寻找可理解的共识算法（扩展版）。

Diego Ongaro和John Ousterhout
斯坦福大学

摘要

筏子是一种共识算法，用于管理一个复制的日志。它产生的结果等同于（多）Paxos，它和Paxos一样高效，但它的结构与Paxos不同；这使得Raft比Paxos更容易理解，也为构建实用系统提供了更好的基础。为了提高可理解性，Raft分离了共识的关键要素，如领导者选举、日志复制和安全，它执行了更强的一致性，以减少必须考虑的状态数量。一项用户研究的结果表明，Raft比Paxos更容易让学生学习。Raft还包括一个改变集群成员资格的新机制，该机制使用重叠的主体来保证安全。

1 简介

共识算法允许一个机器的集合作为一个连贯的团体工作，它可以在一些成员的失败中幸存下来。正因为如此，它们在构建可靠的大规模软件系统中发挥了关键作用。在过去的十年中，Paxos[15, 16]主导了关于共识算法的讨论：大多数共识的实现都是基于Paxos或受其影响，而Paxos已经成为教学生了解共识的主要工具。

不幸的是，Paxos是相当难理解的，尽管有许多尝试使它更容易接近。此外，它的架构需要复杂的变化来支持实际的系统。因此，系统建设者和学生都在为Paxos而奋斗。

在与Paxos的斗争中，我们开始寻找一种新的共识算法，为系统建设和教育提供一个更好的基础。我们的方法是不寻常的，因为我们的主要目标是*可理解性*：我们能否为实用系统定义一个共识算法，并以一种比Paxos更容易学习的方式描述它？此外，我们希望该算法能够促进直觉的发展，这对系统建设者来说是至关重要的。重要的是，不仅要让算法发挥作用，而且要让它显而易见，为什么它能发挥作用。

这项工作的结果是一种名为Raft的共识算法。在设计Raft时，我们应用了一些特殊的技术来提高可理解性，包括分解（Raft将领导者选举、日志复制和安全分开）和减少状态空间（相对于Paxos，Raft减少了非确定性的程度和服务器之间的不一致的方式）。对两所大学的43名学生进行的用户研究表明，Raft明显比Paxos更容易理解：在学习了两种算法后，这些学生中有33人能够更好地回答有关Raft的问题，而不是有关Paxos的问题。

Raft在许多方面与现有的共识算法（最值得注意的是Oki和Liskov的Viewstamped Replication[29, 22]）相似，但它有几个新的特点。

- 强大的领导者。与其他共识算法相比，Raft使用了一种更强的领导者形式。例如，日志条目只从领导者流向其他服务器。这简化了对复制日志的管理，使Raft更容易理解。
- 领导人选举。筏子使用随机的计时器来选举领导人。这只增加了少量的在任何共识算法中已经需要的心跳机制，同时简单而快速地解决冲突。
- 成员变化。Raft改变集群中的服务器集的机制使用一个新的*联合共识*方法，其中两个不同配置的多数在过渡期间重叠。这使得集群在配置变化期间能够继续正常运行。

我们认为Raft比Paxos和其他consensus算法更胜一筹，无论是出于教育目的还是作为实施的基础。它比其他算法更简单，更容易理解；它的描述足够全面，可以满足实际系统的需要；它有几个开源的实现，并被几个公司使用；它的安全属性已经被正式规定和证明；它的效率与其他算法相比是很高的。

本文的其余部分介绍了复制的状态机问题（第2节），讨论了Paxos的优点和缺点（第3节），描述了我们对可理解性的一般方法（第4节），介绍了Raft共识算法（第5-8节），评估了Raft（第9节），并讨论了相关工作（第10节）。

2 复制的状态机

共识算法通常出现在以下情况下*复制的状态机*[37]。在这种方法中，服务器集合上的状态机计算相同状态的相同副本，即使一些服务器停机也能继续运行。复制的状态机是

本技术报告是[32]的扩展版本；额外的材料在空白处用灰条注明。发表于2014年5月20日。

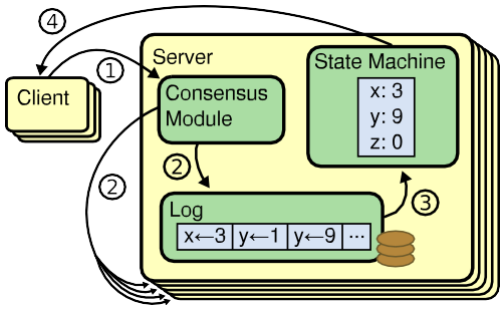


图 1 : 复制的状态机架构。consensus算法管理着一个包含来自客户的状态机命令的复制日志。状态机处理来自日志的相同的命令序列，因此它们亲...
形成相同的产出。

用来解决分布式系统中的各种容错问题。例如，拥有单一集群领导者的大规模系统，如GFS[8]、HDFS[38]和RAMCloud[33]，通常使用一个单独的复制状态机来管理领导者的选举，并存储必须在领导者崩溃后生存的配置信息。复制状态机的例子包括Chubby [2] 和 ZooKeeper [11]。

复制的状态机通常使用复制的日志来实现，如图1所示。每个服务器存储一个包含一系列命令的日志，其状态机按顺序执行这些命令。每个日志都包含相同顺序的命令，所以每个状态机处理相同的命令序列。由于状态机是确定性的，每个状态机都计算出相同的状态和相同的输出序列。

保持复制的日志的一致性是共识算法的工作。服务器上的共识模块接收来自客户端的命令，并将它们添加到其日志中。它与其他服务器上的共识模块进行通信，以确保每条日志最终都包含相同顺序的请求，即使一些服务器失败了。一旦命令被正确复制，每个服务器的状态机就会按照日志顺序处理它们，并将结果返回给客户。因此，这些服务器似乎形成了一个单一的、高度可靠的状态机。

实用系统的共识算法通常具有以下特性。

- 在所有非拜占庭条件下，包括网络延迟、分区和数据包丢失，它们都能*确保安全*（永远不会返回错误的重新结果）。
  拓展，以及重新排序。
- 只要大多数服务器都在运行，并能相互通信和与客户通信，它们就能完全发挥作用（*可用*）。因此，一个
  典型的五台服务器的集群可以容忍任何两台服务器的故障。服务器被假定为通过停止而失败；它们后来可能从稳定存储的状态中恢复并重新加入集群。
- 它们不依赖时间来确保一致性。
  日志的及时性：错误的时钟和极端的信息延迟在最坏的情况下会导致可用性问题。
- 在普通情况下，只要集群中的大多数人响应了一个命令，一个命令就可以完成。
  单一回合的远程过程调用；少数缓慢的服务器不需要影响整个系统的性能。

3 帕克索斯有什么问题？

在过去的十年里，Leslie Lamport的Paxos protocol[15]几乎成了共识的代名词：它是课程中最常讲授的协议，大多数共识的实现都以它为起点。Paxos首先定义了一个能够在单一决策上达成协议的协议，例如单一复制的日志条目。我们把这个子集称为*单决策的Paxos*。然后，Paxos结合这个协议的多个实例，以促进一系列的决定，如日志（*多Paxos*）。Paxos既保证了安全性，又保证了有效性，而且它支持集群成员的变化。它的正确性已被证明，而且在正常情况下是有效的。

不幸的是，Paxos有两个显著的缺点。第一个缺点是，Paxos特别难理解。完整的解释[15]在理论上是不透明的；很少有人能成功地理解它，而且是在付出巨大努力之后。因此，已经有一些人试图用更简单的术语来解释Paxos [16, 20, 21]。这些解释集中在单法令子集上，然而它们仍然具有挑战性。在对2012年NSDI的与会者进行的非正式调查中，我们发现很少有人能够适应Paxos，即使在经验丰富的研究人员中。我们自己也在为Paxos挣扎；在阅读了七种简化的解释和设计了我们自己的替代性协议之后，我们才能够理解完整的协议，这个过程几乎花了一年时间。

我们假设，Paxos的不透明性来自于它选择单法令子集作为基础。单法令Paxos是密集而微妙的：它分为两个阶段，没有简单的直观解释，不能被独立理解。正因为如此，很难发展出关于单法令协议为什么能工作的直觉。多Paxos的组成规则增加了大量的额外复杂性和微妙性。我们认为，就多个决定达成共识的整体问题（即一个日志而不是一个条目）可以用其他更直接和明显的方式进行分解。

Paxos的第二个问题是，它没有为建立实际的实施方案提供一个良好的基础。原因之一是没有广泛认同的多Paxos的算法。Lamport的描述主要是关于单法令Paxos的；他勾画了多Paxos的可能方法，但许多细节是错误的。已经有一些尝试来充实和操作Paxos，如[26]、[39]和[13]，但这些尝试都不一样。

从相互之间以及从Lamport的草图中。像Chubby[4]这样的系统已经实现了类似Paxos的算法，但在大多数情况下，它们的细节还没有被公布出来。

此外，Paxos架构对于构建实用系统来说是一个糟糕的架构；这是单法令分解的另一个后果。例如，独立地选择一系列的日志条目，然后将它们拼接成一个连续的日志，这没有什么好处；这只是增加了复杂性。围绕日志设计一个系统更简单、更有效，新的条目以受限的顺序依次递增。另一个问题是，Paxos在其核心中使用了对称的对等方法（尽管它最终建议采用弱的领导形式作为性能优化）。这在只有一个决定的简化世界中是有意义的，但很少有实际的系统使用这种方法。如果必须做出一系列的决定，首先选举一个领导者，然后让领导者协调这些决定，这样做更简单、更快捷。

因此，实际系统与Paxos几乎没有相似之处。每一个实现都是从Paxos开始的，包括实现它的困难，然后再设计一个明显不同的架构。这样做既耗时又容易出错，而了解Paxos的困难更加剧了这个问题。Paxos的公式可能是一个很好的证明其正确性的定理，但真正的实现与Paxos有很大的不同，所以证明的价值不大。以下来自Chubby实现者的评论是典型的。

> Paxos算法的描述与真实世界系统的需求之间存在着巨大的差距，最终的系统将基于一个未开发的系统。
> 经过验证的协议[4]。

由于这些问题，我们得出结论，Paxos并没有为系统建设或教育提供一个良好的基础。鉴于共识在大规模软件系统中的重要性，我们决定看看我们是否能设计出一种比Paxos有更好特性的替代性共识算法。Raft就是这个实验的结果。

4 设计的可理解性

我们在设计Raft时有几个目标：它必须提供一个完整的、实用的系统建设基础，以便大大减少开发人员所需的设计工作量；它必须在所有条件下都是安全的，并且在典型的操作条件下可用；它必须对普通操作有效。但我们最重要的目标和最困难的挑战是*不可理解性*。必须让广大听众能够舒适地理解该算法。此外，还必须有可能发展关于该算法的直觉，以便系统建设者能够在现实世界的实现中进行可预见的扩展。

