寻找一个可理解的共识算法（扩展版本）

Diego Ongaro 和 John Ousterhout
斯坦福大学

2025 年 10 月 16 日

摘要

Raft 是一种用于管理复制日志的共识算法。它产生的结果等价于（多）Paxos，效率与 Paxos 相当，但其结构不同于 Paxos；这种结构差异使得 Raft 比 Paxos 更容易理解，同时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 将共识的关键要素（如领导者选举、日志复制和安全性）进行分离，并强制更强的一致性，从而减少需要考虑的状态数量。用户研究结果表明，Raft 比 Paxos 更容易被学生学习。Raft 还包含一种用于改变集群成员的新机制，该机制通过重叠多数来保证安全性。

1 引言

共识算法使一组机器能够作为一个整体协同工作，即使其中一些成员发生故障也能继续运行。因此，它们在构建可靠的大规模软件系统中起着关键作用。Paxos [15, 16] 在过去十年中主导了共识算法的讨论：大多数共识实现基于 Paxos 或受其影响，Paxos 已成为教授学生学习共识的主要工具。

然而，尽管已有诸多尝试使其更易于理解，Paxos 仍然非常难以掌握。此外，其架构在支持实际系统时需要复杂的修改。因此，系统构建者和学生都面临巨大的挑战。

在自己长期困扰于 Paxos 之后，我们开始寻找一种新的共识算法，以提供更优越的系统构建和教学基础。我们的方法与众不同，其主要目标是可理解性：我们能否为实际系统定义一种共识算法，并以一种远比 Paxos 更容易学习的方式描述它？此外，我们希望该算法能帮助构建系统构建者所必需的直觉。不仅算法本身必须有效，而且其工作原理必须清晰明了。

这项工作的成果是一种名为 Raft 的共识算法。在设计 Raft 时，我们应用了多种技术以提升可理解性，包括分解（Raft 将领导者选举、日志复制和安全性分离）和

状态空间缩减（相对于 Paxos，Raft 降低了非确定性程度以及服务器之间不一致的可能方式）。一项在两所大学共 43 名学生中进行的用户研究表明，Raft 明显比 Paxos 更容易理解：在学习了两种算法之后，其中有 33 名学生在回答关于 Raft 的问题时表现优于 Paxos 的问题。

Raft 在许多方面与现有共识算法相似（最显著的是 Oki 和 Liskov 的 Viewstamped Replication [29, 22]），但它具有几个新颖的特性：

- 强领导者：Raft 使用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，也使 Raft 更容易理解。

- 领导者选举：Raft 使用随机定时器来选举领导者。这仅在已有共识算法必须实现的心跳机制基础上增加少量机制，同时能快速且简单地解决冲突。

- 成员变更：Raft 改变集群中服务器集合的机制采用了一种新的联合共识方法，在状态转换过程中，两个不同配置的多数相互重叠。这使得集群在配置变更期间能够正常运行。

我们相信，Raft 在教学和实际实现方面均优于 Paxos 及其他共识算法。它比其他算法更简单、更易理解；其描述完整，足以满足实际系统的需求；已有多个开源实现，并被多家公司采用；其安全性属性已形式化指定并得到证明；其效率与其他算法相当。

本文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），阐述了我们关于可理解性的总体方法（第 4 节），介绍了 Raft 共识算法（第

[0]本技术报告是 [32] 的扩展版本；边栏中用灰色条注释了额外内容。发布于 2014 年 5 月 20 日。

5-8 节），评估了 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背景下 [37]。在这种方法中，一组服务器上的状态机计算出相同状态的副本，并且即使部分服务器宕机，仍能继续运行。复制状态机是

