寻找一种可理解的一致性算法
(扩展版本)

迭戈·翁加罗和约翰·奥特鲁特
斯坦福大学

摘要

Raft 是管理复制日志的通用算法。它产生一个等价于(多)Paxos 的结果，和 Paxos 一样高效，但结构和 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 将共识的关键元素(如领导者选举、日志复制和安全)分开，并加强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的新机制，该机制使用重叠多数来保证安全性。

一 介绍

共识算法允许一组机器作为一个连贯的组工作，可以在一些成员失败后幸存下来。正因为如此，它们在构建可靠的大规模软件系统中起着关键作用。在过去的十年里，Paxos [15，16]一直主导着共识算法的讨论:大多数共识的实现都是基于 Paxos 或受其影响，Paxos 已经成为教授学生共识的主要工具。

不幸的是，Paxos 很难理解，尽管无数次试图使它更容易接近。此外，它的架构需要复杂的变化来支持实际的系统。因此，无论是系统构建者还是学生都在与 Paxos 做斗争。

经过与 Paxos 自己的斗争，我们开始寻找一种新的共识算法，可以为系统构建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一致的算法，并以一种比 Paxos 更容易学习的方式来描述它？此外，我们希望该算法能够促进直觉的发展，而直觉对于系统构建者来说是必不可少的。重要的不仅仅是算法的工作，还有它为什么工作的原因。

这项工作的结果是一个被称为 Raft 的共识算法。在设计 Raft 时，我们应用了特定的技术来提高可理解性，包括组合(Raft 将领导者选举、日志复制和安全性分开)和

该技术报告是[32]的扩展版本；其他材料在页边空白处用灰色条标出。2014 年 5 月 20 日出版。

状态空间减少(相对于 Paxos，Raft 减少了不确定性的程度和服务器之间不一致的方式)。一项针对两所大学 43 名学生的用户研究显示，Raft 比 Paxos 更容易理解:在学习了这两种算法后，其中 33 名学生能够更好地回答关于 Raft 的问题，而不是关于 Paxos 的问题。

Raft 在许多方面类似于现有的一致算法(最著名的是 Oki 和 Liskov 的视图标记复制[29，22])，但它有几个新的特点:

- 强领导:Raft 使用了比其他共识算法更强的领导形式。例如，日志条目仅从领导者流向其他服务器。这简化了复制日志的管理，并使 Raft 更容易理解。
- 领袖选举:Raft 使用随机计时器选举领袖。这仅仅给任何一致算法所需的心跳增加了少量的机制，同时简单快速地解决冲突。
- 成员变化:Raft 改变集群中服务器集的机制使用了一种新的联合一致方法，其中两种不同配置的大多数在转换过程中重叠。这允许群集在配置更改期间继续正常运行。

我们认为，无论是出于教育目的还是作为实施基础，Raft 都优于 Paxos 和其他共识算法。比其他算法更简单易懂；它的描述完全足以满足实际系统的需要；它有几个开源实现，并被几家公司使用；其安全性能已经过正式规定和证明；其效率与其他算法相当。

本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)，描述了我们理解的一般方法(第 4 节)，介绍了 Raft 一致性算法(第 5-8 节)，评估 Raft(第 9 节)，并讨论了相关工作(第 10 节)。

2 复制状态机

共识算法通常出现在复制状态机的环境中[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，并且即使某些服务器停机，也可以继续运行。复制的状态机是

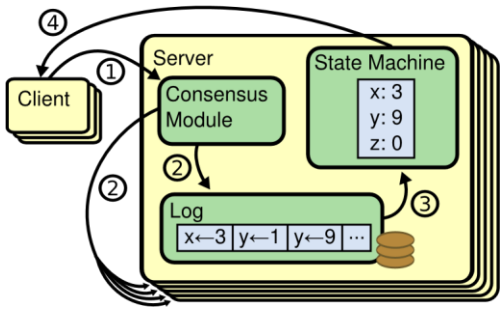

图 1:复制状态机架构。一致性算法管理包含来自客户端的状态机命令的复制日志。状态机从日志中处理相同的命令序列，因此它们产生相同的输出。

用于解决分布式系统中的各种容错问题。例如，只有一个集群领导者的大规模系统，如 GFS [8]、HDFS [38]和 RAMCloud [33]，通常使用单独的复制状态机来管理领导者选举和存储必须在领导者崩溃后仍然存在的配置信息。复制状态机的例子包括 Chabby[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。每台服务器都存储一个日志，其中包含一系列命令，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。因为状态机是确定性的，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一致性算法的工作。服务器上的一致性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一致模块通信，以确保每个日志最终都以相同的顺序包含相同的请求，即使有些服务器出现故障。一旦命令被正确复制，每个服务器的状态机就按日志顺序处理它们，并将输出返回给客户端。结果，服务器看起来形成了一个单一的、高度可靠的状态机。

实际系统的一致性算法通常具有以下特性:

- 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、数据包丢失、复制和重新排序。
- 只要大多数服务器都可以运行，并且可以相互通信和与客户端通信，它们就可以完全正常工作(可用)。因此，一个典型的五个服务器的集群可以容忍任何两个服务器的故障。通过停止，服务器被假定为失败；稍后，它们可能会从稳定存储的状态中恢复并重新加入群集。
- 它们不依赖定时来确保日志的一致性:错误的时钟和极端的消息延迟在最坏的情况下会导致可用性问题。
- 在一般情况下，只要集群的大部分成员响应了一轮远程过程调用，命令就可以完成；少数速度慢的服务器不需要影响整体系统性能。

3 帕克斯怎么了？

在过去的十年里，莱斯利·兰波特的 Paxos 协议[15]几乎已经成为共识的同义词:它是课程中最常用的协议，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个决策上达成一致的协议，比如一个复制的日志条目。我们称这个子集为单指令派克斯。Paxos 然后组合该协议的多个实例，以促进一系列决策，如日志(多 Paxos)。Paxos 确保了安全性和活性，并且它支持集群成员的改变。其正确性已被证明，在正常情况下是有效的。

不幸的是，Paxos 有两个明显的缺点。第一个缺点是 Paxos 异常难懂。充分的解释[15]是出了名的不透明；很少有人能成功地理解它，只有付出巨大的努力。因此，有几种尝试用更简单的术语来解释帕克斯[16，20，21]。这些解释侧重于单个法令子集，但它们仍然具有挑战性。在对 2012 年 NSDI 年会与会者的非正式调查中，我们发现很少有人对帕克斯感到满意，即使是在经验丰富的研究人员中。我们自己和帕克斯一起奋斗；直到阅读了几个简单的解释并设计了我们自己的替代方案，我们才能够理解完整的方案，这个过程花了将近一年的时间。

我们假设帕克斯的不透明性源于其选择单一法令子集作为基础。单令 Paxos 密集而微妙:分为两个阶段，没有简单直观的解释，无法独立理解。正因为如此，很难对单一命令协议的工作原理有直观的了解。多参数组合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的总体问题(即，。日志而不是单个条目)可以以其他更直接和明显的方式分解。

Paxos 的第二个问题是它没有为构建实际的实现提供良好的基础。一个原因是没有广泛认同的多 Paxos 算法。兰波特的描述大多是关于单令派克斯的；他概述了多派和平的可能途径，但缺少了许多细节。有几个尝试充实和优化帕克斯，如[26]、[39]和[13]，但这些彼此不同，也不同于兰波特的草图。像查比[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，它们的细节还没有公布。

此外，Paxos 体系结构对于构建实用系统来说是一个糟糕的体系结构；这是单一法令分解的另一个结果.例如，独立地选择一组日志条目，然后将它们合并成一个连续的日志，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、更有效，在这个系统中，新的条目以受约束的顺序依次追加。另一个问题是 Paxos 在其核心使用了对称的对等方法(尽管它最终暗示了一种弱领导形式作为性能优化)。这在一个简化的世界中是有意义的，在这个世界中，只有一个决定会被做出，但是很少有实际的系统使用这种方法。如果一定要做一系列的决策，先选一个领导，然后让领导协调决策，这样更简单快捷。

因此，实用系统与 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体系结构。这既耗时又容易出错，理解 Paxos 的困难加剧了问题。帕克斯的公式可能是一个很好的公式，可以提供关于其正确性的理论，但是

真正的实现与帕克斯的不同之处在于证明没有什么价值.来自胖胖实现者的以下命令是典型的:

> 在Paxos算法的描述和现实世界系统的需求之间有很大的差距.... 最终的系统将基于一个未经证实的协议[4]。

由于这些问题，我们得出结论，Paxos无论是对系统构建还是对教育都没有提供良好的基础。鉴于共识在大规模软件系统中的重要性，我们决定看看是否可以设计一种性能优于 Paxos 的替代共识算法。筏是那个实验的结果。

四　　为易懂而设计

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和实用的基础，这样它就可以显著减少开发人员所需的设计工作量；它在井下条件下必须是安全的，并且在典型操作条件下可用；并且对于普通操作必须是有效的。但是我们最重要的目标——也是最困难的挑战——是可理解性。一定要有可能让大量观众舒服地理解算法。此外，必须能够开发关于算法的直觉，以便系统构建者能够进行在现实世界实现中不可避免的扩展。