在Raft的设计中，有许多地方我们必须在备选方法中做出选择。在这些情况下，我们根据可理解性来评估备选方案：解释每个备选方案有多难（例如，它的状态空间有多复杂，它是否有微妙的影响？

我们认识到在这种分析中存在着高度的主观性；尽管如此，我们还是使用了两种普遍适用的技术。第一种技术是众所周知的问题分解方法：在可能的情况下，我们将问题分成独立的部分，可以相对独立地进行解决、解释和理解。例如，在Raft中，我们将领袖选举、日志复制、安全和成员变更分开。我们的第二个方法是通过减少需要考虑的状态数量来简化状态空间，使系统更加连贯，并尽可能消除非确定性。具体来说，不允许日志有漏洞，而且Raft限制了日志相互之间不一致的方式。尽管在大多数情况下，我们试图消除非确定性，但在某些情况下，非确定性实际上提高了可预测性。特别是，随机化的方法引入了非确定性，但它们倾向于通过以类似的方式处理所有可能的选择来减少状态空间（"选择任何；这并不重要"）。我们使用随机化以简化Raft领袖选举算法。

5 筏式共识算法

筏子是一种管理复制的日志的算法，它的目的是第2节中描述的形式。图2概括了该算法的浓缩形式以供参考，图3列出了该算法的关键属性；这些数字的元素将在本节的其余部分逐一讨论。

Raft通过首先选举出一个杰出的*领导者来*实现共识，然后让领导者全权负责管理复制的日志。领导接受来自客户端的日志条目，将其复制到其他服务器上，并告诉服务器何时可以安全地将日志条目应用到他们的状态机。拥有一个领导者可以简化对复制日志的管理。例如，领导者可以决定在哪里放置新的日志条目，而不需要咨询其他服务器，并且数据以一种简单的方式从领导者流向其他服务器。一个领导者可能会失败或与其他服务器断开连接，在这种情况下，会选出一个新的领导者。

考虑到领导者的方法，Raft将consensus问题分解为三个相对独立的子问题，这些问题将在后面的小节中讨论。

- 领袖选举：当现有领袖失败时，必须选择一个新的领袖（第5.2节）。
- 日志复制：领导者必须接受日志条目

国家

所有服务器上的持久性状态。
(在响应RPC之前在稳定的存储上进行更新)

当前服务器看到的 最新术语（初始化为0
在第一次启动时，单调地增加)
在当前任期内获得投票**的**
候选者ID（如果没有则为空）。
log[] 日志条目；每个条目包含命令
为状态机，当领导者收到条目时为
术语（第一个索引为1）。

所有服务器上的易失性状态。

承诺指数（commitIndex）是已知的最高日志条目的
指数。
承诺（初始化为0，单调增加）。
最后应用于状态的最高日志条目的 索引
机器（初始化为0，单调增加）
。

关于领导人的不稳定状态。
(选举后重新初始化)

nextIndex[]
对于每个服务器，下一个日志条目的索
引。
发送到该服务器（初始化为领导者最
后的日志索引+1）。

matchIndex[]
对于每个服务器，最高日志条目的索引
。
已知在服务器上被复制的数据（初始
化为0，单调地增加）。

AppendEntries RPC

由领导者调用以复制日志条目（§5.3）；也作为心跳（
§5.2）使用。

争论。
领导人的任期
leaderIdso追随者可以重定向客户
prevLogIndex紧随其后的日志条目的索引。
新的
prevLogIndex条目的 prevLogTermterm
entries[]要存储的 日志条目（心跳时为空。
为提高效率，可多送一份)
leaderCommitleader的commitIndex

结果。
termcurrentTerm，供领导者自我更新。
successtrue 如果follower包含匹配的条目
prevLogIndex 和 prevLogTerm

接收器的实施。
1. 如果 term < currentTerm，则回复 false (§5.1)
2. 如果日志在prevLogIndex处不包含术语与prevLogTerm
匹配的条目，则回复false（§5.3）。
3. 如果一个现有的条目与一个新的条目相冲突（相同
的索引但不同的术语），请删除现有的条目和后面
的所有条目（§5.3）。
4. 添加日志中尚未出现的任何新条目
5. 如果leaderCommit >
commitIndex，则设置commitIndex =

请求投票RPC

候选人为收集选票而调用（§5.2）。

争论。

term 候选人的任期
candidateId 候选人要求投票
lastLogIndex 候选人最后一条日志记录的索引（§5
lastLogTerm .4）
候选人最后一条日志记录的期限（§5
.4）

结果:
currentTerm，对于候选者来说，自身
期限投票已 更新为true意味着候选者收到了投票。
获批准

接收器的实施。
1. 如果 term < currentTerm，则回复 false (§5.1)
2. 如果 votedFor 是 null 或
candidateId，并且候选人的日志至少与接收者的日志
一样是最新的，则接受投票权（§5.2，§5.4）

服务器的规则

lastApplied]应用于状态机（§5.3）。
- 如果RPC请求或响应包含术语T > currentTerm。设置currentTerm = T，转换为follower（§5.1）。

追随者（§5.2）。
- 对候选人和领导人的RPC作出回应
- 如果选举超时，没有收到现任领导的AppendEntries RPC，也没有授予候选人投票权：转换为候选人

候选人（§5.2）。
- 在转换为候选人时，开始选举。
  - 递增 currentTerm
  - 投给自己
  - 重置选举定时器
  - 向所有其他服务器发送RequestVote RPCs
- 如果获得大多数服务器的投票：成为领导者
- 如果收到来自新领导的AppendEntries RPC：转换为跟随者
- 如果选举超时：开始新的选举

领导人。
- 选举时：向每个服务器发送最初的空AppendEntries RPC（心跳）；在空闲期间重复，以防止选举超时（§5.2）。
- 如果收到来自客户端的命令：将条目追加到本地日志，在条目应用于状态机后做出响应（§5.3）。
- 如果一个追随者的最后日志索引≥nextIndex：发送AppendEntries RPC，从NextIndex开始的日志条目。
  - 如果成功：为跟随者更新nextIndex和matchIndex（§5.3）。
  - 如果AppendEntries因为日志不一致而失败：递减NextIndex并重试（§5.3）。
- 如果存在一个N，使得N>commitIndex，大多数的matchIndex[i]≥N，并且log[N].term==currentTerm：设置commitIndex = N（§5.3，§5.4）。

min(leaderCommit, 最后一个新条目的索引)

图2：Raft共识算法的浓缩摘要（不包括成员变化和日志压实）。左上角方框中的服务器行为被描述为一组独立和重复触发的规则。第5.2节等章节编号表示讨论特定功能的地方。一个正式的规范[31]更精确地描述了Raft算法。

选举安全：在一个特定的任期内最多可以选出一名领导人。§5.2

Leader Append-Only：领导者从不覆盖或删除其日志中的条目；它只附加新条目。§5.3

日志匹配：如果两个日志包含一个具有相同索引和术语的条目，那么这两个日志的所有条目到给定索引都是相同的。§5.3

领导者的完整性：如果一个日志条目在某一条款中被承诺，那么该条目将出现在所有更高编号条款的领导者的日志中。§5.4

状态机安全：如果一个服务器在其状态机上应用了一个给定索引的日志条目，那么其他服务器将永远不会为同一索引应用不同的日志条目。§5.4.3

图 3：Raft保证这些属性中的每一项在任何时候都是真的。节号表示每个属性的讨论位置。

在整个集群中复制它们，迫使其他日志与它自己的日志一致（第5.3节）。

- 安全性：Raft的关键安全属性是图3中的状态机安全属性：如果任何服务器在它的状态机中应用了一个特定的日志条目，那么其他服务器就不能对相同的日志索引应用不同的命令。第5.4节描述了Raft如何确保这一属性；该解决方案涉及对第5.2节中描述的选举机制的额外限制。

在介绍了共识算法之后，本节将讨论可用性问题和系统中的计时作用。

5.1 筏子基础知识

一个Raft集群包含几个服务器；五个是典型的数量，这使得系统可以容忍两个故障。在任何时候，每个服务器都处于三种状态之一：*领导者*、*追随者*或*候选者*。在正常操作中，正好有一个领导者，其他所有的服务器都是追随者。跟随者是被动的：他们自己不发出任何请求，只是对领导者和候选人的请求做出回应。领导者处理所有客户的请求（如果客户联系追随者，追随者将其重定向到领导者）。第三种状态，候选人，被用来选举一个新的领导者，如第5.2节所述。图4显示了这些状态和它们的转换；下面将讨论这些转换。

如图5所示，Raft将时间划分为任意长度的*项*。期限用连续的整数来编号。每个任期以*选举*开始，其中一个或多个候选人试图成为领袖，如第5.2节所述。如果一个候选人在选举中获胜，那么他将在剩下的任期内担任领袖。在某些情况下，选举的结果是分裂票。在这种情况下，任期将在没有领袖的情况下结束；新的任期（重新选举）。

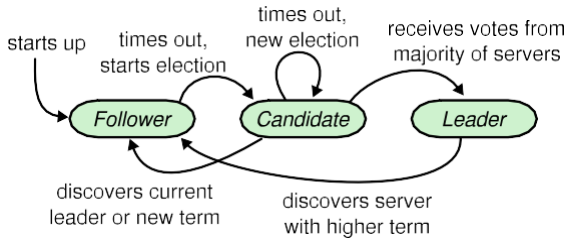


图 4：服务器状态。追随者只对来自其他服务器的请求做出回应。如果一个追随者没有收到任何通信，它就会成为一个候选人并发起选举。收到整个集群中大多数人的投票的候选人成为新领导人。领导人通常会一直运作到失败为止。

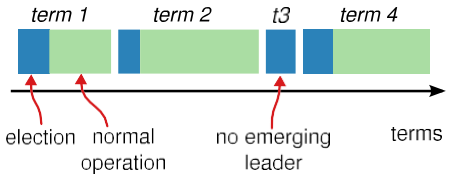


图 5：时间被划分为几个学期，每个学期以选举开始。选举成功后，由一个领导者管理集群，直到任期结束。有些选举失败了，在这种情况下，任期结束时没有选择领导者。在不同的服务器上，可能会在不同的时间观察到术语之间的转换。

将很快开始。筏子确保在一个给定的任期内最多有一个领导人。

不同的服务器可能会在不同的时间观察术语之间的转换，在某些情况下，一个服务器可能不会观察一个选举甚至整个术语。条款在Raft中充当了逻辑时钟[14]，它们允许服务器检测过时的信息，如过时的领导。每个服务器都存储一个*当前的术语*编号，该编号随时间单调地增加。每当服务器进行通信时，就会交换当前条款；如果一个服务器的当前条款比另一个服务器的小，那么它就会将其当前条款更新为较大的值。如果一个候选者或领导者发现它的术语已经过期，它将立即恢复到下面的状态。如果一个服务器收到的请求是一个过时的术语，它将拒绝该请求。

