

# 寻找可理解的共识算法（扩展版）

斯坦福大学 Diego Ongaro 和 John  
Ousterhout

## 摘要

Raft 是一种管理复制日志的共识算法。它产生的结果等同于（多）Paxos，其效率与 Paxos 相当，但其结构与 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

增强易懂性，Raft 分离了共识的关键要素，如领导者选举、日志复制和安全性，并加强了一致性，以减少必须考虑的状态数量。一项用户研究结果表明，对于学生来说，Raft 比 Paxos 更容易学习。Raft 还包括一种新的集群成员变更机制，它使用重叠的主要关系来保证安全性。

## 1 引言

共识算法允许机器集合作为一个连贯的群体工作，即使其中一些成员出现故障也能继续工作。正因为如此，共识算法在构建可靠的大型软件系统中发挥着关键作用。在过去十年中，Paxos [15, 16] 在共识算法的讨论中占据了主导地位：大多数共识算法的实现都基于 Paxos 或受其影响，Paxos 已成为向学生讲授共识算法的主要工具。

遗憾的是，尽管多次尝试使 Paxos 更易于理解，但它还是相当难懂。此外，它的架构需要复杂的更改才能支持实际系统。因此，系统构建者和学生都很难理解 Paxos。

在与 Paxos 进行了一番较量之后，我们开始寻找一种新的共识算法，为系统构建和教育提供更好的基础。我们的方法与众不同，因为我们的首要目标是 *可理解性*：我们能否为实用系统定义一种共识算法，并以比 Paxos 更易于学习的方式对其进行描述？此

外，我们还希望该算法能够促进直觉的发展，这对系统构建者来说至关重要。重要的是，算法不仅要有效，还要让人明白它为什么有效。

这项工作的成果就是一种名为 Raft 的共识算法。在设计 Raft 时，我们采用了特定的技术来提高可理解性，包括分解（Raft 分离了领导者选举、日志复制和安全）和

本技术报告是[32]的扩展版本；空白处用灰条标注了补充材料。发表于 2014 年 5 月 20 日。

减少状态空间（相对于 Paxos，Raft 减少了非确定性程度，以及服务器之间保持一致的方式）。对两所大学的 43 名学生进行的用户研究表明，Raft 比 Paxos 更容易理解：在学习了这两种算法后，其中 33 名学生回答有关 Raft 的问题比回答有关 Paxos 的问题要好。

Raft 在很多方面与现有的共识算法（最著名的是 Oki 和 Liskov 的 Viewstamped Replication [29, 22]）相似，但它有几个新特点：

- 强大的领导者与其他共识算法相比，Raft 使用更强的领导者形式。例如  
日志条目只从领导者流向其他服务器。这简化了复制日志的管理，使 Raft 更容易理解。
- 领导人选举：Raft 使用随机计时器选举领导人。这增加了少量  
这种机制与任何共识算法所需的心跳机制相辅相成，同时又能简单快速地解决冲突。
- 成员变更：Raft 更改集群中服务器集的机制使用新的  
联合共识法，即两种不同配置的多数在转换过程中重叠。这样，集群就能在配置变化时继续正常运行。

我们认为，无论是出于教育目的，还是作为实施的基础，Raft 都优于 Paxos 和其他同感算法。与其他算法相比，它更简单、更易懂；它的描述足够全面，能满足实际系统的需要；它有多个开源实现，并被多家公司使用；它的安全属性已得到正式规定和证明；它的效率可与其他算法相媲美。

本文其余部分将介绍复制状态机问题（第 2 节），讨论 Paxos 的优缺点（第 3 节），介绍我们的可理解性总体方法（第 4 节），介绍 Raft 共识算法（第 5-8 节），评估 Raft（第 9 节），并讨论相关工作（第 10 节）。

## 2 复制状态机

共识算法通常出现在复制状态机中[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，即使部分服务器宕机，状态机也能继续运行。复制状态机

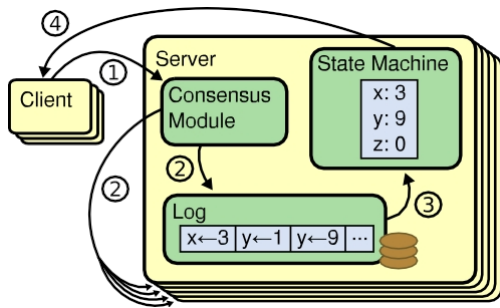


图 1：复制状态机架构。Consensus 算法管理一个复制日志，其中包含来自客户端的状态机命令。状态机处理来自日志的相同命令序列，因此产生相同的输出。

用于解决分布式系统中的各种容错问题。例如，GFS[8]、HDFS[38]和 RAMCloud[33]等拥有单一集群领导者的大型系统，通常使用单独的复制状态机来管理领导者选举，并存储必须在领导者崩溃后仍能存活的配置信息。复制状态机的例子包括 Chubby [2] 和 ZooKeeper [11]。

复制状态机通常使用复制日志来实现，如图 1 所示。每台服务器都存储着包含一系列命令的日志，其状态机按顺序执行这些命令。每个日志包含的命令顺序相同，因此每个状态机处理的命令顺序也相同。由于状态机是确定的，因此每个状态机计算的状态和输出序列都是相同的。

保持复制日志的一致性是一致的算法的工作。服务器上的共识模块接收来自客户端的命令，并将其添加到日志中。它与其他服务器上的共识模块进行通信，以确保每个日志最终都以相同的顺序包含相同的请求，即使某些服务器出现故障也是如此。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这些命令，并将结果返回给客户端。这样，服务器就形成了一个高度可靠的状态机。

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，它们都能确保安全（永远不会返回错误结果），包括网络延迟、分区、数据包丢失、重复和重新排序。
- 只要大部分服务器都能正常运行并能与其他服

务器通信，它们就能充分发挥作用（可用）。

因此，一个典型的五台服务器集群可以承受任意两台服务器的故障。因此，一个由五台服务器组成的典型群集可以承受任意两台服务器的故障。假定服务器因停止而发生故障；它们随后可能从稳定存储上的状态恢复，并重新加入群集。

- 它们并不依赖于时间来确保一致性。

日志的及时性：错误的时钟和极度的信息延迟在最坏的情况下也会导致可用性问题。

- 在常见情况下，只要群集的大多数成员都响应了命令，命令就能完成。

单轮远程过程调用；少数速度慢的服务器不会影响整个系统的性能。

### 3 帕克斯斯怎么了？

在过去十年中，莱斯利-兰波特的 Paxos 原型[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 首先定义了一种能够就单一决策（如单一复制日志条目）达成一致的协议。我们将这个子集称为 *单决定 Paxos*。然后，Paxos 将此协议的多个实例结合起来，以促进日志等一系列决策的达成（*多 Paxos*）。Paxos 可确保安全性和有效性，并支持集群成员的变更。其正确性已得到证实，而且在正常情况下效率很高。

遗憾的是，Paxos 有两个重大缺陷。第一个缺点是 Paxos 极难理解。完整的解释[15]显然是不透明的；很少有人能成功理解它，而且要付出很大的努力。因此，有人尝试用更简单的术语来解释 Paxos [16、20、21]。这些解释主要针对单一法令子集，但仍然具有挑战性。在对 2012 年 NSDI 与会者的非正式调查中，我们发现即使是经验丰富的研究人员，也很少有人能熟练掌握 Paxos。我们自己也在与 Paxos 作斗争；在阅读了七份简化解释并设计了自己的原生协议之后，我们才理解了完整的协议，这个过程花费了将近一年的时间。

我们假设，Paxos 的不透明性源于它选择了单一法令子集作为其基础。单法令 Paxos 密实而微妙：它分为两个阶段，没有简单直观的解释，也无法独立理解。正因为如此，我们很难从直觉上理解单法令协议的工作原理。多 Paxos 的组成规则增加了额外的复杂性和微妙性。我们认为，就多项决定达成

共识（即用日志代替单一条目）这一整体问题可以用其他更直接、更明显的方式进行分解。

Paxos 的第二个问题是，它没有为构建实用的实施方案提供良好的基础。其中一个原因是多 Paxos 算法没有得到广泛认可。Lamport 所描述的大多是单自由度 Paxos；他勾画了多自由度 Paxos 的可能方法，但许多细节都有遗漏。已经有一些尝试对 Paxos 进行充实和操作化，如 [26]、[39] 和 [13]，但这些尝试都有不同之处。

这些算法彼此互不相同，也与 Lamport 的草图不同。Chubby [4] 等系统已经实现了类似 Paxos 的算法，但在大多数情况下，其细节尚未公布。

此外，Paxos 架构在构建实用系统方面并不理想；这也是单目标分解的另一个后果。例如，独立选择日志条目集合，然后将它们合并成一个顺序日志，这样做的好处微乎其微；这只会增加复杂性。更简单、更高效的方法是围绕日志设计一个系统，在这个系统中，新条目按照受限的顺序依次排列。另一个问题是，Paxos 的核心是对称的点对点方法（尽管它最终建议采用弱领导形式作为性能优化）。这在只需做出一个决定的简化世界中是合理的，但很少有实际系统使用这种方法。如果必须做出一系列决策，那么首先选出一个领导者，然后由领导者协调决策，这样会更简单快捷。

因此，实际系统与 Paxos 几乎没有相似之处。每一个实施方案都以 Paxos 为起点，先解决实施过程中的困难，然后再开发出截然不同的架构。这样做既耗时又容易出错，而理解 Paxos 的困难又加剧了问题的严重性。Paxos 的公式可能是证明其正确性定理的好方法，但实际实现与 Paxos 相差甚远，证明的价值微乎其微。Chubby 实现者的以下评论很有代表性：

Paxos 算法的描述与实际系统的需求之间存在很大差距。  
经过验证的协议 [4]。

由于存在这些问题，我们认为 Paxos 无法为系统构建或教育提供良好的基础。鉴于共识在大型软件系统中的重要性，我们决定看看能否设计出一种比 Paxos 性能更好的替代共识算法。Raft 就是这一实验的成果。