在筏子的设计中，有许多地方我们必须在可供选择的方法中进行选择。在这些情况下，我们基于可理解性来评估选项:解释每个选项有多难(例如，它的状态空间有多复杂，它是否有微妙的含义？，读者完全理解这种方法及其含义有多容易？

我们认识到这种分析具有高度的主观性；尽管如此，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方法:只要有可能，我们就把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，原木不允许有孔，Raft 限制了原木相互不一致的方式。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，随机化方法引入了不确定性，但是它们倾向于通过以相似的方式处理所有可能的选择来减少状态空间(“选择任意；没关系”)。我们使用随机化来简化 Raft 领导者选举算法。

5　　筏共识算法

Raft 是一种算法，用于管理第 2 节所述形式的复制日志。图 2 以浓缩的形式总结了算法供参考，图 3 列出了算法的关键属性；这些图的元素将在本节的其余部分中逐段讨论。

Raft 通过首先选举一个杰出的领导者，然后赋予领导者管理复制日志的全部责任来实现共识。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，并告诉服务器何时将日志条目应用到它们的状态机是安全的。有一个领导者简化了复制日志的管理。例如，领导者可以决定在日志中何处放置新条目，而无需咨询其他服务器，数据以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。

给定领导方法，Raft 将共识问题分解为三个相对独立的子问题，这些子问题将在下面的小节中讨论:

- 领导人选举:当现有领导人失败时，必须选择新领导人(第 5.2 节)。
- Log replication: the leader must accept log entries

状态

请求投票

Invoked by candidates to gather votes (§5.2).

Arguments:

term	candidate’s term
candidateId	candidate requesting vote
lastLogIndex	index of candidate’s last log entry (§5.4)
lastLogTerm	term of candidate’s last log entry (§5.4)

Results:

term	currentTerm, for candidate to update itself
voteGranted	true means candidate received vote

Receiver implementation:

1. Reply false if term < currentTerm (§5.1)
2. If votedFor is null or candidateId, and candidate’s log is at least as up-to-date as receiver’s log, grant vote (§5.2, §5.4)

Persistent state on all servers:
(Updated on stable storage before responding to RPCs)

currentTerm latest term server has seen (initialized to 0 on first boot, increases monotonically)

votedFor candidateId that received vote in current term (or null if none)

log[] log entries; each entry contains command for state machine, and term when entry was received by leader (first index is 1)

Volatile state on all servers:

commitIndex index of highest log entry known to be committed (initialized to 0, increases monotonically)

lastApplied index of highest log entry applied to state machine (initialized to 0, increases monotonically)

Volatile state on leaders:
(Reinitialized after election)

nextIndex[] for each server, index of the next log entry to send to that server (initialized to leader last log index + 1)

matchIndex[] for each server, index of highest log entry known to be replicated on server (initialized to 0, increases monotonically)

追加条目

Invoked by leader to replicate log entries (§5.3); also used as heartbeat (§5.2).

Arguments:

term leader's term

leaderId so follower can redirect clients **prevLogIndex** index of log entry immediately preceding new ones

prevLogTerm term of prevLogIndex entry **entries[]** log entries to store (empty for heartbeat; may send more than one for efficiency)

leaderCommit leader's commitIndex

Results:

term currentTerm, for leader to update itself **success** true if follower contained entry matching prevLogIndex and prevLogTerm

Receiver implementation:

1. Reply false if term < currentTerm (§5.1)
2. Reply false if log doesn't contain an entry at prevLogIndex whose term matches prevLogTerm (§5.3)
3. If an existing entry conflicts with a new one (same index but different terms), delete the existing entry and all that follow it (§5.3)
4. Append any new entries not already in the log
5. If leaderCommit > commitIndex, set commitIndex = min(leaderCommit, index of last new entry)

服务器规则

All Servers:

- If commitIndex > lastApplied: increment lastApplied, apply log[lastApplied] to state machine (§5.3)
- If RPC request or response contains term T > currentTerm: set currentTerm = T, convert to follower (§5.1)

Followers (§5.2):

- Respond to RPCs from candidates and leaders
- If election timeout elapses without receiving AppendEntries RPC from current leader or granting vote to candidate: convert to candidate

Candidates (§5.2):

- On conversion to candidate, start election:
- Increment currentTerm
- Vote for self
- Reset election timer
- Send RequestVote RPCs to all other servers
- If votes received from majority of servers: become leader
- If AppendEntries RPC received from new leader: convert to follower
- If election timeout elapses: start new election

Leaders:

- Upon election: send initial empty AppendEntries RPCs (heartbeat) to each server; repeat during idle periods to prevent election timeouts (§5.2)
- If command received from client: append entry to local log, respond after entry applied to state machine (§5.3)
- If last log index ≥ nextIndex for a follower: send AppendEntries RPC with log entries starting at nextIndex
- If successful: update nextIndex and matchIndex for follower (§5.3)
- If AppendEntries fails because of log inconsistency: decrement nextIndex and retry (§5.3)
- If there exists an N such that N > commitIndex, a majority of matchIndex[i] ≥ N, and log[N].term == currentTerm: set commitIndex = N (§5.3, §5.4).

图 Raft 一致性算法的浓缩总结(不包括成员变化和日志压缩)。左上角框中的服务器行为被描述为一组独立且重复触发的规则。章节号，如 5.2，表示讨论特定功能的地方。形式规范[31]更精确地描述了算法。

选举安全:在给定的任期内，最多只能选出一名领导人。5.2	
仅添加领导者:领导者从不覆盖或删除其日志中的条目；它只追加新条目。5.3	
日志匹配:如果两个日志包含具有相同索引和术语的条目，则在给定索引的所有条目中，日志是相同的。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被提交，那么该条目将出现在所有高编号术语的领导者的日志中。5.4	
状态机安全性:如果一个服务器已经在给定的索引处为其状态机应用了一个日志条目，则没有其他服务器会为同一个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正确的。章节号表示每个属性的讨论位置。

并在集群中复制它们，迫使其他日志与自己的一致(第 5.3 节)。

- 安全:Raft 的关键安全属性是图 3 中的状态机安全属性:如果任何服务器已经对其状态机应用了特定的日志条目，则没有其他服务器可以对同一日志索引应用不同的命令。第 5.4 节描述了 Raft 如何确保这一特性；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。

在介绍了一致性算法之后，本节讨论了可用性问题和系统中定时的作用。

5.1 筏基础

一个 Raft 集群包含几个服务器；五是一个典型的数字，它允许系统容忍两次故障。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。在正常操作中，只有一个领导者，所有其他服务器都是追随者。追随者是被动的:他们不会主动提出要求，只是简单地回应领导和候选人的要求。领导者处理所有的客户请求(如果客户联系一个跟随者，跟随者将其重定向到领导者)。第三种状态，候选人，用于选举新的领导人，如第 5.2 节所述.图 4 显示了状态及其转换；下面讨论过渡。

Raft 将时间划分为任意长度的项，如图 5 所示。术语用连续整数编号。每个任期始于一次选举，其中一名或多名候选人试图成为第 5.2 节所述的领导人.如果一名候选人赢得选举，那么他将在剩余的任期内担任领导人。在某些情况下，选举会导致分裂投票。在这种情况下，任期结束时没有领导人；新的任期(新的选举)

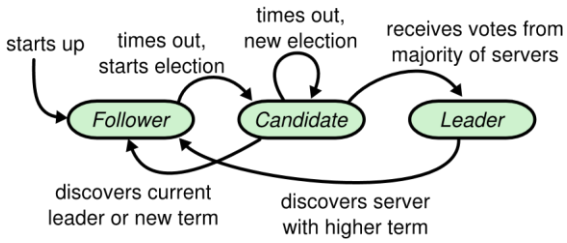

图 4:服务器状态。关注者只响应其他服务器的请求。如果一个追随者没有收到任何信息，它就成为候选人并开始选举。从全体成员中获得多数票的候选人将成为新的领导者。领导者通常运作到失败。

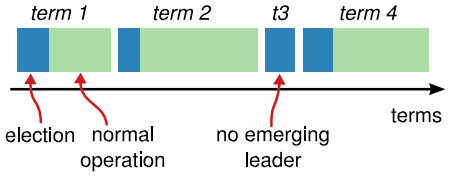

图 5:时间分为任期，每个任期以一次选举开始。在一次成功的选举后，一个领导人将管理这个集群，直到任期结束。一些选举失败，在这种情况下，任期结束时没有选择领导人。术语之间的转换可以在不同服务器上的不同时间观察到。

很快就会开始。Raft 确保在给定的任期内最多有一个领导。

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选举甚至整个术语。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，如过时的领导者。每个服务器存储一个当前术语编号，该编号随时间单调增加。只要服务器通信，就交换当前术语；如果一台服务器的当前期限小于另一台服务器的期限，则它会将其当前期限更新为更大的值。如果一个候选人或领导者发现他的任期已经过期，他会立即回复到追随者状态。如果服务器收到一个带有过期术语编号的请求，它会拒绝该请求。

Raft 服务器使用远程过程调用(RPC)进行通信，基本的一致性算法只需要两种类型的 RPC。候选人在选举期间发起请求投票 RPC(第 5.2 节)，而追加条目 RPC 由领导者发起，以复制日志条目并提供一种心跳形式(第 5.3 节)。第 7 节添加了第三个用于在服务器之间传输快照的 RPC。如果服务器没有及时收到响应，它们会重试远程过程调用，并并行发出远程过程调用以获得最佳性能。