Raft服务器使用远程过程调用（RPCs）进行通信，基本的共识算法只需要两种类型的RPCs。Request Vote RPCs由候选人在选举期间发起（第5.2节），Append Entries RPCs由领导者发起，用于复制日志内容并提供一种心跳形式（第5.3节）。第7节增加了第三个RPC，用于在服务器之间传输快照。如果服务器没有及时收到重新响应，它们会重试RPC，并且为了获得最佳性

能，它们会并行地发出RPC。

5.2 领导人选举

Raft使用心跳机制来触发领导者的电。当服务器启动时，它们开始是跟随者。一个服务器只要收到有效的

来自领导者或候选人的RPC。领导者定期向所有追随者发送心跳（AppendEntries
RPCs，不携带日志条目），以维持他们的权威。如果追随者在一段被称为*选举超时*的时间内没有收到任何通信，那么它就认为没有可行的领导者，并开始选举以选择一个新的领导者。

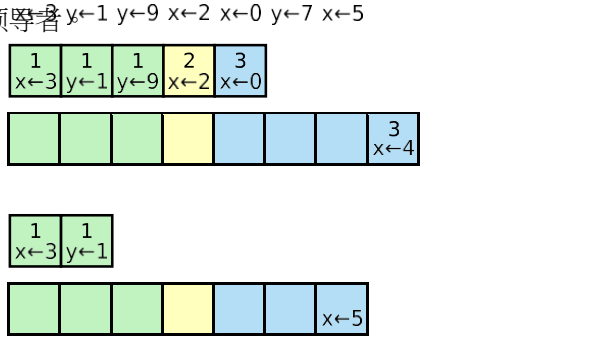
为了开始选举，追随者增加它的当前任期并过渡到候选状态。然后，它为自己投票，并向集群中的每个其他服务器发出RequestVote
RPCs。候选者继续处于这种状态，直到发生三种情况之一。(a)它赢得了选举，(b)另一个服务器确立了自己的领导者地位，或
(c)
一段时间过去了，没有赢家。这些结果将在下面的段落中分别讨论。

如果一个候选人在同一任期内获得了整个集群中大多数服务器的投票，那么它就赢得了选举。每台服务器在给定的任期内最多只为一名候选人投票，以先来后到为原则（注意：第5.4节对投票增加了一个额外的限制）。少数服从多数的规则保证了在某一任期内最多只有一名候选人能够赢得选举（图3中的选举安全原则）。一旦一个候选人在选举中获胜，他就成为领袖。然后，它向所有其他服务器发送心跳信息，以建立其权威并防止新的选举。

在等待投票的过程中，候选人可能会收到另一个服务器的AppendEntries
RPC，声称自己是领导者。如果领导者的任期（包括在其RPC中）至少与候选人的当前任期一样大，那么候选人就会承认领导者是合法的，并返回到跟随者状态。如果RPC中的术语小于候选者当前的术语，那么候选者拒绝RPC并继续处于候选状态。

第三种可能的结果是，一个候选人既没有赢得也没有输掉选举：如果许多追随者同时成为候选人，票数可能被分割，因此没有候选人获得多数票。当这种情况发生时，每个候选人都会超时，并通过增加其任期和启动新一轮的请求-
投票RPC来开始新的选举。然而，如果没有额外的措施，票数分裂可能会无限期地重复。

Raft使用随机的选举超时，以确保分裂投票很少发生，并能迅速解决。为了从一开始就防止分裂投票，选举超时是从一个固定的时间间隔中随机选择的（例如，150-
300ms）。这就分散了服务器，所以在大多数情况下，只有一个服务器会超时；它赢得了选举，并在任何其他服务器超时之前发送心跳信号。同样的机制被用来处理分裂投票。每个候选人在选举开始时重新启动其随机的选举超时，并等待超时过后再开始下一次选举；这减少了在新的选举中再次出现分裂票的可能性。第9.3节显示，这种方法可以迅速选出一个领导者。



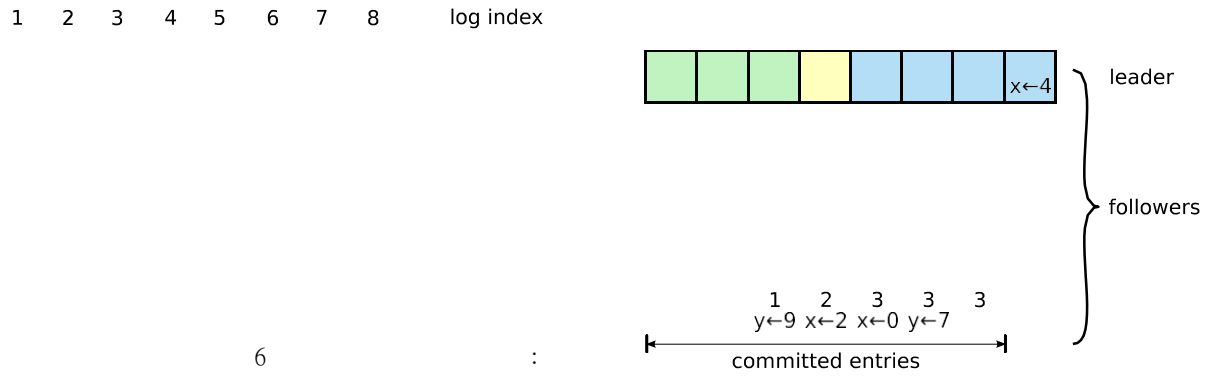


图 6：日志是由条目组成的，这些条目按顺序编号。每个条目都包含创建它的术语（每个方框中的数字）和状态机的命令。一个条目被认为是*承诺的*，如果它对该条目是安全的
条目，以应用于状态机。

选举是一个例子，说明可理解性是如何指导我们在设计方案之间进行选择的。最初我们计划使用一个排名系统：每个候选人被分配一个独特的排名，用来在竞争的候选人之间进行选择。如果一个候选人发现了另一个排名更高的候选人，它就会回到追随者的状态，这样排名更高的候选人就能更容易地赢得下一次选举。我们发现这种方法在可用性方面产生了一些微妙的问题（如果一个排名较高的服务器失败了，一个排名较低的服务器可能需要超时并再次成为候选人，但如果它过早地这样做，它可能会重置选举领导者的进展）。我们对算法进行了多次调整，但每次调整后都会出现新的角落案例。最终我们得出结论，随机重试的方法更加明显和容易理解。

5.3 日志复制

一旦一个领导者被选出，它就开始为客户请求提供服务。每个客户请求都包含一个要由复制的状态机执行的命令。领导者将命令作为一个新条目附加到它的日志中，然后平行地将AppendEntries RPCs发送给其他每个服务器以复制该条目。当条目被安全复制后（如下所述），领导者将条目应用于其状态机，并将执行结果返回给客户端。如果跟随者崩溃或运行缓慢，或者网络数据包丢失，领导者会无限期地重试Append- Entries RPCs（甚至在它回应了客户端之后），直到所有跟随者最终存储所有日志条目。

日志的组织方式如图6所示。每个日志条目都存储了一个状态机命令，以及领导者收到该条目时的术语编号。日志条目中的术语编号被用来检测日志之间的不一致，并确保图3中的一些属性。每个日志条目也有一个整数的索引标识。

说明其在日志中的位置。

领导决定何时将日志条目应用于状态机是安全的；这样的条目被称为"*提交*"。Raft保证所提交的条目是持久的，最终会被所有可用的状态机执行。一旦创建该条目的领导者将其复制到大多数服务器上，该日志条目就会被提交（例如，图6中的条目7）。这也会提交领导者日志中所有之前的条目，包括之前领导者创建的条目。第5.4节讨论了在领导者变更后应用这一规则时的一些微妙之处，它还表明这种承诺的定义是安全的。领导者会跟踪它所知道的已承诺的最高索引，并且它在未来的AppendEntries RPC（包括心跳）中包括该索引，以便其他服务器最终发现。一旦跟随者得知一个日志条目被提交，它就会将该条目应用到它的本地状态机（按日志顺序）。

我们设计的Raft日志机制在不同服务器上的日志之间保持高度的一致性。这不仅简化了系统的行为，使其更具可预测性，而且是确保安全的重要组成部分。Raft维护了以下特性，它们共同构成了图3中的日志匹配特性。

- 如果不同日志中的两个条目具有相同的索引和术语，那么它们存储的是同一个命令。
- 如果不同日志中的两个条目具有相同的索引和术语，那么日志中的所有前面的条目都是相同的。

第一个属性来自于这样一个事实，即一个领导者在一个给定的术语中最多创建一个具有给定的日志索引的条目，并且日志条目永远不会改变它们在日志中的位置。第二个属性由AppendEntries执行的简单一致性检查来保证。当发送AppendEntries RPC时，领导者包括其日志中紧接新条目之前的条目的索引和术语。如果跟随者在其日志中没有找到具有相同索引和术语的条目，那么它将拒绝新条目。一致性检查作为一个归纳步骤：日志的初始空状态满足了日志匹配属性，并且每当日志被扩展时，一致性检查都会保留日志匹配属性。因此，每当AppendEntries成功返回时，领导者知道跟随者的日志与自己的日志在新条目之前是相同的。

在正常运行期间，领导者和追随者的日志保持一致，所以AppendEntries一致性检查不会失败。然而，领导者崩溃会使日志不一致（老领导者可能没有完全复制其日志中的所有条目）。这些不一致会在一系列领导者和追随者的崩溃中加剧。图7说明了追随者的日志与新领导者的日志的不同方式。一个追随者可能

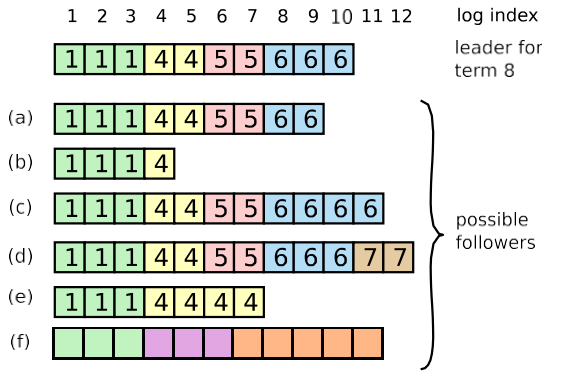


图7：当顶端的领导者掌权时，在追随者的日志中可能会出现（a-f）的任何一种情况。每个盒子代表一个日志条目；盒子里的数字是它的术语。一个追随者可能缺少条目（a-b），可能有额外的未承诺的条目（c-d），或者两者都有（e-f）。举例来说如果该服务器是第二学期的领头羊，在其日志中增加了几个条目，然后在提交任何条目之前就崩溃了；它迅速重新启动，成为第三学期的领头羊，并在其日志中增加了几个条目；在第二学期或第三学期的任何条目被提交之前，该服务器再次崩溃，并在几个学期内一直处于瘫痪状态，则可能发生情景（f）。