## 4 可理解性设计

我们在设计 Raft 时有几个目标：它必须为系统构建提供一个完整而实用的基础，从而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，并且在典型的运行条件下都是可用的；它必须在普通操作中都是高效的。但是，我们最重要的目标--也是最艰巨的挑战--是系统的 *可理解性*。必须让广大受众能够轻松理解算法。此外，还必须能够建立起对算法的直觉，以便系统构建者能够在实际应用中进行扩展。

在 Raft 的设计过程中，有许多地方我们必须在备选方案中做出选择。在这种情况下，我们根据可理解性来评估替代方案：解释每种替代方案有多难（例如，其状态空间有多复杂，是否有微妙的影响？

我们认识到，这种分析具有高度的主观性；不过，我们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法：在可能的情况下，我们将问题分成可以相对独立地解决、解释和理解的不同部分。例如，在 Raft 中，我们将领导者选举、日志复制、安全和成员变更分开。我们的第二种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，日志不允许有漏洞，Raft 限制了日志之间不一致的方式。虽然在大多数情况下，我们都试图消除非确定性，但在某些情况下，非确定性实际上会提高可理解性。特别是，随机化方法会引入非确定性，但它们倾向于以类似的方式处理所有可能的选择（“任选其一；无所谓”），从而缩小状态空间。我们使用随机化

来简化 Raft 领导者选举算法。

## 5 Raft 共识算法

Raft 是一种管理第 2 节所述复制日志的算法。图 2 以浓缩的形式概述了该算法，以供参考；图 3 列出了该算法的关键属性；这些图中的元素将在本节的其余部分逐一讨论。

Raft 实现共识的方法是，首先选出一个杰出的领导者，然后让领导者全权负责管理复制的日志。领导者接受来自客户端的日志条目，将它们复制到其他服务器上，并告诉服务器何时可以将日志条目应用到它们的状态机中。领导者简化了复制日志的管理。例如，领导者可以决定将新条目放在日志的哪个位置，而无需咨询其他服务器，数据也能以简单的方式从领导者流向其他服务器。领导者可能出现

故障或与其他服务器断开连接，在这种情况下，会选出新的领导者。

根据领导者方法，Raft 将同感问题分解为三个相对独立的子问题，并在下面的小节中进行讨论：

- 领导者选举：当现有领导者失败时，必须选出新的领导者（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状态:

(在响应 RPC 之前在稳定存储上更新)

`currentTerm` 服务器看到的最新术语 (初始化为 0, 首次启动时, 单调递增)  
在当前任期内获得投票的候选人标识 (如果没有, 则为空)。  
`log[]` `log` 条目; 每个条目包含命令为状态机, 以及领导者收到条目时的术语 (第一个索引为 1)

### 所有服务器的易失性状态:

已知最高日志条目的 `commitIndex` (已提交 (初始化为 0, 单调递增))  
应用于状态的最高日志条目的 `lastAppliedIndex` (机器 (初始化为 0, 单调递增))

### 领导人状态不稳定:

(选举后重新初始化)

`nextIndex[]` 为每台服务器的下一条日志条目索引 (将发送给该服务器 (初始化为领导者最后的日志索引 + 1))  
每个服务器的 `matchIndex[]`, 最高日志条目的索引 (已知在服务器上复制 (初始化为 0, 单调递增))

## AppendEntries RPC

由领导者调用, 复制日志条目 (§5.3); 也用作心跳 (§5.2)。

### 论据:

**任期** 领导任期  
**领导者** `Id` 追随者可以重新定向客户  
前一个日志条目的 `prevLogIndex`  
新  
`prevLogIndex` 条目的 `prevLogTerm`  
**条目[]** 要存储的 日志条目 (心跳时为空; 为效率, 可发送多个)  
**领导者提交** 领导者的提交索引

### 结果

**任期** 当前术语, 以便领导者自行更新  
如果追随者包含匹配的条目, 则 `success` 为 `true`  
`prevLogIndex` 和 `prevLogTerm`

### 接收器的实施:

- 如果 `term < currentTerm`, 则回复 `false` (§5.1)
- 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)
- 如果现有条目与新条目冲突 (索引相同但术语不同), 则删除现有条目及其后面的所有条目 (§5.3)

## 请求投票 RPC

被候选人用来收集选票 (§5.2)。

### 论据:

**term** 候选人任期 要求投票的  
**candidateId** 候选人  
**lastLogIndex** 候选人最后日志条目的索引 (§5.4)  
**lastLogTerm** 候选人最后日志条目的期限 (§5.4)  
**结果: 任期** `currentTerm`, 候选人更新为 "true" 表示候选人获得投票  
**投票通过**

### 接收器的实施:

- 如果 `term < currentTerm`, 则回复 `false` (§5.1)
- 如果 `votedFor` 为空或候选人 `Id`, 且候选人的日志至少与接收方的日志一样新, 则同意投票 (§5.2, §5.4)

4. 添加日志中尚未记录的新条目
5. 如果  $\text{leaderCommit} > \text{commitIndex}$ ，则设置  $\text{commitIndex} = \min(\text{leaderCommit}, \text{最后一个新条目索引})$

- 所有服务器。
- 如果  $\text{commitIndex} > \text{lastApplied}$ ：递增  $\text{lastApplied}$ ，将  $\log[\text{lastApplied}]$  应用于状态机 (§5.3)
  - 如果 RPC 请求或响应包含术语  $T > \text{currentTerm}$ ：设置  $\text{currentTerm} = T$ ，转换为从众 (§5.1)

#### 追随者 (§5.2)：

- 回复候选人和领导人的 RPC
- 如果选举超时，但未收到当前领导者的  $\text{AppendEntries}$  RPC，也未将投票权授予候选人：转换为候选人

#### 候选人 (§5.2)：

- 转为候选人后，开始选举：
  - 递增  $\text{currentTerm}$
  - 为自己投票
  - 重置选举计时器
  - 向所有其他服务器发送请求投票 RPC
- 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者的  $\text{AppendEntries}$  RPC：转换为跟随者
- 如果选举超时：开始新的选举

#### 领队

- 当选后：向每个服务器发送初始空  $\text{AppendEntries}$  RPC（心跳）；在空闲期间重复发送，以防止选举超时 (§5.2)
- 如果从客户端接收到命令：将条目添加到本地日志，在条目应用到状态机后进行响应 (§5.3)

图 2：Raft 共识算法的浓缩摘要（不包括成员变更和日志压缩）。从左上到右框中的服务器行为被描述为一般独立重复触发的规则。章节编号（如第 5.2 节）表示讨论特定功能的位置。正式规范 [1] 更精确地描述了这个算法。

- 如果追随者的最后日志索引  $\geq \text{nextIndex}$ ：发送  $\text{AppendEntries}$  RPC
- 如果  $\text{nextIndex}$  开始为 0，则递增  $\text{nextIndex}$  并重新发送
- 如果  $\text{AppendEntries}$  成功：更新追随者的  $\text{matchIndex}$  和  $\text{commitIndex}$  (第 5.3 节)
- 如果  $\text{AppendEntries}$  因日志不一致而失败：递减  $\text{nextIndex}$  并重试 (第 5.3 节)
- 如果存在一个  $N$ ，使得  $N > \text{commitIndex}$ ， $\text{matchIndex}[i]$  的大部分  $\geq N$ ，并且  $\log[N].\text{term} == \text{currentTerm}$ ：则设置  $\text{commitIndex} = N$  (§5.3, §5.4)



选举安全：一个任期内最多只能选出一名领导人。§5.2

仅领导者附加：领导者从不覆盖或删除日志中的条目，只附加新条目。§5.3

日志匹配：如果两个日志中包含具有相同索引和术语的条目，那么直到给定索引为止的所有条目都是相同的。§5.3

领导者完整性：如果在某一术语中记录了日志条目，那么该条目将出现在所有更高术语的领导者日志中。§5.4

状态机安全：如果一台服务器在其状态机中应用了给定

图 3：索引的日志条目，其他服务器将不会在更高索引编号处应用不同的属性条目。

§5.4.3

并在整个集群中复制，迫使其他日志与自己的日志保持一致（第 5.3 节）。

- 安全性：Raft 的关键安全特性是图 3 中的状态机安全特性：如果任何服务器则其他服务器不得对同一日志索引应用不同的命令。第 5.4 节介绍了 Raft 如何确保这一特性；该解决方案涉及对第 5.2 节所述选举机制的额外限制。

在介绍了共识算法之后，本节将讨论系统中的可用性问题 and 定时的作用。

## 5.1 筏子基础知识

Raft 集群包含多台服务器；典型的数量为五台，这样系统就能承受两次故障。在任何时候，每台服务器都处于三种状态之一：领导者、追随者或候选者。正常运行时，只有一个领导者，其他所有服务器都是追随者。追随者是被动的：它们自己不发出请求，只是响应领导者和候选者的请求。领导者处理所有客户请求（如果客户联系追随者，追随者会将其重定向到领导者）。第三个状态是候选状态，用于选举新的领导者，详见第 5.2 节。图 4 显示了这些状态及其转换；下文将讨论这些转换。

如图 5 所示，Raft 将时间划分为任意长度的项。

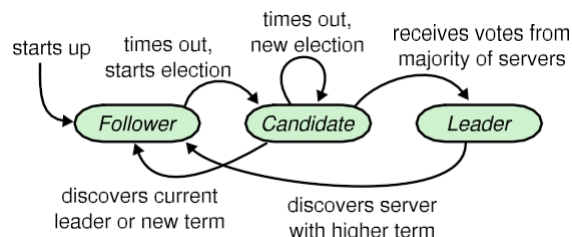
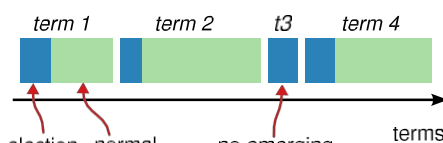


