寻找一种可理解的共识算法

(扩展版)

Diego Ongaro 和 John Ousterhout 斯
坦福大学

摘要

Raft 是一种管理复制日志的共识算法。它产生的结果相当于(multi-)Paxos，其效率与 Paxos 一样高，但其结构与 Paxos 不同;这使得 Raft 比 Paxos 更容易理解，也为构建实际系统提供了更好的基础。为了增强可理解性，Raft 分离了共识的关键元素，如领导人选举、日志复制和安全，并强制实施更强的一致性程度，以减少必须考虑的状态数量。一项用户研究的结果表明，对于学生来说，Raft 比 Paxos 更容易学习。Raft 还包括一种更改集群成员的新机制，它使用重叠多数来保证安全性。

1 介绍

共识算法允许一组机器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着关键作用。Paxos[15, 16]在过去十年中主导了共识算法的讨论:大多数共识的实现都是基于 Paxos 或受其影响，Paxos 已经成为用于教授学生关于共识的主要工具。

不幸的是，尽管人们多次尝试让 Paxos 变得更平易近人，但 Paxos 还是相当难以理解。此外，它的架构需要复杂的变化来支持实际的系统。因此，无论是系统建设者还是学生都在与 Paxos 斗争。

在我们自己与 Paxos 斗争之后，我们开始寻找一种新的共识算法，它可以为系统建设和教育提供更好的基础。我们的方法不同寻常，因为我们的主要目标是可理解*性*:我们能否为实际系统定义一个共识算法，并以一种比 Paxos 更容易学习的方式来描述它?此外，我们希望算法能够促进对系统构建者至关重要的直觉的开发。重要的不仅是算法能起作用，而且它能明显地说明它起作用的原因。

这项工作的结果是一个共识算法，叫做 Raft。在设计 Raft 时，我们应用了具体的技术来提高可理解性，包括分解(Raft 分离领导者选举、日志复制和安全性)和状态空间减少(相对于 Paxos, Raft 降低了不确定性程度和服务器之间可能不一致的方式)。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 人能够更好地回答关于 Raft 的问题，而不是关于 Paxos 的问题。

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

- **强大的领导者:**Raft 使用了比其他共识算法更强大的领导形式。例如，日志条目只从 leader 流向其他服务器。这简化了复制日志的管理，也让 Raft 更容易理解。
- **领导人选举:**Raft 使用随机定时器选举领导人。这只在任何共识算法已经需要的心跳基础上增加了少量机制，同时简单而快速地解决冲突。
- **成员关系变化:**Raft 用于更改集群中服务器集合的机制使用了一种新的*联合共识*方法，其中两种不同配置的大多数在转换期间重叠。这允许集群在配置更改期间继续正常运行。

我们相信，无论是出于教育目的还是作为实现的基础，Raft 都优于 Paxos 和其他共识算法。它比其他算法更简单，更容易理解;它的描述完全足以满足实际系统的需求;它有几个开源实现，被几家公司使用;它的安全属性已经被正式指定和证明;其效率与其他算法相当。

论文的其余部分介绍了复制状态机问题(第 2 节)，讨论了 Paxos 的优缺点(第 3 节)，描述了我们对可理解性的一般方法(第 4 节)，提出了 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 个复制状态机

共识算法通常出现在*复制状态机*[37]的上下文中。在这种方法中，一组服务器上的状态机计算相同状态的相同副本，即使某些服务器宕机，也可以继续运行。复制的状态机是

本技术报告是[32]的扩展版本;附加材料在空白处标有灰色条。发布于 2014 年 5 月 20 日。

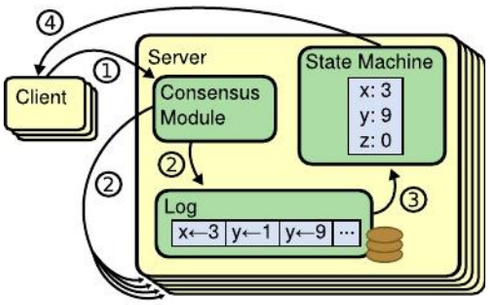

图 1:复制状态机架构。共识算法管理一个包含来自客户端的状态机命令的复制日志。状态机从日志中处理相同的命令序列，因此它们产生相同的输出。

用于解决分布式系统中的各种容错问题。例如，具有单一集群 leader 的大型系统，如 GFS[8]、HDFS[38]、RAMCloud[33]等，通常使用单独的复制状态机来管理 leader 选举，并存储必须在 leader 崩溃时存活的配置信息。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。每个日志以相同的顺序包含相同的命令，因此每个状态机处理相同的命令序列。由于状态机是确定的，所以每个状态机都计算相同的状态和相同的输出序列。

保持复制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，并将其添加到自己的日志中。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以相同的顺序包含相同的请求，即使有些服务器失败。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，并将输出返回给客户端。因此，这些服务器看起来形成了一个单一的、高度可靠的状态机。

实际系统的共识算法通常具有以下属性:

- 它们确保在所有非拜占庭条件下的*安全性*(从不返回错误的结果)，包括网络延迟、分区和数据包丢失、复制和重排。
- 只要大多数服务器运行正常，并且可以相互通信和与客户端通信，它们就具有完整的功能*(可用)*。因此，一个由 5 台服务器组成的典型集群可以容忍任意两台服务器的故障。假定服务器通过停止;它们可能稍后在稳定存储上从状态中恢复并重新加入集群。
- 它们不依赖于时间来确保一致性日志的紧张:错误的时钟和极端的消息延迟，在最坏的情况下，可能会导致可用性问题。
- 在一般情况下，当大多数集群对一轮远程过程调用作出响应时，一个命令可以尽快完成;少数慢速服务器不需要影响整体系统性能。

3 Paxos 怎么了？

在过去的十年中，Leslie Lamport 的 Paxos 协议[15]几乎已经成为共识的代名词:它是课程中最常教授的协议，大多数共识的实现都将其作为起点。Paxos 首先定义了一个协议，该协议能够就单一的决策达成一致，比如单一的复制日志条目。我们把这个子集称为*单法令 Paxos*。Paxos 然后结合该协议的多个实例，以促进一系列决策，如日志*(multi-Paxos)*。Paxos 确保了安全性和活性，并支持集群成员关系的变化。其正确性已被证明，在正常情况下是高效的。

不幸的是，Paxos 有两个显著的缺陷。第一个缺点是，Paxos 极其难以理解。完整解释[15]是出了名的不透明;很少有人能成功地理解它，而且只有付出巨大的努力。因此，有几次尝试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 NSDI 2012 与会者的非正式调查中，我们发现很少有人对 Paxos 感到舒服，即使在经验丰富的研究人员中也是如此。我们自己也在与 Paxos 做斗争;直到阅读了几个简化的解释并设计了我们自己的替代协议后，我们才能够理解完整的协议，这个过程花了几乎一年的时间。

假设 Paxos 的不透明性源于其选择单法令子集作为其基础。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，无法独立理解。正因为如此，我们很难对单法令协议为何有效发展出直观的理解。多 Paxos 的组合规则增加了显著的额外复杂性和微妙性。我们认为，在多个决策(即用日志代替单一条目)上达成共识的整体问题，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，它没有为构建实际的实现提供一个良好的基础。一个原因是，对于 multi-Paxos，目前还没有一个被广泛认可的算法。Lamport 的描述主要是关于单法令 Paxos;他勾画了多 paxos 的可能方法，但很多细节都不见了。人们曾多次尝试充实和优化 Paxos，如[26]、[39]和[13]，但这些都有所不同

从彼此和兰波特的素描中。Chubby[4]等系统已经实现了类似 paxos 的算法，但在大多数情况下，它们的细节尚未公布。

此外，Paxos 架构对于构建实际系统来说是一个很差的架构;这是单法令分解的另一个后果。例如，独立选择一组日志条目，然后将它们融合成顺序日志，几乎没有什么好处;这只会增加复杂性。围绕日志设计一个系统更简单、更高效，在这个系统中，新的条目以受限的顺序顺序地添加。另一个问题是，Paxos 在其核心使用了对称的点对点方法(尽管作为性能优化，它最终建议采用一种弱形式的领导)。在一个只会做出一个决定的简化世界中，这是有意义的，但很少有实际系统使用这种方法。如果必须做出一系列的决策，那么首先选出一个领导者，然后让领导者协调这些决策，这样更简单、更快。

因此，实际系统与 Paxos 几乎没有相似之处。每个实现都从 Paxos 开始，发现实现它的困难，然后开发一个明显不同的架构。这既耗时又容易出错，而理解 Paxos 的困难又加剧了问题。Paxos 的公式化可能是证明关于其正确性的定理的好方法，但真正的实现与 Paxos 相差甚远，证明的价值不大。下面这些胖乎乎的实现者的评论很有代表性:

> 的描述之间有明显的差距
>
> Paxos算法和现实世界的需求
>
> 系统....最终的系统将以联合国-
>
> 经过验证的协议[4]。

由于这些问题，我们得出结论，Paxos 无论在系统建设还是在教育方面都没有提供良好的基础。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 具有更好属性的替代共识算法。Raft 就是那个实验的结果。

4 为可理解性而设计

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，这样就可以显著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是可用的;而且对于常见操作必须是高效的。但我们最重要的目标，也是最困难的挑战是*可理解性*。必须有可能让大量观众轻松地理解算法。此外，必须有可能开发关于算法的直觉，以便系统建设者可以做出在现实实现中不可避免的扩展。

在 Raft 的设计中，有许多点需要我们在可供选择的方法中进行选择。在这些情况下，我们基于可理解性对备选方案进行评估:解释每个备选方案有多难(例如，其状态空间有多复杂，是否有微妙的含义?)，以及读者完全理解该方法及其含义的难度有多大?