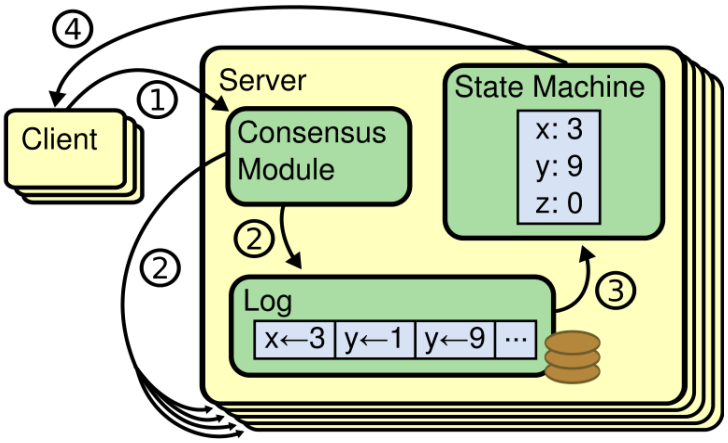

图 1：复制状态机架构。共识算法管理一个包含来自客户端状态机命令的复制日志。各个状态机从日志中处理相同的命令序列，因此产生相同的输出。

用于解决分布式系统中的多种容错问题。例如，具有单一集群领导者的大型系统，如 GFS [8]、HDFS [38] 和 RAMCloud [33]，通常会使用独立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。每个日志包含相同命令的相同顺序，因此每个状态机处理相同的命令序列。由于状态机是确定性的，它们计算出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任务。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通信，以确保每个日志最终都包含相同请求的相同顺序，即使某些服务器发生故障。一旦命令被正确复制，每台服务器的状态机按日志顺序处理这些命令，并将输出返回给客户端。因此，这些服务器看起来像是构成一个单一、高度可靠的状态机。

实际系统中的共识算法通常具有以下特性：

- 在所有非拜占庭条件下（包括网络延迟、分区、数据包丢失、重复和重排序）都能保证安全性（永不返回错误结果）。

- 只要服务器中的多数节点处于运行状态，并且能够相互通信以及与客户端通信，则系统始终可用。因此，一个典型的五台服务器集群可以容忍任意两台服务器

的故障。服务器的故障被假设为停止运行；它们可能从稳定存储中的状态恢复并重新加入集群。

- 不依赖时间同步来保证日志的一致性：即使时钟异常或消息延迟极端，最坏情况下仅会导致可用性问题。

- 在常规情况下，一旦集群中的多数节点对一次远程过程调用（RPC）做出响应，命令即可完成；少数慢速服务器的延迟不会影响整体系统性能。

3 Paxos 有什么问题?

在过去十年中，莱斯利 · 兰波特（Leslie Lamport）提出的 Paxos 协议 [15] 几乎已成为共识协议的代名词：它是课程中最常教授的协议，大多数共识实现都以它为起点。Paxos 首先定义了一种能够就单个决策达成一致的协议，例如单个复制日志条目。我们将这一子集称为单决策 Paxos。Paxos 随后将多个此类协议实例组合起来，以支持一系列决策，例如日志（多实例 Paxos）。Paxos 保证了安全性和活性，并支持集群成员的变化。其正确性已被证明，在正常情况下效率也较高。

然而，Paxos 存在两个显著的缺点。第一个缺点是 Paxos 非常难以理解。完整的解释 [15] 众所周知晦涩难懂；很少有人能真正理解它，且需要付出巨大努力。因此，已有若干尝试用更简单的语言来解释 Paxos [16, 20, 21]。这些解释集中于单决策子集，但仍然具有挑战性。在对 NSDI 2012 会议参会者进行的一项非正式调查中，我们发现很少有人对 Paxos 感到熟悉，即使是在经验丰富的研究人员中。我们自己也难以理解 Paxos；直到阅读了多个简化版本的解释并设计出自己的替代协议后，才最终理解了完整的协议，这一过程耗时接近一年。