遗失的条目在领头人上是存在的，它可能有领头人上不存在的额外条目，或者两者都有。日志中的缺失和不相干的条目可能跨越多个术语。

在Raft中，领导者通过强迫追随者的日志重复自己的日志来处理不一致的情况。这意味着追随者日志中的冲突条目将被领导者日志中的条目覆盖。第5.4节将表明，如果再加上一个限制，这就是安全的。

为了使跟随者的日志与自己的日志保持一致，领导者必须找到两个日志一致的最新日志条目，删除跟随者日志中此后的任何条目，并向跟随者发送此后领导者的所有条目。所有这些动作都是为了响应AppendEntries RPCs所进行的一致性检查而发生的。领导为每个跟随者维护一个*nextIndex*，它是领导将发送给该跟随者的下一个日志条目的索引。当领导者第一次上台时，它将所有的nextIndex值初始化为其日志中最后一条的索引（图7中的11）。如果跟随者的日志与领导者的日志不一致，AppendEntries一致性检查将在下一个AppendEntries RPC中失败。在拒绝之后，领导者会递减NextIndex并重试AppendEntries RPC。最终，nextIndex将达到一个领导者和追随者日志匹配的点。当这种情况发生时，AppendEntries将成功，这将删除跟随者日志中任何冲突的条目，并附加领导者日志中的条目（如果有的话）。一旦AppendEntries成功，追随者的日志就与领导者的日志一致了，并且在剩下的时间里，它将保持这种状态。

如果需要，该协议可以被优化以减少被拒绝的AppendEntries RPC的数量。例如，当拒绝一个AppendEntries请求时，跟随者

可以包括冲突条目的术语和它为该术语存储的第一个索引。有了这些信息，领导者可以递减nextIndex以绕过该术语中的所有冲突条目；每个有冲突条目的术语需要一个AppendEntries RPC，而不是每个条目一个RPC。在实践中，我们怀疑这种操作是否有必要，因为失败不常发生，而且不太可能有许多不一致的条目。

有了这种机制，领导者在上电时不需要采取任何特别的行动来恢复日志的一致性。它只是开始正常的操作，而日志会自动收敛以应对AppendEntries一致性检查的失败。领头羊从不覆盖或删除自己日志中的条目（图3中的领头羊仅应用的属性）。

这种日志复制机制表现出第2节中所描述的理想的共识特性：只要大多数的服务器是正常的，Raft就可以接受、复制和应用新的日志条目；在正常情况下，一个新的条目可以通过一轮RPCs复制到集群的大多数；一个缓慢的跟随者不会影响性能。

5.4 安全问题

前面的章节描述了Raft如何选举领导和复制日志条目。然而，到目前为止所描述的机制还不足以确保每个状态机以相同的顺序执行完全相同的命令。例如，当领导者提交几个日志条目时，一个跟随者可能无法使用，然后它可能被选为领导者，并用新的条目覆盖这些条目；因此，不同的状态机可能执行不同的命令序列。

本节对Raft算法进行了完善，增加了对哪些服务器可以被选为领导者的限制条件。该限制确保了任何给定条款的领导者都包含了之前条款中承诺的所有条目（图3中的领导者完整性属性）。考虑到选举的限制，我们会使承诺的规则更加精确。最后，我们提出一个领导者完整性属性的证明草图，并说明它如何导致复制状态机的正确行为。

5.4.1 选举限制

在任何基于领导者的共识算法中，领导者最终必须存储所有承诺的日志条目。在一些共识算法中，例如Viewstamped Replication [22]，即使最初不包含所有承诺的条目，也可以选出一个领导者。这些算法包含额外的机制来识别缺失的条目，并将它们传送给新的领导者，例如在选举过程中或之后不久。不幸的是，这导致了相当多的额外机制和复杂性。Raft使用了一种更简单的方法，它保证所有先前承诺的条目都是在选举过程中获得的。

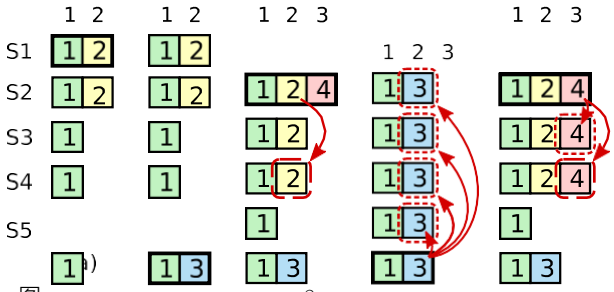


图 8 :
一个时间序列显示了为什么领导者不能使用旧条款的日志条目来确定承诺。在
(a) S1是领导者，部分复制了索引处的日志条目
2.在(b)中，S1崩溃了；S5在S3、S4和它自己的投票中被选为第三任期的领导者，并接受了日志索引2的不同条目。在（c）中，S5崩溃了；S1重新启动，被选为领导者，并继续复制。在这一点上，第2项的日志条目已经在大多数服务器上复制，但它没有被提交。如果S1像(d)那样崩溃，S5可以当选为领导者（由S2、S3和S4投票），并用它自己的第3期日志条目覆盖该条目。然而，如果S1在崩溃前在大多数服务器上复制了其当前任期的一个条目，如(e)，那么这个条目就被承诺了（S5不能赢得选举）。在这一点上，日志中所有前面的条目也被提交。

每一个新的领导者从当选的那一刻起就存在，而不需要将这些条目转移到领导者身上。这意味着日志条目只在一个方向上流动，即从领导者到追随者，而且领导者永远不会在他们的日志中过度写入现有条目。

Raft使用投票程序来防止候选人赢得选举，除非其日志包含所有承诺的条目。候选人必须与集群中的大多数人联系才能当选，这意味着每个已承诺的条目必须至少存在于其中一个服务器中。如果候选人的日志至少和该多数人中的任何其他日志一样是最新的（这里的"最新"在下面有精确的定义），那么它将包含所有承诺的条目。RequestVote RPC实现了这一限制：RPC包括关于候选人日志的信息，如果投票人自己的日志比候选人的日志更及时，则拒绝投票。

Raft通过比较日志中最后条目的索引和术语来确定两个日志中哪一个是最新的。如果日志的最后条目有不同的术语，那么术语较晚的日志是最新的。如果日志以相同的期限结束，那么哪一个日志的期限更长，哪一个就更加最新。

5.4.2 承诺以前的条款的条目

如第5.3节所述，一旦一个条目被存储在大多数服务器上，领导者就知道其当前任期的一个条目被提交。如果一个领导者在提交一个条目之前崩溃了，未来的领导者将试图完成对该条目的复制。然而，

领导者不能立即断定，一旦前一个任期的条目被存储在大多数服务器上，该条目就会被提交。图

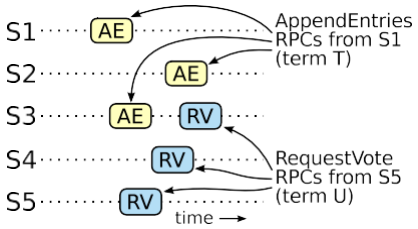


图 9 : 如果S1（任期T的领导者）在其任期内提交了一个新的日志条目，而S5被选为后来任期U的领导者，那么至少有一个服务器（S3）接受了该日志条目，并且也为S5投票。

图8说明了这样一种情况：一个旧的日志条目被存储在大多数服务器上，但仍然可以被未来的领导者所覆盖。

为了消除类似图8中的问题，Raft从不通过计算副本来提交以前的日志条目。只有领导者当前任期的日志条目是通过计算副本来提交的；一旦当前任期的条目以这种方式被提交，那么由于日志匹配特性，所有之前的条目都被间接提交。在某些情况下，领导者可以安全地断定一个较早的日志条目已被提交（例如，如果该条目存储在每台服务器上），但Raft为简单起见采取了更保守的方法。

Raft在承诺规则中产生了这种额外的复杂性，因为当领导者复制以前条款的条目时，日志条目会保留其原始条款编号。在其他共识算法中，如果一个新的领导者重新复制之前 "条款"中的条目，它必须用新的 "条款号"来做。Raft的方法使得对日志条目的推理更加容易，因为它们在不同的时间和不同的日志中保持着相同的术语编号。此外，与其他算法相比，Raft中的新领导者从以前的条款中发送的日志条目更少（其他算法必须发送重新冗余的日志条目，在它们被提交之前对其重新编号）。

5.4.3 安全论证

鉴于完整的Raft算法，我们现在可以ar-更确切地说，领导者完备性原则是成立的（这个论证是基于安全证明的，见第9.2节）。我们假设领导者完备性属性不成立，那么我们就证明一个矛盾点。假设术语T的领导者（$leader_T$）从其术语中提交了一个日志条目，但该日志条目没有被未来某个术语的领导者所存储。请考虑最小的术语U > T，其领导者($leader_U$)不存储该条目。

1. 承诺的条目在当选时必须不在领导者$_U$的日志中（领导者从不删除或改写条目）。
2. 领导者$_T$，在集群的大多数上复制了该条目，领导者$_U$，从集群的大多数上获得投票。因此，至少有一台服务器（"投票者"）既接受了领导者$_T$的条目，又投票给了

领导者$_U$
，如图9所示。选民是达成矛盾的关键。

3. *在*投票给领导者$_U$
*之前，*投票者必须接受领导者$_T$
的承诺条目；否则，它将拒绝领导者$_T$ 的AppendEntries 请求（它的当前任期将高于T）。
4. 投票者在投票给领袖$_U$
，仍然储存了这个条目，因为每一个介入的领袖都包含这个条目（根据假设），领袖从不删除条目，而追随者只有在与领袖冲突时才删除条目。
5. 选民将自己的票授予了领袖$_U$ ，因此领袖$_U$ ，他的日志肯定和选民的一样是最新的。这导致了两个矛盾中的一个。
6. 首先，如果投票人和领导者$_U$
共享相同的最后一个日志项，那么领导者$_U$
的日志肯定至少和投票人的一样长，所以它的日志包含了投票人日志中的每个条目。这是一个矛盾，因为投票人包含了已承诺的条目，而领导者$_U$ 被认为不包含。
7. 否则，领导者$_U$
，他的最后一个对数项一定比投票者的大。此外，它比T大，因为投票人的最后一个日志项至少是T（它包含了T项中的承诺条目）。创建领导者$_U$'s last log entry的早期领导者，在其日志中一定包含了已承诺的条目（根据假设）。那么，根据日志匹配属性，领导者$_U$
的日志也必须包含已承诺的条目，这就是一个矛盾之处。
8. 这就完成了矛盾。因此，所有大于T的术语的领导必须包含术语T的所有条目，这些条目在术语T中被承诺。
9. 日志匹配属性保证了未来的领导也将包含间接承诺的条目，如图8(d)中的索引2。