我们认识到，在这样的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，我们将问题分成可以相对独立地解决、解释和理解的独立部分。例如，在 Raft 中，我们将领导人选举、日志复制、安全、成员变更分开。

我们的第二种方法是通过减少要考虑的状态数量来简化状态空间，使系统更加一致并尽可能消除不确定性。具体来说，日志不允许有洞，Raft 限制了日志可能相互不一致的方式。虽然在大多数情况下我们试图消除不确定性，但在某些情况下，不确定性实际上提高了可理解性。特别是，随机化的方法引入了不确定性，但它们倾向于通过类似的方式处理所有可能的选择(“选择任何;无所谓”)。我们使用随机化来简化 Raft leader 选举算法。

5 Raft 共识算法

Raft 是一种用于管理第 2 节中描述的复制日志的算法。图 2 以浓缩形式总结了该算法以供参考，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中进行分段讨论。

Raft 通过首先选举一位杰出的*领导者*来实现共识，然后将管理复制日志的全部责任交给领导者。leader 接受来自客户端的日志条目，将它们复制到其他服务器上，并告诉服务器何时将日志条目应用到其状态机是安全的。有了 leader，就简化了对复制日志的管理。例如，leader 可以决定在日志中放置新条目的位置，而无需咨询其他服务器，数据以一种简单的方式从 leader 流向其他服务器。一个 leader 可能会失败或与其他服务器断开连接，在这种情况下，一个新的 leader 就会被选举出来。

考虑到领导者的方法，Raft 将共识问题分解为三个相对独立的子问题，在接下来的小节中讨论:

- **领导人选举:**当现有领导人失败时，必须选择一个新的领导人(第 5.2 节)。
- **日志复制:**leader 必须接受日志条目

状态

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** 服务器最近看到的 term(初始化为 0

在第一次启动时，单调增加)

votedFor 在当前任期内收到投票的 candidateId(如果没有则为 null)

日志 日志条目;每个条目包含状态机的命令，以及 leader接收条目时的术语(第一个索引为 1)

所有服务器上的状态都是不稳定的:

commitIndex 已知要提交的最高日志条目的索引(初始化为 0，单调增加)

lastApplied 应用于状态的最高日志条目的索引

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[]对于每个服务器，发送到该服务器的下一个日志条目的索引(初始化为 leader last log index + 1)

leader 调用来复制日志条目(§ 5.3);也用作心跳(§ 5.2)。

参数:term 领导人的任期

leaderId

prevLogIndex 因此，follower可以将客户的日志条目索引重定向到新条目之前

prevLogTerm

entries[] leader prevLogIndex 条目的词条

提交结果:term 要存储的日志条目(empty for heartbeat;可以发送多份以提高效率)leader 的

成功

currentTerm，如果 follower包含匹配 prevLogIndex和 prevLogTerm 的条目，leader更新自己为 true

接收机实现:

1.如果 term < currentTerm(§ 5.1)则回复 false

2.如果日志在 prevLogIndex不包含词条匹配 prevLogTerm 的条目，则回复 false(§ 5.3)

3.如果现有条目与新条目冲突(相同的索引但不同的词条)，删除现有条目及其之后的所有条目(§ 5.3)

4.添加任何尚未在日志中出现的新条目

5.如果 leaderCommit > commitIndex，设置 commitIndex = min(leaderCommit，上一个新条目的索引)

RequestVote RPC

候选人调用来收集选票(§ 5.2)。

参数:

候选人的任期

candidate id 要求投票的候选人

lastLogIndex候选人最后一次日志记录的索引(§ 5.4)

lastLogTerm候选人最近一次日志条目的术语(§ 5.4)

结果:

term currentTerm，用于候选人更新自身

voteGranted true表示候选人获得了选票

接收机实现:

1. 如果 term < currentTerm(§ 5.1)则回复 false
2. 如果 votedFor是 null或 candidateId，并且候选人的日志是at

至少与接收人的日志一样最新，授予投票(§ 5.2，§ 5.4)

服务器规则

所有服务器:

- 如果 commitIndex > lastApplied:增量 lastApplied，将日志[lastApplied]应用到状态机(§ 5.3)
- 如果 RPC请求或响应包含词条 T > currentTerm:设置 currentTerm = T，转换为 follower(§ 5.1)

追随者(§ 5.2):

- 回应候选人和领导人的 prc
- 如果选举超时，没有收到来自现任领导人的附加条目 RPC或授予候选人投票:转换为候选人

候选人(§ 5.2):

- 转换为候选人后，开始选举:
  - 增加 currentTerm
  - 给自己投票
  - 重置选举计时器

  向所有其他服务器发送 RequestVote rpc

如果收到大多数服务器的投票:成为 leader

- 如果 AppendEntries RPC从新的 leader收到:转换为 follower
- 如果选举超时:开始新的选举

领导:

- 当选时:向每个服务器发送初始的空附件 rpc(心跳);在空闲期间重复以防止选举超时(§ 5.2)
- 如果从客户端收到命令:将条目添加到本地日志，在条目应用到状态机后响应(§ 5.3)
- 如果 last log index≥ nextIndex对于一个 follower:发送 AppendEntries RPC，日志条目从 nextIndex开始
  - 如果成功:为关注者更新 nextIndex和 matchIndex(§ 5.3)
  - 如果 AppendEntries失败，因为日志不一致:减少 nextIndex和重试(§ 5.3)

•如果存在一个 N，使 N > commitIndex，大多数
matchIndex[i]≥N，和 log[N]。term == currentTerm：set commitIndex = N(§5.3，§5.4)。

图 2: Raft 共识算法的概要(不包括成员变更和日志合并)。左上角框中的服务器行为被描述为一组独立、反复触发的规则。章节编号(如§5.2)指明了讨论特定功能的位置。形式规范[31]更精确地描述了算法。

matchIndex 对于每个服务器，已知在服务器上复制的最高日志条目的索引

(初始化为 0，单调递增)

AppendEntries RPC

选举安全:每一届最多可以选出一名领导人。§5.2

Leader - only: Leader 从不重写或删除其日志中的条目;它只追加新的条目。§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，那么日志在给定索引的所有条目中都是相同的。§5.3

领导者完整性:如果一个日志条目在一个给定的术语中提交，那么这个条目将出现在所有更高编号的术语的领导者的日志中。§5.4

状态机安全:如果一个服务器已经将给定索引上的日志条目应用到它的状态机，没有其他服务器将永远不会为相同的索引应用不同的日志条目。§5.4.3

图 3: Raft 保证这些属性在任何时候都是真实的。节号表示每个 prop-的位置

Erty 被讨论。

从客户端并在整个集群中复制它们，迫使其他日志同意自己的日志(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:如果任何服务器已将特定的日志条目应用于其状态机，则任何其他服务器都不能对相同的日志索引应用不同的命令。5.4 节描述了 Raft 如何确保这一属性;解决方案涉及对 5.2 节描述的选举机制的额外限制。

在介绍了共识算法之后，本节讨论了可用性问题和时间在系统中的作用。

5.1 Raft 基础知识

一个 Raft 集群包含多个服务器;5 是一个典型的数字，这使得系统可以容忍两次故障。在任何给定的时间，每个服务器都处于三种状态之一:*领导者*、*追随者*或*候选人*。在正常运行中，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:他们自己不发出任何请求，只是简单地回应领导者和候选人的请求。leader 处理所有的客户请求(如果客户联系了 follower，那么 follower 会将其重定向给 leader)。第三个状态，candidate，用于选举一个新的 leader，如 5.2 节所述。图 4 显示了状态及其转换;下面讨论这些转换。

Raft 将时间*按*任意长度进行划分，如图 5 所示。Terms 用连续的整数进行编号。每一届任期以*选举*开始，一名或多名候选人试图成为第 5.2 节所述的领导人。如果一位候选人赢得选举，那么他将在剩下的任期内担任领导人。在某些情况下，选举将导致投票分裂。在这种情况下，任期结束时没有领导人;新任期(重新选举)

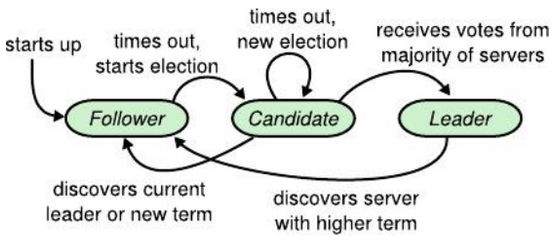

图 4:服务器状态。关注者只响应其他服务器的请求。如果一个 follower 没有收到任何通信，它就会成为一个候选人，并发起选举。得到整个集群多数人投票的候选人成为新的 leader。领导者通常会一直运作到失败为止。

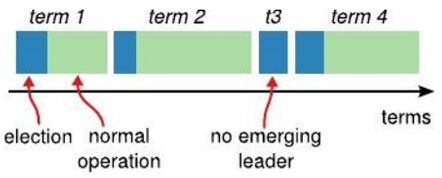

图 5:时间分为任期，每个任期都以选举开始。选举成功后，单个 leader 管理集群，直到任期结束。有些选举失败，在这种情况下，任期结束时没有选出领导人。在不同的服务器上，可以在不同的时间观察到任期之间的过渡。

马上开始。Raft 确保在给定的任期内最多有一个领导人。

不同的服务器可能在不同的时间观察到任期之间的过渡，在某些情况下，服务器可能不观察选举甚至整个任期。术语在 Raft 中充当逻辑时钟[14]，它们允许服务器检测过时的信息，如陈旧的领导人。每个服务器存储一个*当前的 term* 编号，该编号随时间单调增加。每当服务器通信时，都会交换当前词条;如果一个服务器的当前项小于另一个服务器，那么它将其当前项更新为较大的值。如果某个候选者或 leader 发现自己的 term 已经过时，则会立即恢复到 follower 状态。如果一个服务器收到一个带有过时词条编号的请求，它会拒绝该请求。

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(章节 5.2)，Append- Entries rpc 由领导人发起，用于复制日志条目并提供一种心跳形式(章节 5.3)。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响应，则会重试 rpc，并并行发出 rpc 以获得最佳性能。

