阶段，在这个阶段，新的服务器以没有投票权的身份加入到新的集群中来。（领导人复制日志给他们，但是不考虑他们是大多数）。一旦新的服务器追赶上了集群中的其他机器，重新配置可以和上面一样处理。

第二个问题是，集群的领导人可能不是新配置的一员。在这种情况下，在这种情况下，领导者会在提交c-new之后退位（回到跟随者的状态）。这意味着有这样的一段时间，领导人管理着集群，但不包括他自己；它复制日志，但不把自己当作大多数之一，当c-new被提交时，会发生领导人过渡，因为这时是最早新的配置可以独立工作的时间点（将总是能够在c-new配置下选出新的领导人）。在此之前，只能从c-old中选出候选人。

第三个问题是，移除不在c-new中的服务器可能会扰乱集群。这些服务器可能不会再收到心跳。所以当选举超时，他们就会进行新的选举。他们会发送拥有新的任期号的请求投票RPCs，这样会导致当前的领导人回退成跟随者状态。新的领导人最终会被选举出来，但是被移除的服务器会再次超时，然后这个过程会再次重复，导致整体可用性大幅下降。

为了避免这个问题，当服务器确认领导人存在时，服务器会忽略请求投票RPCs，特别的，当服务器在当前最小选举超时时间内只收到一个请求投票RPC，他不会更新当前的任期号或投出选票，这不会影响正常的选举，每个服务器在开始选举之前，至少等待一个最小选举超时时间，然而，这有利于被移除的服务器的干扰：如果领导人能够发送心跳给集群，他们他就不会被更大的任期号所废黜。

* 1. 日志压缩

Raft日志在正常的操作中不断的增长。但是在实际的系统中，日志不能无限制的增长。随着日志不断增长，他们会占用越来越多的空间，花费越来越多的时间来重置。如果没有一定的机制去清除日志里日积月累的陈旧的信息，那么，会带来可用性的问题。

快照是最简单的压缩方法。在快照系统中，整个系统的状态都以快照的形式写入到稳定的自动化存储中，然后到那个时间点之前的日志全部丢弃。快照技术被使用在Chubby和Zookeeper中，接下来的章节会介绍Raft中的快照技术。

增量压缩的方法，例如日志清理或者日志结构合并树，都是可行的。这些方法每次只对一小部分数据进行操作。这样就分散了压缩的负载压力。首先，他们会选择一个已经积累的大量已经被删除或者被覆盖对象的区域，然后重写那个区域还活跃的对象，之后释放那个区域。和简单操作整个数据集合的快照相比，需要增加复杂的机制来实现。状态机可以实现LSM tree使用和快照相同的接口，但是日志清除方法就需要修改Raft了。

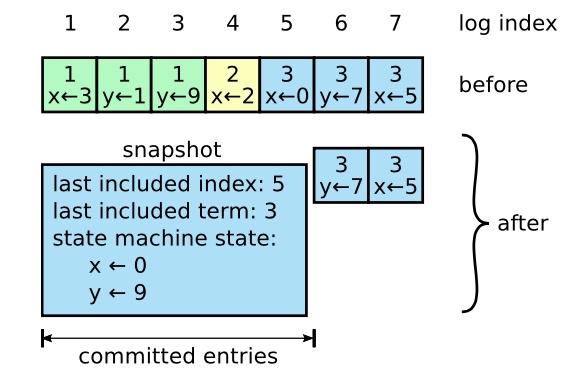


图12中展示了Raft中快照的基础思想。每个服务器独立的创建快照，只包括已经被提交的日志。主要的工作包括将状态机的状态写入到快照中。Raft也包含一些少量的元数据到快照中：最后被包含索引指的是最后的条目在日志中的索引值（状态机最后应用的日志），最后包含的任期指的是该条目的任期号。保留这些数据是为了支持快照后紧接着第一个条目的附加日志请求时的一致性检查，因为这个条目需要前一日志的索引值和任期号。为了支持集群成员变更（第6节），快照中也将最后的一次配置作为最后一个条目存了下来。一旦服务器完成一次快照，那就可以删除最后索引位置之前的所有日志和快照了。