图 4：服务器状态。追随者只回应其他服务器的请求。

如果一个跟随者没有收到任何通信，它就会成为候选者并发起选举。获得整个集群多数选票的候选者将成为新的领导者。领导者通常一直运行到失败为止。



任期用连续的整数编号。每个任期以选举开始，在选举中，一个或多个候选人试图成为领导者，如第 5.2 节所述。如果一名候选人在选举中获胜，那么他将在任期的剩余时间内担任领导者。在某些情况下，选举会导致票数相差悬殊。在这种情况下，任期结束时将没有领导者；新的任期（重新选举）

图 5：时间被划分为若干个任期，每个任期以选举开始。选举成功后，由一名领导者管理集群，直到任期结束。有些选举会失败，在这种情况下，任期结束时不会选出领导者。在不同的服务器上，可以在不同的时间段观察到任期之间的转换。

将很快开始。Raft 可确保在特定任期内最多有一名领导者。

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，服务器可能不会观察一次选举甚至整个术语。术语在 Raft 中充当逻辑时钟 [14]，允许服务器检测过时信息，如陈旧的领导人。每个服务器都存储一个 *当前术语* 编号，该编号随时间单调递增。每当服务器进行通信时，就会交换当前项；如果一个服务器的当前项小于另一个服务器的当前项，那么它就会将自己的当前项更新为较大的值。如果候选者或领导者发现自己的术语已经过时，就会立即恢复到较低的状态。如果一个服务器收到一个过期的任期号请求，它就会拒绝该请求。

Raft 服务器使用远程过程调用 (RPC) 进行通信，基本的共识算法只需要两种类型的 RPC。RequestVote RPC 由候选人在选举期间发起（第 5.2 节），而 Append-Entries RPC 则由领导者发起，用于复制日志和提供一种心跳形式（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回复，就会重试 RPC，并且并行发布 RPC 以获得最佳性能。

## 5.2 领导人选举

Raft 使用心跳机制来触发领导者选举。当服务器启动时，它们开始处于跟随者状态。只要收到有效的心跳信号，服务器就会保持跟随者状态。

来自领导者或候选者的 RPC。领导者会定期向所有追随者发送心跳信息（AppendEntries RPC，不带日志条目），以维护自己的权威。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，那么它就会认为没有可行的领导者，并开始选举新的领导者。

要开始选举，追随者会递增其当前任期并转换为候选状态。然后，它为自己投票，并向集群中的其他每台服务器发送并行的 RequestVote RPC。候选者会一直处于这种状态，直到以下三种情况之一发生：(a) 它赢得了选举；(b) 另一台服务器确立了自己的领导者地位；或(c) 另一台服务器确立了自己的领导者地位。

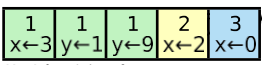
(c) 一段时间后没有获胜者。下文将分别讨论这些情况。

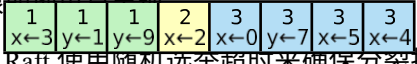
如果一个候选者在同一任期内获得整个群集中大多数服务器的投票，则该候选者将赢得选举。每台服务器在给定任期内最多为一名候选人投票，先到先得（注：第 5.4 节增加了对投票的额外限制）。多数规则确保在某一任期内最多只有一名候选人能赢得选举（图 3 中的 "选举安全提议"）。候选人一旦赢得选举，就成为领导者。然后，它将向所有其他服务器发送心跳信息，以建立自己的权威并防止出现新的选举。

在等待投票期间，候选者可能会收到来自其他服务器的自称为领导者的 AppendEntries RPC。如果领导者的任期（包含在其 RPC 中）至少与候选者当前任期一样大，那么候选者就会承认领导者是合法的，并返回追随者状态。如果 RPC 中的任期小于候选者的当前任期，则候选者拒绝 RPC 并继续保持候选者状态。

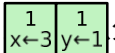
第三种可能的结果是，候选人既没有赢得选举，也没有输掉选举：如果许多追随者同时成为候选人，选票可能会被分割，从而没有候选人获得多数票


。当这种情况发生时，每个候选人都会超时，并通过增加任期和发起新一轮请求投票 RPC 开始新的选举。

。  的措施，分裂的选票可能会无限期地重复出现。

 Raft 使用随机选举超时来确保分裂投票的罕见性和快速解决。

为了从一开始就防止分裂投票，选举超时是从一个固定间隔（例如 150-300ms）中随机选择的。

。  分散服务器，在大多数情况下只有一台服

务  在其他服务器超时之前发送心跳。同样的机制也用于处理分裂投票。

每个候选人都会在选举开始时重启其随机选举超时，并等待超时结束后再开始下一次选举；这就降低了在新的选举中再次出现分裂投票的可能性。第 9.3 节显示，这种方法能迅速选出领导者。

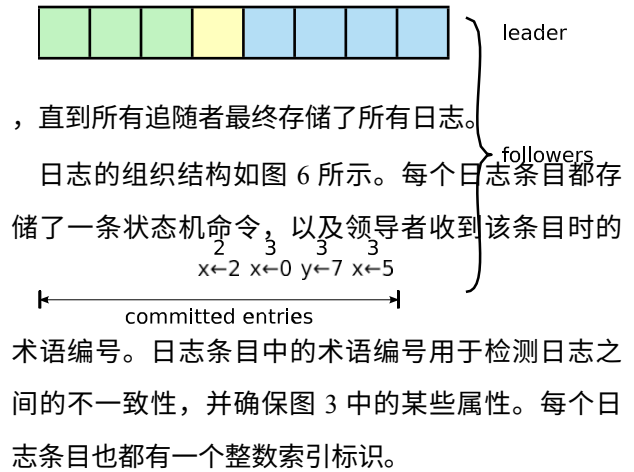
1 2 3 4 5 6 7 8 log index

图 6：日志由按顺序编号的条目组成。每个条目都包含创建时的术语（每个方框中的数字）和状态机的命令。如果一个条目可以安全地应用于状态机，那么该条目就被视为 *已提交*。

选举就是一个例子，说明可理解性是如何指导我们在各种设计方案之间做出选择的。起初，我们计划使用一个排名系统：为每个候选人分配一个唯一的排名，用于在相互竞争的候选人中进行选择。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（如果排名较高的服务器出现故障，排名较低的服务器可能需要超时并再次成为候选者，但如果超时过早，就会重置选举领导者的进程）。我们对算法进行了多次调整，但每次调整后都会出现新的死角。最终我们得出结论，随机重试的方法更明显、更容易理解。

### 5.3 日志复制

一旦选出领导者，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状态机执行的命令。领导者将命令作为一个新条目添加到日志中，然后并行向其他每个服务器发送 `AppendEntries` RPC，以复制该条目。当条目被安全复制后（如下所述），领导者会将条目应用到其状态机，并将执行结果返回给客户端。如果追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 "添加条目" RPC（即使在对客户端做出响应之后）



确定其在日志中的位置。

领导者决定何时可以安全地将日志应用于状态机；这样的条目称为已提交条目 (commit-*ted*)。Raft 保证提交的条目是持久的，最终将由所有可用的状态机执行。一旦创建日志条目的领导者在大多数服务器上复制了该条目 (如图 6 中的条目 7)，该条目即被提交。这也会提交领导者日志中的所有先前条目，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该规则的一些微妙之处，同时也说明了这种承诺定义是安全的。领导者会跟踪它所知道的已提交的最高索引，并在未来的 AppendEntries RPC (包括心跳) 中包含该索引，以便其他服务器最终发现。追随者一旦得知日志条目已提交，就会将该条目应用到其本地状态机 (按日志顺序)。

我们设计 Raft 日志机制的目的是在不同服务器上的日志之间保持高度一致性。这不仅简化了系统行为，使其更具可预测性，而且也是确保安全的重要组成部分。Raft 维护以下适当关系，它们共同构成了图 3 中的日志匹配属性：

- 如果不同日志中的两个条目具有相同的索引和术语，那么它们存储的是相同的命令。
- 如果不同日志中的两个条目具有相同的索引和项，那么这两个日志在前面的所有内容中都是相同的。

条目。

第一个特性源于这样一个事实，即领导者在给定术语中最多创建一个具有给定日志索引的条目，而且日志条目在日志中的位置永远不会改变。第二个特性由 AppendEntries 执行的简单一致性检查来保证。在发送 AppendEntries RPC 时，领导者会在其日志中加入紧跟新条目之前的条目的索引和术语。如果跟随者在其日志中找不到具有相同索引和术语的条目，则拒绝接收新条目。一致性检查是一个归纳步骤：日志的初始空状态满足日志匹配属性，无论

日志何时扩展，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，领导者就会知道，从新条目开始，跟随者的日志与自己的日志完全相同。

(c) 在正常运行期间，所有复制的日志保持一致，因此 AppendEntries 的一致性检查不会失败。但是，领导者崩溃会导致日志不一致 (旧领导者可能没有完全复制其日志中的所有条目)。这些不一致性会随着一系列领导者和追随者的崩溃而加剧。图 7 展示了追随者日志与新领导者日志的不同之处。追随者可能

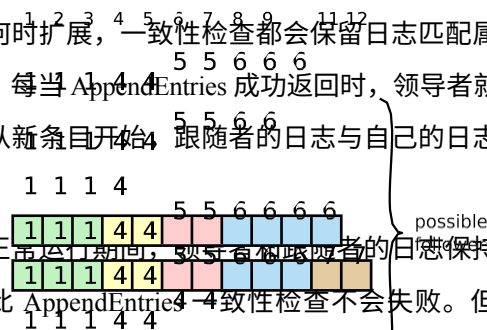
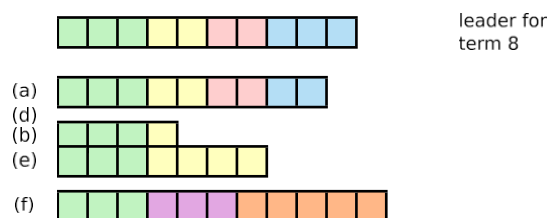