5.2 领导人选举

Raft 使用心跳机制触发领导人选举。当服务器启动时，它们一开始是追随者。只要接收到有效，服务器就会保持跟随者状态

来自领导或候选人的 rpc。领导者定期向所有追随者发送心跳(不带日志记录的附录 rpc)，以维护自己的权威。如果一个追随者在一段称为*选举超时*的时间内没有收到任何通信，那么它假定没有可行的领导者，并开始选举以选择一个新的领导者。

为了开始选举，追随者增加其当前任期并过渡到候选人状态。然后它为自己投票，并并行地向集群中的每个其他服务器发出 RequestVote rpc。一个候选服务器继续保持这种状态，直到发生以下三种情况之一:(A)它赢得了选举，(b)另一个服务器建立了自己的领导者，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单独讨论。

如果一个候选人在同一任期内收到了整个集群中大多数服务器的投票，那么他就赢得了选举。每个服务器在给定的任期内最多投票给一个候选人，按照先到先得的原则(注:第 5.4 节对投票增加了额外的限制)。多数原则确保最多有一名候选人可以赢得特定任期的选举(图 3 中的选举安全属性)，一旦候选人赢得选举，他就成为领袖。然后，它向所有其他服务器发送心跳消息，以建立其权威，并防止新的选举。

在等待投票的过程中，候选人可能会收到来自另一台自称为领导者的服务器的 AppendEntries RPC。如果 leader 的任期(包括在它的 RPC 中)至少和候选人的当前任期一样大，那么候选人就会承认 leader 是合法的，并返回到 follower 状态。如果 RPC 中的任期小于候选人的当前任期，则候选人拒绝 RPC，继续处于候选人状态。

第三种可能的结果是候选人在选举中既不获胜也不失败:如果有许多追随者同时成为候选人，选票就可能被分割，这样就没有候选人获得多数。当这种情况发生时，每个候选人将暂停并开始新的选举，通过增加其任期并发起另一轮请求-投票 rpc。然而，如果没有额外的措施，分裂投票可能会无限期重复。

Raft 使用随机的选举超时来确保分裂票是罕见的，并且能够快速解决。为了首先防止分裂选票，选举超时是从固定的间隔(例如 150-300ms)中随机选择的。这将服务器分散开来，因此在大多数情况下只有一个服务器会超时;它赢得了选举，并在其他服务器超时之前发送心跳。同样的机制也被用于处理分裂投票。每个候选人在选举开始时重新启动其随机选举超时，并等待该超时过去后再开始下一次选举;这就降低了在新的选举中出现另一次分裂选票的可能性。9.3 节表明，这种方法可以快速选出领导人。

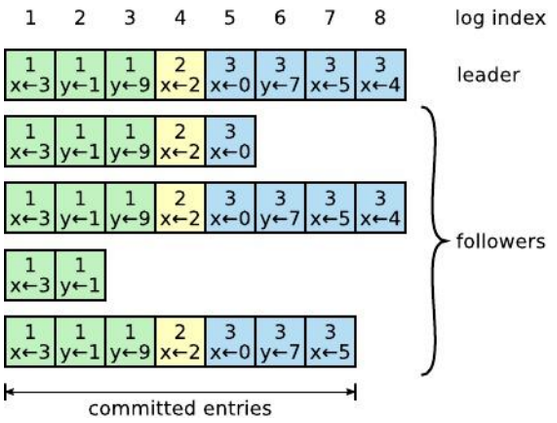

图 6:日志由条目组成，条目按顺序编号。每个条目包含创建它的词条(每个框中的数字)和状态机的一个命令。如果*将*某个条目应用于状态机是安全的，则认为该条目已*提交*。

选举是一个例子，说明可理解性如何指导我们在设计备选方案之间的选择。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，用于在相互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，它就会回到追随者状态，这样排名更高的候选人就可以更轻松地赢得下一次选举。我们发现，这种方法在可用性方面产生了微妙的问题(如果排名较高的服务器失败，排名较低的服务器可能需要超时并再次成为候选人，但如果它这么做得太早，它可以重新设置选举领导人的进度)。我们对算法进行了几次调整，但每次调整后都会出现新的拐角情况。最终我们得出结论，随机重试的方法更明显，也更容易理解。

5.3 日志复制

一旦 leader 被选出，它就开始为客户端请求提供服务。每个客户端请求包含一个命令，由复制的状态机执行。leader 将命令作为一个新条目附加到它的日志中，然后并行地向每个其他服务器发出 AppendEntries rpc 来复制条目。当条目被安全复制后(如下所述)，leader 将条目应用到它的状态机中，并将执行的结果返回给客户端。如果 follower 崩溃或运行缓慢，或者网络数据包丢失，leader 会无限期地重试 Append- Entries rpc(即使在它已经响应客户端之后)，直到所有 follower 最终存储所有日志条目。

日志的组织如图 6 所示。当条目被 leader 接收时，每个日志条目存储了一个状态机命令以及词条编号。日志条目中的词条号用于检测日志之间的不一致性，并确保图 3 中的一些属性。每个日志条目也有一个整数索引 id

-

确认其在日志中的位置。

领导者决定何时可以安全地将日志条目应用到状态机;这样的条目被称为*已提交*。Raft 保证已提交的条目是持久的，并且最终将由所有可用的状态机执行。一旦创建条目的 leader 将其复制到大多数服务器上(例如图 6 中的条目 7)，日志条目就会被提交。这也会提交 leader 日志中的所有先前条目，包括以前的 leader 创建的条目。5.4 节讨论了在 leader 变更后应用这一规则时的一些微妙之处，它也显示了这种承诺的定义是安全的。leader 跟踪它知道要提交的最高索引，并在未来的 AppendEntries rpc(包括心跳)中包括该索引，以便其他服务器最终发现。一旦一个 follower 得知一个日志条目被提交了，它就会将这个条目应用到它的本地状态机中(按照日志顺序)。

我们设计了 Raft 日志机制，以保持不同服务器上日志之间的高水平一致性。这不仅简化了系统的行为，使其更具有可预测性，而且是确保安全性的重要组成部分。Raft 维护着以下属性，这些属性共同构成了图 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术语，那么在所有前面的条目中日志是相同的。

第一个属性源于这样一个事实:一个 leader 在一个给定的词条中，用给定的日志索引最多创建一个条目，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。当发送一个 AppendEntries RPC 时，leader 在它的日志中包含了这个条目的索引和词条，这个索引和词条就在新条目的前面。如果 follower 在它的日志中没有找到索引和项相同的条目，那么它就会拒绝新条目。一致性检查充当归纳步骤:日志的初始空状态满足日志匹配属性，而一致性检查在日志扩展时保留日志匹配属性。因此，每当 AppendEntries 成功返回时，leader 就会通过新条目知道 follower 的日志与自己的日志是相同的。

正常情况下，leader 和 followers 的日志保持一致，因此 AppendEntries 一致性检查不会失败。然而，leader 崩溃可能会使日志不一致(旧的 leader 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的 leader 和 follower 崩溃中变得更加复杂。图 7 说明了追随者日志与新领导者日志的不同之处。跟随者可能

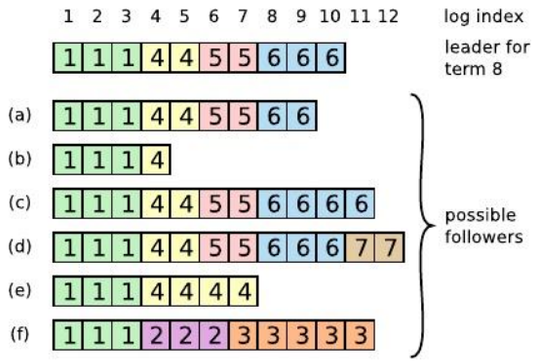

图 7:当高层领导上台后，追随者日志中可能会出现任何一种情况(a-f)。每个方框代表一个日志条目;方框内的数字就是它的词条。一个跟随者可能缺少条目(A - b)，可能有额外的未提交条目(c-d)，或者两者都有(e-f)。例如，如果该服务器是 term 2 的 leader，在其日志中添加了几个条目，然后在提交任何条目之前崩溃，则可能发生场景(f);它迅速重启，成为 term 3 的 leader，并在它的日志中增加了一些条目;在 term 2 或 term 3 的任何条目提交之前，服务器再次崩溃，并保持了几个 term。

如果缺少 leader 上存在的条目，它可能会有 leader 上不存在的额外条目，或者两者都有。日志中缺失的和多余的条目可能跨越多个术语。

在 Raft 中，领导者通过强迫跟随者的日志复制自己的日志来处理不一致的地方。这意味着追随者日志中冲突的条目将被来自领导者日志的条目覆盖。5.4 节将说明，当再加上一个限制时，这是安全的。

为了使跟随者的日志与自己的一致，领导者必须找到两个日志一致的最新日志条目，删除跟随者日志中该点之后的任何条目，并将该点之后领导者的所有条目发送给跟随者。所有这些动作的发生都是为了响应 AppendEntries rpc 执行的一致性检查。leader 为每个 follower 维护一个*nextIndex*，这是 leader 将发送给该 follower 的下一个日志条目的索引。当 leader 第一次掌权时，它将所有的 nextIndex 值初始化到其日志中最后一个条目之后的索引中(图 7 中的 11 个)。如果一个 follower 的日志与 leader 的日志不一致，在下一个 AppendEntries RPC 中，AppendEntries 一致性检查将失败。在被拒绝之后，leader 将 nextIndex 减 1，并重新尝试 AppendEntries RPC。最终，nextIndex 将达到 leader 和 follower 日志匹配的点。当这种情况发生时，AppendEntries 将会成功，它会删除跟随者日志中任何冲突的条目，并从 leader 的日志中追加条目(如果有的话)。一旦 AppendEntries 成功，跟随者的日志与 leader 的日志是一致的，并且在余下的学期中会保持这种状态。