尽管通常服务器都是独立的创建快照，但是领导者必须偶尔的发送快照给有些落后的跟随者。这通常发生在当领导人已经丢弃了下一条需要发送给跟随者的日志条目的时候，幸运的是这种情况不是常规操作：一个与领导人保持同步的跟随者通常都会有这个条目。然而一个运行非常缓慢的跟随者或者新加入集群的服务器（第6节）将不会有这个条目。这时让这个跟随者更新到最新状态的方式就是通过网络发送给它们。

安装快照RPC:

|  |  |
| --- | --- |
| Term | 领导人任期号 |
| LeaderId | 领导者ID |
| lastIncludedIndex | 快照中包含的最后日志条目的索引值 |
| LastIncludedTerm | 快照中包含的最后日志条目的任期号 |
| Offset | 分块在快照中的字节偏移量 |
| Data[] | 从偏移量开始的快照分块的原始字节 |
| Done | 如果这是最后一个分块则为true |

结果

|  |  |
| --- | --- |
| 参数 | 解释 |
| Term | 当前任期号，方便领导人更新自己 |

接收者实现：

1. 如果term < currentTerm就立即回复
2. 如果是一个新的分块（offset=0）就创建一个新的快照
3. 在指定的偏移量写入数据
4. 如果done=false,则继续等待更多的数据
5. 保存快照文件，丢弃具有较小索引的任何现有或部分快照
6. 如果现存的日志条目与快照中最后包含的日志条目具有相同索引值和任期号，则保留其后的日志条目并进行回复。
7. 丢弃整个日志
8. 使用快照重置状态机（并加载快照的集群配置）

图13是一个关于快照安装的简单概述。为了方便传输，快照都是被分成分块的。每个分块都给了跟随者生命的迹象，所以跟随者可以重置选举超时计时器。

在这种情况下领导者使用一种叫做安装快照的新的RPC来发送快照给太落后的跟随者；见图13。当跟随者通过这种RPC接收到快照时，它必须决定对于已经存在日志如何处理。通常快照会包含没有在接收者日志存在的信息。在这种情况下，跟随者会丢弃其整个日志；它全部被快照取代，并且可能包含与快照冲突的未提交条目。如果接收到的快照是自己日志的前面部分（由于网络重传或者错误），那么被快照包含的条目会被全部删除，但是快照后面的条目仍然有效，必须保留。

这种快照的方式背离了Raft的强领导人原则，即只有领导人创建快照，因为跟随者可以在不知道领导人的情况下创建快照。但是我们认为这种背离是值得的。领导人的存在，是为了解决在达成一致性时候的冲突，但是在创建快照的时候，一致性已经达成，这时不存在冲突了，所以没有领导人也是可以的。数据依然是从领导人传给跟随者，只是跟随者可以重新组织他们的数据了。

我们考虑过一种替代的基于领导人的快照方案。即只有领导人创建快照，然后发送给所有的跟随者。但是这样做有两个缺点。第一，发送快照会浪费网络带宽并且延缓了快照处理的时间。每个跟随者都已经拥有了虽有产生快照的信息，而且很显然，自己从本地的状态中创建快照比通过网络接收发送过来的经济。第二，领导人的实现会更加复杂。例如，领导人需要发送快照的同时将新的日志发送给跟随者，这样才不会阻塞新的客户请求。

还有两个问题影响了快照的性能。首先，服务器必须决定什么时候创建快照。如果过于频繁的创建快照，那么就会浪费大量的磁盘带宽和其他资源；如果创建的频率过低，它就要承担耗尽存储容量的风险，同时也增加了从日志重建的时间。一个简单的策略就是当日志达到一个固定大小的时候就创建一次快照。如果这个阈值设置的明显大于期望的快照的大小，那么快照对于磁盘压力的影响就会很小了。

第二个影响性能的问题就是，写入快照需要花费显著的一段时间，并且我们还不希望影响到正常的操作。解决方案是通过写时复制的技术（CopyOnWrite）,这样新的更新就可以被接收而不影响到快照。例如，具有函数式数据结构的状态机天然支持这样的功能。另外，操作系统写时复制技术的支持（如Linux上的fork）可以被用来创建完整的状态机的内存快照（我们的实现就是这样的）。

* 1. 客户端交互

这一节将介绍客户端是如何与Raft交互的，包括客户端如何发现领导人和Raft是如何支持线性化语义的。这些问题对于所有基于一致性的系统都存在，并且Raft的解决方案和其他的也差不多。