图 7：当高层领导掌权时，追随者日志中可能出现（a-f）中的任何一种情况。每个方框代表一个日志条目；方框中的数字是其期限。追随者可能缺少条目（a-b），可能有额外的未承诺条目（c-d），或两者皆有（e-f）。例如，(f) 情况可能发生在以下情况：该服务器是第 2 学期的领导者，在日志中添加了几个条目，但在提交任何条目之前就崩溃了；它很快重新启动，成为第 3 学期的领导者，并在日志中添加了几个条目；在提交第 2 学期或第 3 学期的任何条目之前，该服务器再次崩溃，并在几个学期内一直处于宕机状态。

日志可能缺少领导者中存在的条目，也可能有领导者中不存在的额外条目，或两者兼而有之。日志中的缺失条目和额外条目可能跨越多个学期。

在 Raft 中，领导者处理不一致问题的方式是强迫跟随者的日志复制自己的日志。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 节将说明，如果再加上一个限制条件，这种方法是安全的。

要使追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近日志条目，删除追随者日志中该点之后的任何条目，并向追随者发送领导者日志中该点之后的所有条目。所有这些操作都是为了响应 AppendEntries RPC 执行的一致性检查。领导者会为每个跟随者维护一个 *nextIndex*，它是领导者将发送给该跟随者的下一个日志条目的索引。领导者首次启动时，会将所有 *nextIndex* 值初始化为其日志中最后一条日志（图 7 中的 11）之后的索引。如果追随者的日志与领导者的不一致，下一次 AppendEntries RPC 的一致性检查就会失败。被拒绝后，领导者会递减 *nextIndex* 并重试 AppendEntries RPC。最终，*nextIndex* 会达到领导者



日志和跟随者日志相匹配的程度。这时，AppendEntries 就会成功，删除追随者日志中任何冲突的条目，并从领导者日志中添加条目（如果有的话）。一旦 AppendEntries 成功，追随者的日志就会与领导者的日志保持一致，并在任期内一直如此。

如果需要，可以对协议进行优化，以减少被拒绝的 AppendEntries RPC 数量。例如，在拒绝 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，领导者就可以递减 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将需要一个 AppendEntries RPC，而不是每个条目一个 RPC。在实践中，我们怀疑是否有必要进行这种优化，因为失败发生的频率很低，而且不太可能出现很多不一致的尝试。

有了这种机制，领导者在启动时无需采取任何特殊措施来恢复日志一致性。它只需开始正常运行，日志就会自动收敛，以应对 Append-Entries 一致性检查的失败。领导者从不覆盖或删除自己日志中的条目（图 3 中的“领导者仅应用属性”）。

这种日志复制机制具有第 2 节所述的理想共识特性：只要大多数服务器正常运行，Raft 就能接收、复制和应用新的日志条目；在正常情况下，只需向集群中的大多数服务器发送一轮 RPC，就能复制一个新条目；单个慢速跟随者不会影响性能。

## 5.4 安全

前面几节介绍了 Raft 如何选举领导者和复制日志条目。然而，迄今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，当领导者提交若干日志条目时，跟随者可能无法使用，然后它可能被选为领导者并用新条目覆盖这些条目；因此，不同的状态机可能会执行不同的命令序列。

本节通过增加对哪些服务器可以当选领导者的限制，完善了 Raft 算法。该限制确保任何给定任期的领导者都包含之前任期的所有承诺条目（图 3 中的“领导者完整性属性”）。有了选举限制，我们就能更精确地制定提交规则。最后，我们给出了领导者完备性属性的证明草图，并展示了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，领导者必须最终存储所有已提交的日志条目。在某些共识算法（如 View-hopping [22]）中，即使领导者最初不包含所有承诺条目，也可以当选。这些算法包含额外的机制来识别缺失的条目，并在选举过程中或选举结束后不久将其传输给新的领导者。不幸的是，这导致了相当大的额外机制和复杂性。Raft 使用了一种更简单的方法，它能保证所有已提交的条目都能在选举过程中或选举结束后的短时间内传送给新的领导者。





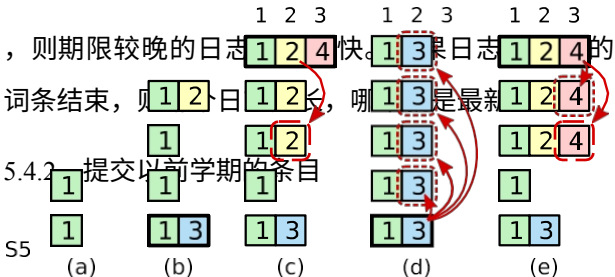
图 8：时间序列显示领导者为何无法使用旧任期的日志条目来终止承诺。在

(a) S1 是领导者，部分复制索引处的日志条目  
2.在 (b) 中，S1 崩溃；S5 在 S3、S4 和它自己的投票下当选为第 3 期的领导者，并接受日志索引 2 的不同条目。在 (c) 中，S5 崩溃；S1 重新启动，当选领导并继续复制。此时，第 2 项日志条目已在大多数服务器上复制，但尚未提交。如果 S1 如 (d) 所示崩溃，S5 可以当选领导者（获得 S2、S3 和 S4 的投票），并用自己的第 3 学期条目覆盖该条目。但是，如果 S1 在崩溃前在大多数服务器上复制了当前任期的条目（如 (e) 所示），那么该条目就会被提交（S5 无法赢得选举）。此时，日志中前面的所有条目也都被提交。

每个新的领导者从当选开始，其日志中就会出现这些条目，而无需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重复写入其日志中的现有条目。

Raft 利用投票流程阻止候选人赢得选举，除非其日志包含所有已提交条目。候选人必须联系到集群中的大多数服务器才能当选，这意味着每个已提交条目都必须至少出现在其中一个服务器中。如果候选者的日志至少与大多数服务器中的其他日志一样是最新的（"最新"的定义见下文），那么它就会包含所有已提交的条目。请求投票 RPC 实现了这一限制：RPC 包括候选者日志的信息，如果投票者自己的日志比候选者的日志更新，投票者就会拒绝投票。

Raft 通过比较日志中最后条目的索引和术语来确定哪个日志更及时。如果日志的最后条目期限不同



如第 5.3 节所述，一旦当前任期内的某个条目存储在大多数服务器上，领导者就会知道该条目已被提交。如果领导者在提交条目前崩溃，未来的领导者会尝试完成条目的复制。但是，一旦上一任期的条目存储在大多数服务器上，领导者就无法准确断定该条目已被提交。图



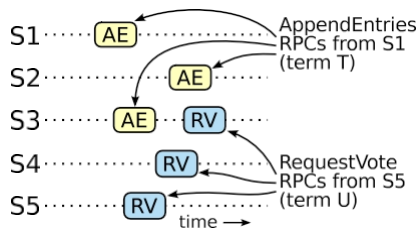


图 9: 如果 S1 (任期 T 的领导者) 提交了其任期内的新日志条目, 而 S5 当选为以后任期 U 的领导者, 那么肯定至少有一台服务器 (S3) 接受了日志条目, 并且也投票给了 S5。

图 8 展示了这样一种情况: 旧日志条目存储在大多数服务器上, 但仍有可能被未来的领导者覆盖。

为了消除类似图 8 中的问题, Raft 从不通过计算副本的方式提交前一术语的日志条目。只有领导者当前任期内的日志条目才会通过计算副本的方式提交; 一旦当前任期内的条目以这种方式提交, 那么之前的所有条目都会因为日志匹配属性而间接提交。在某些情况下, 领导者可以有把握地断定较早的日志条目已被提交 (例如, 如果该条目存储在每一台服务器上), 但 Raft 为简单起见采取了更为保守的方法。

Raft 在承诺规则中产生了这种额外的复杂性, 因为当领导者复制前几期的条目时, 日志条目会保留其原来的期号。而在其他共识算法中, 如果新的领导者重新复制之前 "术语 "中的条目, 则必须使用新的 "术语编号"。Raft 的方法使日志条目更容易推理, 因为它们在不同时间和不同日志中保持相同的术语编号。此外, 与其他算法相比, Raft 中的新领导者从以前的术语中发送的日志条目更少 (其他算法必须发送重复的日志条目以重新编号, 然后才能提交)。

#### 5.4.3 安全论据

有了完整的 Raft 算法, 我们现在可以更精确地论证领导者完备性定理是否成立 (这一论证基于安全性证明; 见第 9.2 节)。我们假设领导者完备性属性不成立, 然后证明一个矛盾。假设术语 T 的领导者

(领导者<sub>T</sub>) 提交了其术语的日志条目, 但未来某个术语的领导者没有存储该日志条目。考虑最小的术语 U > T 的领导者 (领导者<sub>U</sub>) 不存储条目。

1. 已提交的条目在当选时必须不在领导人<sub>U</sub> 的日志中 (领导人从不删除或覆盖条目)。
2. 领导者<sub>T</sub> 在大多数群集上复制了该条目, 而领导者<sub>U</sub> 收到了来自大多数群集的投票。因此, 至少有一台服务器 ("投票者") 既接受了领导者<sub>T</sub> 的条目, 又投了赞成票。

$U$ ，如图 9 所示。选民是达成矛盾的关键。

3. 投票者在投票给领导者 $U$ 之前，必须接受了领导者 $T$ 的承诺条目；否则，投票者会拒绝领导者 $T$ 的 AppendEntries 请求（其当前任期会大于 $T$ ）。
4. 投票者在投票给领导者 $U$ 时仍存储了该条目，因为每一位介入的领导者都包含该条目（根据假设），领导者从不删除条目，而追随者只有在与领导者冲突时才会删除条目。
5. 投票人将其投票权授予了领导人 $U$ ，因此领导人 $U$ 的日志一定和投票人的日志一样是最新的。这就导致了两个矛盾中的一个。
6. 首先，如果投票人和领导者 $U$ 共享同一个最后一个日志项，那么领导者 $U$ 的日志肯定至少和投票人的日志一样长，所以它的日志包含了投票人日志中的每一个条目。这是一个矛盾，因为投票者包含了提交的条目，而领导者 $U$ 假设不包含。
7. 否则，领导者 $U$ 的最后一个对数项一定大于投票者的最后一个对数项。此外，它还大于 $T$ ，因为投票者的最后一个日志项至少是 $T$ （它包含了 $T$ 项中的已提交条目）。创建领导者 $U$ 的最后一个日志项的早先的领导者的日志中一定包含了已提交的条目（根据推断）。那么，根据日志匹配属性，领导者 $U$ 的日志也必须包含已提交条目，这是一个矛盾。
8. 这就完成了矛盾。因此，所有大于 $T$ 的术语的首项必须包含术语 $T$ 中的所有条目。
9. 日志匹配属性保证了未来的领导者也会包含间接提交的条目，如图 8(d) 中的索引 2。