如果需要，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，follower

可以包含冲突条目的词条和它为该词条存储的第一个索引。有了这些信息，leader 就可以减 nextIndex 来绕过该词条中的所有冲突条目;对于每个条目有冲突的条目，将需要一个 AppendEntries RPC，而不是每个条目一个 RPC。在实践中，我们怀疑这种优化是否必要，因为故障很少发生，不太可能有许多不一致的 en-

尝试。

有了这种机制，领导者在掌权时不需要采取任何特殊行动来恢复日志一致性。它只是开始正常运行，日志会自动收敛以响应追加-条目一致性检查的失败。leader 从不重写或删除自己日志中的条目(图 3 中的 leader 仅追加属性)。

这种日志复制机制展示了第 2 节中描述的理想共识属性:只要大多数服务器都在运行，Raft 就可以接受、复制和应用新的日志条目;在正常情况下，一个新条目可以通过单轮 rpc 复制到集群的大多数;而单个慢跟随者不会影响性能。

5.4 安全

前面的章节描述了 Raft 如何选举领导人和复制日志条目。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 leader 提交几个日志条目时，一个 follower 可能是不可用的，那么它可以被选为 leader，并用新的条目覆盖这些条目;因此，不同的状态机可能会执行不同的命令序列。

本节通过添加一个限制哪些服务器可以被选为领导者来完成 Raft 算法。这个限制确保了任何给定词条的 leader 包含了之前词条中提交的所有条目(来自图 3 的 leader 完备性属性)。给定选举限制，然后我们让承诺的规则更精确。最后，我们给出了 Leader 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，领导者最终必须存储所有提交的日志条目。在一些共识算法中，比如 Viewstamped Replication[22]，即使一个 leader 最初没有包含所有提交的条目，也可以被选举出来。这些算法包含额外的机制来识别缺失的条目，并将它们传递给新的领导者，无论是在选举过程中还是在选举之后不久。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，它保证以前提交的所有条目

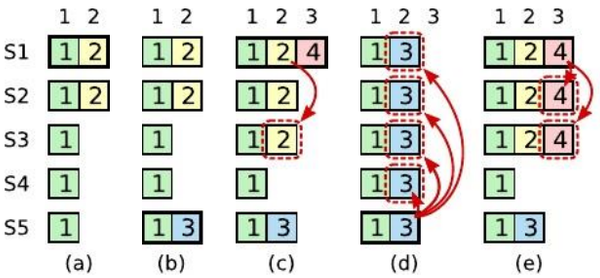

图 8:一个时间序列，显示了为什么领导不能使用来自旧术语的日志条目来确定承诺。在(a)中 S1 是 leader，并部分复制了索引 2 处的日志条目。在(b)中 S1 崩溃;S5 通过 S3、S4 和自身的投票当选为任期 3 的领导人，并在日志索引 2 处接受一个不同的条目。In (c) S5 崩溃;S1 重启，被选为 leader，并继续复制。此时，term 2 的日志条目已经在大多数服务器上进行了复制，但没有提交。如果 S1 像(d)那样崩溃，S5 可以被选为 leader(通过来自 S2、S3 和 S4 的投票)，并用来自 term 3 的自己的条目覆盖该条目。然而，如果 S1 在崩溃前在大多数服务器上复制了当前词条中的一个条目，如(e)，那么这个条目是提交的(S5 不能赢得选举)。在这一点上，日志中所有之前的条目也被提交。

条款在每个新领导人当选时就已经存在，无需将这些条目转交给领导人。这意味着日志条目只向一个方向流动，从领导者到追随者，领导者永远不会覆盖其日志中的现有条目。

Raft 使用投票过程来防止候选人赢得选举，除非它的日志包含所有提交的条目。为了当选，候选人必须联系集群的大多数成员，这意味着每个已提交的条目必须出现在这些服务器中的至少一个。如果候选人的日志至少与该多数中的任何其他日志一样是最新的(“最新”在下面精确定义)，那么它将保存所有已提交的条目。RequestVote RPC 实现了这个限制:RPC 包含了关于候选人日志的信息，如果投票者自己的日志比候选人的日志更最新，那么投票者就会拒绝投票。

Raft 通过比较日志中最后一个条目的索引和项来确定两个日志中哪一个更最新。如果日志的最后条目具有不同的词条，则具有较晚词条的日志更与时俱进。如果日志以相同的词条结束，那么较长的日志就更与时俱进。

5.4.2 提交前一个词条的条目

如 5.3 节所述，一旦当前词条的条目存储在大多数服务器上，leader 就知道该条目被提交了。如果一个 leader 在提交条目之前崩溃，未来的 leader 将尝试完成条目的复制。然而，一旦前一个词条的条目被存储在大多数服务器上，leader 就不能立即得出结论，认为它已经提交。图- - - - - -

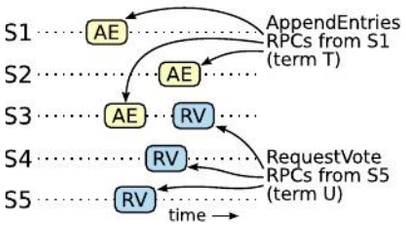

图 9:如果 S1 (term T 的 leader)从其 term 提交了一个新的日志条目，而 S5 被选为后期 term U 的 leader，那么必须至少有一台服务器(S3)接受了该日志条目，并且也投票给了 S5。

Ure 8 说明了这样一种情况，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 leader 覆盖。

为了消除图 8 所示的问题，Raft 从不通过计算副本来提交以前词条的日志条目。仅通过计数副本提交领导当前任期的日志条目;一旦以这种方式提交了来自当前任期的条目，那么由于日志匹配属性，所有之前的条目都是间接提交的。在某些情况下，leader 可以安全地得出一个较旧的日志条目被提交了(例如，如果该条目存储在每个服务器上)，但 Raft 为了简单起见采取了更保守的方法。

Raft 在提交规则中增加了这种额外的复杂性，因为当 leader 复制前一个词条时，日志条目将保留其原始词条编号。在其他共识算法中，如果一个新的 leader 从以前的“词条”重新复制条目，它必须用它的新“词条数”来这样做。Raft 的方法使得对日志条目的推理变得更容易，因为它们随着时间的推移和跨日志保持相同的词条数。此外，与其他算法相比，Raft 中的新领导人从以前的词条发送更少的日志条目(其他算法必须发送冗余的日志条目重新编号后才能提交)。

5.4.3 安全性论证

给定完整的 Raft 算法，我们现在可以更精确地论证领导者完备性是成立的(此论证基于安全性证明;参见 9.2 节)。我们假设 Leader 完备性不成立，那么我们证明了一个矛盾。假设词条 T 的 $leader_T$(leader)从它的词条中提交了一个日志条目，但是这个日志条目不是由某个未来词条的 leader 存储的。考虑最小的词条 U > T，它的 $leader_U$(leader)不存储条目。

1.提交的条目必须在 $leader_U$ 选举时从其日志中缺失(leader 从不删除或覆盖条目)。

2.$leader_T$ 在大多数集群上复制了条目，leader 收到 $_U$ 了大多数集群的投票。因此，至少有一个服务器(“投票人”)同时接受了 leader 的条目并 $_T$ 为之投票

$leader_U$，如图 9 所示。选民是达成矛盾的关键。

3.投票人在投票选领袖 *前*，必须接受领袖 $_U$ 的承诺参选 $_T$;否则它将拒绝 $leader_T$ 的追加条目请求(其当前任期将高于 T)。

4.投票人在投票给 $leader_U$ 的时候仍然保存这个条目，因为每一个介入的 leader 都包含这个条目(假设)，leader 永远不会删除条目，follower 只会在条目与 leader 冲突的时候删除条目。

5.投票者把票投给了 $leader_U$，所以 $leader_U$ 的日志必须和投票者的一样是最新的。这就引出了两个矛盾之一。

6.首先，如果选民和领导共享 $_U$ 同一个日志项，那么领导 $_U$ 的日志必须至少和选民的一样长，所以它的日志包含了选民日志中的每一个条目。这是一个矛盾，因为投票者包含了承诺的条目，而 $leader_U$ 被假定不包含。

7.否则，领导人 $_U$ 的上一个 log 任期必须比选民的大。而且，它大于 T，因为投票人的最后一个 log 项至少是 T(它包含来自 T 项的 committed 条目)。创建 leader 最后一个 log 条目的较早的 $leader_U$ 在它的 log 中一定包含了 committed 条目(根据假设)。那么，根据日志匹配属性，$leader_U$ 的日志中也必须包含 commit 条目，这是一个矛盾。

8.这就完成了矛盾。因此，所有大于 T 的项的首项必须包含在项 T 中提交的来自项 T 的所有项。

9.日志匹配属性保证未来的领导者也将包含间接提交的条目，例如图 8(d)中的索引 2。

有了 Leader 完整性属性，我们可以从图 3 证明状态机安全属性，图 3 指出，如果一个服务器已经将给定索引上的日志条目应用到它的状态机，其他服务器将永远不会为相同的索引应用不同的日志条目。当一台服务器将日志条目应用到它的状态机时，它的日志必须通过该条目与 leader 的日志相同，并且必须提交该条目。现在考虑任何服务器应用给定日志索引时的最低项;日志完整性属性保证所有较高词条的 leader 都将存储相同的日志条目，因此在较低词条中应用索引的服务器将应用相同的值。因此，状态机安全属性是成立的。

最后，Raft 要求服务器按日志索引顺序应用条目。结合状态机安全属性，这意味着所有服务器将以相同的顺序，将完全相同的一组日志条目应用到它们的状态机。

5.5 Follower 和 candidate 崩溃

到目前为止，我们一直关注领导者的失败。跟随者和候选人崩溃比领导者崩溃处理起来要简单得多，而且它们的处理方式都是一样的。如果一个 follower 或者 candidate 崩溃了，那么以后发送给它的 RequestVote 和 AppendEntries rpc 就会失败。Raft 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 之后但在响应之前崩溃，那么它将在重启后再次收到相同的 RPC。Raft rpc 是幂等的，所以这不会造成任何伤害。例如，如果一个 follower 收到一个 AppendEntries 请求，其中包含了它的日志中已经存在的日志条目，那么它会忽略新请求中的那些条目。