Raft中的客户端发送所有请求给领导人。当客户端启动的时候，它会随机挑选一个服务器进行通信，如果客户端第一次挑选的服务器不是领导人，那么这个服务器会拒绝客户端的请求，并且会提供它最近接收到的领导人的信息（append log请求包含了领导人的网络地址）。如果领导人崩溃了，那么客户端的请求就会超时，客户端之后会再次重试随机挑选服务器的过程。

我们Raft的目标是实现线性化语义（每一次操作立即执行，只执行一次，在它调用和收到回复之前）。但是，如上述，Raft是可以执行同一条命令多次的：例如，如果领导者在提交这条日志之后，但是在响应客户端之前崩溃了，那么，客户端会和新的领导人重试这条指令，导致这条指令被再次执行了。解决方案是客户端对于每一条指令都赋予一个唯一的序列号。然后状态机跟踪每条指令最新的序列号和相应的响应。如果接收到一条指令，它的序列号已经被执行了。那么就立即返回结果，而不重复执行指令。

只读的操作可以直接处理而不需要记录日志。但是，在不增加任何限制的情况下，这么做可能会冒着返回脏数据的风险，因为领导人响应客户端请求的时候可能已经被新领导人作废了，但是他还不知道。线性化的读操作必须不能返回脏数据。Raft需要使用两个额外的在不使用日志的情况下保证这一点。

首先，领导人必须有关于被提交日志的最新信息。领导人完全特性保证了领导人一定拥有所有已经被提交的日志条目。但是在他任期开始的时候，他可能不知道哪些是已经被提交的。为了知道这些信息，他需要在他的任期里提交一条日志条目。Raft通过领导人在任期开始的时候提交一个空白的没有任何操作的日志条目到日志中去来实现。

第二，领导人在处理只读的请求之前必须检查自己是否已经被废黜了（他自己的信息已经变脏了，如果一个更新的领导人被选举出来）。Raft中通过让领导人在响应只读请求之前，先和集群中的大部分节点交换一次心跳来处理这个问题。可选的，领导人可以依赖心跳机制来实现一种租约的机制，但是这种方法依赖时间来保证安全性（假设时间误差是有界的）。

* 1. 算法实现和评估

我们已经为RAMCloud实现了Raft算法作为存储配置信息的复制状态机的一部分。并且帮助RAMCloud协调故障转移。这个Raft实现大约包含2000行C++代码，其中不包括测试、注释和空行。这些代码是开源的。同时也有大约25个其他独立的第三方的基于这篇论文草稿的开源实现，针对不同的开发场景。同时，很多公司部署了基于Raft的系统。

这一节会从三个方面来评估Raft算法：可理解性、正确性 和性能。

* + 1. 可理解性

为了和Paxos比较Raft算法的可理解能力，我们针对高层次的本科生和研究生，在斯坦福大学的高级操作系统课程和加州大学伯克利分校的分布式计算课程上，进行了一次学习的实验。我们分别拍了针对Paxos和Raft的视频课程，并准备了响应的小测验。Raft的视频讲课覆盖了这篇论文的所有内容包括日志压缩；Paxos讲课包含了足够的资料来创建一个等价的复制状态机，包括单决策 Paxos,多决策Paxos，重新配置和一些实际系统需要的性能优化（例如领导者选举）。

小测试测试一些对算法的基本理解和理解一些边角的示例。每个学生都是看完第一个视频，回到相应的测试，再看第二个视频，回答相应的测试。大约有一半的学生先进行 Paxos部分，然后另一半先进行Raft部分，这是为了说明两者从第一部分的算法学习中获得的表现和经验的差异。我们计算参加人员的每一个小测验的得分，来看参与者是否在Raft算法上更加容易理解。

我们尽可能的使得 Paxos 和 Raft 的比较更加公平。这个实验偏爱 Paxos 表现在两个方面：43 个参加者中有 15 个人在之前有一些 Paxos 的经验，并且 Paxos 的视频要长 14%。如表格 1 总结的那样，我们采取了一些措施来减轻这种潜在的偏见。我们所有的材料都可供审查。