有了领导者完备性属性，我们就可以证明图 3 中的状态机安全属性，即如果一台服务器在其状态机中应用了给定索引的日志条目，那么其他服务器将

不会在同一索引中应用不同的日志条目。当一台服务器在其状态机中应用日志条目时，它的日志必须与领导者的日志完全相同，直到该条目为止，而且该条目必须提交。现在考虑任何服务器应用给定日志索引的最低子项；日志完整性属性保证所有更高子项的领导者都将存储相同的日志条目，因此在后面子项中应用该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 要求服务器按日志顺序应用条目。结合状态机安全特性，这意味着所有服务器都将以相同的顺序，在其状态机中应用完全相同的日志条目集。

## 5.5 追随者和候选人崩溃

在此之前，我们一直专注于领导者故障。追随者和候选者的崩溃比领导者的崩溃更容易处理，两者的处理方式也相同。如果追随者或候选者崩溃，那么发送给它的 RequestVote 和 AppendEntries RPC 就会失败。Raft 会通过无限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，那么它将在重启后再次收到相同的 RPC。RPC 是幂等的，因此不会造成任何损害。例如，如果追随者收到的 AppendEntries 请求包含了其日志中已有的日志条目，那么它就会在新的重新请求中忽略这些条目。

## 5.6 时间和可用性

我们对 Raft 的要求之一是安全性不能依赖于时间：系统不能因为某些事件发生得比预期快或慢而产生不正确的结果。但是，可用性（系统及时响应客户的能力）不可避免地取决于时间。例如，如果信息交流的时间超过服务器崩溃的典型间隔时间，候选人就无法保持足够的时间赢得选举；没有稳定的领导者，Raft 就无法取得进展。

领导者选举是 Raft 最关键的计时环节。只要系统满足以下 *计时要求*，Raft 就能选出并保持稳定的领导者：

$$\text{broadcastTime} \ll \text{electionTimeout} \ll \text{MTBF}$$

在这个不等式中，*广播时间* (*broadcastTime*) 是一台服务器并行向集群中的每台服务器发送 RPC 并接收其响应所需的平均时间；*选举超时* (*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。广播时间应比选举超时小一个数量级，这样领导者就能可靠地发送心跳信息，以防止从属者开始选举；考虑到选举超时采用的随机方法，这种不等式也使得分裂投票不太可能发生。选举超时应该比 MTBF 小几个数量级，这样系统才能稳步发展。当领导者崩溃时，系统将在大

约选举超时时间内不可用；我们希望这只占总时间的一小部分。

广播时间和平均无故障时间是未衍生系统的属性，而选举超时则是我们必须选择的。Raft 的 RPC 通常要求接收方将信息持久保存到稳定的存储中，因此广播时间可能在 0.5ms 到 20ms 之间，具体取决于存储技术。因此，选举超时可能介于 10ms 和 500ms 之间。典型情况

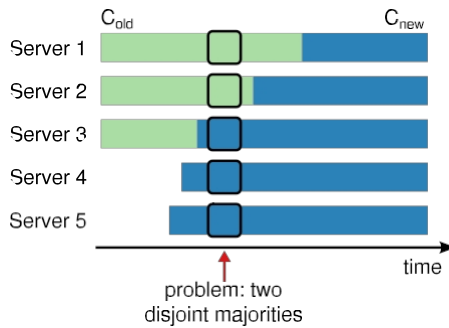


图 10：直接从一种配置切换到另一种配置是不安全的，因为不同的服务器会在不同的时间切换。在这个例子中，群集从三台服务器增加到五台。不幸的是，在某个时间点上，会有两个不同的领导者当选为同一任期的领导者，其中一个拥有旧配置的多数（ $C_{old}$ ），另一个拥有新配置的多数（ $C_{new}$ ）。

服务器的 MTBF 为几个月或更长，很容易满足时间要求。

## 6 群组成员变更

到目前为止，我们一直假设集群配置（参与共识算法的服务器集）是固定的。在实践中，偶尔需要更改配置，例如在服务器出现故障时更换服务器或更改复制程度。虽然可以通过下线整个群集、更新配置文件然后重新启动群集来完成，但这将导致群集在切换期间不可用。此外，如果有任何手动步骤，操作员就有可能出错。为了避免这些问题，我们决定自动更改配置，并将其纳入 Raft 共识算法。

为了保证配置变更机制的安全性，在过渡期间必须不存在两个领导者在同一任期内当选的可能。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。不可能一次原子式地切换所有服务器，因此在过渡期间，群集有可能分裂成两个独立的多数（见图 10）。

为确保安全，配置更改必须采用两阶段方法。实现这两个阶段的方法多种多样。例如，有些系统（如 [22]）会在第一阶段禁用旧配置，使其无法处理客户端请求；然后在第二阶段启用新配置。在 Raft 系统中，集群首先切换到我们称之为联合共识的过

渡配置；一旦联合共识被提交，系统就会切换到新配置。联合共识结合了新旧配置：

- 日志条目会复制到两种配置中的所有服务器上。

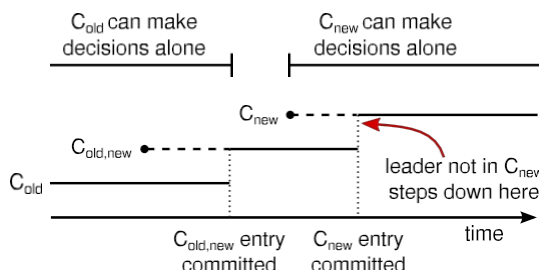


图 11：配置更改时间表。虚线表示已创建但未提交的配置项，实线表示最新提交的配置项。领导者首先在日志中创建  $C_{old,new}$  配置条目，并将其提交到  $C_{old,new}$  ( $C_{old}$  的多数和  $C_{new}$  的多数)。然后，它创建  $C_{new}$  条目，并将其提交到  $C_{new}$  的大部分。在任何时候， $C_{old}$  和  $C_{new}$  都无法独立做出决定。

- 任一配置的任何服务器都可担任领导者。
- 协议（选举和加入承诺）要求新老双方分别获得多数票。

配置。

联合共识允许单个服务器在不同时间在不同配置之间转换，而不保证安全。此外，联合共识还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。

集群配置通过复制日志中的特殊条目进行存储和通信；图 11 展示了配置更改过程。当领导者收到将配置从  $C_{old}$  更改为  $C_{new}$  的请求时，它会用于达成联合共识的配置（图中的  $C_{old,new}$ ）存储为日志条目，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日志中，它就会在今后的所有决策中使用该配置（无论条目是否已提交，服务器始终使用其日志中的最新配置）。这意味着，领导者将使用  $C_{old,new}$  的规则来决定何时提交  $C_{old,new}$  的日志条目。如果领导者崩溃，则会在  $C_{old}$  或  $C_{old,new}$  下选择新的领导者，这取决于获胜的候选者是否收到了  $C_{old,new}$ 。在任何情况下， $C_{new}$  都不能在此期间做出单方面决定。

一旦提交了  $C_{old,new}$ ， $C_{old}$  和  $C_{new}$  都不能在未经对方同意的情况下做出决定，而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当选为领导

者。现在，领导者可以安全地创建描述  $C_{new}$  的日志条目，并将其复制到群集中。同样，该配置一旦被看到，就会在每台服务器上生效。当新配置根据  $C_{new}$  的规则提交后，旧配置就无关紧要了，不在新配置中的服务器可以关闭。如图 11 所示，在任何时候， $C_{old}$  和  $C_{new}$  都不能做出单方面的决定；这就保证了安全性。

重新配置还需要解决三个问题。第一个问题是，新服务器最初可能不会存储任何日志条目。如果在这种状态下将它们添加到群集，它们可能需要相当长的时间才能跟上，在此期间可能无法获取新的日志条目。为了避免出现可用性缺口，Raft 在更改配置前引入了一个额外阶段，在这一阶段中，新服务器作为无投票权成员加入集群（领导者向它们复制日志条目，但它们不被视为多数）。一旦新服务器跟上群集其他服务器的步伐，重新配置就可以如上所述进行。

第二个问题是，群集领导者可能不是新配置的一部分。在这种情况下，一旦领导者提交了  $C_{\text{new}}$  日志条目，它就会退出（返回追随者状态）。这意味着，在一段时间内（提交  $C_{\text{new}}$  时），领导者将管理一个不包括自己在内的群集；它复制日志条目，但不将自己计入多数。当  $C_{\text{new}}$  提交后，领导者就会发生转换，因为这是新配置可以独立运行的第一个时间点（始终可以从  $C_{\text{new}}$  中选择领导者）。在此之前，可能只有来自  $C_{\text{old}}$  的服务器才能当选领导者。

第三个问题是被移除的服务器（不在  $C_{\text{new}}$  中的服务器）会扰乱群集。这些服务器不会重新接收心跳，因此会超时并开始新的选举。然后，它们将发送带有新任期编号的 RequestVote RPC，这将导致当前的领导者恢复到追随者状态。最终将选出新的领导者，但被移除的服务器将再次超时，这一过程将重复进行，导致可用性很差。