5.6 定时和可用性

我们对 Raft 的要求之一是，安全性绝不能依赖于时间:系统绝不能仅仅因为某些事件发生得比预期的快或慢而产生不正确的结果。然而，可用性(系统及时响应客户的能力)必须不可避免地依赖于时间。例如，如果消息交换花费的时间超过了服务器崩溃之间的典型时间，候选人就不会熬夜到赢得选举的时间;如果没有稳定的领导者，Raft 就无法取得进展。

领导人选举是 Raft 最讲究时机的一个方面。只要系统满足以下*时序要求*，Raft 就能选出并维持稳定的 leader:

$$broadcastTime \ll electionTimeout \ll MTBF$$

在这个不等式中，*broadcastTime* 是服务器向集群中的每个服务器并行发送 rpc 并接收它们的响应所需的平均时间;*electionTime- out* 是 5.2 节中描述的选举超时;*MTBF* 是单个服务器的平均故障间隔时间。广播时间应该比选举超时时间少一个数量级，以便领导者能够可靠地发送阻止追随者开始选举所需的心跳消息;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太可能发生。选举超时时间应该比 MTBF 小几个数量级，这样系统才能稳步前进。当 leader 崩溃时，系统将在大致的选举超时时间内无法使用;我们希望这只代表总体时间的一小部分。

广播时间和 MTBF 是底层系统的属性，而选举超时是我们必须选择的。Raft 的 rpc 通常需要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，具体取决于存储技术。因此，选举超时时间很可能在 10ms 到 500ms 之间。典型的

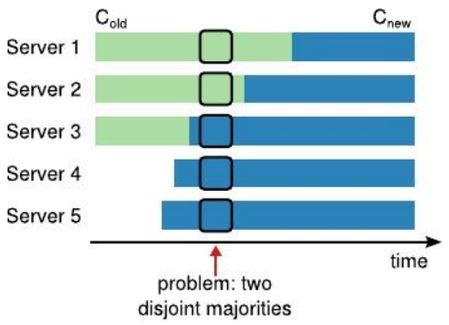

图 10:直接从一种配置切换到另一种配置是不安全的，因为不同的服务器会在不同的时间切换。在这个例子中，集群从 3 台服务器增长到 5 台。不幸的是，有一个时间点可以选出两个不同的领导人担任同一任期，一个以旧配置*(Cold)*为多数，另一个以新配置(Cnew)为多数。

服务器 MTBFs 是几个月甚至更长的时间，这很容易满足时间要求。

6 集群成员变更

到目前为止，我们假设集群*配置*(参与共识算法的服务器集合)是固定的。在实际操作中，偶尔会需要改变配置，例如服务器出现故障时更换服务器，或者改变复制程度。虽然这可以通过让整个集群离线，更新配置文件，然后重新启动集群来实现，但这将使集群在切换期间不可用。此外，如果有任何手动步骤，它们会有操作失误的风险。为了避免这些问题，我们决定将配置更改自动化，并将其纳入 Raft 共识算法中。

为了使配置更改机制安全，在过渡期间必须没有任何意义，即有可能出现两个领导人同时当选的情况。不幸的是，任何服务器直接从旧配置切换到新配置的方法都是不安全的。原子性地一次性切换所有服务器是不可能的，因此集群可能会在过渡期间分裂成两个独立的多数(参见图 10)。

为了确保安全性，配置更改必须使用两阶段方法。实现这两个阶段的方法有很多种。例如，一些系统(例如[22])使用第一阶段禁用旧的配置，因此它不能处理客户端请求;然后第二阶段启用新配置。在 Raft 中，集群首先切换到我们称为*联合共识*的过渡配置;一旦提交了联合共识，系统就会过渡到新的配置。联合共识结合了旧的和新的配置:

日志条目被复制到所有服务器在这两个 con-

形状。

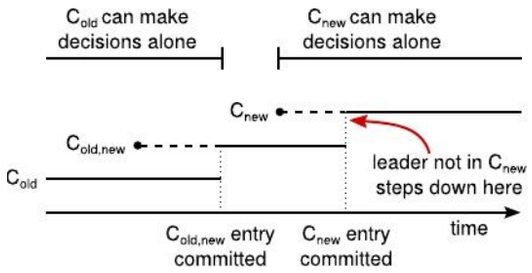

图 11:配置更改的时间线。虚线显示已创建但未提交的配置条目，实线显示最新提交的配置条目。leader 首先在自己的日志中创建 *Cold,*new 配置条目，并将其提交到 Cold,new(*大部分是* Cold，大部分是 *Cnew*)。然后它创建 Cnew *条目*，并将其提交给大多数 Cnew。没有一个时间点可以让 Cold *和* Cnew 都能独立做出决定。

任一配置下的任何服务器都可以作为 leader。

- 协议(用于选举和进入承诺)需要从新旧配置 *中*分离出多数票。

联合共识允许单个服务器在不同的时间在不同的配置之间过渡，而不影响安全性。此外，联合共识允许集群在整个配置变化期间继续为客户端请求提供服务。

集群配置使用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 leader 收到 C_{old} *to*C_{new} 更改配置的请求时，它将联合共识的配置(图中 *Cin*$_{old,new}$)存储为日志条目，并使用前面描述的机制复制该条目。一旦给定的服务器将新的配置条目添加到它的日志中，它就会将该配置用于所有未来的决策(服务器总是在其日志中使用最新的配置，无论该条目是否提交)。这意味着 leader 将使用 Cto 的规则来*确定* old,new Cis 的日志条目何时提交。old,new 如果 leader 崩溃，可能会在 Cold 或 Cold,new 下选择一个新的 leader，取决于获胜的候选人*是否*收到了 Cold,new。无论如何，Cnew 在*此*期间都不能单方面*做出*决定。

*OnceCold,*new 已经承诺，严寒的 *norCnew* 都不能未经对方批准就*做出*决定，Leader 完备性保证只有拥有 Clog 入口的服务器才能*当选* old,new Leader。现在 leader 可以安全地创建一个描述 c 的日志条目，并将其复制到集群。new 同样，这个配置一看到就会在每台服务器上生效。当新配置按照 C 的规则提交完毕后，旧配置就无关紧要了，new 不在新配置中的服务器就可以关闭了。如图 11 所示，没有时间让 c 和 Ccan 都做出单方面的决定;old new 这保证了安全性。

对于重新配置，还有三个问题需要解决。第一个问题是新服务器最初可能不会存储任何日志条目。如果以这种状态添加到集群中，它们可能需要相当长的时间才能跟上，在此期间可能无法提交新的日志条目。为了避免可用性缺口，Raft 在配置更改之前引入了一个额外的阶段，在这个阶段中，新服务器作为无投票成员加入集群(leader 将日志条目复制给它们，但不考虑多数)。一旦新服务器赶上了集群的其他服务器，重新配置就可以像上面描述的那样进行。

第二个问题是，集群 leader 可能不在新配置的范围内。在这种情况下，leader 一旦提交了 *Clo*new *g* 入口，就会下台(回到 follower 状态)。这意味着，当 leader 正在管理一个不包括自己的集群时，会有一段时间(*当* new 它正在提交时 c);它复制日志条目，但不将自己统计为多数。leader transition 发生在 Cis 提交的时候，*因为* new 这是新配置可以独立运行的第一个点(总是可以从 C 中选择一个 leader)。在这一点之前，new 可能只有来自 c_的服务器才能被选为 leader。old

第三个问题是被移除的服务器(不在 C_{new} 中的服务器)可能会破坏集群。这些服务器不会接收到心跳，因此它们会超时并开始新的选举。然后，它们会发送带有新任期编号的 RequestVote rpc，这将导致当前的 leader 恢复到 follower 状态。最终会选出一个新的 leader，但是被移除的服务器会再次超时，这个过程会重复，导致可用性差。

为了防止这个问题的发生，当服务器认为当前的 leader 存在时，会忽略 RequestVote rpc。具体来说，如果服务器在听取当前 leader 的最小选举超时时间内收到 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，在正常的选举中，每个服务器在开始选举前至少等待一个最低选举超时时间。但是，它有助于避免被移除的服务器造成的中断:如果一个 leader 能够获得其集群的心跳，那么它将不会被更大的任期数罢免。

7.日志合并

Raft 的日志在正常运行期间增长，以合并更多的客户端请求，但在实际系统中，它不能无限制地增长。随着日志的增长，它会占用更多的空间，需要更多的时间来重放。如果没有某种机制来丢弃日志中积累的过时信息，这最终将导致可用性问题。

快照是最简单的压缩方法。在快照中，将整个当前系统状态写入稳定存储上的*快照*，然后将整个日志写入到

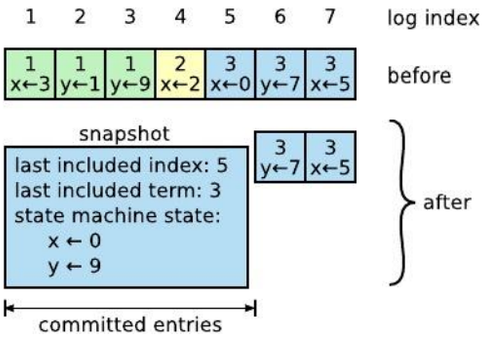

图 12:服务器将其日志(索引 1 到 5)中提交的条目替换为一个新的快照，该快照仅存储当前状态(本例中的变量 *x* 和 *y*)。快照的最后一个包含的索引和术语用于将快照定位在日志条目 6 之前。