鉴于领导者完备性属性，我们可以证明图3中的状态机安全属性，即如果一个服务器在其状态机上应用了一个给定索引的日志条目，那么没有其他服务器会在同一索引上应用一个不同的日志条目。当一个服务器在其状态机上应用一个日志条目时，其日志必须与领导者的日志相同，直到该条目，并且该条目必须被提交。现在考虑任何服务器应用一个给定的日志索引的最低条款；日志完整性属性保证所有更高条款的领导者将存储相同的日志条目，因此在以后的条款中应用索引的服务器将应用相同的值。因此，状态机安全属性成立。

最后，Raft要求服务器按照日志的顺序应用条目。结合状态机安全条款，这意味着所有服务器都将以相同的顺序将相同的日志条目应用到他们的状态机中。

5.5 追随者和候选人崩溃

在这之前，我们一直专注于领导者的失败。跟随者和候选者的崩溃比领导者的崩溃更容易处理，而且它们的处理方式都是一样的。如果一个追随者或候选人崩溃了，那么发给它的RequestVote和AppendEntries
RPC就会失败。Raft通过无限期地重试来处理这些失败；如果崩溃的服务器重新启动，那么RPC将成功完成。如果服务器在完成RPC后但在响应前崩溃，那么它将在重新启动后再次收到相同的RPC。筏式RPC是等效的，所以这不会造成任何伤害。例如，如果一个跟随者收到一个AppendEntries请求，其中包括已经存在于其日志中的日志条目，那么它在新的重新探索中会忽略这些条目。

5.6 时间和可用性

我们对Raft的要求之一是安全不能依赖于时间：系统不能因为某些事件发生得比预期快或慢就产生不正确的结果。然而，可用性（系统及时响应客户的能力）必须不可避免地取决于时间。例如，如果信息交流的时间超过了服务器崩溃之间的典型时间，那么候选人就不会保持足够长的时间来赢得选举；没有一个稳定的领导者，Raft就不能取得进展。

领袖选举是Raft中时间最关键的方面。只要系统满足以下*时间要求*，Raft就能选出并维持一个稳定的领导者。

$$broadcastTime \ll selectionTimeout \ll MTBF$$

在这个不等式中，*broadcastTime*是一台服务器向集群中的每台服务器并行发送RPC并接收其响应的平均时间；*electionTimeout*是第5.2节中描述的选举超时；*MTBF*是单台服务器的平均故障间隔时间。广播时间应该比选举超时少一个数量级，这样领导者就可以可靠地发送心跳信息，以防止支持者开始选举；考虑到选举超时使用的随机方法，这种不平等也使得分裂投票不可能发生。选举超时应该比MTBF小几个数量级，这样系统才能稳步前进。当领导者崩溃时，系统将在大约选举超时的时间内不可用；我们希望这只占总体时间的一小部分。

广播时间和平均无故障时间是无关联系统的属性，而选举超时是我们必须选择的。Raft的RPC通常要求接收者将信息持久化到稳定的存储中，因此广播时间可能在0.5ms到20ms之间，这取决于存储技术。因此，选举超时可能是在10ms和500ms之间。典型情况

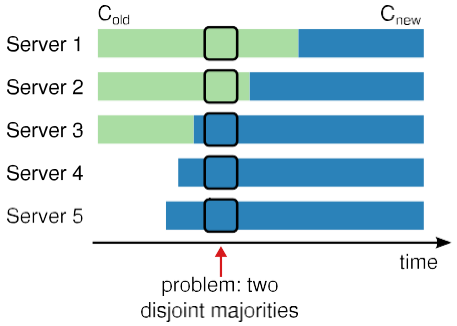


图 10 : 直接从一个配置切换到另一个配置是不安全的，因为不同的服务器会在不同的时间切换。在这个例子中，集群从三台服务器增长到五台。不幸的是，有一个时间点，即
两个不同的领导人可以在同一任期内当选，一个是旧组合的多数（C_{old}），另一个是新组合的多数（C_{new}）。

服务器的平均无故障时间为几个月或更长，这很容易满足时间要求。

6 集群成员的变化

到目前为止，我们假设集群的*配置*（参与共识算法的服务器集合）是固定的。在实践中，偶尔有必要改变配置，例如在服务器故障时更换服务器或改变复制的程度。虽然这可以通过关闭整个集群，更新配置文件，然后重新启动集群来完成，但这将使集群在改变期间不可用。此外，如果有任何手动步骤，就有可能出现操作错误。为了避免这些问题，我们决定将配置变更自动化，并将其纳入Raft共识算法中。

为了使配置变化机制安全，在过渡期间必须没有任何一个点可以让两个领导人在同一任期内当选。不幸的是，任何让服务器直接从旧配置切换到新配置的方法都是不安全的。不可能一次性地切换所有的服务器，所以集群有可能在过渡期间分裂成两个独立的多数（见图10）。

为了确保安全，配置变更必须使用两阶段的方法。实现这两个阶段的方法有很多种。例如，一些系统（如[22]）使用第一阶段禁用旧的配置，使其无法处理客户端请求；然后第二阶段启用新的配置。在Raft中，集群首先切换到一个过渡性配置，我们称之为*联合共识*；一旦联合共识被承诺，系统就会过渡到新的配置。联合共识结合了新旧两种配置。

- 日志条目会被复制到两个构架中的所有服务器。

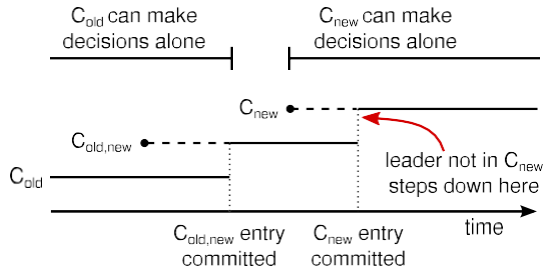


图 11：配置变更的时间线。虚线表示已经创建但未提交的配置条目，实线表示最新提交的配置条目。领导者首先在其日志中创建 $C_{old,new}$ 配置条目，并将其提交给 $C_{old,new}$（C_{old} 的大多数和 C_{new} 的大多数）。然后，它创建C_{new}条目，并将其提交给C_{new}的大多数。在这个时间点上，C_{old}和C_{new}都不能独立做出决定。

- 任何一个配置的服务器都可以作为领导者。
- 达成协议（选举和加入承诺）需要新旧两个组合的单独多数。

联合共识允许单个服务器在不同的时间在配置之间转换，而不需要保证安全。此外，联合共识允许集群在整个配置变化过程中继续为客户请求提供服务。

集群配置是通过复制日志中的特殊条目来存储和交流的；图11说明了配置的变化过程。当领导者收到将配置从C_{old}改为C_{new}的请求时，它将联合共识的配置（图中的$C_{old,new}$）存储为一个日志条目，并使用前面描述的机制复制该条目。一旦某个服务器将新的配置条目添加到其日志中，它就会将该配置用于所有未来的决策（一个服务器总是使用其日志中的最新配置，无论该条目是否被提交）。这意味着领导者将使用 $C_{old,new}$ 的规则来决定 $C_{old,new}$ 的日志条目何时被提交。如果领导者崩溃了，可以在C_{old}或$C_{old,new}$下选择一个新的领导者，这取决于获胜的候选人是否得到了 $C_{old,new}$。在任何情况下，C_{new}都不能在这期间做出单方面的决定。

一旦C_{old}和C_{new}被提交，没有*对方的批准，*C_{old}和C_{new}都不能做出决定，领导者完整性属性确保只有拥有$C_{old,new}$日志条目的服务器才能被选举为领导者。现在领导者创建一个描述C_{new}的日志条目并将其复制到集群中是安全的。同样，这个配置一旦被看到，就会在每个服务器上生效。当新的配置在C_{new}的规则下被提交后，旧的配置就不重要了，不在新配置中的服务器可以被关闭了。如图11所示，没有任何时候C_{old}和C_{new}可以同时做出单边决定；这保证了安全。

在重新配置方面还有三个问题需要解决。第一个问题是，新的服务器最初可能不会存储任何日志条目。如果它们在这种状态下被添加到集群中，可能需要相当长的时间才能赶上，在此期间，可能不可能收集新的日志条目。为了避免可用性差距，Raft在配置改变之前引入了一个额外的阶段，在这个阶段，新的服务器作为非投票成员加入集群（领导者将日志条目复制给他们，但他们不被考虑为多数）。一旦新的服务器赶上了集群的其他部分，重新配置就可以如上所述进行。

第二个问题是，集群领导者可能不是新配置的一部分。在这种情况下，一旦它提交了C_{new}的日志条目，领导者就会下台（返回到跟随者状态）。这意味着将有一段时间（当它提交C_{new}时），领导者正在管理一个不包括自己的集群；它复制日志条目，但不把自己算在多数中。领导者过渡发生在C_{new}被提交时，因为这是新配置可以依赖性地运行的第一个点（它总是可以从C_{new}中选择一个领导者）。在这之前，可能只有C_{old}中的一个服务器可以被选为领导者。

第三个问题是，被移除的服务器（那些不在C_{new}）会扰乱集群。这些服务器不会重新接收心跳，所以它们会超时并开始新的选举。然后他们将发送带有新任期号码的RequestVote RPCs，这将导致当前的领导者恢复到追随者状态。一个新的领导者最终将被选出，但被移除的服务器将再次超时，这个过程将重复，导致可用性差。

为了防止这个问题，当服务器认为存在一个当前的领导者时，它们会忽略RequestVote RPC。具体来说，如果一个服务器在听到当前领导者的最小选举超时内收到RequestVote RPC，它不会更新其任期或授予其投票。这并不影响正常的选举，每个服务器在开始选举前至少要等待一个最小选举超时。然而，这有助于避免来自重新移动的服务器的干扰：如果一个领导者能够得到其集群的心跳，那么它将不会被更大的任期人数所废黜。

7 原木压实

筏子的日志在正常运行中不断增长，以纳入但在一个实际的系统中，它不可能无限制地增长。随着日志的增长，它占用了更多的空间，需要更多的时间来重放。如果没有某种机制来丢弃已经累积在日志中的过时信息，这最终会导致可用性问题。