5.2 领导人选举

Raft 使用心跳机制来触发领导人选举。当服务器启动时，它们开始成为追随者。只要服务器从领导者或候选人那里接收到有效的远程过程控制，它就保

持在从者状态。领导者定期向所有追随者发送心跳信号(不带日志条目的 AppendEntriesRPCs)，以维护他们的权威。如果一个跟随者在一段被称为选举超时的时间内没有收到任何通信，那么它就假设没有可行的领导者，并开始选举来选择一个新的领导者。

要开始选举，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，并向集群中的每个其他服务器并行发出请求投票 RPC。候选人继续处于这种状态，直到发生三件事之一:(a)它赢得了选举，(b)另一个服务器确立了自己的领导地位，或者(c)一段时间过去了，没有赢家。这些结果将在以下段落中单独讨论。

如果候选人在同一任期内获得整个集群中大多数服务器的投票，他就赢得了选举。在给定的任期内，每个服务器最多将投票给一名候选人，以先到先得的方式进行(注意:第 5.4 节增加了投票的额外限制)。多数规则确保在特定的任期内，最多有一名候选人能够赢得选举(图 3 中的选举安全属性)。候选人一旦赢得选举，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。

在等待投票时，候选人可能会从另一台声称是领导者的服务器上收到一个追加条目 RPC。如果领导者的任期(包含在其 RPC 中)至少与候选人当前的任期一样长，那么候选人将认为领导者是合法的，并返回到追随者状态。如果 RPC 中的术语小于候选人的当前术语，则候选人拒绝 RPC 并继续处于候选人状态。

第三种可能的结果是，一名候选人既不会赢得选举，也不会输掉选举:如果许多追随者同时成为候选人，选票可能会被分割，因此没有候选人获得多数票。当这种情况发生时，每个候选人都将超时，并通过增加任期和启动另一轮请求投票 RPC 来开始新的选举。然而，如果没有额外的措施，分裂的投票可能会无限期地重复。

Raft 使用随机选举超时来确保分裂的投票很少，并且很快得到解决。为了首先防止分裂投票，从固定间隔中随机选择选举超时(例如，。150–300 毫秒)。这将服务器分散开来，因此在大多数情况下，只有一台服务器会超时；它赢得选举，并在任何其他服务器超时之前发送心跳。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选举超时，并在开始下一次选举之前等待该超时过去；这降低了新选举中再次出现分裂投票的可能性。第 9.3 节表明，这种方法可以快速选出领导人。

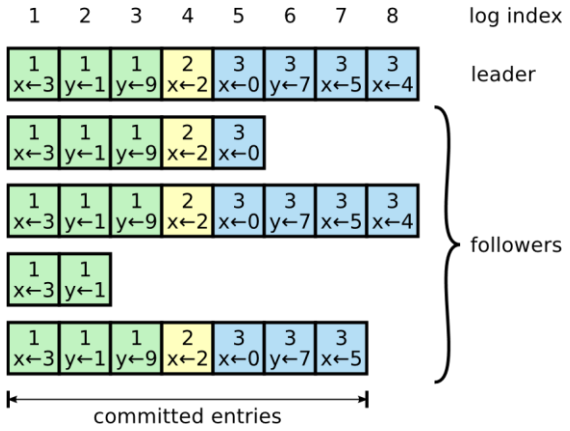

图 6:日志由条目组成，条目按顺序编号。每个条目包含创建它的术语(每个框中的数字)和状态机的命令。如果条目应用于状态机是安全的，则该条目被视为已提交。

选举是一个例子，说明可理解性如何指导我们在设计方案之间做出选择。最初，我们计划使用排名系统:每个候选人被分配一个唯一的排名，用于在竞争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，它会返回到追随者状态，这样排名更高的候选人就可以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微妙的问题(如果排名较高的服务器失败了，排名较低的服务器可能需要暂停并再次成为候选服务器，但是如果它过早这样做，它可以重新设置选举领导者的进度)。我们对算法进行了几次调整，但每次调整后都会出现新的角点情况。最终我们得出结论，随机重试方法更明显，更容易理解。

5.3 日志复制

一旦领导者被选出，它就开始服务客户请求。每个客户端请求都包含一个由复制的状态机执行的命令。领导者将命令作为新条目附加到其日志中，然后并行地向每个其他服务器发出 AppendEntries RPCs 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，或者如果网络数据包丢失，领导者无限期地重试追加条目 RPC(即使它已经响应了客户端)，直到所有追随者最终存储所有日志条目。

日志的组织如图 6 所示。每个日志条目存储一个状态机命令，以及领导者收到该条目时的术语号。日志条目中的术语编号用于检测日志之间的不一致，并确保图3中的一些属性.每个日志条目还有一个标识其在日志中位置的整数索引。

领导者决定何时将日志条目应用到状态机是安全的；这样的条目称为提交。Raft 保证提交的条目是持

久的，并将最终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(例如，。图 6 中的条目 7)。这还会提交领导日志中的所有先前条目，包括由先前领导创建的条目。第 5.4 节讨论了在领导变动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。领导者跟踪它知道要提交的最高索引，并将该索引包含在未来的附加条目 RPC(包括心跳)中，以便其他服务器最终发现。一旦跟随者得知日志条目被提交，它就将该条目应用到其本地状态机(按日志顺序)。

我们设计了 Raft 日志机制来维护不同服务器上日志之间的高度一致性。这不仅简化了系统的行为，使其更加可预测，而且是确保安全的重要组成部分。Raft 维护以下属性，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，则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，则日志在所有前面的条目中是相同的。

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用一个给定的日志索引创建最多一个条目，并且日志条目从不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。当发送一个追加条目 RPC 时，前导在它的日志中包括紧接在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，则拒绝新条目。一致性检查作为一个归纳步骤:日志的初始空状态满足日志匹配属性，无论何时扩展日志，一致性检查都会保留日志匹配属性。因此，当 wheneverAppendEntriesreturns 成功时，领导者通过新条目知道跟随者的日志与其自己的日志相同。

在正常操作期间，领导者和追随者的日志保持一致，因此附录条目一致性检查永远不会失败。但是，引导崩溃会导致日志不一致(旧的引导可能没有完全复制其日志中的所有条目)。这些不一致会加剧一系列领导者和追随者的崩溃。图 7 说明了追随者的日志可能与新领导者的不同之处。追随者可以

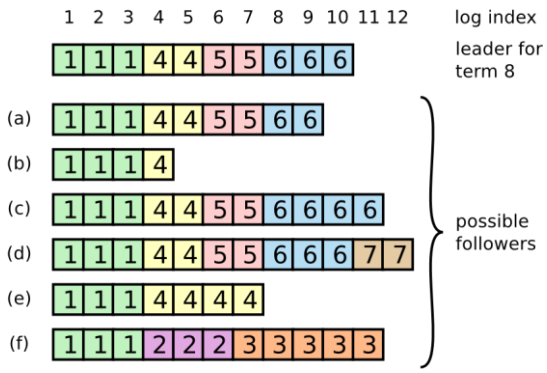

图 7:当高层领导掌权时，任何场景(a–f)都有可能出现在跟随者日志中。每个框代表一个日志条目；盒子里的数字是它的术语。跟随者可能缺少条目(a–b)，可能有额外的未提交条目(c–d)，或者两者都有(e–f)。例如，场景(f)可能会发生，如果服务器是第 2 项的领导者，在其日志中添加了几个条目，然后在提交任何条目之前崩溃；它很快重新启动，成为第三任期的领导者，并在其日志中增加了一些条目；在提交第 2 项或第 3 项中的任何条目之前，服务器再次崩溃，并在几项中保持关闭。

可能缺少引线上存在的条目，也可能有引线上不存在的额外条目，或者两者都有。日志中缺失和无关的条目可能跨越多个术语。

在 Raft 中，领导者通过强制追随者的日志复制自己的日志来处理不一致。这意味着跟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，当再加上一个限制时，这是安全的。

为了使跟随者的日志与其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，在该点之后删除跟随者日志中的任何条目，并在该点之后将领导者的所有条目发送给跟随者。所有这些操作都是为了响应 AppendEntries RPCs 执行的一致性检查而发生的。领导者为每个跟随者维护一个下一个索引，这是领导者将发送给跟随者的下一个日志条目的索引。当一个领导者第一次掌权时，它将所有的下一个索引值初始化为其日志中最后一个值之后的索引值(图 7 中的 11)。如果跟随者的日志与领导者的日志不一致，则在下一个追加条目 RPC 中，追加条目一致性检查将失败。拒绝后，领导者递减下一个索引，并重试追加条目 RPC。最终，下一个索引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 将会成功，这将从跟随者的日志中删除任何冲突的条目，并从领导者的日志中追加条目(如果有)。一旦 AppendEntries 成功，跟随者的日志和领导者的日志是一致的，并且在剩余的任期内将保持这种状态。

如果需要，可以优化协议以减少被拒绝的附加条目 RPC 的数量。例如，当拒绝一个 AppendEntries 请

求时，跟随者可以包含冲突条目的术语以及它为该术语存储的第一个索引。有了这个信息，领导者可以减少下一个索引，以绕过该术语中所有冲突的条目；对于每个有冲突条目的术语，将需要一个附加条目 RPC，而不是每个条目一个 RPC。在实践中，我们怀疑这种优化是必要的，因为故障很少发生，也不太可能有许多不一致的条目。