为了防止出现这个问题，服务器在认为有现任领导人存在时，会忽略 RequestVote RPC。具体来说，如果服务器在听到当前领导者消息后的最短选举超时内收到 RequestVote RPC，则不会更新其任期或授予其投票权。这不会影响正常选举，因为在正常选举中，每个服务器在开始选举前至少要等待一个最小选举超时。不过，这有助于避免重新移动服务器造成的干扰：如果领导者能够向其群集发送心跳

，那么它就不会被更大的任期数所取代。

## 7 原木压实

Raft 的日志在正常运行期间会不断增长，以容纳更多的客户端请求，但在实际系统中，日志不可能无限制地增长。日志越长，占用的空间就越大，重放所需的时间也就越长。如果没有某种机制来丢弃日志中累积的过时信息，这最终会导致可用性问题。

快照是最简单的压缩方法。在快照过程中，整个系统的当前状态会被写入稳定存储上的快照，然后整个日志直到

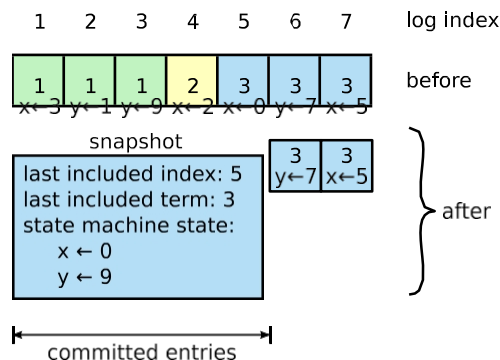


图 12：服务器用新快照替换日志中已提交的条目（索引 1 至 5），新快照只存储当前状态（本例中为变量  $x$  和  $y$ ）。快照最后包含的索引和术语用于将快照定位在条目 6 之前的日志中。

该点将被丢弃。Chubby 和 ZooKeeper 中使用了快照功能，本节其余部分将介绍 Raft 中的快照功能。

增量压缩方法也是可行的，例如日志清理 [36] 和日志结构合并树 [30, 5]。这些方法一次只对部分数据进行操作，因此能在一段时间内更均匀地分散压缩负荷。它们首先会选择一个数据区域，该区域已累积了许多被删除和覆盖的对象，然后它们会更有效地重写该区域的实时对象，并释放该区域。与快照相比，这需要大量额外的机制和复杂性，而快照总是对整个数据集进行操作，从而简化了问题。日志清理需要对 Raft 进行修改，而状态机可以使用与快照相同的接口来实现 LSM 树。

图 12 显示了 Raft 中快照的基本思想。每台服务器都独立拍摄快照，只覆盖其日志中已提交的条目。大部分工作包括将状态机的当前状态写入快照。Raft 还在快照中包含少量元数据：**最后包含的索引**是快照所取代的日志中最后一个条目的索引（状态机应用的最后一次尝试），**最后包含的术语**是该条目术语。保留这些信息是为了对快照后的第一个日志条目进行 AppendEntries 一致性检查，因为该条目需要先前的日志索引和术语。为实现群集成员变更（第 6 节），快照还包括日志中截至上次包含索引时的最新配置。服务器完成写入快照后，可删除截至最后

包含索引的所有日志条目以及之前的快照。

虽然服务器通常都会独立拍摄快照，但领导者偶尔也必须向落后的追随者发送快照。当领导者已经丢弃了需要发送给跟随者的下一个日志条目时，就会发生这种情况。幸运的是，这种情况在正常运行中不太可能发生：跟上领导者快照的跟随者可能会在下一个日志条目中丢弃快照。

<b>论据:</b>	
<b>领导者任期</b>	追随者可以重新定向客户
<b>领导者Id</b>	追随者可以重新定向客户
<b>最后包含的索引 (lastIncludedIndex)</b>	快照会替换包括该 lastIncludedTerm 的术语
<b>偏移量</b>	数据块在快照有序位置的字节偏移量
<b>data[]</b>	快照块的原始字节, 从
<b>done</b>	如果这是最后一个数据块, 则为 true
<b>结果</b>	当前术语, 以便领导者自行更新
<b>安装器的实施:</b>	
1. 如果 term < currentTerm, 立即回复	
2. 如果是第一个数据块 (偏移量为 0), 则创建新的快照文件	
3. 在给定偏移量处将数据写入快照文件	
4. 如果 done 为 false, 则回复并等待更多数据块	
5. 保存快照文件, 丢弃索引较小的任何现有或部分快照	
6. 如果现有日志条目与快照最后包含的条目具有相同的索引和术语, 则保留其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 (并加载快照的群集配置)

图 13: InstallSnapshot RPC 的摘要。快照会被分割成若干小块进行传输; 这就为后续程序提供了每个小块的生命迹象, 以便重置其选举计时器。

领导者已经有了这个条目。但是, 速度特别慢的追随者或新加入群集的服务器 (第 6 节) 就没有了。让这样的跟随者更新的方法是由领导者通过网络向其发送快照。

领导者使用名为 InstallSnapshot 的新 RPC 向落后太多的跟随者发送快照; 见图 13。当追随者通过此 RPC 收到快照时, 它必须决定如何处理其现有的日志记录。通常, 快照会包含收件人日志中尚未包含的新信息。在这种情况下, 跟随者会丢弃其全部日志; 这些日志已被快照取代, 而且可能有与快照冲突的未提交条目。相反, 如果追随者收到的快照描述了其日志的前缀 (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。

这种快照方法背离了 Raft 的强势领导原则, 因为追随者可以在领导者不知情的情况下拍摄快照。不过, 我们认为这种偏离是合理的。虽然有一个领导者有助于避免在达成共识时出现决策冲突, 但在快照时已经达成了共识, 因此不会出现决策冲突。数据仍然只能从领导者流向跟随者。



现在，只有追随者可以重组他们的数据。

我们考虑过另一种基于领导者的方法，即只有领导者创建快照，然后将快照发送给每个追随者。然而，这种方法有两个缺点。首先，向每个追随者发送快照会浪费网络带宽，并减慢快照处理速度。每个追随者都已经掌握了生成自己快照所需的信息，而且服务器从本地状态生成快照的成本通常要比通过网络发送和接收快照的成本低得多。其次，领导者的实施将更加复杂。例如，领导者在向追随者发送快照的同时，还需要向追随者回复新的日志条目，以免阻塞新的客户端请求。

还有两个问题会影响快照的性能。首先，服务器必须决定何时快照。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，就会有耗尽存储容量的风险，而且会增加重启时重放日志所需的时间。一种简单的策略是在日志达到以字节为单位的固定大小时进行快照。如果将该大小设定为明显大于快照的预期大小，那么快照所需的磁盘带宽就会很小。

第二个性能问题是，写快照可能需要大量时间，我们不希望因此耽误正常操作。解决方法是使用写入时复制技术，这样可以在不影响正在写入的快照的情况下接受新的更新。例如，使用功能数据结构构建的状态机自然支持这种方法。另外，也可以使用操作系统的写入时复制支持（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

本节将介绍客户端与 Raft 的交互方式，包括客户端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，Raft 的解决方案与其他系统类似。

Raft 的客户端将所有请求发送给领导者。客户端

首次启动时，会连接到随机选择的服务器。如果客户端首先选择的不是领导者，该服务器将拒绝客户端的

请求，并提供它最近听到的领导者的信息（AppendEntries 请求包括领导者的网络地址）。如果领导者崩溃，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试一次。我们对 Raft 的目标是实现可线性化的语义（每个操作在调用和响应之间的某一时刻似乎都是瞬时执行的，恰好执行一次）。

不过，如前所述，Raft 可以多次执行一条命令：例如，如果领导者

在提交日志条目后但在响应客户端之前，客户端会使用新的领导者重试命令，导致命令被第二次执行。解决方法是让客户端为每条命令分配唯一的序列号。然后，状态机会跟踪每个客户端处理的最新序列号，以及相关的响应。如果收到序列号已被执行过的命令，它会立即做出响应，而不会重新执行请求。

只读操作可以在不向日志写入任何内容的情况下进行处理。但是，由于没有额外的监控措施，这将面临返回陈旧数据的风险，因为响应请求的领导者可能已经被新的领导者取代，而它却不知道。Linearizable 读取必须不会返回过期数据，而 Raft 需要两个额外的预防措施，才能在不使用日志的情况下保证这一点。首先，领导者必须掌握已提交条目的最新信息。领导者完整性属性（Leader Completeness Property）保证领导者拥有所有已提交的条目，但在任期开始时，它可能不知道哪些是已提交的条目。要想知道，它需要提交其任期内的条目。Raft 的处理方法是，让每个领导者在任期开始时向日志中提交一个空白的无操作条目。其次，在处理只读请求之前，领导者必须检查自己是否已被删除（如果有新的领导者当选，其信息可能已经过时）。Raft 的处理方法是让领导者在回应只读请求前与集群中的大多数人交换心跳信息。或者，领导者也可以依靠心跳机制来提供一种租约形式 [9]，但这需要依赖时间来保证安全性（它假定时钟偏移是有界的）。

## 9 实施和评估

我们将 Raft 作为复制状态机的一部分来实现，该状态机用于存储 RAMCloud [33] 的配置信息，并协助 RAMCloud 协调器的故障转移。Raft 的实现包含大约 2000 行 C++ 代码，不包括测试、注释或空行。源代码可免费获取 [23]。根据本文的草稿，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。此外，多家公司也在部署基于 Raft 的系统 [34]。

本节的其余部分将使用三个标准对 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，我们使用斯坦福大学高级操作系统课程和加州大学伯克利分校分布式计算课程的高年级本科生和研究生进行了一项实验研究。我们录制了 Raft 和 Paxos 的视频讲座，并制作了相应的测验。除日志压缩外，Raft 讲座涵盖了本文的所有内容；Paxos 讲座涵盖了本文的所有内容。