快照是最简单的压缩方法。在快照中，整个当前系统状态被写入稳定存储的*快照*中，然后整个日志直到

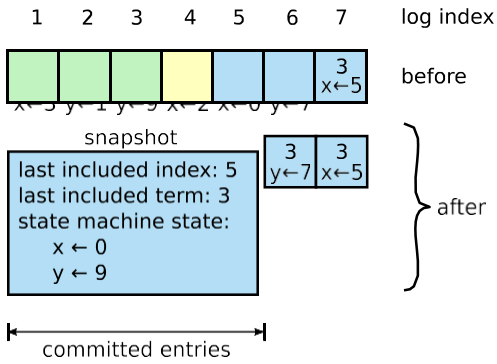


图12：一个服务器用一个新的快照来替换其日志中的已提交条目（索引1到5），该快照只存储当前状态（本例中的变量*x*和*y*）。快照的最后包含的索引和术语用于定位快照。

在第6条之前的日志中拍摄。

该点被丢弃。快照在Chubby和ZooKeeper中使用，本节的其余部分描述了Raft的快照。

递增的压缩方法，如日志清理[36]和日志结构的合并树[30，5]，也是可能的。这些方法一次性对部分数据进行操作，因此它们将压缩的负荷更均匀地分散在一段时间内。它们首先选择一个已经有许多被删除和覆盖的对象的数据区域，然后将该区域中的活对象更有效地重写，并释放该区域。与快照相比，这需要大量的附加机制和复杂性，因为快照总是对整个数据集进行操作，从而简化了问题。虽然日志清理需要对Raft进行修改，但状态机可以使用与快照相同的接口实现LSM树。

图12显示了Raft中快照的基本思路。每台服务器独立进行快照，只覆盖其日志中已提交的条目。大部分工作是由状态机将其当前状态写入快照中。Raft还包括快照中的少量元数据：*最后包含的索引*是快照取代的日志中最后一个条目的索引（状态机应用的最后一个entry），而*最后包含的术语*是这个条目的术语。这些被保留下来是为了支持快照后第一个日志条目的AppendEntries一致性检查，因为该条目需要一个先前的日志索引和术语。为了实现集群成员的变化（第6节），快照还包括日志中最新的配置，即最后包含的索引。一旦服务器完成写入快照，它可以删除所有的日志条目，直到最后包含的索引，以及任何先前的快照。

尽管服务器通常会独立地拍摄快照，但领导者偶尔必须向落后的跟随者发送快照。这种情况发生在领导者已经丢弃了它需要发送给追随者的下一个日志条目。幸运的是，这种情况在正常操作中是不可能发生的：一个追随者已经跟上了

InstallSnapshot RPC

由领导者调用，向跟随者发送快照的块。领导者总是按顺序发送块。

争论。

	领导人的任期
	leaderIdso追随者可以重定向客户
lastIncludedIndex	快照会替换所有的条目，直到并包括这个索引
最后包含的术语	lastIncludedIndex的术语
的偏移量	大块在快照文件中定位的字节偏移量
	快照块的原始字节数，从
data[]	补偿
	如果这是最后一块，则为true
done	

结果。

	currentTerm, for leader to update itself

术语

接收器的实施。

1. 如果期限<当前期限，则立即回复
2. 创建新的快照文件，如果第一块（偏移量为0）。
3. 在给定的偏移处将数据写入快照文件
4. 如果完成是假的，则回复并等待更多的数据块
5. 保存快照文件，丢弃任何现有的或部分快照的较小索引
6. 如果现有的日志条目与快照最后包含的条目具有相同的索引和术语，则保留它后面的日志条目并回复
7. 丢弃整个日志
8. 使用快照内容重置状态机（并加载快照的集群配置）。

图13：InstallSnapshot RPC的摘要。快照被分成几块进行传输；这给了Follower一个每块的生命迹象，所以它可以重置其选举计时器。

领导者已经有了这个条目。然而，一个特别慢的跟随者或一个新加入集群的服务器（第6节）则没有。让这样的追随者更新的方法是，领导者通过网络向它发送一个快照。

领导者使用一个名为InstallSnapshot的新RPC来向落后于它的追随者发送快照；见图13。当跟随者收到这个RPC的快照时，它必须决定如何处理其现有的日志内容。通常情况下，快照将包含新的信息，而不是在接收者的日志中。在这种情况下，跟随者会丢弃它的整个日志；它都被快照取代了，而且可能有与快照冲突的未提交的条目。相反，如果跟随者收到描述其日志前缀的快照（由于重传或错误），那么快照所涵盖的日志条目将被删除，但快照之后的条目仍然有效，必须保留。

这种快照方法偏离了Raft的强势领导原则，因为跟随者可以在领导不知情的情况下进行快照。然而，我们认为这种偏离是合理的。虽然有一个领导者有助于在达成共识时避免冲突的决定，但在快照时已经达成了共识，所以没有决定冲突。数据仍然只从领导那里流向下属。

低者，只是追随者现在可以重新组织他们的数据。

我们考虑了另一种基于领导者的方法，即只有领导者会创建一个快照，然后它将这个快照发送给它的每个追随者。然而，这有两个缺点。首先，向每个追随者发送快照会浪费网络带宽并减慢快照过程。每个追随者都已经拥有产生自己快照所需的信息，而且对于服务器来说，从其本地状态产生快照通常比通过网络发送和接收快照要便宜得多。第二，领导者的实施将更加复杂。例如，领导者需要在向追随者发送快照的同时，向他们回复新的日志条目，这样就不会阻碍新的客户请求。

还有两个影响快照性能的问题。首先，服务器必须决定何时进行快照。如果服务器快照的频率过高，就会浪费磁盘带宽和能源；如果快照的频率过低，就会有耗尽其存储容量的风险，而且会增加重启时重放日志的时间。一个简单的策略是，当日志达到一个固定的字节大小时进行快照。如果这个大小被设定为明显大于快照的预期大小，那么快照所占用的磁盘带宽就会很小。

第二个性能问题是，写一个快照可能需要相当长的时间，我们不希望这延迟正常的操作。解决方案是使用写时复制技术，这样新的更新可以被接受而不影响正在写入的快照。例如，用功能数据结构构建的状态机自然支持这一点。另外，可以使用操作系统的写时拷贝支持（例如Linux上的fork）来创建整个状态机的内存快照（我们的实现采用了这种方法）。

8 客户互动

本节介绍客户如何与Raft互动。包括客户如何找到集群领导者以及Raft如何支持可线性化语义[10]。这些问题适用于所有基于共识的系统，而Raft的解决方案与其他系统类似。

筏子的客户将其所有的请求发送给领导者。当客户端第一次启动时，它连接到一个随机选择的服务器。如果客户端的第一选择不是领导者，该服务器将拒绝客户端的请求，并提供最近的领导者的信息（AppendEntries请求包括领导者的网络地址）。如果领导者崩溃了，客户端的请求就会超时；然后客户端会在随机选择的服务器上再次尝试。我们对Raft的目标是实现可线性化的语义（每个操作看起来都是瞬时执行的，在其调用和响应之间的某个点上正好执行一次）。然而，正如到目前为止所描述的，Raft可以多次执行一个命令：例如，如果领导者如果在提交日志条目后但在回应客户之前发生崩溃，客户将用一个新的领导者重试该命令，导致它被第二次执行。解决方案是让客户为每个命令分配唯一的序列号。然后，状态机跟踪为每个客户处理的最新序列号，以及相应的响应。如果它收到一个序列号已经被执行的命令，它会立即响应，而不重新执行该请求。

只读操作可以在不向日志中写入任何内容的情况下进行处理。然而，由于没有额外的监控措施，这将有返回陈旧数据的风险，因为响应请求的领导者可能已经被一个它不知道的较新的领导者所吸纳。可以读取的数据必须不返回陈旧的数据，Raft需要两个额外的预防措施来保证这一点而不使用日志。首先，领导者必须拥有关于哪些条目被提交的最新信息。领导者完整性属性保证领导者拥有所有已提交的条目，但在其任期开始时，它可能不知道哪些是。为了找到答案，它需要从其任期内提交一个条目。Raft通过让每个领导者在其任期开始时向日志中提交一个空白的*no-op*条目来处理这个问题。第二，领导者必须在处理只读请求之前检查它是否已经被删除（如果最近的领导者已经当选，那么它的信息可能是过时的）。Raft通过让领导者在重新响应只读请求之前与集群中的大多数人交换心跳信息来处理这个问题。另外，领导者可以依靠心跳机制来提供一种租赁形式[9]，但这要依靠时间来保证安全（它假定有约束的时钟偏移）。

9 实施和评估

我们将Raft作为复制的一部分来实施。状态机用于存储RAMCloud的配置信息[33]，并协助RAMCloud协调器的故障转移。Raft的实现包含大约2000行的C++代码，不包括测试、注释或空行。源代码可以免费获得[23]。还有大约25个独立的第三方开源Raft实现[34]，处于不同的开发阶段，以本文的草案为基础。此外，各种公司正在部署基于Raft的系统[34]。

本节的其余部分使用三个标准对Raft进行评估：可理解性、正确性和性能。

9.1 可理解性

为了衡量Raft相对于Paxos的可理解性，我们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中对高年级的本科生和研究生进行了实验研究。我们录制了Raft和Paxos的视频讲座，并制作了相应的测验。Raft的讲座涵盖了本文中除日志压缩以外的内容；Paxos的讲座涵盖了本文中除日志压缩以外的内容。

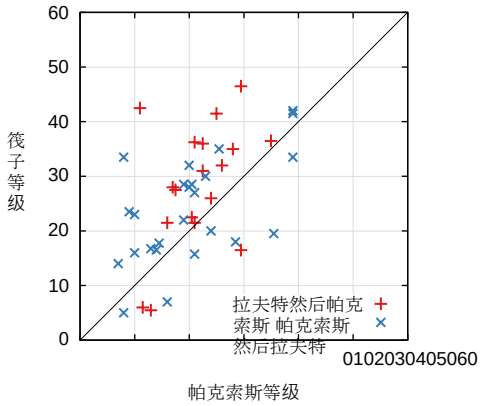


图14：比较43名参与者在Raft和Paxos测验中的表现的散点图。图表上方的点(33)代表在Raft中得分较高的参与者。

讲座涵盖了足够的材料来创建一个等效的复制状态机，包括单法令Paxos，多法令Paxos，重新配置，以及实践中需要的一些优化（如领导者选举）。测验测试了对算法的基本理解，还要求学生对角落的情况进行推理。每个学生都看了一个视频，做了相应的测验，看了第二个视频，做了第二个测验。大约一半的参与者先做Paxos部分，另一半先做Raft部分，以便考虑到个人表现的差异和从研究的第一部分获得的经验。我们比较了参与者在每次测验中的得分，以确定参与者是否对Raft有更好的理解。

我们试图使Paxos和Raft之间的比较尽可能的公平。实验在两个方面对Paxos有利。43名参与者中的15名报告说以前有一些使用Paxos的经验，而且Paxos的视频比Raft的视频长14%。正如表1所总结的，我们采取了措施来减少潜在的偏见来源。我们所有的材料都可供查阅[28, 31]。