通过这种机制，领导者在掌权时不需要采取任何特殊的行动来恢复日志的一致性。它只是开始正常操作，日志自动收敛以响应 AppendEntries 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 中的领导者仅附加属性)。

这种日志复制机制展示了第 2 节中描述的合意的一致属性:只要大多数服务器运行，Raft 就可以接受、复制和应用新的日志条目；在正常情况下，一个新条目可以用一轮远程过程调用复制到集群的大部分；单个慢速跟随器不会影响性能。

5.4 安全

前面几节描述了 Raft 如何选举领导者和复制日志条目。然而，到目前为止描述的机制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，跟随者可能不可用，那么它可能被选为领导者，并用新条目覆盖这些条目；因此，不同的状态机可能执行不同的命令序列。

本节通过添加对哪些服务器可以被选为领导者的限制来完成 Raft 算法。该限制确保任何给定术语的前导包含在先前术语中提交的所有条目(图 3 中的前导完整性属性)。给定选举限制，我们就能使承诺的规则更加精确。最后，我们给出了一个引线完整性属性的证明草图，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一致性算法中，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，如视图标记复制[22]，即使领导者最初不包含所有提交的条目，也可以选择领导者。这些算法包含额外的机制来识别丢失的条目，并在选举过程中或之后不久将它们传送给新的领导人。不幸的是，这导致相当大的额外机制和复杂性。Raft 使用了一种更简单的方法，它可以保证所有以前提交的条目

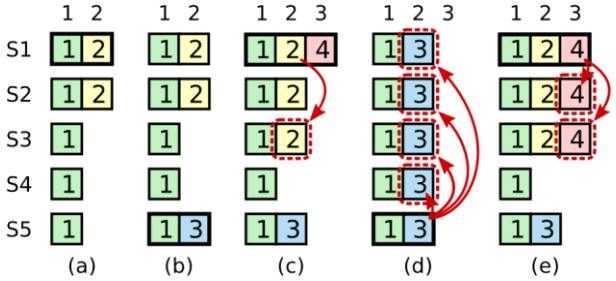

图 8:一个时间序列，显示了为什么领导者不能使用旧术语的日志条目来确定承诺。在(a)中，S1 处于领先地位，并部分复制了索引 2 处的日志条目.1986 年，S1 发生了车祸；S5 凭借 S3、S4 和自己的选票当选为第三任期的领导人，并在日志索引 2 中接受不同的条目.在(c) S5 中，崩溃；S1 重新开始，当选领袖，并继续复制。此时，术语 2 中的日志条目已在大多数服务器上复制，但并未提交。如果 S1 像(d)中那样崩溃，S5 可能会被选为领导人(由 S2、S3 和 S4 投票)，并在第三任期用自己的条目覆盖该条目.但是，如果 S1 在崩溃之前在大多数服务器上复制了当前任期的条目，如(e)中所示，则该条目被提交(S5 不能赢得选举)。此时，日志中所有先前的条目也被提交。

每一位新领导人从当选之时起就有任期，而不需要将这些条目转给领导人。这意味着日志条目只在一个方向上流动，从领导者到追随者，领导者永远不会覆盖他们日志中的现有条目。

Raft 使用投票过程来阻止候选人赢得选举，除非其日志包含所有提交的条目。候选人必须联系集群的大多数成员才能当选，这意味着每个提交的条目必须至少出现在其中一个服务器上。如果候选人的日志至少与大多数其他日志一样是最新的(其中“最新”定义如下)，那么它将保存所有提交的条目。request 投票 RPC 实现了这一限制:该 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新，则投票人拒绝投票。

Raft 通过比较日志中最后一个条目的索引和术语来确定两个日志中哪一个更新。如果日志中的最后一个条目具有不同的术语，则具有较晚术语的日志会更新。如果日志以相同的术语结尾，那么无论哪个日志更长，都是最新的。

5.4.2 提交以前条款的条目

如第 5.3 节所述，一旦条目存储在大多数服务器上，领导者就知道其当前任期的条目已被提交。如果领导者在提交条目之前崩溃，未来的领导者将尝试完成条目的复制。然而，一旦前一个术语的条目存储在大多数服务器上，领导者就不能立即断定它已被提交。图-

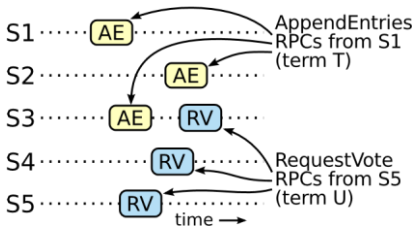

图 9:如果 S1(任期 T 的领导者)从其任期提交了一个新的日志条目，并且 S5 被选为下一个任期 U 的领导者，那么必须至少有一个服务器(S3)接受了该日志条目并投票支持 S5。

ure 8 说明了一种情况，其中旧的日志条目存储在大多数服务器上，但仍可能被未来的领导者覆盖。

为了消除类似图 8 中的问题，Raft 从不通过计算副本来提交以前的日志条目。通过计数副本，仅提交领导者当前任期的日志条目；一旦以这种方式提交了当前术语中的条目，则由于日志匹配属性，所有先前的条目都会被间接提交。在某些情况下，领导者可以有把握地断定提交了一个旧的日志条目(例如，如果该条目存储在每个服务器上)，但是为了简单起见，Raft 采取了一种更保守的方法。

Raft 在承诺规则中引入了这种额外的复杂性，因为当领导者复制以前术语的条目时，日志条目会保留其原始术语号。在其他共识算法中，如果一个新的领导者从以前的“术语”中复制条目，它必须用新的“术语号”来复制。Raft 的方法使得对日志条目进行推理变得更加容易，因为它们在一段时间内和跨日志保持相同的术语号。此外，Raft 中的新领导者从以前的术语发送的日志条目比其他算法中的少(其他算法必须发送冗余的日志条目以对它们重新编号，然后才能提交)。

5.4.3 安全论证

给定完整的Raft算法，我们现在可以更精确地论证 Leader 完备性性质成立(这个论证基于安全证明；见第 9.2 节)。我们假设领导者完备性不成立，那么我们证明一个矛盾。假设期限 T 的领导者(leaderT)提交了其期限的日志条目，但该日志条目不是由某个未来期限的领导者存储的。考虑最小的术语 U > T，其领导者(leaderU)不存储条目。

1. 提交的条目必须在选举时不在领导的日志中(领导从不删除或覆盖条目)。
2. leandt 在群集的大多数成员上复制条目，leandu 从群集的大多数成员那里获得投票。因此，至少有一个服务器(“投票者”)既接受了来自领导的条目，又投票给了领导，如图 9 所示.选民是达成矛盾的关键。
3. 投票人在投票给领导之前，必须已经接受了领导提交的条目；否则，它将拒绝来自 leaderT 的追加条目请求(它的当前模式将高于 T)。
4. 投票者在投票给领导者时仍然存储条目，因为每个插入的领导者都包含条目(假设)，领导者从不删除条目，追随者只有在与领导者冲突时才会删除条目。
5. 选民将其投票授予了 LeadU，因此 LeadU 的日志必须与选民的日志一样最新。这就导致了两个矛盾之一。
6. 首先，如果投票人和领导共享相同的最后一个日志项，那么领导的日志必须至少与投票人的日志一样长，因此其日志包含投票人日志中的每个条目。这是一个矛盾，因为投票者包含承诺的条目，而 leaderU 被假定不包含。
7. 否则，leaderU 的最后一个日志项一定比投票人的日志项大。此外，它比 T 大，因为投票者的最后一个日志项至少是 T(它包含来自项 T 的提交条目)。创建 leaduu 的最后一个日志条目的较早领导者必须在其日志中包含已提交的条目(根据假设)。然后，通过日志匹配属性，leaderU 的日志还必须包含提交的条目，这是一个矛盾。
8. 这就完成了矛盾。因此，所有大于 T 的术语的前导必须包含在术语 T 中提交的术语 T 的所有条目。
9. 日志匹配属性保证未来的分发者也将包含间接提交的条目，如图 8(d)中的索引 2。

给定 Leader 完整性属性，我们可以从图 3 中证明状态机安全属性，该属性声明如果一个服务器已经在给定的索引处将一个日志条目应用到其状态机，则没有其他服务器将会为相同的索引应用不同的日志条目。当服务器将日志条目应用到其状态机时，其日志必须通过该条目与领导者的日志相同，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的最低术语；日志完整性属性保证所有更高术语的前导将存储相同的日志条目，因此在后面的术语中应用索引的服务器将应用相同的值。因此，国家机器安全属性成立。

最后，Raft 要求服务器按照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向其状态机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，我们一直关注领导者的失败。跟随者和候选人崩溃比领导者崩溃更容易处理，而且两者

处理方式相同。如果一个跟随者或候选人崩溃，那么以后发送给它的请求投票和追加条目 RPC 将失败。Raft 通过无限期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。Raft RPCs 是幂等的，所以这不会造成伤害。例如，如果一个跟随者收到一个包含日志条目的 AppendEntries 请求，它会忽略新请求中的那些条目。