这个点被丢弃了。在 Chubby 和 ZooKeeper 中使用了快照，本节的其余部分描述了 Raft 中的快照。

增量的压缩方法，如日志清理[36]和日志结构合并树[30,5]，也是可能的。这些一次性对一小部分数据进行操作，因此它们会随着时间的推移更均匀地分散压缩的负载。他们首先选择一个已经积累了许多删除和覆盖对象的数据区域，然后他们从该区域更紧凑地重写活动对象，并释放该区域。与快照相比，这需要显著的额外机制和复杂性，快照总是在整个数据集上操作，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 中快照的基本思想。每个服务器独立地获取快照，只覆盖其日志中已提交的条目。大部分工作由状态机将其当前状态写入快照组成。Raft 还在快照中包含少量元数据:*最后包含的索引*是快照所替换的日志中最后一个条目的索引(状态机应用的最后一个条目)，*最后包含的词条*是这个条目的词条。保留这些元数据是为了支持对快照之后的第一个日志条目的 AppendEntries 一致性检查，因为该条目需要一个先前的日志索引和词条。为了实现集群成员关系的更改(第 6 节)，快照还包括日志中截至最近包含的索引的最新配置。一旦服务器完成了快照的写入，它可以删除最后包含的索引中的所有日志条目，以及任何之前的快照。

虽然服务器通常独立拍摄快照，但 leader 必须偶尔将快照发送给滞后的 follower。当 leader 已经丢弃了它需要发送给 follower 的下一个日志条目时，就会发生这种情况。幸运的是，这种情况在正常操作下是不太可能发生的:一个跟随者已经跟上了

InstallSnapshot RPC

leader 调用，将快照的块发送给 follower。leader 总是按顺序发送分块。

参 数:

任 期 领导人任期

leader 所以 follower 可以重定向客户

lastIncludedIndex 快照会替换所有包含这个索引的条目

lastIncludedTerm lastIncludedIndex **的 词 条**

Offset 字节 chunk 在快照文件中的**偏 移 量**

Data[]快照块的原始字节，从 offset 开始

Done true，如果这是最后一块

结 果:

术 语 currentTerm，为 leader 更新自己

接 收 机 实 现:

1.如果 term < currentTerm，请立即回复

2.创建新的快照文件，如果第一个块(偏移量为 0)

3.在给定的偏移量将数据写入快照文件

4.如果 done 为 false，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的具有较小索引的快照

6.如果现有的日志条目与快照的上一个包含条目具有相同的索引和术语，则保留它之后的日志条目并回复

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。快照被分割成块进行传输;这样每一个分块都会给跟随者一个生命的迹象，这样它就可以重置选举计时器。

Leader 应该已经有这个条目了。然而，异常缓慢的 follower 或新加入集群的服务器(第 6 节)则不会。让这样一个 follower 更新到最新状态的方法是 leader 通过网络给它发送一个快照。

leader 使用一种名为 InstallSnapshot 的新型 RPC 将快照发送给落后太远的 follower;参见图 13。当一个 follower 通过这个 RPC 接收到一个快照时，它必须决定如何处理它现有的日志条目。通常，快照将包含接收者的日志中尚未包含的新信息。在这种情况下，关注者会丢弃自己的整个日志;它全部被快照取代，并且可能有未提交的条目与快照冲突。相反，如果 follower 接收到描述其日志前缀的快照(由于重传或错误)，则快照覆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。

这种快照方式偏离了 Raft 的强领导者原则，因为追随者可以在领导者不知情的情况下拍摄快照。然而，我们认为这种背离是合理的。虽然有一个领导者有助于在达成共识时避免冲突的决策，但快照时已经达成共识，所以没有决策冲突。数据仍然只从领导者流向 fol-

降低者，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，只有领导者会创建一个快照，然后它会将这个快照发送给它的每个追随者。然而，这种方式有两个缺点。首先，将快照发送给每个关注者会浪费网络带宽，并减慢快照过程。每个关注者已经拥有了自己生成快照所需的信息，而且对于服务器来说，从本地状态生成快照通常比通过网络发送和接收快照便宜得多。其次，leader 的实现会更加复杂。例如，leader 需要在向 follower 复制新日志条目的同时，并行地向 follower 发送快照，这样才不会阻塞新的客户端请求。

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。如果服务器快照太频繁，会浪费磁盘带宽和能量;如果快照频率过低，则有耗尽存储容量的风险，并且会增加重启期间重放日志所需的时间。一个简单的策略是，当日志达到固定大小(以字节为单位)时，进行快照。如果将此大小设置为显著大于快照的预期大小，那么用于快照的磁盘带宽开销将很小。

第二个性能问题是，写入快照可能会花费大量的时间，我们不希望这延误正常的操作。解决方案是使用写时复制(copy-on-write)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，用函数式数据结构构建的状态机自然就支持这一点。或者，操作系统的写时复制支持(例如，Linux 上的 fork)可以用来创建整个状态机的内存快照(我们的实现使用了这种方法)。

8 客户端交互

本节描述了客户端如何与 Raft 交互，包括客户端如何找到集群领导者以及 Raft 如何支持线性化语义[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 的客户将所有的请求发送给领导者。当客户端第一次启动时，它会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 leader，该服务器将拒绝客户端的请求，并提供它从最近的 leader 那里听到的信息(AppendEntries 请求包括 leader 的网络地址)。如果 leader 崩溃，客户端请求就会超时;然后客户端会用随机选择的服务器再次尝试。

我们对 Raft 的目标是实现可线性化的语义(每个操作似乎在调用和响应之间的某个点上立即执行一次)。然而，正如目前所描述的，Raft 可以多次执行一个命令:例如，如果领导者在提交日志条目后崩溃，但在响应客户端之前，客户端将用新的 leader 重试命令，导致它被第二次执行。解决方案是让客户端为每条命令分配唯一的序列号。然后，状态机跟踪为每个客户端处理的最新序列号，以及相关的响应。如果它接收到一个序列号已经被执行过的命令，它会立即响应，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。然而，如果没有额外的措施，这将会有返回过期数据的风险，因为响应请求的 leader 可能已经被它不知道的新 leader 取代了。线性化读取一定不能返回陈旧的数据，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，leader 必须掌握提交哪些条目的最新信息。Leader 完备性保证 Leader 已经提交了所有的条目，但在其任期开始时，它可能不知道那些是哪些条目。为了找出答案，它需要从它的任期提交一个条目。Raft 通过让每个领导在任期开始时提交一个*空白的 no-op* 条目到日志中来处理这个问题。其次，在处理只读请求之前，领导者必须检查自己是否已经被罢免(如果选举的是更近的领导者，那么它的信息可能是陈旧的)。Raft 通过在响应只读请求之前让 leader 与大多数集群交换心跳消息来处理这个问题。或者，leader 可以依靠心跳机制来提供一种形式的 lease[9]，但这将依赖于时间的安全性(它假设有界的时钟偏差)。

~~9~~ 实施和评估

我们已经将 Raft 实现为复制状态机的一部分，该状态机为 RAMCloud[33]存储配置信息，并协助 RAMCloud 协调器的故障转移。Raft 实现包含大约 2000 行 c++代码，不包括测试、注释或空行。源代码可免费获得[23]。基于本文草稿，还有大约 25 个处于不同开发阶段的 Raft 的独立第三方开源实现[34]。此外，各公司正在部署基于 raft 的系统[34]。

本节的其余部分使用三个标准来评估 Raft:可理解性、正确性和性能。

~~9.1~~ 可理解性

为了衡量相对于 Paxos 而言，Raft 的可理解性，我们对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。我们录制了一个 Raft 的视频讲座和另一个 Paxos 的视频讲座，并创建了相应的小测验。Raft 讲座涵盖了本文除了 log compaction 之外的内容;Paxos

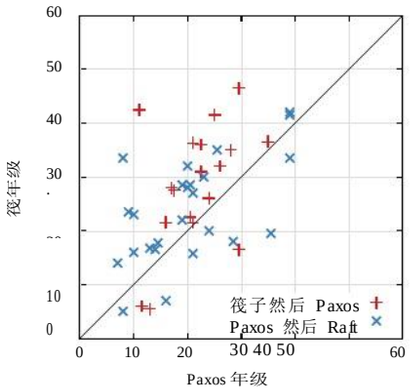

图 14:比较 43 名参与者在 Raft 和 Paxos 测试中的表现的散点图。对角线以上的点(33)代表在 Raft 中得分较高的参与者。讲座涵盖了足够的材料来创建一个等效的复制状态机，包括单法令 Paxos，多法令 Paxos，重新配置，以及实践中需要的一些优化(如领导人选举)。测验测试了对算法的基本理解，也要求学生对边缘情况进行推理。每个学生看一个视频，做相应的测验，看第二个视频，再做第二个测验。大约一半的参与者先做 Paxos 部分，另一半先做 Raft 部分，以便解释个体在表现上的差异和从第一部分研究中获得的经验。我们比较了参与者在每个测验中的得分，以确定参与者是否对 Raft 表现出了更好的理解。

我们试图让 Paxos 和 Raft 之间的比较尽可能公平。实验在两个方面对 Paxos 有利:43 名参与者中有 15 人报告说有一些

之前使用 Paxos 的经验，Paxos 的视频比 Raft 的视频要长 14%。如表 1 所述，我们已经采取措施来减少潜在的偏见来源。我们所有的材料都可以查阅[28,31]。

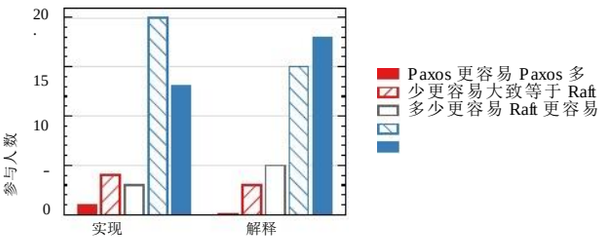

图 15:使用 5 分制量表，参与者被问及(左)他们觉得哪种算法在一个有效、正确和高效的系统中更容易实现，以及(右)哪种算法更容易向 CS 研究生解释。

他们学习算法的顺序。该模型预测，quiz 的选择产生了对 Raft 的 12.5 分的偏好差异。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 Paxos 的经验，这对 Paxos 的帮助很大，而对 Raft 的帮助略小。奇怪的是，该模型还预测，已经参加过 Paxos 测试的人在 Raft 上的得分要低 6.3 分;虽然我们不知道原因，但这似乎确实具有统计学意义。