* + 1. 正确性

在第5节，我们已经制定了正式的规范，和对一致性的安全性证明。这个正式规范使用 TLA+ 规范语言使图 2 中总结的信息非常清晰。它长约400行，并作为证明的主题。同时对于任何想实现 Raft 的人也是十分有用的。我们通过 TLA 证明系统非常机械的证明了日志完全特性。然而，这个证明依赖的约束前提还没有被机械证明（例如，我们还没有证明规范的类型安全）。而且，我们已经写了一个非正式的证明关于状态机安全性是完备的，并且是相当清晰的（大约 3500 个词）。

* + 1. 性能

Raft和其他一致性算法例如Paxos有着差不多的性能。在性能方面，最重要的关注点是，当领导人被选举成功时，什么时候复制新的日志条目。Raft通过很少数量的消息包（一轮从领导人到集群大多数机器的消息）就达成了找个目的。同时，进一步提升Raft的性能也是可行的。例如，很容易通过支持批量操作和管道操作来提高吞吐量和降低延迟。对于其他一致性算法已经提出过的很多优化方案，其中很多也可以应用到Raft中来，但是我们暂时把这个问题放到未来的工作中去。

我们使用我们自己的Raft实现衡量Raft领导人选举的性能并回答领个问题。首先，领导者选举的过程收敛是否快速？第二，在领导者宕机之后，最小的系统宕机时间是多久？

图16：发现并替换一个已经崩溃的领导者的时间。上面的图考察了在选举超时时间上的随机化程度，下面的图考察了最小选举超时时间。每条线代表了1000次实验，和相应的确定的选举超时时间。例如，150-155毫秒意思是，选举超时时间从这个区间范围内随机选择并确定下来。这个实验在一个拥有 5 个节点的集群上进行，其广播时延大约是 15 毫秒。对于 9 个节点的集群，结果也差不多。

为了衡量领导者选举，我们反复的使一个5个节点的服务器集群的领导人宕机，并计算需要多久才能发现领导人已经宕机并选举出一个新的领导人。（见图16），为了构建一个最坏的场景，在每一次尝试里，服务器都有不同长度的日志，意味着有些候选人是没有成为领导人的资格的。另外为了促成选票瓜分的情况，我们的测试脚本在终止领导人之前同步的发送了一次心跳广播。（这大约和领导人在崩溃前复制一个新的日志给其他机器很像），领导人均匀的，随机的在心跳间隔里宕机，也就是最小选举超时时间的一半。因此，最小宕机时间大约就是最小选举超时时间的一半。

图16中上面的图表明，只需要在选举超时时间上使用很少的随机化，就可以大大避免选票被瓜分的情况。在没有随机化的情况下，在我们的测试里，选举过程往往都需要花费10秒的时间，仅仅增加 5毫秒的随机化时间，就大大改善了选举过程，现在平均的宕机时间只有287毫秒。增加更多的随机化时间可以大大改善最坏情况：通过增加50毫秒的随机化时间，最坏的完成情况（1000次尝试）只要513毫秒。

图 16 中下面的图显示，通过减少选举超时时间可以减少系统的宕机时间。在选举超时时间为 12-24 毫秒的情况下，只需要平均 35 毫秒就可以选举出新的领导人（最长的一次花费了 152 毫秒）。然而，进一步降低选举超时时间的话就会违反 Raft 的时间不等式需求：在选举新领导人之前，领导人就很难发送完心跳包。这会导致没有意义的领导人改变并降低了系统整体的可用性。我们建议使用更为保守的选举超时时间，比如 150-300 毫秒；这样的时间不大可能导致没有意义的领导人改变，而且依然提供不错的可用性。

* 1. 相关工作

已经有很多关于一致性算法的工作被发表出来，其中很多都可以归到下面的类别中：

* Lamport关于Paxos的原始描述，和尝试描述的更清晰。
* 关于Paxos的更详尽的描述，补充遗漏的细节并修改算法，使得可以提供更加容易的实现基础
* 实现一致性算法的系统，例如Chubby，Zookeeper和Spanner。对于Chubby和Spanner的算法并没有公开发表其技术细节，尽管他们声称是基于Paxos的。Zookeeper的算法细节已经发表，但是和Paxos着实有着很大的差别。
* Paxos可以应用的性能优化
* Oki和Liskov的Viewstamped Replication（VR），一种和Paxos差不多的替代算法，原始的算法描述和分布式传输协议耦合在一起，但是核心的一致性算法在最近的更新里被分离了出来。VR使用了一种基于领导人的算法，和Raft有很多相似之处。