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:系统不能仅仅因为某些事件发生得比预期的更快或更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)必然取决于时间。例如，如果消息交换比服务器崩溃之间的典型时间更长，候选人就不会熬夜到足以赢得选举；没有一个稳定的领导者，Raft 无法取得进步。

领导人选举是 Raft 最关键的方面。只要系统满足以下时间要求，Raft 将能够选出并保持稳定的领导者:

$$broadcast\ time \ll electron\ time\ out \ll mtbf$$

在这个不等式中，broadcastTime 是服务器并行发送 RPC 到集群中的每台服务器并接收它们的响应所花费的平均时间；选举超时是指第 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时间少一个数量级，以便领导人能够可靠地发送阻止追随者开始选举所需的心跳消息；鉴于用于选举暂停的随机方法，这种不平等也使得分裂投票不太可能发生。选举超时应该比 MTBF 大几个数量级，这样系统才能稳步前进。当领导崩溃时，系统将在大约选举超时时间内不可用；我们希望这只是整个时间的一小部分。

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 通常要求接收方将信息保存到稳定的存储中，因此广播时间可能在 0.5 毫秒到 20 毫秒之间，具体取决于存储技术。因此，选举超时可能在 10 毫秒到 500 毫秒之间。典型的

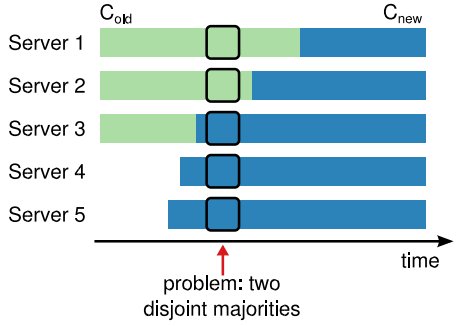

图 10:直接从一种配置切换到另一种配置是不安全的，因为不同的服务器会在不同的时间切换。在这个例子中，集群从三个服务器增加到五个。不幸的是，有一个时间点，两个不同的领导人可以当选为同一任期，一个拥有旧配置的大多数(冷)，另一个拥有新配置的大多数(Cnew)。

服务器 MTBFs 是几个月或更长，这很容易满足时间要求。

6 群集成员身份更改

到目前为止，我们假设集群配置(参与一致算法的服务器集)是固定的。实际上，有时需要更改配置，例如在服务器出现故障时更换服务器，或者更改复制程度。虽然这可以通过使整个集群脱机、更新配置文件，然后重新启动集群来完成，但这将使集群在转换期间不可用。此外，如果有任何手动步骤，它们会有操作员出错的风险。为了避免这些问题，我们决定自动化配置更改，并将它们合并到 Raft 共识算法中。

为了确保配置更改机制的安全性，在过渡期间，不能有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。不可能一次自动切换所有服务器，因此在转换过程中，集群可能会分裂成两个独立的主体(参见图 10)。

为了确保安全，配置更改必须使用两阶段方法。有多种方法可以实现这两个阶段。例如，一些系统(例如，。[22])使用第一阶段禁用旧配置，使其无法处理客户端请求；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一致的过渡配置；一旦达成一致意见，系统就会转换到新的配置。联合共识结合了新旧配置:

- 在这两种配置中，日志条目都被复制到所有服务器。

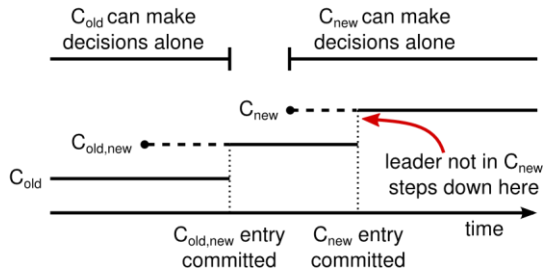

图 11:配置变更的时间线。虚线显示已创建但未提交的配置条目，实线显示最新提交的配置条目。领导者首先在其日志中创建冷的新配置条目，并将其提交给冷的新(冷的大部分和新的大部分)。然后，它创建 Cnew 条目，并将其提交给 Cnew 的大多数成员。没有一个时间点是 Cold 和 Cnew 都可以独立做决定的。

- 任何一种配置的服务器都可以作为领导者。
- 协议(选举和参赛承诺)要求新旧组合都有单独的多数。

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转换，而不会影响安全性。此外，联合共识允许集群在整个配置更改过程中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。当领导收到将配置从冷更改为新的请求时，它会将联合共识(冷，图中为新)的配置存储为日志条目，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其日志中，它就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，而不管该条目是否已提交)。这意味着领导者将使用冷、新的规则来确定冷、新的日志条目何时被提交。如果领导崩溃，根据获胜候选人是否获得，可以在冷或冷，新下选择新领导

冷，新。无论如何，Cnew 在此期间不能单方面决策。

一旦提交了冷的新日志条目，新的冷的新日志条目可以在不经过另一个的批准的情况下做出决定，并且领导者完整性属性确保只有具有冷的新日志条目的服务器可以被选为领导者。现在，领导者可以安全地创建描述 Cnew 的日志条目，并将其复制到集群中。同样，此配置将在每台服务器上看到后立即生效。当新的配置已经在 Cnew 的规则下提交时，旧的配置是不相关的，并且不在新配置中的服务器可以被关闭。如图 11 所示，不存在 Cold 和 Cnew 都可以单方面决策的时候；这保证了安全。

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。如果以这种状态将它们添加到集群中，它们可能需要相当长的时间才能赶上，在此期间可能无法提交新的日志条目。为了避免可用性差距，Raft 在配置更改之前引入了一个额外的阶段，在这个阶段中，新服务器作为非投票成员加入集群(领导者向它们复制日志条目，但它们不被视为多数)。一旦新服务器赶上了集群的其余部分，重新配置就可以按照上述方式进行。

第二个问题是集群领导可能不是新配置的一部分。在这种情况下，一旦提交了 Cnew 日志条目，领导者就下台(返回到追随者状态)。这意味着当领导者管理一个不包括其自身的集群时，将会有一段时间(当它正在提交新的集群时)；它复制日志条目，但不计算自己的多数。当 Cnew 被提交时会发生引线转换，因为这是新配置可以独立运行的第一点(总是可以从 Cnew 中选择引线)。在这一点之前，可能只有来自 Cold 的服务器才能当选为领导者。

第三个问题是删除的服务器(不在 *Cnew)会破坏集群*。这些服务器将不会接收心跳，因此它们将超时并开始新的选举。然后，他们将发送带有新术语号的请求投票 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，并且该过程将重复，导致可用性差。

为了防止这个问题，当服务器认为当前的领导者存在时，它们会忽略请求投票。具体来说，如果服务器在听到当前领导者的最小选举超时内收到请求投票 RPC，它不会更新其任期或授予其投票。这不影响正常选举，在正常选举中，每台服务器在开始选举之前至少要等待一段最短的选举超时时间。然而，它有助于避免被移除的服务器造成的中断:如果一个领导者能够获得其集群的心跳，那么它将不会被更大的术语数取代。

七　原木压实

Raft 的日志在正常运行期间增长，以包含更多的客户端请求，但在实际系统中，它不能无限制地增长。随着日志变得越来越长，它占用的空间越来越大，重放时间也越来越长。如果没有某种机制来丢弃日志中积累的过时信息，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，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

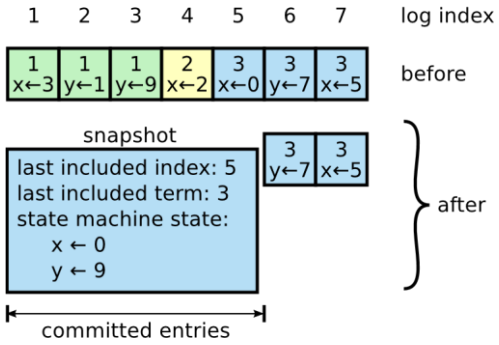

图 12:服务器用一个新的快照替换其日志中提交的条目(索引 1 到 5)，该快照只存储当前状态(在这个例子中是变量 x 和 y)。快照最后包含的索引和术语用于将快照定位在日志条目 6 之前。

这一点被放弃了。在《查比》和《动物园管理员》中使用了快照，本节的其余部分描述了《筏》中的快照。

增量压缩方法，如日志清理[36]和日志结构合并树[30，5]，也是可能的。它们同时对一小部分数据进

行操作，因此随着时间的推移，它们会更均匀地分布压缩负载。他们首先选择一个已经积累了许多已删除和已覆盖对象的数据区域，然后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，这需要额外的机制和复杂性，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来实现 LSM 树。

图 12 显示了 Raft 中快照的基本思想。 每台服务器独立拍摄快照，只覆盖其日志中提交的条目。大部分工作包括状态机将其当前状态写入快照。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)，最后包含的术语是这个条目的术语。保留这些条目是为了支持快照后第一个日志条目的AppendEntries 一致性检查，因为该条目需要以前的日志索引和术语。要启用群集成员身份更改(第 6 节)，快照还会在日志中包括截至上次包含索引的最新配置。一旦服务器完成了快照的写入，它可能会删除最后一次包含的索引中的所有日志条目，以及任何以前的快照。

虽然服务器通常独立拍摄快照，但领导者必须偶尔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追随者的下一个日志条目时，就会发生这种情况。幸运的是，这种情况在正常操作中不太可能发生:一个跟随者跟上了