平均而言，参与者在Raft测验中的得分比Paxos测验高4.9分（在可能的60分中，Raft的平均得分是25.7分，Paxos的平均得分是20.8分）；图14显示了他们的个人得分。成对的*t*检验表明，在95%的置信水平下，Raft分数的真实分布比Paxos分数的真实分布至少大2.5分。

我们还创建了一个线性回归模型，根据以下三个因素预测新学生的测验分数：他们参加了哪次测验，他们以前的Paxos经验程度，和

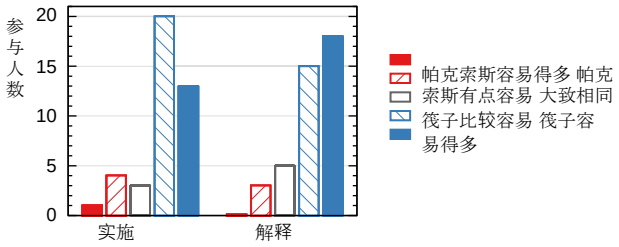


图15：使用5分制，参与者被问及(左)他们认为哪种算法更容易在一个有效的、正确的、高效的系统中实现，(右)哪种算法更容易向一个CS研究生解释。

他们学习算法的顺序。该模型预测，测验的选择会产生有利于Raft的12.5分的差异。这明显高于观察到的4.9分的差异，因为许多实际的学生之前有Paxos的经验，这对Paxos有很大的帮助，而对Raft的帮助略小。奇怪的是，模型还预测已经参加过Paxos测验的人在Raft上的得分要低6.3分；虽然我们不知道为什么，但这似乎是有统计学意义的。

我们还在测验后对参与者进行了调查，以了解他们认为哪种算法更容易实现或解释；这些结果显示在图15。绝大多数参与者认为Raft更容易实施和解释（每个问题41人中有33人）。然而，这些自我报告的感觉可能不如参与者的测验分数可靠，而且参与者可能因为知道我们的假设--
Raft更容易理解而产生偏差。

关于Raft用户研究的详细讨论，可参见[31]。

9.2 正确性

我们已经为第5节中描述的共识机制开发了一个正式的规范和安全证明。形式化规范[31]使用TLA+规范语言[17]使图2中总结的信息完全精确。它长约400行，作为证明的主题。对于实现Raft的人来说，它本身也是很有用的。我们已经用TLA证明系统[7]机械地证明了对数完整性属性。然而，这个证明依赖于没有经过机械检查的不变量（例如，我们没有证明规范的类型安全）。此外，我们已经写了一个关于状态机安全属性的非正式证明[31]，它是完整的（它仅仅依赖于规范）和相关的。

关注	为减少偏见	而采取的措施供审查的材料
[28, 31]		
	平等的讲座 质量两者的讲师 视频	
	相同。帕克索斯的讲座基于并改进了现有的...	
	在一些大学中使用的材料。帕克索斯的讲座长14%。	
测验	难度相等	
	问题按难度分组，并在各次考试中配对。	

	测验公平 使用了评分表。以随机顺序评分，测验之间交替进行 评分标准	评分 。

表1：对研究中可能存在的对Paxos的偏见的关注，为应对每种偏见而采取的措施，以及可用的补充材料。

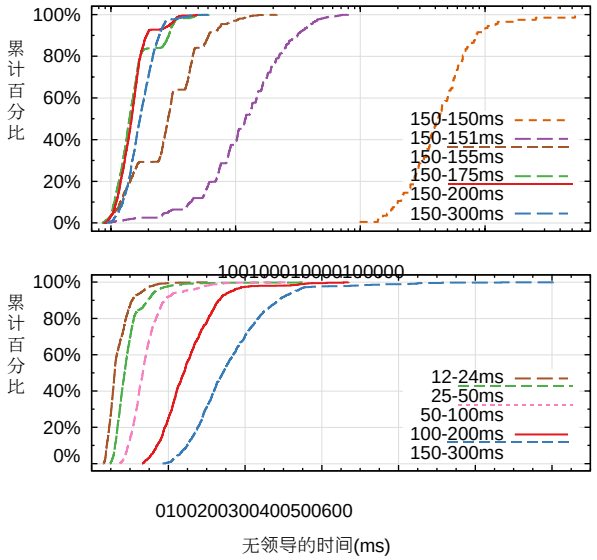


图 16：检测和替换一个崩溃的领导者的时间。上图改变了选举超时的随机性，下图是对最小选举超时的缩放。每条线代表1000次试验（除了100次三选一之外）。例如，"150-155ms"意味着选举超时是在150ms和155ms之间随机和均匀地选择的。测量是在一个由五台服务器组成的集群上进行的，广播时间大约为15ms。9台服务器的集群的结果也类似。

准确地说，它的长度约为3500字。

9.3 业绩

Raft的性能与其他共识算法（如Paxos）相似。对于性能来说，最重要的情况是当一个已建立的领导者正在复制新的日志条目。Raft使用最少的信息数量（从领导者到一半集群的单次往返）实现了这一点。进一步提高Raft的性能也是可能的。例如，它很容易支持批处理和管道化请求，以提高吞吐量和降低延迟。文献中已经为其他算法提出了各种优化方案；其中许多方案可以应用于Raft，但我们将此留给未来的工作。

我们用我们的Raft实现来衡量Raft的领导人选举算法的性能，并回答两个问题。首先，选举过程是否快速收敛？第二，领导者崩溃后可实现的最小停机时间是多少？

为了测量领导者的选举，我们反复地让五个服务器集群的领导者崩溃，并计时检测崩溃和选举新领导者所需的时间（见图16）。为了产生一个最坏的情况，每次试验中的服务器都有不同的日志长度，所以一些候选人没有资格成为领导者。此外，为了鼓励分裂的投票，我们的测试脚本在终止其进程之前，从领导者那里触发了同步的心跳RPC广播（这接近于领导者在崩溃前复制新的日志条目的行为）。领导者在其心跳间隔内被均匀地随机撞毁，该间隔是所有测试中最小选举超时的一半。因此，最小可能的停机时间是最小选举超时的一半左右。

图16中的顶部图表显示，选举超时中的少量随机化足以避免选举中的分裂票。在没有随机性的情况下，在我们的测试中，由于许多分裂的投票，领导选举的时间一直超过10秒。仅仅增加5毫秒的随机性就有很大帮助，导致中位数停机时间为287毫秒。使用更多的随机性可以改善最坏情况下的行为：使用50ms的随机性，最坏情况下的完成时间（超过1000次试验）是513ms。

图16中的底图显示，停机时间可以通过减少选举超时来减少。在选举超时12-24ms的情况下，平均只需要35ms就能选出一个领导者（最长的一次试验用了152ms）。然而，将超时时间降低到超过这个点就违反了Raft的时间要求：领导者很难在其他服务器开始新的选举之前广泛地投下心跳。这可能会导致不必要的领导者更换，降低系统的整体可用性。我们建议使用Conservative选举超时，如150-300ms；这样的超时不太可能导致不必要的领导者变更，并且仍然会提供良好的可用性。

10 相关工作

已有许多出版物涉及到了与"中国"有关的问题。感的算法，其中许多属于以下类别之一。

- Lamport对Paxos的原始描述[15]，并在诱惑下更清楚地解释它[16，20，21]。
- Paxos的阐述，填补了缺失的细节，并修改了算法，为实施提供了更好的基础[26, 39, 13]。
- 实现共识算法的系统，如Chubby [2, 4]，ZooKeeper [11, 12]，和Spanner [6]。Chubby和Spanner的算法还没有详细发表，尽管两者都声称是基于Paxos的。ZooKeeper的算法已经公布了更多细节，但它与Paxos有很大不同。
- 可以应用于Paxos的性能优化[18，19，3，25，1，27]。
- Oki和Liskov的Viewstamped Replication (VR)，是与Paxos差不多同时开发的另一种共识方法。原始描述[29]在过去的几年中，VR与分布式传输协议交织在一起，但在最近的更新中，核心共识协议已被分离出来[22]。VR使用了一种基于领导者的方法，与Raft有许多相似之处。

Raft和Paxos的最大区别是Raft的强大领导力。Raft将领导者选举作为共识协议的一个重要部分，并且它集中于

尽可能多的功能在领导者身上得到体现。这种方法导致了一种更简单的算法，更容易理解。例如，在Paxos中，领袖选举与基本的共识协议无关：它只是作为一种性能优化，并不是达成共识的必要条件。然而，这导致了额外的机制。Paxos包括一个两阶段的基本共识协议和一个单独的领袖选举机制。相比之下，Raft将领袖选举直接纳入共识算法，并将其作为共识的两个阶段中的第一阶段。这导致了比Paxos更少的机制。

与Raft一样，VR和ZooKeeper也是基于领导者的，因此与Paxos相比，Raft有很多优势。然而，Raft的机制不如VR或ZooKeeper，因为它将非领导者的功能降到最低。例如，Raft中的日志条目只向一个方向流动：从AppendEntries RPCs的领导者向外流动。在VR中，日志条目是双向流动的（领导者可以在选举过程中接收日志条目）；这导致了额外的机制和复杂性。ZooKeeper公布的描述也是将日志条目传送给领导者和从领导者那里传送，但其实现显然更像Raft[35]。

Raft的消息类型比我们所知的任何其他基于共识的日志复制的算法都要少。例如，我们计算了VR和ZooKeeper用于基本共识和成员变化的消息类型（不包括日志压缩和客户端互动，因为这些几乎是独立于算法的）。VR和ZooKeeper各自定义了10种不同的消息类型，而Raft只有4种消息类型（两个RPC请求和它们的响应）。Raft的消息比其他算法的消息更密集一些，但它们的结构更简单。此外，VR和ZooKeeper的描述是在领导者变化期间传输整个日志；需要额外的消息类型来优化这些机制，使其实用。

Raft的强领导方法简化了算法，但它排除了一些性能优化。例如，Egalitarian Paxos（EPaxos）在某些条件下可以通过无领导方法获得更高的性能[27]。EPaxos利用了状态机命令的交换性。只要其他同时提出的命令与之相换，任何服务器都可以只用一轮通信来提交一个命令。然而，如果提出的命令目前没有相互交换，EPaxos就会重新要求进行额外的一轮通信。由于任何服务器都可以提交命令，EPaxos可以很好地平衡服务器之间的负载，并能够在广域网环境中实现比Raft更低的延迟。然而，它给Paxos增加了很大的复杂性。

其他工作中已经提出或实施了几种不同的集群成员变更方法，包括Lamport的原始提议[15]、VR[22]和SMART[24]。我们为Raft选择了联合共识的方法，因为它利用了consensus协议的其他部分，所以成员变更所需的额外机制很少。Lamport的基于α的方法不是Raft的选择，因为它假定没有领导者也能达成共识。与VR和SMART相比，Raft的重新配置算法的优点是，成员变化可以在不限制正常请求的处理的情况下发生；相反，VR在配置变化期间停止所有的正常处理，而SMART对未处理的请求数量施加了类似α的限制。Raft的方法也比VR或SMART增加了更少的机制。