Raft和Paxos最大的不同之处在于它的强领导性，Raft使用领导者选举作为一致性协议里必不可少的一部分，并且将尽可能多的功能集成到了领导者身上。这样就可以使得算法更加容易理解。例如，在Paxos中，领导人选举和基本的一致性是正交的：领导人选举仅仅是性能优化的手段，而不是一致性所必须要求的。但是，这样就增加了多余的机制：Paxos同时针对基本一致性要求的两阶段提交和针对领导人选举的独立的机制。想比较而言，Raft直接将领导者选举纳入到一致性算法中，并作为两阶段一致性的第一步。这就减少了很多的机制。

像Raft一样，VR和Zookeeper也是基于领导者的，因此他们也拥有一些Raft的优点。但是，Raft比VR和Zookeeper拥有更少的机制，因为Raft尽可能地减少了非领导者的功能。例如，Raft中日志条目都遵循从领导者发送给其他人这一方向：append log RPC是向外发送的。在VR中，日志条目的流向是双向的（领导人可以在选举过程中接收日志）；这就导致了额外的机制和复杂性。根据Zookeeper公开的资料看，他的日志条目也是双向流动的，但是它的实现更像Raft。

和上述我们提及的其他基于一致性的日志复制算法中，Raft的消息类型更少，例如，我们数了一下VR和Zookeeper使用的用来维护一致性需要和成员改变的消息数（排除了客户端交互和日志压缩，因为这些都比较独立，且和一致性关系不大），VR和ZK都定义了10种不同的消息类型，相对的，Raft只有4种消息类型，（两种RPC请求和对应的响应）。Raft的消息都稍微比其他算法的信息量大，但是都很简单。另外，VR和ZK都在领导人改变时传输了整个日志；所以为了能够实践中使用，额外的消息都很必要了。

Raft的强领导人模型简化了整个算法，但是也排斥了一些性能优化的方法。例如，平等主义Paxos（EPaxos）在没有领导人的情况下达到了很高的性能。EPaxos充分发挥了在状态机指令中的交换性。任何服务器都可以在一轮通信下就提交指令，除非其他指令被同时提出了。然而，如果指令都是并发的提出，并且相互之间都不通信沟通，那么，EPaxos就需要额外的一轮通信。因为任何服务器都可以提交指令，所以EPaxos在服务器之间的负载均衡做的很好，并且容易在WAN网络下获得很低的延迟。但是，它在Paxos上增加了非常明显的复杂性。

一些集群成员变换的方法已经被提出或者在其他的工作中被实现，包括 Lamport 的原始的讨论，VR 和 SMART。我们选择使用共同一致的方法因为他对一致性协议的其他部分影响很小，这样我们只需要很少的一些机制就可以实现成员变换。Lamport 的基于 α 的方法之所以没有被 Raft 选择是因为它假设在没有领导人的情况下也可以达到一致性。和 VR 和 SMART 相比较，Raft 的重新配置算法可以在不限制正常请求处理的情况下进行；相比较的，VR 需要停止所有的处理过程，SMART 引入了一个和 α 类似的方法，限制了请求处理的数量。Raft 的方法同时也需要更少的额外机制来实现，和 VR、SMART 比较而言。

* 1. 结论

算法的设计通常会把正确性、效率和简洁作为主要目标。尽管这些都是很有意义的目标，但是我们相信，可理解性也是一样的重要。在开发者把算法应用到实际的系统中之前，这些目标没有一个会被实现，这些都会必然的偏离发表时的形式。除非开发人员对这个算法有着很深的理解，并且有着直观的感觉，否则将会对他们而言，很难在实现的时候保持原有期望的特性。

在这篇论文中，我们尝试解决分布式一致性问题。但是作为一个广为接受但是十分令人费解的算法，Paxos已经困扰了无数学生和开发者很多年了。我们创造了一种新的算法Raft，显而易见比Paxos更容易理解。我们同时也相信，Raft也可以为实际的实现提供坚实的基础，把可理解性作为设计的目标改变了我们设计Raft的方式；随着设计的进展，我们发现自己重复使用了一些技术，比如分解问题和简化状态空间。这些技术不仅提升了Raft的可理解性，同时也使我们坚信其正确性。

* 1. 感谢

略

1. **参考**

略