InstallSnapshot RPC

1. 由领导者调用，将快照的大块发送给追随者。领导总是按顺序发大块。
2. 参数:
3. 学期　　领导任期
4. leaderId 所以跟随者可以重定向客户
5. 最后包含索引快照将替换该索引中包含的所有条目
6. lastIncludedTerm　　lastIncludedIndex 的术语
7. 抵消　　块在
8. 快照文件
9. 数据[]　快照区块的原始字节，从
10. 抵消
11. 完成的　如果这是最后一块，则为 true
12. 结果:期限　　当前模式，供领导自行更新
13. 接收器实现:
14. 如果术语<当前模式，请立即回复
15. 如果第一个区块(偏移量为 0)，则创建新的快照文件
16. 将数据写入给定偏移量处的快照文件
17. 如果完成为假，则回复并等待更多数据块
18. 保存快照文件，放弃索引较小的任何现有快照或部分快照
19. 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，请保留其后面的日志条目并回复
20. 丢弃整个日志
21. 使用快照内容重置状态机(并加载快照的群集配置)

图 13:安装快照 RPC 的概要。快照被分割成块进行传输；这给了跟随者一个生命迹象，所以它可以重置它的选举计时器。

leader 应该已经有这个条目了。然而，特别慢的跟随者或加入集群的新服务器(第 6 节)不会。让这样的跟随者更新的方法是让领导者通过网络给它发送一个快照。

领导用一个新的叫做 InstallSnapshot 的 RPC，把快照发给落后太多的跟随者；参见图 13。当一个跟随者收到带有这个 RPC 的快照时，它必须决定如何处理它现有的日志条目。通常，快照将包含收件人日志中尚未包含的新信息。在这种情况下，跟随者丢弃它的整个日志；它全部被快照取代，并且可能具有与快照冲突的未提交条目。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)，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。

这种抓拍方法背离了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。但是，我们认为这种背离是有道理的。虽然有一个领导者有助于在达成共识时避免决策冲突，但在拍摄快照时已经达成共识，因此没有决策冲突。数据仍然只从领导者流向追随者，只是追随者现在可以重组他们的数据。

我们考虑了另一种基于领导者的方法，其中只有领导者会创建一个快照，然后它会将此快照发送给它的每个追随者。然而，这有两个缺点。首先，向每个跟随者发送快照会浪费网络带宽并减慢快照过程。每个跟随者都已经有了生成自己的快照所需的信息，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要便宜得多。其次，领导者的实施会更加复杂。例如，领导者需要向追随者发送快照，同时向他们复制新的日志条目，以便不阻止新的客户端请求。

还有两个问题会影响快照性能。首先，服务器必须决定何时拍摄快照。如果服务器快照太频繁，就会浪费磁盘带宽和能量；如果它的快照频率太低，就有耗尽其存储容量的风险，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的字节大小时拍摄快照。如果此大小设置为明显大于快照的预期大小，则快照的磁盘带宽开销将会很小。

第二个性能问题是写快照会花费大量时间，我们不希望这会延迟正常操作。解决方案是使用写入时拷贝技术，以便可以接受新的更新，而不会影响正在写入的快照。例如，具有功能数据结构的状态机自然支持这一点。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 的客户将他们的所有请求发送给领导。当客户端第一次启动时，它连接到一个随机连接的服务器。如果客户的第一选择不是领导者，服务器将拒绝客户的请求，并提供其最近听到的领导者的信息(追加条目请求包括领导者的网络地址)。如果领导崩溃，客户请求将超时；客户端然后用随机选择的服务器再试一次。

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个时刻看起来是瞬间执行的，恰好一次)。 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，如果领导者在提交日志条目之后但在响应客户端之前崩溃，客户端将使用新的领导者重试该命令，导致它被第二次执行。解决方案是让客户端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机处理的最新序列号，以及相关的响应。如果它收到一个序列号已经被执行的命令，它会立即响应，而不重新执行请求。

只读操作可以在不将任何内容写入日志的情况下处理。但是，在没有额外措施的情况下，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其不知道的新领导者所取代。可线性化的读取不能返回过时的数据，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，领导者必须拥有提交条目的最新信息。领导者完整性属性保证领导者拥有所有提交的条目，但是在其任期开始时，它可能不知道哪些是提交的条目。要找到答案，它需要从它的术语中提交一个条目。Raft 通过让每个领导者在任期开始时在日志中提交一个空白的禁止操作条目来处理这个问题。其次，领导者必须在处理只读请求之前检查其是否已被废黜(如果最近的领导者当选，其信息可能会过时)。Raft 通过让领导者在响应只读请求之前与集群的大多数成员交换心跳消息来处理这个问题。或者，领导者可以依靠心跳机制来提供一种租赁形式[9]，但这将依赖于安全的定时(它假设有限的时钟偏差)。

9 实施和评价

我们已经将Raft实现为复制状态机的一部分，该复制状态机存储 RAMCloud 的配置信息[33]，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、注释或空白行。源代码是免费提供的[23]。根据本文的草稿，Raft 在不同的开发阶段还有大约 25 个独立的第三方开源实现[34]。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来评估 Raft:可理解性、正确性和性能。

9.1 易懂

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使用上层本科生和研究生进行了一项实验研究。我们录制了 Raft 和另一个 Paxos 的视频讲座，并制作了相应的小测验。筏式讲座涵盖了本文的内容，但原木压实除外；帕克斯

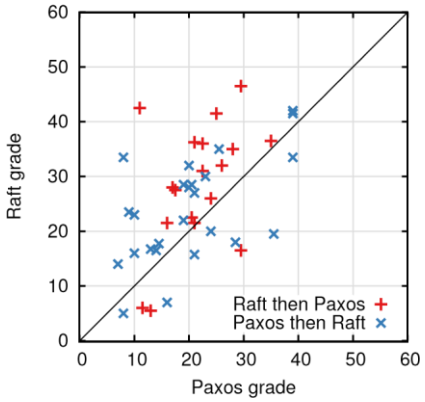

图 14:散点图，比较了 43 名参与者在 Raft 和 Paxos 测验中的表现。对角线上方的点(33)代表 Raft 得分较高的参与者。

讲座涵盖了足够多的材料来创建一个等效的复制状态机，包括单指令 Paxos、多指令 Paxos、重新配置和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，还要求学生对角落案例进行推理。每个学生看一个视频，做相应的小考，看第二个视频，做第二个小考。大约一半的参与者先做了 Paxos 部分，另一半先做了 Raft 部分，以说明从第一部分研究中获得的个人表现和经验的差异。我们比较了参与者在每次测验中的分数，以确定参与者是否对 Raft 有更好的理解。

我们试图尽可能公平地比较帕克斯和 Raft。这项实验在两个方面对 Paxos 有利:43 名参与者中有 15 人报告说以前有过使用 Paxos 的经历，Paxos 视频比 Raft 视频长 14%。如表 1 所示，我们已经采取措施减少潜在的偏差来源。我们所有的资料都可供查阅[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，Raft 分数的真实分布比 Paxos 分数的真实分布平均大至少 2.5 分，置信度为 95%。

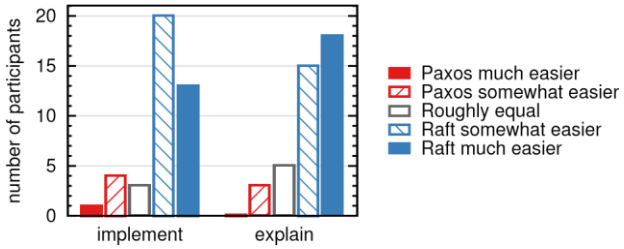

图 15:使用 5 分制，参与者被问及(左)他们认为哪种算法在一个有效、正确和高效的系统中更容易实现，以及(右)哪种算法更容易向计算机科学研究生解释。

我们还创建了一个线性回归模型，根据三个因素预测新学生的测验分数:他们参加的测验、他们以前的 Paxos 体验程度以及他们学习算法的顺序。该模型预测，测验的选择会产生 12.5 分的差异，有利于 Raft。这明显高于观察到的 4.9 分的差异，因为许多实际的学生都有 Paxos 的经验，这对 Paxos 有很大帮助，而对 Raft 的帮助稍小。奇怪的是，该模型还预测，已经参加过 Paxos 测验的人在 Raft 上的分数会低 6.3 分；虽然我们不知道为什么，但这似乎在统计上是有意义的。

我们还在测验后调查了参与者，看看他们认为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 更容易实施和解释(每个问题 41 个中有 33 个)。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，参与者可能被我们的假设 Raft 更容易理解的知识所偏见。

关于 Raft 用户研究的详细讨论见[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性证明。形式规范[31]使用 TLA+规范语言[17]使图 2 中总结的信息完全精确。它大约有 400 行长，是证明的主题。对于任何实现 Raft 的人来说，它本身也很有用。我们已经用 TLA 证明系统机械地证明了对数完备性[7]。然而，这种证明依赖于未经机械检查的不变量(例如，我们没有证明规范的类型安全性)。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(仅依赖于规范)并且与

涉及	Steps taken to mitigate bias	Materials for review [28, 31]
Equal lecture quality	Same lecturer for both. Paxos lecture based on and improved from existing materials used in several universities. Paxos lecture is 14% longer.	videos
Equal quiz difficulty	Questions grouped in difficulty and paired across exams.	quizzes
Fair grading	Used rubric. Graded in random order, alternating between quizzes.	rubric

Table 1: Concerns of possible bias against Paxos in the study, steps taken to counter each, and additional materials available.