我们还在测试后对参与者进行了调查，看看他们觉得哪种算法更容易实现或解释;这些结果如图 15 所示。绝大多数参与者报告说 Raft 更容易实现和解释(每个问题 41 个中的 33 个)。然而，这些自我报告的感受可能不如参与者的测验分数可靠，参与者可能因为知道我们的假设 Raft 更容易理解而产生了偏见。

关于 Raft 用户研究的详细讨论可以在[31]上找到。

9.2 正确性

我们已经为第 5 节中描述的共识机制制定了正式规范和安全性证明。正式规范[31]使图 2 中总结的信息完全精确使用

参与者在 TLA+规范语言[17]上的平均得分高出 4.9 分。它比 Paxos 测验(一个可能的 60 长，并作为证明的主题)大约是 400 行 Raft 测验。它也是使用-分，筏的平均分数是 25.7，对于任何实施筏的人来说，平均 Paxos 分数都是自己的。我们的得分是 20.8);图 14 显示了他们的个人分数。机械证明 Log 完备性美国*配对 t* 检验表明，95%的置信度，真实的 TLA 证明系统[7]。然而，这一证明依赖于 Raft 分数的分布在没有经过机械检查的不变量上的平均至少 2.5 分大于 Paxos 分数的真实分布。(例如，我们还没有证明 the 的类型安全性

我们还创建了一个预测规格的线性回归模型)。此外，我们根据三个因素编写了一个非正式的新学生的测试分数:他们参加了哪些测试的状态机安全属性的[31]证明，他们之前的 Paxos 经验的程度，并且是完整的(它仅依赖于规范)和**采取的相关步骤来减轻审查的偏见材料[28,31]**相等的讲座质量相同的讲师。Paxos 讲座基于现有视频，并从现有视频中改进

Ing 材料在几所大学使用。Paxos 讲座要长 14%。

同等难度的试题按难度分组，并在考试中配对。小测验

公平评分采用量规。评分是随机的，在测验之间交替进行。标题

表 1:研究中对 Paxos 可能存在偏见的担忧、应对措施以及可获得的额外材料。

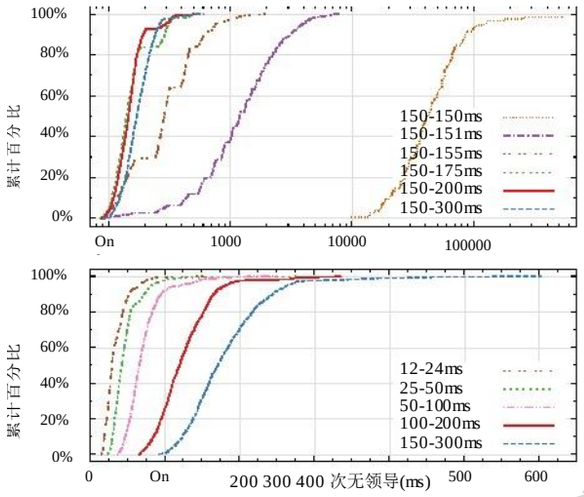

图 16:检测并替换坠毁 leader 的时间。上面的图改变了选举超时的随机性，下面的图缩放了最小选举超时时间。每一行代表 1000 次试验(除了“150-150ms”的 100 次试验)，对应于选举超时的特定选择;例如，“150-155ms”意味着选举超时是在 150ms 和 155ms 之间随机均匀选择的。测量是在一个由 5 台服务器组成的集群上进行的，广播时间大约为 15ms。对于一个由 9 台服务器组成的集群，测量结果是相似的。

相当精确(长度约为 3500 字)。

9.3 性能

Raft 的性能类似于 Paxos 等其他共识算法。对于性能来说，最重要的情况是当一个既定的领导者正在复制新的日志条目时。Raft 使用最少的消息数量(从 leader 到半个集群的一次往返)实现了这一点。此外，还可以进一步提高 Raft 的性能。例如，它很容易支持批处理和流水线请求，以获得更高的吞吐量和更低的延迟。文献中针对其他算法提出了各种优化;其中许多可以应用于 Raft，但我们把这个留给未来的工作。

我们使用我们的 Raft 实现来测量 Raft 的领导人选举算法的性能，并回答两个问题。第一，选举过程是否快速收敛?第二，领袖崩溃后能实现的最小停机时间是多少?

为了衡量 leader 选举，我们反复让 5 台服务器组成的集群中的 leader 宕机，并计时检测宕机并选举新 leader 所需的时间(见图 16)。为了生成一个最坏的情况，每次试验中的服务器有不同的日志长度，因此一些候选人没有资格成为 leader。此外，为了鼓励分裂投票，我们的测试脚本在终止其进程之前从 leader 触发了心跳 rpc 的同步广播(这近似于 leader 在崩溃之前复制新日志条目的行为-荷兰国际集团(ing))。leader 在其心跳间隔内均匀随机地崩溃，心跳间隔是所有测试最小选举超时时间的一半。因此，最小可能的宕机时间大约是最小选举超时时间的一半。

图 16 中最上面的图表显示，选举超时中少量的随机化就足以避免选举中的选票分裂。在没有随机性的情况下，由于存在很多分裂选票，在我们的测试中，领导人选举持续时间超过 10 秒。仅仅增加 5 毫秒的随机性就有很大帮助，导致平均停机时间为 287 毫秒。使用更多的随机性可以改善最坏情况下的行为:在 50ms 的随机性下，最坏情况下的完成时间(超过 1000 次试验)是 513ms。

图 16 的底部图表显示，可以通过减少选举超时时间来减少停机时间。在选举超时时间为 12-24ms 的情况下，选举领导人平均只需要 35ms(最长的试验需要 152ms)。然而，将选举超时时间降低到超过这一点违反了 Raft 的计时要求:领导者在其他服务器开始新的选举之前很难广播心跳。这可能会导致不必要的 leader 变更，降低系统的整体可用性。我们建议使用保守的选举超时时间，如 150-300ms;这样的超时不太可能造成不必要的领导人变化，仍然会提供良好的可用性。

10 相关工作

已经有很多与共识算法相关的出版物，其中许多属于以下类别之一:

- Lamport 对 Paxos[15]的原始描述，并试图更清楚地解释它[16,20,21]。
- Paxos 的阐述，填补缺失的细节并修改算法，为实现提供更好的基础[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 的算法尚未详细公布，尽管两者都声称基于 Paxos。ZooKeeper 的算法已经发表了更详细的内容，但与 Paxos 有很大的不同。
- 可应用于 Paxos 的性能优化[18,19,3,25,1,27]。
- Oki 和 Liskov 的 Viewstamped Replication (VR)，一种与 Paxos 几乎同时开发的共识替代方法。最初的描述[29]与分布式交易协议交织在一起，但核心共识协议在最近的更新[22]中被分离。VR 使用的是基于领导者的方式，与 Raft 有很多相似之处。

Raft 与 Paxos 最大的区别在于 Raft 强大的领导力:Raft 将领导者选举作为共识协议的重要组成部分，它关注-

尽可能多地发挥领导者的功能。这种方法产生了更简单、更容易理解的算法。例如，在 Paxos 中，领导者选举与基本共识协议正交:它仅作为性能优化，并不是实现共识所必需的。然而，这导致了额外的机制:Paxos 既包括用于基本共识的两阶段协议，也包括用于领导人选举的单独机制。相比之下，Raft 将领导人选举直接纳入共识算法，并将其作为两阶段共识的第一阶段。这导致了比 Paxos 更少的机制。

像 Raft 一样，VR 和 ZooKeeper 都是基于领导者的，因此与 Paxos 相比，它们具有许多 Raft 的优势。然而，与 VR 或 ZooKeeper 相比，Raft 的机制更少，因为它将非领导者的功能最小化。例如，Raft 中的日志条目只向一个方向流动:在 AppendEntries rpc 中从 leader 向外流动。在 VR 中，日志条目是双向流动的(领导人可以在选举过程中收到日志条目);这导致了额外的机制和复杂性。发布的 ZooKeeper 描述也在 leader 和 leader 之间传输日志条目，但实现显然更像 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(不包括日志合并和客户端交互，因为这些几乎独立于算法)。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(2 个 RPC 请求和它们的响应)。Raft 的消息比其他算法更密集一些，但它们总体上更简单。此外，VR 和 ZooKeeper 被描述为在 leader 变化期间传输整个日志;将需要额外的消息类型来优化这些机制，使其具有实用性。

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)可以在某些条件下用无领导的方法[27]实现更高的性能。EPaxos 利用了状态机命令中的交换性。任何服务器只需一轮通信就可以提交一个命令，只要同时提出的其他命令与之交换即可。然而，如果并发提出的命令不相互通勤，EPaxos 就需要额外一轮的通信。因为任何服务器都可能提交命令，所以 EPaxos 可以很好地平衡服务器之间的负载，并且能够在 WAN 设置中实现比 Raft 更低的延迟。然而，它为 Paxos 增加了显著的复杂性。

在其他工作中，已经提出或实施了几种不同的集群成员变化方法，包括 Lamport 的原始提案[15]、VR[22]和 SMART[24]。我们为 Raft 选择了联合共识的方法，因为它利用了共识协议的其余部分，因此对于成员关系的更改几乎不需要额外的机制。Lamport 的基于 α 的方法不是 Raft 的选择，因为它假定在没有领导者的情况下可以达成共识。与 VR 和 SMART 相比，Raft 的重构算法有一个优势，即可以在不限制正常请求处理的情况下发生成员变化;相比之下，VR 在配置变化时停止所有正常处理，SMART 对未完成请求的数量施加了 α -like 限制。与 VR 或 SMART 相比，Raft 的方法添加的机制也更少。