11 总结

算法的设计通常以正确性、有效性和/或简洁性为主要目标。虽然这些都是有价值的目标，但我们认为理解性也同样重要。除非开发者将算法转化为实际的实现，否则其他目标都无法实现，而实际的实现将不可避免地偏离和扩展已公布的形式。除非开发者对算法有深刻的理解，并能建立起对它的直觉，否则他们很难在实现中保留其理想的特性。

在这篇论文中，我们讨论了分布式康采恩的问题，其中一个被广泛接受但难以理解的算法--Paxos，多年来一直在挑战学生和开发者。我们开发了一种新的算法，Raft，我们已经证明它比Paxos更容易理解。我们还认为，Raft为系统建设提供了一个更好的基础。将可理解性作为主要的设计目标，改变了我们处理Raft的方式；随着设计的进展，我们发现自己反复使用一些技术，如分解问题和简化状态空间。这些技术不仅提高了Raft的可理解性，而且也使我们更容易相信它的正确性。

12 鸣谢

如果没有Ali Ghodsi、David Mazières以及伯克利大学CS 294-91和斯坦福大学CS 240的学生的支持，这项用户研究是不可能完成的。Scott Klemmer帮助我们设计了用户研究，Nelson Ray为我们提供了统计分析方面的建议。用户研究的Paxos幻灯片在很大程度上借鉴了Lorenzo Alvisi最初制作的幻灯片。特别感谢David Mazières和Ezra Hoch发现了Raft中的细微错误。许多人对论文和用户研究材料提供了有益的反馈，包括Ed Bugnion、Michael Chan和Hugues Evrard。

Daniel Giffin, Arjun Gopalan, Jon Howell, Vimalkumar Jeyakumar, Ankita Kejriwal, Aleksandar Kracun, Amit Levy, Joel Martin, Satoshi Matsushita, Oleg Pesok, David Ramos, Robbert van Renesse, Mendel Rosenblum。Nico-las Schiper, Deian Stefan, Andrew Stone, Ryan Stutsman, David Terei, Stephen Yang, Matei Zaharia, 24位匿名会议评审员（有重复的），特别是我们的牧羊人Eddie Kohler。Werner Vogels在推特上提供了一个早期草案的链接，这给了Raft很大的曝光机会。这项工作得到了Gigascale Systems Research Center和Multiscale Systems Center的支持，这是由Fo-cus Center Research Program（半导体研究公司计划）资助的六个研究中心中的两个；STARnet（由MARCO和DARPA赞助的半导体再搜索公司计划）；国家科学基金会第0963859号拨款；以及Facebook、Google、Mellanox、NEC、NetApp、SAP和三星的拨款。Diego Ongaro由Junglee公司的斯坦福毕业生奖学金支持。

参考文献

[1] BOLOSKY, W. J., BRADShaw, D., HAAGENS, R. B。KUSTERS, N. P., AND LI, P. Paxos复制的状态机作为高性能数据存储的基础。In *Proc.NSDI'11*, *USENIX网络系统设计与实施会议*（2011），USENIX，第141-154页。

[2] BURROWS, M. The Chubby lock service for loosely-coupled distributed systems.In *Proc.OSDI'06, Symposium on Operating Systems Design and Implementation* (2006), USENIX, pp.335-350.

[3] CAMARGOS, L. J., SCHMIDT, R. M., AND PEDONE, F. Multicoordinated Paxos.In *Proc.PODC'07, ACM Symposium on Principles of Distributed Computing* (2007), ACM, pp.316-317.

[4] CHANDRA, T. D., GRIESEMER, R., AND REDSTONE, J. Paxos made live: an engineering perspective.In *Proc.PODC'07, ACM分布式计算原理研讨会*（2007），ACM，第398-407页。

[5] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. Bigtable: a distributed storage system for structured data.In *Proc.OSDI'06, USENIX操作系统设计与实现研讨会*（2006），USENIX，第205-218页。

[6] Corbett, J. C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J. J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P., Hsieh, W., Kan-thak, S., Kogan, E. 。LI, H., LLOYD, A., MELNIK, S., MWAURA, D., NAGLE, D., QUINLAN, S., RAO, R., ROLIG, L., SAITO, Y., SZYMANIAK, M., TAYLOR, C., WANG, R., and WOODFORD, D. Spanner。谷歌的全球分布式数据库。在*Proc.OSDI'12, USENIX Conference on Operating Systems Design and Implemen- tation* (2012), USENIX, pp.251-264.

[7] COUSINEAU, D., DOLIGEZ, D., LAMPORT, L., MERZ, S., RICKETTS, D., AND VANZETTO, H. TLA+证明。In *Proc.FM'12，形式化方法研讨会*（2012）。D.D. Giannakopoulou和D. Méry, Eds., vol. 7436 of *Lecture Notes in Computer Science*, Springer, pp.147-154。

[8] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T.The Google file system.In *Proc. SOSP'03, ACM Symposium on Operating Systems Principles* (2003), ACM, pp.29-43.

[9] GRAY, C., AND CHERITON, D. Leases:分布式文件缓存一致性的有效容错机制。In *Proceedings of the 12th ACM Ssymposium on Operating Systems Principles* (1989), pp.202-210.

[10] HERLIHY, M. P., AND WING, J. M. Linearizability: a correctness condition for concurrent objects.*ACM Transactions on Programming Languages and Systems 12* (July 1990), 463-492.

[11] Hunt, P., Konar, M., Junqueira, F. P., and Reed, B.ZooKeeper：互联网规模系统的无等待协调。In *Proc ATC'10, USENIX Annual Technical Con- ference* (2010), USENIX, pp.145-158.

[12] JUNQUEIRA, F. P., REED, B. C., AND SERAFINI, M. Zab:主备份系统的高性能广播。In *Proc.DSN'11, IEEE/IFIP Int'l Conf. on Depend- able Systems & Networks* (2011), IEEE Computer Society, pp.245-256.

[13] KIRSCH, J., AND AMIR, Y. Paxos for system builders.Tech.CNDS-2008-2, Johns Hopkins University, 2008.

[14] LAMPORT, L. Time, clocks, and the ordering of events in a distributed system.*Commununications of the ACM 21*, 7 (July 1978), 558-565.

[15] LAMPORT, L. The part-time parliament.*ACM Transactions on Computer Systems 16*, 2 (May 1998), 133-169.

[16] LAMPORT, L. Paxos made simple.*ACM SIGACT News 32*, 4 (Dec. 2001), 18-25.

[17] LAMPORT, L. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*.Addison-Wesley, 2002.

[18] LAMPORT, L. Generalized consensus and Paxos.Tech.MSR-TR-2005-33, Microsoft Research, 2005.

[19] LAMPORT, L. Fast paxos.*Distributed Computing 19*, 2 (2006), 79-103.

[20] LAMPSON, B. W. How to build a highly available system using consensus.In *Distributed Algorithms*, O. Baboaglu and K. Marzullo, Eds.Springer-Verlag, 1996, pp.1-17.

[21] LAMPSON, B. W. The ABCD's of Paxos.In *Proc.PODC'01, ACM分布式计算原理研讨会*（2001），ACM，第13-13页。

[22] LISKOV, B., AND COWLING, J. Viewstamped replica- tion revisited.Tech.MIT-CSAIL-TR-2012-021, MIT, July 2012.

[23] `http://github.com/ logcabin/logcabin` 的

源代码。

[24] LORCH, J. R., ADYA, A., BOLOSKY, W. J., CHAIKEN, R., DOUCEUR, J. R., AND HOWELL, J. The SMART way to migrate replicated stateful services.In *Proc.Eu- roSys'06, ACM SIGOPS/EuroSys欧洲计算机系统会议*（2006），ACM，第103-115页。

[25] MAO, Y., JUNQUEIRA, F. P., AND MARZULLO, K. Mencius: building efficient replicated state machines for WANs.In *Proc.OSDI'08，USENIX操作系统设计与实现会议*（2008），USENIX，第369-384页。

[26] MÈRES, D. Paxos made practical. http://www.scs.stanford.edu/~dm/home/papers/paxos.pdf，2007年1月。

[27] MORARU, I., ANDERSEN, D. G., AND KAMINSKY, M. There is more consensus in egalitarian parliaments.在*Proc. SOSP'13, ACM操作系统原理研讨会*（2013）上，ACM。

[28] 筏子用户研究。http://ramcloud.stanford.edu/~ongaro/userstudy/。

[29] OKI, B. M., AND LISKOV, B. H. Viewstamped replication:支持高可用分布式系统的一种新的主拷贝方法。In *Proc.PODC'88，ACM分布式计算原理研讨会*（1988），ACM，第8-17页。

[30] O'NEIL, P., CHENG, E., GAWLICK, D., AND ONEIL, E. The log-structured merge-tree（LSM-tree）.*Acta Informatica 33*, 4 (1996), 351-385.

[31] ONGARO, D. *共识。沟通理论与实践*。斯坦福大学博士论文，2014年（工作正在进行中）。http://ramcloud.stanford.edu/~ongaro/thesis.pdf。

[32] ONGARO, D., AND OUSTERHOUT, J. In search of an understandable consensus algorithm.In *Proc ATC'14, USENIX Annual Technical Conference* (2014), USENIX.

[33] OUSTERHOUT, J., AGRAWAL, P., ERICKSON, D., KOZYRAKIS, C., LEVERICH, J., MÈRES, D., MI- TRA, S., NARAYANAN, A., ONGARO, D., PARULKAR, G., ROSENBLUM, M., RUMBLE, S. M., STRATMANN, E., AND STUTSMAN, R. The case for RAMCloud。*ACM的通讯54*（2011年7月），121-130。

[34] 筏式共识算法网站。http://raftconsensus.github.io。

[35] REED, B. 个人通信，2013年5月17日。

[36] ROSENBLUM, M., AND OUSTERHOUT, J. K. The design and implementation of a log-structured file system.*ACM Trans.Comput.Syst.10* (February 1992), 26-52.

[37] SCHNEIDER, F. B. Implementing fault-tolerant services using the state machine approach: a tutorial.*ACM Computing Surveys 22*, 4 (Dec. 1990), 299-319.

[38] SHVACHKO, K., KUANG, H., RADIA, S., AND CHANSLER, R. The Hadoop distributed file system.In *Proc.MSST'10，大规模存储系统和技术研讨会*（2010），IEEE计算机协会，第1-10页。

[39] VAN RENESSE, R. Paxos made moderately complex.Tech. rep., Cornell University, 2012.