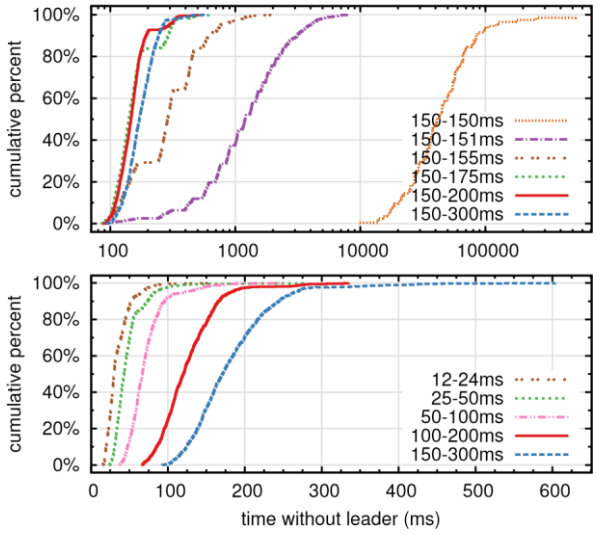

图 16:检测和替换崩溃的领导者的时间。上图显示了选举超时的随机性，下图显示了最小选举超时。每一行代表 1000 次试验(除了“150–150 毫秒”的 100 次试验)，并对应于选举超时的特定选择；例如,“150–155 毫秒”意味着选举超时是在 150 毫秒和 155 毫秒之间随机均匀选择的。测量是在一个由五台服务器组成的集群上进行的，广播时间约为 15 毫秒。由 9 台服务器组成的集群的结果相似。相当精确(大约 3500 字长)。

9.3 表演

Raft 的性能类似于 Paxos 等其他共识算法。对于性能而言，最重要的情况是当一个既定的领导者复制新的日志条目时。Raft 使用最少数量的消息(从领导者到一半集群的一次往返)来实现这一点。也有可能进一步提高 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。文献中已经为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 实现来衡量 Raft 的领导者选举算法的性能，并回答了两个问题。第一，选举过程收敛快吗？第二，领导者崩溃后能达到的最小停机时间是多少？

为了衡量领导者的选举，我们反复崩溃了一个由五个服务器组成的集群的领导者，并计算了检测崩溃和选举一个新领导者所花费的时间(见图 16)。为了产生最坏的情况，每个试验中的服务器具有不同的日志长度，因此一些候选人没有资格成为领导者。此外，为了鼓励分裂投票，我们的测试脚本在终止其进程之前触发了来自领导者的心跳 RPC 的同步广播(这近似于领导者在崩溃之前复制新日志条目的行为)。领导者在其心跳间隔内被均匀随机地崩溃，这是所有测试的最小选举超时的一半。因此，最小可能的停机时间大约是最小选举超时时间的一半。

图 16 中的上图显示，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，由于许多分裂的选票，在我们的测试中，领导人选举持续花费了 10 秒钟以上。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:50 毫秒的随机性，最坏情况下的完成时间(超过 1000 次试验)是 513 毫秒。

图 16 中的下图显示了通过减少选举超时时间可以减少停机时间。选举超时为 12–24 毫秒，平均只需 35 毫秒就能选出一名领导人(最长的审判需要 152 毫秒)。然而，将超时时间降低到超过这一点违反了 Raft 的时间要求:在其他服务器开始新的选举之前，领导者很难广播心跳。这可能导致不必要的领导变动和整体系统可用性降低。我们建议使用保守的选举超时，如 150–300 毫秒；这种超时不太可能导致不必要的前导改变，并且仍然会提供良好的可用性。

10 相关著作

有许多与共识算法相关的出版物，其中许多属于以下类别之一:

- 兰波特对帕克斯的最初描述[15]，并试图更清楚地解释它[16，20，21]。
- Paxos 的细化，填补了缺失的细节，修改了算法，为实现提供了更好的基础[26，39，13]。
- 实现一致性算法的系统，如查比[2，4]，动物园管理员[11，12]和扳手[6]。查比和扳手的算法尚未详细公布，尽管两者都声称基于 Paxos。ZooKeeper 的算法已经发布的比较详细了，但是和 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 的观点复制(VR)，一种替代共识的方法，与 Paxos 同时发展。最初的描述[29]与分布式事务的协议交织在一起，但核心的一致协议在最近的更新[22]中被分开了。虚拟现实使用一种基于领导的方法，与 Raft 有许多相似之处。

Raft 和 Paxos 最大的区别在于 Raft 的强大领导力:Raft 将领导人选举作为共识协议的重要组成部分，它尽可能将功能集中在领导人身上。这种方法导致更简单的算法，更容易理解。例如，在 Paxos 中，领导者选举与基本共识协议正交:它仅用作性能优化，不需要用于实现共识。然而，这导致了额外的机制:Paxos 包括基本共识的两阶段协议和单独的领导人选举机制。相比之下，Raft 将领导人选举直接纳入共识算法，并将其用作共识两个阶段的第一个阶段。这导致比 Paxos 更少的机制。

像 Raft 一样，VR 和 ZooKeeper 是基于领导者的，因此与 Paxos 相比，Raft 有许多优势。然而，Raft 的机制不如 VR 或 ZooKeeper，因为它最大限度地减少了非领导者的功能。例如，Raft 中的日志条目仅在一个方向上流动:从追加条目 RPC 中的前导向外。VR 中日志条目双向流动(选举过程中领导可以接收日志条目)；这导致了额外的机制和复杂性。发布的关于动物园管理员的描述也将日志条目传递给领导者和从领导者那里传递，但是实现显然更像 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(两个 RPC 请求及其响应)。Raft 的消息比其他算法的消息要密集一些，但总体来说更简单。此外，VR 和 ZooKeeper 是按照领导者变更时传输整个日志来描述的；需要额外的消息类型来优化这些机制，使它们切实可行。

Raft 强有力的领导方法简化了算法，但排除了一些性能优化。例如，平等派在某些情况下可以通过无领导的方式获得更高的绩效[27]。EPaxos 利用状态机命令的交换性。任何服务器都可以只通过一轮通信提交一个命令，只要同时提出的其他命令与之交换即可。然而，如果同时提出的命令不能相互交换，电子考绩制度需要额外的一轮沟通。因为任何服务器都可以提交命令，所以 EPaxos 可以很好地平衡服务器之间的负载，并且能够在广域网设置中实现比 Raft 更低的延迟。然而，它给 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更方法，包括兰波特的原始提议[15]、虚拟现实[22]和智能技术[24]。我们为 Raft 选择了联合协商一致的方法，因为它利用了协商一致协议的其余部分，因此几乎不需要额外的机制来改变成员。兰波特基于 α 的方法不是 Raft 的选择，因为它假设在没有领导者的情况下可以达成共识。与 VR 和 SMART 相比，Raft 的重新配置算法的优势在于，可以在不限制正常请求处理的情况下发生成员资格变化；相比之下，VR 会在配置更改期间停止所有正常处理，而 SMART 会对未完成请求的数量施加类似 α 的限制。Raft 的方法也比 VR 或 SMART 增加了更少的机制。

11 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也同样重要。除非开发人员将算法转化为实际的实现，否则其他目标都无法实现，这将不可避免地偏离并扩展到已发布的形式。除非开发人员对算法有深刻的理解，并能对其产生直觉，否则他们很难在实现中保留其理想的属性。

在本文中，我们讨论了分布式一致性问题，其中一个被广泛接受但不可理解的算法 Paxos 多年来一直在挑战学生和开发人员。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。我们还认为 Raft 为系统构建提供了更好的基础。以可理解性为主要设计目标改变了我们对筏形基础的设计方式；随着设计的进展，我们发现自己在重复使用一些技术，比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，也使我们更容易相信它的正确性。

12 感谢

如果没有阿里·古兹、大卫·马齐尔斯以及伯克利的 CS 294-91 和斯坦福的 CS 240 的学生的支持，这项用户研究是不可能的。斯科特·克莱默帮助我们设计了用户研究，尼尔森·雷为我们提供了统计分析方面的建议。用于用户研究的 Paxos 幻灯片大量借用了洛伦佐·阿尔维斯最初创建的幻灯片。特别感谢大卫·马泽尔和以斯拉·霍奇在筏子上发现了微妙的漏洞。许多人对论文和用户研究材料提供了有益的反馈，包括埃德·布格尼翁、陈宸、胡格埃夫拉尔德、丹尼尔·贾诩、阿尔琼·戈帕兰、乔恩·豪厄尔、维马尔库马尔·耶亚库马尔、安基塔·凯杰瓦尔、亚历山大·克拉村、阿米特·列维、乔尔·马丁、松下幸之助、奥列格·比索克、大卫·拉莫斯、罗布特·范·雷尼塞、孟德尔·罗森布鲁姆、尼古拉斯·希佩尔、戴安·斯特凡、安德鲁·斯通、瑞安·斯图茨曼、大卫·特雷、斯蒂芬·杨、马泰·扎哈里亚、24 名匿名。沃纳·威格尔在推特上发布了一个早期草稿的链接，这给了 raft 很大的曝光度。这项工作得到了 Gigascale 系统研究中心和多尺度系统中心的支持，这两个中心是由半导体研究公司项目焦点中心研究计划资助的六个研究中心中的两个，由半导体研究公司项目 STARnet 资助，该项目由 MARCO 和 DARPA 赞助，由国家科学基金会资助，赠款号为。0963859，以及脸书、谷歌、梅兰诺、日本电气、NetApp、思爱普和三星的资助。迭戈·翁加罗得到了荣格利公司斯坦福研究生奖学金的支持。

参考

[1] BOLOSKY。J.,。布拉德肖，。HAAGENS。B.,。KUSTERS。页（page 的缩写）,。还有李，p。Paxos 复制状态机作为高性能数据存储的基础。在 Proc 中。*NSDI'11，USENIX 网络系统设计和实施会议(2011)，USENIX，页*。141–154.