11 的结论

算法的设计通常以正确性、效率和/或简洁为主要目标。虽然这些都是有价值的目标，但我们相信可理解性同样重要。除非开发者将算法转化为实际实现，否则其他目标都无法实现，而实际实现将不可避免地偏离并在公布的形式上进行扩展。除非开发人员对算法有深刻的理解，并能创造出关于它的直觉，否则他们将很难在实现中保留其理想的属性。

在这篇文章中，我们讨论了分布式共识的问题，其中一个被广泛接受但无法穿透的算法 Paxos，多年来一直挑战着学生和开发人员。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。我们也相信，Raft 为系统构建提供了更好的基础。将可理解性作为主要设计目标改变了我们进行 Raft 设计的方式;随着设计的进展，我们发现自己重复使用了一些技术，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，而且更容易让我们相信它的正确性。

12 致谢

如果没有 Ali Ghodsi、David Mazi 'eres 以及伯克利 CS 294-91 课程和斯坦福 CS 240 课程的学生的支持，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 幻灯片大量借鉴了洛伦佐·阿尔维西(Lorenzo Alvisi)最初创建的幻灯片。特别感谢 David Mazi 'eres 和 Ezra Hoch 在 Raft 中发现了微妙的 bug。许多人对论文和用户研究材料提供了有用的反馈，包括 Ed Bugnion、Michael Chan、Hugues Evrard、

Daniel Giffin, Arjun Gopalan, Jon Howell, Vimalkumar Jeyakumar, Ankita Kejriwal, Aleksandar Kracun, Amit Levy, Joel Martin, Satoshi Matsushita, Oleg Pesok, David Ramos, robert van Renesse, Mendel Rosenblum, Nicolas Schiper, Deian Stefan, Andrew Stone, Ryan Stutsman, David Terei, Stephen Yang, Matei Zaharia, 24 名匿名会议评审员(有重复)，尤其是我们的牧羊人 Eddie Kohler。Werner Vogels 在推特上发了一个早期草稿的链接，这给了 Raft 很大的曝光率。这项工作得到了 Gigascale 系统研究中心和多尺度系统中心(半导体研究公司项目 Focus Center Research Program 资助的六个研究中心中的两个)、STARnet (MARCO 和 DARPA 赞助的半导体研究公司项目)、美国国家科学基金会(No. 0963859)的资助，以及 Facebook、谷歌、Mellanox、NEC、NetApp、SAP 和三星的资助。Diego Ongaro 得到了 Junglee Corporation 斯坦福研究生奖学金的支持。

参考文献

[1]BOLOSKY, W. J.， BRADSHAW, D.， HAAGENS, R. B.， KUSTERS,N. P.和 LI, P. Paxos 复制了状态机作为高性能数据存储的基础。在 Proc. NSDI’*11 中，《USENIX 网络系统设计与实现会议》(2011)，USENIX*，第 141-154 页。

[2]BURROWS, M.《松耦合分布式系统的 Chubby lock 服务》。In Proc. *OSDI’06,Symposium on Operating Systems Design and Implementation (2006)*， USENIX，第 335-350 页。

[3]CAMARGOS, L. J.， SCHMIDT,R. M.，和 PEDONE, F.MulticoordinatedPaxos 。在 Proc. PODC’07,ACM Symposium on Principles of Distributed Computing (2007)， ACM, pp. 316-317。

[4]钱德拉，T. D.，GRIESEMER,R.，和 REDSTONE,J. Paxos made live: an engineering perspective。在 Proc. PODC’*07,ACM Symposium on Principles of Distributed Computing (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.， ANDGRUBER,R. E. Bigtable:一个结构化数据的分布式存储系统。在 Proc. OSDI’06 中，USENIX Symposium on Operating Systems Design and Implementation (2006)， USENIX, pp. 205-218。

[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.， ANDWOODFORD, D.Spanner:谷歌的全球分布式数据库。在 Proc. *OSDI’*12，《USENIX 操作系统设计与实现会议》(2012)，USENIX，第 251-264 页。

[7]COUSINEAU, D.，DOLIGEZ,L.， LAMPORT, MERZ, S.， RICKETTS,D.， AND VANZETTO, H.TLA+ proofs。在 Proc. FM’12,Symposium *on Formal Methods (2012)， D. Giannakopoulou* 和 D. M´ery, Eds.;，《计算机科学讲义》第 7436 卷，施普林格，第 147-154 页。

[8] S .ghemawat,gobioff, h .和 leung, S.-t .。谷歌文件系统。在 Proc. *SOSP’03,ACM Symposium on Operating Systems Principles (2003)*， ACM, pp. 29-43。

[9] GRAY,C.和 CHERITON,D.:一种高效的分布式文件缓存一致性容错机制。在*第 12 届 ACM Ssymposium on Operating Systems Principles 会议论文集(1989)中，pp. 202-210*。

[10]HERLIHY, m.p.， AND WING,j.m. Linearizability:并发对象的正确性条件。美国计算机学会*程序设计语言与系统学报 12(1990 年 7 月)，463-492*。

[11] HUNT,P.， KONAR,M.，JUNQUEIRA, F. P.， AND REED, B.ZooKeeper:面向互联网规模系统的无等待协调。《Proc ATC’10,*USENIX 年度技术会议(2010)》，USENIX*，第 145-158 页。

[12]JUNQUEIRA, F.P.， REED,B. C.和 SERAFINI, M. Zab:主备系统的高性能广播。在 Proc. *DSN’11 中，IEEE/IFIP 国际会议 on reliable Systems & Networks (2011)*， IEEE 计算机学会，pp. 245-256。

KIRSCH, J.，和 AMIR, Y. Paxos 为系统建设者。技术众议员 CNDS-2008-2，约翰·霍普金斯大学，2008 年。

L. [14]LAMPORT，《分布式系统中的时间、时钟和事件排序》。*ACM 通讯 21,7(1978 年 7 月)*，558-565。

兼职议会。ACM *计算机系统学报 16,2(*1998 年 5 月)，133-169。

[16]LAMPORT, L. Paxos made simple 。ACM*SIGACT 新闻 32,4(*2001 年 12 月)，18-25。

L.指定系统，*用于硬件和软件工程师的 TLA+ 语言和工具*。Addison- Wesley，2002。

[18]LAMPORT, L.广义共识与 Paxos。技术。众议员 MSR-TR-2005-33，微软研究院，2005。

[19]LAMPORT, L. Fast paxos。分布式计算 19,2

(2006), 79 - 103。

[20]LAMPSON, B. W.如何使用共识构建高可用性系统。在分布式*算法中，O. Baboaglu* 和 K. Marzullo, Eds.;Springer-Verlag, 1996，第 1-17 页。

[21]LAMPSON, B. W. The ABCD 的 Paxos 。在 Proc. *PODC’01,ACM Symposium on Principles of Distributed Computing (2001)*， ACM, pp. 13-13。

[22]LISKOV, B.，和 COWLING,J. Viewstampedreplication revisited。科技众议员 MIT- csail - tr -2012-021，麻省理工学院，2012 年 7 月。

[23] LogCabin 源代码。http://github.com/好像好像。

[24]LORCH, J. R.， ADYA,A.，BOLOSKY, W. J.， CHAIKEN,R.， DOUCEUR,J. R.， AND HOWELL, J.TheSMART way to migration replicated stateful services。《Proc. EuroSys’06》，*ACM SIGOPS/EuroSys 欧洲计算机系统会议(2006)，ACM，第* 103-115 页。

《Proc. *OSDI’08,USENIX 操作系统设计与实现会议》(2008*)，USENIX，第 369-384 页。

[26]MAZIE’RES, D. Paxos 使之实用。http: / / www.scs.stanford.edu/~dm /home /

论文/paxos.pdf, 2007 年 1 月。

在平等主义的议会中有更多的共识。在 Proc. 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.和 LISKOV,B. H. Viewstamped replication:一种支持高可用分布式系统的新主复制方法。在 Proc. *PODC’88,ACM Symposium on Principles of Distributed Computing (1988*)， ACM, pp. 8-17。

[30] O 'NEIL,P.， CHENG,E.， GAWLICK,D.，和 ONEIL, E.log-structured merge-tree (LSM-tree)。信息学报 *33,4(1996)，351*-385。

[31]ONGARO, D.共识:*理论与实践的桥梁*。博士论文，斯坦福大学，2014 年(工作中)。

http://ramcloud.stanford.edu/~ongaro / thesis.pdf。

[32]ONGARO, D.和 OUSTERHOUT, J.寻求一种可理解的共识算法。在 Proc ATC’14,*USENIX 年度技术会议(2014)，USENIX*。

[33]OUSTERHOUT, J.， AGRAWAL, P.， ERICKSON, D.， KOZYRAKIS, C.， LEVERICH, J.， MAZIE' RES, D.， MITRA,S.， NARAYANAN, A.， ONGARO,D.， PARULKAR,G.， ROSENBLUM,M.， RUMBLE, S. M.， STRATMANN, E.， AND STUTSMAN, R.RAMCloud 的案例。*《ACM 通讯》54(2011 年 7 月)，121*-130。

[34] Raft 共识算法网站。

http://raftconsensus.github.io 。

[35] REED,B.Personal communications, 2013 年 5 月 17 日。

[36]ROSENBLUM, M.和 OUSTERHOUT, J. K.《一个日志结构文件系统的设计与实现》。*ACM 反式。第一版。系统 10(1992 年 2 月)*，26-52。

[37] SCHNEIDER,F. B.使用状态机方法实现容错服务:教程。ACM 计算调查 *22,4(1990 年 12 月)，299-319*。

[38]SHVACHKO, K.， KUANG, H.， RADIA, S.， AND CHANSLER, R.《Hadoop 分布式文件系统》。In Proc. *MSST’10,Symposium on Mass Storage Systems and Technologies (2010)*， IEEE 计算机学会，pp. 1-10。

[39] VANRENESSE, R.Paxos made moderate complex。康奈尔大学技术代表，2012 年。