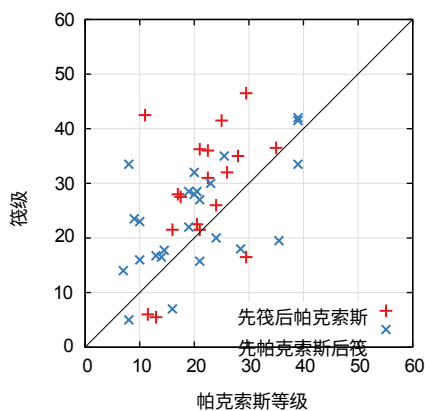
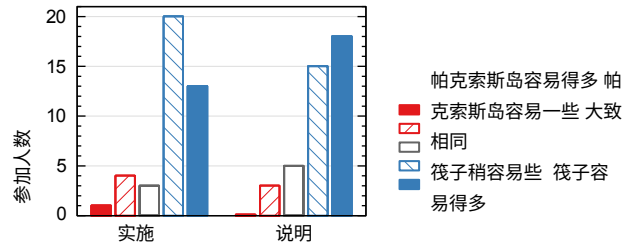


图 14：比较 43 位参与者在 Raft 和 Paxos 测验中表现的散点图。对角线上方的点（33）代表 Raft 得分较高的参与者。

讲座的内容足以创建一个等效的复制状态机，包括单自由度 Paxos、多自由度 Paxos、重新配置和一些实践中需要的优化（如领导者选举）。测验测试了学生对算法的基本理解，还要求学生从拐角情况进行推理。每个学生观看一段视频，参加相应的测验，然后观看第二段视频，参加第二次测验。大约一半的学员先做了 Paxos 部分，另一半学员先做了 Raft 部分，以考虑到学员在成绩上的个体差异和从第一部分学习中获得的经验。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。

我们努力使 Paxos 和 Raft 之间的比较尽可能公平。实验在两个方面有利于 Paxos：在 43 名参与者中，有 15 人表示之前曾使用过 Paxos，而且 Paxos 视频比 Raft 视频长 14%。如表 1 所示，我们已采取措施减少潜在的偏差来源。我们的所有材料均可供查阅 [28, 31]。

参与者在 Raft 测验中的平均得分比在 Paxos 测验中的平均得分高出 4.9 分（满分为 60 分，Raft 的平均得分为 25.7 分，Paxos 的平均得分为 20.8 分）；图 14 显示了他们的个人得分。配对  $t$  检验表明，在 95% 的置信度下，Raft 分数的真实分布的平均值至少比 Paxos 分数的真实分布的平均值大 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生的测验分数：他们参加了哪次测验、他们之前的 Paxos 体验程度以及他们的测验成绩。

图 15：采用 5 级评分法，询问参与者  
 （左图）他们认为哪种算法更容易在一个正常、正确和高效的系统中实施，（右图）哪种算法更容易向计算机科学系的研究生解释。

他们学习算法的顺序。根据模型预测，小测验的选择会产生 12.5 分的差异，而 Raft 更有利。这明显高于观察到的 4.9 分的差异，因为许多实际学生之前都有过 Paxos 经验，这对 Paxos 有很大帮助，而对 Raft 的帮助则略小。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；虽然我们不知道原因何在，但这似乎在统计学上是有意义的。

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施或解释；结果如图 15 所示。绝大多数参与者都认为 Raft 更容易实施和解释（每个问题 41 人中有 33 人）。不过，这些自我

报告的感受可能不如参与者的测验分数可靠，而且参与者可能会因为知道我们的假设 Raft 更容易理解而产生偏差。

有关 Raft 用户研究的详细讨论见 [31]。

## 9.2 正确性

我们为第 5 节中描述的共识机制开发了正式规范和安全证明。正式规范[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，是证明的主题。对于实现 Raft 的任何人来说，它本身也很有用。我们使用 TLA 证明系统 [7] 机械地证明了日志完备性属性。然而，该证明依赖于未经机械检查的不变式（例如，我们尚未证明规范的类型安全性）。此外，我们还编写了状态机安全属性的非正式证明 [31] ，该证明是完整的（它仅依赖于规范），并且是与我们的证明系统[2]相关的。

关注	为减少偏见而采取的措施	审查材料[28, 31]
授课	质量相同，授课教师相同。Paxos 讲座基于多所大学使用的现有教材，并对其进行了改进。Paxos 讲座时间长 14%。	视频
测验难度相等	试题按难度分组并在各次考试中配对。	小测验
公平评分标准	使用评分标准按随机顺序评分，小测验之间交替进行。	评分

表 1：对研究中可能存在的针对 Paxos 的偏见的担忧、为消除这些担忧而采取的措施以及可获得的补充材料。

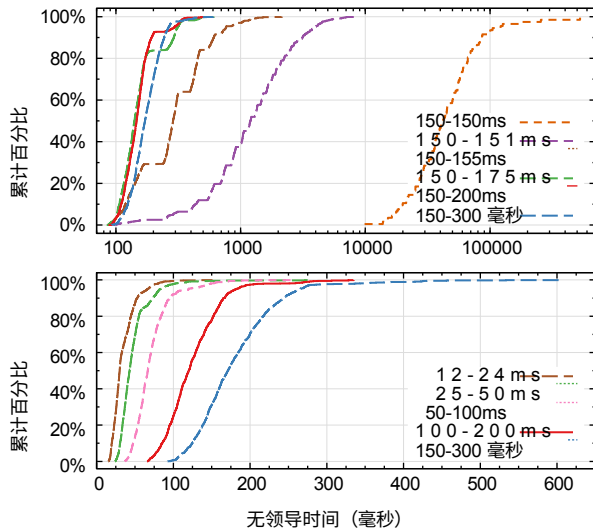


图 16：检测和替换崩溃领导者的时间。上图是选举超时随机性的变化，下图是最小选举超时的缩放。每条线代表 1000 次测试（"150-150ms" 的 100 次测试除外），并对应特定的选举超时选择；例如，"150-155ms" 表示选举超时在 150ms 和 155ms 之间随机均匀选择。测量是在由五台服务器组成的群集上进行的，广播时间约为 15ms。九台服务器集群的结果与此类似。

它非常精确（长约 3500 字）。

### 9.3 性能

Raft 的性能与 Paxos 等其他共识算法类似。对于性能而言，最重要的情况是已建立的领导者正在复制新的日志条目。Raft 使用最少的信息量（从领导者到半个集群的单次往返）实现了这一目标。还可以进一步提高 Raft 的性能。例如，它可以轻松支持批处理和流水线处理请求，以提高吞吐量和降低延迟。文献中对其他算法提出了各种优化建议；其中许多都可以应用于 Raft，但我们将其留待未来工作中进行。

我们使用 Raft 实现来衡量 Raft 领导者选举算法的性能，并回答了两个问题。第一，选举过程是否能快速收敛？第二，领导者崩溃后的最短停机时间是

ing)。在所有测试中，领导者都是在心跳间隔内随机崩溃的，心跳间隔是最小选举超时的一半。因此，最短的宕机时间约为最短选举超时的一半。

图 16 的上图显示，选举超时中的少量随机化足以避免选举中的分裂投票。在没有随机性的情况下，我们的测试中领导者选举的时间一直超过 10 秒，原因是出现了许多分裂投票。仅增加 5 毫秒的随机性就有很大帮助，导致停机时间中位数为 287 毫秒。使用更多随机性可改善最坏情况下的表现：使用 50 毫秒随机性时，最坏情况下的完成时间（超过 1000 次测试）为 513 毫秒。

图 16 的下图显示，减少选举超时可以缩短停机时间。随着多少？

为了衡量领导者选举情况，我们反复让一个由五台服务器组成的群集的领导者崩溃，并计算检测到崩溃和选举新领导者所需的时间（见图 16）。为了产生最坏的情况，每次试验中的服务器都有不同的日志长度，因此一些候选人没有资格成为领导者。此外，为了防止投票分裂，我们的测试脚本在终止进程前触发了领导者的心跳 RPC 同步广播（这近似于领导者在崩溃前复制新日志条目的行为）。

选举超时为 12-24 毫秒，而在选举领导人的平均时间（最长试验耗时 152 毫秒）。然而，将超时时间降低到这一点之后，就违反了 Raft 的计时要求：领导者很难在其他服务器开始新的选举之前发出宽泛的心跳。这会导致不必要的领导者更换，并降低系统的整体可用性。我们建议使用一个有效的选举超时，如 150-300ms；这样的超时不太可能导致不必要的领导者更换，而且仍能提供良好的可用性。

## 10 相关工作

与同感算法有关的出版物不胜枚举，其中很多都属于以下类别：

- 兰波特对帕克索斯的最初描述[15]，以及在诱导下对其进行更清晰的解释[16, 20, 21]。
- Paxos 的阐述，填补了缺失的细节并修改了算法，以提供更好的基础。  
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 的算法虽然都声称基于 Paxos，但详细内容尚未公布。ZooKeeper 的算法已经公布了更多细节，但与 Paxos 截然不同。
- 可应用于 Paxos 的性能优化 [18, 19, 3, 25, 1, 27]。
- Oki 和 Liskov 的 Viewstamped Replication (VR)，是一种围绕共识开发的替代方法。

与 Paxos 同时诞生。最初的描述[29]与分布式传输协议交织在一起，但最近的更新[22]已将核心共识协议分离出来。VR 采用基于领导者的方法，与 Raft 有许多相似之处。

Raft 与 Paxos 最大的不同在于 Raft 强大的领导力：Raft 将领导者选举作为共识协议的一个重要组成部分，并将领导者选举作为共识协议的一个重要组成部分，并将领导者选举作为共识协议的一个重要组成部分。

在领导者中尽可能多地集成功能。这种方法使得算法更简单，更容易理解。例如，在 Paxos 中，领导者选举与基本共识协议无关：它只是作为一种性能优化，并不是达成共识所必需的。不过，这也导致了额外的机制：Paxos 包括一个两阶段的基本共识协议和一个单独的领导者选举机制。相比之下，Raft 将领导者选举直接纳入共识算法，并将其作为共识两个阶段中的第一个阶段。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 也有许多共同的优势。不过，Raft 的机制不如 VR 或 ZooKeeper，因为它将非领导者的功能最小化了。例如，Raft 中的日志条目只有一个流动方向：在 AppendEntries RPC 中从领导者向外流动。而在 VR 中，日志条目是双向流动的（在选举过程中，领导者可以接收日志条目）；这导致了额外的机制和复杂性。已公布的 ZooKeeper 的描述也将日志条目传输到领导者和从领导者处传输，但其实现方式显然更像 Raft [35]。