[2] BURROWS。松耦合分布式系统的胖锁服务。在 Proc 中。*OSDI 06，操作系统设计和实现研讨会(2006)，USENIX，页*。335–350.

[3] CAMARGOS，l . J.,。SCHMIDT。米（meter 的缩写））,。还有 PEDONE，f。多坐标 Paxos。在 Proc 中。*PODC'07，美国计算机学会分布式计算原理研讨会(2007)，美国计算机学会，页*。316–317.

[4] CHANDRA。D.,。GRIESEMER，r . 还有红石，j。工程观点。在 Proc 中。*PODC'07，美国计算机学会分布式计算原理研讨会(2007)，美国计算机学会，页*。398–407.

[5] CHANG。DEAN，j . GHEMAWAT，s,。谢长廷，w . C.,。WALLACH。A.,。BURROWS,。钱德拉,。FIKES，a。和格鲁伯。E. Bigtable:结构化数据的分布式存储系统。在 Proc 中。*OSDI'06，USENIX 操作系统设计和实现研讨会(2006)，USENIX，页*。205–218.

[6] CORBETT。C.,。DEAN，j . EPSTEIN，m . FIKES，a,。FROST,。FURMAN，j . J.,。GHEMAWAT，s,。GUBAREV，a,。HEISER,。HOCHSCHILD，p . 谢长廷,。KANTHAK，s . KOGAN，e,。李,。LLOYD，a . 梅尔尼克，美国,。MWAURA，d . NAGLE，d . 昆兰，美国,。RAO，r,。ROLIG，l . SAITO，y,。SZYMANIAK，m . TAYLOR，c .
王，右。伍德福德市。扳手:谷歌的全球分布式数据库。在 Proc 中。*OSDI'12，USENIX 操作系统设计和实施会议(2012)，USENIX，页*。251–264.

[7] COUSINEAU，d 。 DOLIGEZ，。洛杉矶兰波特，。MERZ，
南。RICKETTS，。还有范泽托，h。TLA+校样。在 Proc 中 。 *第 12 届形式方法研讨会(2012)，d*。Giannakopoulou 和 d . 对不起，Eds。vol . 《计算机科学讲义》第 7436 页，斯普林格出版社，第 100 页。147–154.

[8] GHEMAWAT，s 。 GOBIOFF，h，。还有梁，s。-T。Google 文件系统。在 Proc 中。*SOSP'03，美国计算机学会操作系统原理研讨会(2003)，美国计算机学会，页*。29–43.

[9] 灰色，c。还有切丽顿，d。租约:分布式文件缓存一致性的有效容错机制。《第 12 届美国计算机学会操作系统原理研讨会论文集》(1989)，页。202–210.

[10] HERLIHY，m . 页（page 的缩写）,。还有 WING，j。米（meter 的缩写））。可线性化:并发对象的正确性条件。*美国计算机学会程序设计语言和系统学报 12(1990 年 7 月)，463–492*。

[11] HUNT。KONAR，。JUNQUEIRA，f . 页（page 的缩写）,。里德，
B. 动物园管理员:互联网规模系统的无等待协调。USENIX 年度技术会议(2010)，USENIX，第 10 页。145–158.

[12] JUNQUEIRA，f . 页（page 的缩写）,。REED。C.,。和塞拉菲尼。
主要备份系统的高性能广播。在 Proc 中。*DSN'11，IEEE/IFIP 国际会议。《可靠的系统和网络》(2011 年)，美国电气和电子工程师协会计算机学会，页*。245–256.

[13] KIRSCH。还有阿米尔，y。系统构建者的 Paxos。技术。有代表性的。CNDS-2008-2，约翰霍普金斯大学，2008。

[14] LAMPORT，l . 分布式系统中的时间、时钟和事件顺序。*美国计算机学会通讯 21，7(1978 年 7 月)，558–565*。

[15] LAMPORT，l . 兼职议会。*美国计算机学会计算机系统学报 16，2(1998 年 5 月)，133–169*。

[16] LAMPORT，l . 帕克斯做得很简单。*ACM SIGACT 新闻 32，4(12 月*。2001), 18–25.

[17] LAMPORT，l . *为硬件和软件工程师指定系统、TLA+ 语言和工具*。爱迪生韦斯利，2002。

[18] LAMPORT，l . 广义共识和帕克斯。技术。有代表性的。MSR-TR-2005-33，微软研究，2005。

[19] LAMPORT，l . 快速派克斯。*分布式计算 19，2 (2006)，79–103*。

[20] LAMPSON，b . W. 如何利用共识构建高可用系统？在分布式算法中。巴巴格鲁和 k。Marzullo，Eds。Springer-Verlag，1996，pp . 1–17.

[21] LAMPSON，b . W. 帕克斯的 ABCD。在 Proc 中。*PODC 01，美国计算机学会分布式计算原理研讨会(2001)，美国计算机学会，页*。13–13.

[22] LISKOV，b 。 和考林。重温视图标记复制。技术。有代表性的。麻省理工学院，2012 年 7 月。

[23] 木屋 来源 代码。`http://github.com/`小木屋。

[24] LORCH，j . R.,。ADYA，a，。BOLOSKY。J.,。CHAIKEN,。DOUCEUR，j . R.,。豪威尔，j。迁移复制的有状态服务的 SMART 方式。在 Proc 中。*EuroSys'06，ACM SIGOPS/EuroSys 欧洲计算机系统会议(2006)，ACM，pp* . 103–115.

[25] 毛，y。JUNQUEIRA，f . 页（page 的缩写）,。还有马祖罗，k。
孟子:建立高效的复制型国家机器广域网。在 Proc 中。*OSDI 08，USENIX 操作系统设计和实现会议(2008)，USENIX，页。*369–384.

[26] MAZIERES `，d . Paxos 使实用。http://www . SCS . Stanford . edu/ḌM/home/papers/Pax OS . pdf，Jan . 2007.

[27] MORARU，我。ANDERSEN。G.,。还有卡明斯基，m。在平等主义的议会中有更多的共识。在 Proc 中。*SOSP'13，美国计算机学会操作系统原理研讨会(2013)，美国计算机学会。*

[28] Raft 用户研究。http://ramcloud.stanford。edu/翁加罗/用户研究/。

[29] OKI。米（meter 的缩写））,。还有利科夫，b。H. 视图标记复制:一种支持高可用性分布式系统的新的主要复制方法。在 Proc 中。*PODC'88，美国计算机学会分布式计算原理研讨会(1988)，美国计算机学会，页。*8–17.

[30] 奥尼尔，p 。 程，e，。GAWLICK，d，。还有 ONEIL，e。日志结构的合并树(LSM 树)。*《信息学报》33，4 (1996)，351–385。*

[31] ONGARO，d . *共识:理论与实践的桥梁。*斯坦福大学博士论文，2014(工作在进行中)。http://ramcloud.stanford.edu/~ongaro/ · thesis.pdf。

[32] ONGARO，d 。 和驱逐出去。寻找一个可以理解的共识算法。在 USENIX 举行的 USENIX 年度技术会议(2014)上。

[33] 驱逐出境，j 。 AGRAWAL，p . 埃里克森，。KOZYRAKIS，c . LEVERICH，j . MAZIERES `，d，。MITRA，s . NARAYANAN，a . 翁加罗，d . PARULKAR，g，。ROSENBLUM，。RUMBLE，s。米（meter 的缩写））,。STRATMANN，
E。 和斯图斯曼，r。RAMCloud 的案例。*ACM 54 的通信(2011 年 7 月)，121–130。*

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

[35] REED。个人通讯，2013 年 5 月 17 日。

[36] ROSENBLUM。和驱逐出去。K. 日志结构文件系统的设计与实现。*ACM Trans。电脑。Syst。10(1992 年 2 月)，26-52。*

[37] SCHNEIDER。B. 使用状态机方法实现容错服务:教程。*美国计算机学会计算调查 22，4(12 月。*1990), 299–319.

[38] ,, KUANG, H.匡， H.,和
CHANSLER。Hadoop 分布式文件系统。在 Proc 中。*MSST'10，大容量存储系统和技术研讨会(2010)，电气和电子工程师协会计算机学会，页。*1–10.

[39] VAN RENESSE。帕克斯做得适度复杂。
技术。代表，。康奈尔大学，2012。