就我们所知，Raft 的消息类型比任何其他基于共识的日志复制算法都要少。例如，我们统计了 VR 和 ZooKeeper 用于基本共识和成员变更的消息类型（不包括日志压缩和客户端交互，因为这些几乎与算法无关）。VR 和 ZooKeeper 各定义了 10 种不同的消息类型，而 Raft 只有 4 种消息类型（两个 RPC 请求及其响应）。Raft 的消息比其他算法的消息更密集一些，但总体上更简单。此外，VR 和 ZooKeeper 在描述领导者变更时会传输整个日志；要优化这些机制，使其切实可行，还需要额外的消息类型。

Raft 的强领导方法简化了算法，但排除了某些性能优化。例如，Egalitarian Paxos (EPaxos) 在某些条件下可以通过无领导方法实现更高的性能[27]。EPaxos 利用了状态机命令的交换性。任何服务器只

需进行一轮通信，就能提交一个命令，只要同时提出的其他命令与之换向即可。但是，如果同时提出的命令相互之间不换向，EPaxos 就需要额外的一轮通信。由于任何服务器都可以提交命令，因此 EPaxos 可以很好地平衡服务器之间的负载，在广域网环境下，其延迟比 Raft 更低。不过，它也大大增加了 Paxos 的复杂性。

在其他工作中，包括 Lamport 的原始提案[15]、VR[22]和 SMART [24]，已经提出或实施了几种不同的集群成员变更方法。我们为 Raft 选择了联合共识方法，因为它利用了 consensus 协议的其他部分，因此成员变更几乎不需要额外的机制。Lamport 基于  $\alpha$  的方法不适合 Raft，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势在于，成员变更可以在不限制正常请求处理的情况下进行；相反，VR 会在配置变更期间停止所有正常处理，而 SMART 则会对未决请求的数量施加类似  $\alpha$  的限制。与 VR 或 SMART 相比，Raft 的方法增加的机制更少。

## 11 结论

算法的设计通常以正确、高效和/或简洁为主要目标。尽管这些都是值得追求的目标，但我们认为易懂性同样重要。在开发人员将算法转化为实际实现之前，其他目标都无法实现，而实际实现将不可避免地偏离并扩展已发布的形式。除非开发人员对算法有深刻的理解，并能建立起对算法的直觉，否则很难在实现过程中保留算法的理想特性。

在本文中，我们探讨了分布式共识的问题，在这个问题上，一种广为接受但难以理解的算法--Paxos--多年来一直困扰着学生和开发人员。我们开发了一种新算法 Raft，并证明它比 Paxos 更容易理解。我们还认为，Raft 为系统构建提供了更好的基础。将可理解性作为首要设计目标改变了我们开发 Raft 的方式；随着设计的深入，我们发现自己在重复使用一些技术，例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，也让我们更容易相信它的正确性。

## 12 致谢

如果没有 Ali Ghodsi、David Mazie`res 以及伯克利 CS 294-91 和斯坦福 CS 240 的学生的支持，这项

用户研究是不可能完成的。斯科特-克莱默（Scott Klemmer）帮助我们设计了用户研究，尼尔森-雷（Nelson Ray）就统计分析为我们提供了建议。用户研究的 Paxos 幻灯片大量借鉴了 Lorenzo Alvisi 最初制作的幻灯片。特别感谢 David Mazie`res 和 Ezra Hoch 发现了 Raft 中的细微错误。Ed Bugnion、Michael Chan、Hugues Evrard 等许多人对论文和用户研究材料提供了有益的反馈、



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 位无名会议审稿人（有重复），特别是我们的牧羊人 Eddie Kohler。沃纳-沃格尔斯（Werner Vogels）在推特上提供了一份早期草案的链接，这让 Raft 获得了极大的曝光率。这项工作得到了千兆级系统研究中心（Gigascale Systems Research Center）和多尺度系统中心（Multiscale Systems Center）的支持，这两个中心是由半导体研究公司（Semiconductor Research Corporation）的 "Focus Center Research Program" 计划资助的六个研究中心中的两个，还得到了 STARnet（由 MARCO 和 DARPA 赞助的 Semiconductor Research Corporation 计划）、美国国家科学基金会（National Science Foundation）的 0963859 号资助，以及 Facebook、Google、Mellanox、NEC、NetApp、SAP 和三星的资助。Diego Ongaro 由 Junglee Corporation 斯坦福大学毕业生奖学金资助。

## 参考资料

- [1] BOLOSKY, W. J., BRADSHAW, D., HAAGENS, R. B., KUSTERS, N. P., AND LI, P. Paxos 复制状态机作为高性能数据存储的基础。In *Proc.NSDI'11, USENIX Conference on Networked Systems Design and Implementation* (2011), USENIX, pp.141-154.
- [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.
- [3] Camargos, L. J., Schmidt, R. M., and Pedone, F. 多协调 PaxosIn *Proc.PODC'07, ACM Symposium on Principles of Distributed Computing* (2007), ACM, pp.
- [4] Chandra, T. D., Griesemer, R., and Redstone, J. 实时 Paxos：工程视角。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., BURROS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. Bigtable: A distributed storage system for structured data.In *Proc.OSDI'06, USENIX Symposium on Operating Systems Design and Implementation* (2006), USENIX, pp.
- [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。In *Proc.OSDI'12, USENIX Conference on Operating Systems Design and Implementation* (2012), USENIX, pp.

- [7] COUSINEAU, D., DOLIGEZ, D., LAMPORT, L., MERZ, S., RICKETTS, D., AND VANZETTO, H.  $TLA^+$  proofs. In *Proc.FM'12, 形式化方法研讨会* (2012 年)、D.Giannakopoulou and D. Me'ry, Eds., vol. 7436 of *Lecture Notes in Computer Science*, Springer, pp.
- [8] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T.的谷歌文件系统。 *SOSP'03, ACM Symposium on Operating Systems Principles* (2003), ACM, pp.
- [9] GRAY, C., AND CHERITON, D. Leases: 分布式文件缓存一致性的容错机制。第 12 届 ACM 操作系统原理研讨会论文集 (1989 年), 第 202-210 页。
- [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., JUNKEIRA, 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 Conference* (2010), USENIX, pp.
- [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 Dependable Systems & Networks* (2011), IEEE Computer Society, pp.245-256.
- [13] KIRSCH, J., AND AMIR, Y. Paxos for system builders. Tech.CNDS-2008-2, 约翰霍普金斯大学, 2008 年。
- [14] LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. *ACM Communications 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. 广义共识与 Paxos. Tech.MSR-TR-2005-33, 微软研究院, 2005 年。
- [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.
- [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, pp.
- [22] LISKOV, B., AND COWLING, J. Viewstamped replication revisited. Tech.MIT-CSAIL-TR-2012-021, 麻省理工学院, 2012 年 7 月。
- [23] LogCabin 源代码。 <http://github.com/logcabin/logcabin>.

- [24] LORCH, J. R., ADYA, A., BOLOSKY, W. J., CHAIKEN, R., DOUCEUR, J. R., AND HOWELL, J. The SMART 迁移复制的有状态服务的方法。In *Proc.EuroSys'06, ACM SIGOPS/EuroSys European Conference on Computer Systems* (2006), ACM, pp.
- [25] MO, Y., JUNKEIRA, F. P., AND MARZULLO, K. Mencius: 为广域网构建高效的复制状态机。In *Proc.OSDI'08, USENIX Conference on Operating Systems Design and Implementation* (2008), USENIX, pp.
- [26] MAZIE`RES, D. Paxos made practical. <http://www.scs.stanford.edu/~dm/home/papers/paxos.pdf>, Jan. 2007.
- [27] Moraru, I., ANDERSEN, D. G., AND KAMINSKY, M. 平等主义议会共识更多。 *SOSP'13, ACM Symposium on Operating System Principles* (2013), ACM.
- [28] Raft 用户研究 <http://ramcloud.stanford.edu/~ongaro/userstudy/>.
- [29] Oki, B. M., and Liskov, B. H. 查看盖章复制: 支持高可用性分布式系统的新主副本方法。In *Proc.PODC'88, ACM 分布式计算原理研讨会* (1988 年), ACM, 第 8-17 页。
- [30] O'NEIL, P., CHENG, E., GAWLICK, D., AND ONEIL, E. 对数结构合并树 (LSM-tree)。 *Acta Informatica* 33, 4 (1996), 351-385.
- [31] ONGARO, D. *Consensus: 连接理论与实践*。 斯坦福大学博士论文, 2014 年 (正在撰写中)。
- <http://ramcloud.stanford.edu/~ongaro/thesis.pdf>.
- [32] ONGARO, D., AND OUSTERHOUT, J. In search of a 可理解的共识算法。 In *Proc ATC'14, USENIX Annual Technical Conference* (2014), USENIX.
- [33] ousterhout, J., agrawal, P., erickson, D., Kozyrakis, C., leverich, J., mazie`res, D., mi-tra, S., narayanan, A., ongaro, D., parulkar, G., rosenblum, M., rumble, S. M., stratmann, E., AND STUTSMAN, R. The case for RAMCloud. *Communications of the ACM* 54 (July 2011), 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 日志结构文件系统的实现。 *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] S H V A C H K O, K., KUANG, H., RADIA, S., AND CHANSLER, R. The Hadoop distributed file system. In *Proc.MSST'10, Mass Storage Systems and Technologies Symposium* (2010), IEEE Computer Society, pp.1-10.
- [39] VAN RENESSE, R. Paxos made moderately complex. 技术报告, 康奈尔大学, 2012 年。