我们推测，Paxos 的晦涩源于其以单决策子集作为基础。单决策 Paxos 密集且微妙：它分为两个阶段，这两个阶段缺乏简单的直观解释，且无法独立理解。因此，很难建立对单决策协议为何有效这一问题的直觉。多实例 Paxos 的组合规则增加了显著的额外复杂性和微妙性。我们相信，关于如何达成多个决策（即日志而非单个条目）的共识这一整体问题，可以通过其他更直接且明显的方式进行分解。

Paxos 的第二个问题是，它并不为构建实用的实现提供良好的基础。其中一个原因是，目前尚无广泛公认的多实例 Paxos 算法。兰波特的描述主要集中在单决策 Paxos 上；他仅简要提到了多实例 Paxos 的可能方法，但许多细节缺失。已有若干尝试对 Paxos 进行完善和优化，例如 [26]、[39] 和 [13]，但这些方案之间存在差异彼此之间以及与 Lamport 的草图之间存在差异。一些系统（如 Chubby [4]）实现了类似 Paxos 的算法，但在大多数情况下，它们的详细信息并未公开。

此外，Paxos 架构对于构建实际系统而言是不佳的；这是单决议分解的另一个后果。例如，独立选择一组日志条目，然后将它们合并成一个顺序日志，几乎没有实际好处；这只会增加复杂性。更简单且更高效的方法是围绕日志来设计系统，其中新条目以受约束的顺序逐个追加。另一个问题是，Paxos 在其核心采用了对称的

点对点方法（尽管最终建议采用一种弱形式的领导者作为性能优化）。这种设计在只有单一决策的简化世界中是合理的，但很少有实际系统采用这种方法。如果需要做出一系列决策，更简单、更快的方法是首先选举出一个领导者，然后由领导者协调这些决策。

因此，实际系统与 Paxos 差异甚大。每个实现都从 Paxos 开始，发现其实施中的困难，进而发展出显著不同的架构。这一过程耗时且容易出错，而理解 Paxos 的困难进一步加剧了这一问题。Paxos 的表述可能在证明其正确性定理方面是合理的，但实际实现与 Paxos 差异巨大，使得这些证明几乎毫无价值。Chubby 实现者的一条评论具有代表性：

Paxos 算法的描述与现实系统的需求之间存在显著差距……最终的系统将基于一个未经验证的协议 [4]。

由于这些问题，我们得出结论：Paxos 既不适合作为系统构建的基础，也不适合作为教育的工具。鉴于在大规模软件系统中共识的重要性，我们决定尝试设计一种具有比 Paxos 更优特性的替代共识算法。Raft 就是这一实验的结果。

4 为可理解性而设计

我们在设计 Raft 时设定了多个目标：它必须为系统构建提供完整且实用的基础，从而显著减少开发者所需的设计工作量；它必须在所有条件下都是安全的，并在典型运行条件下保持可用性；同时它必须在常见操作上具有高效性。但我们的最重要目标——也是最困难的挑战——是可理解性。它必须能够让广大受众轻松理解该算法。此外，必须能够形成对算法的直观认识，以便系统构建者能够在实际应用中做出不可避免的扩展。

在 Raft 的设计过程中，我们面临了许多需要在不同方案之间做出选择的点。在这些情况下，我们基于可理解性来评估各个替代方案：每种方案的解释难度如何（例如，其状态空间的复杂性如何，是否具有微妙的含义？），以及读者能否完全理解该方案及其含义？

我们认识到，此类分析具有高度主观性；尽管如此，我们采用了两种普遍适用的技术。第一种技术是广为人知的问题分解方法：在可能的情况下，我们将问题分解为若干可以独立解决、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、安全性以及成员变更分离开来。

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，从而使系统更加连贯，并尽可能消除非确定性。具体而言，日志不允许出现空洞，Raft 也限制了日志之间变得不一致的方式。虽然在大多数情况下我们试图消除非确定性，但某些情况下非确定性实际上有助于理解。特别是，随机化方法引入了非确定性，但通常通过以相似方式处理所有可能选择（“任选其一，无所谓”）来减少状态空间。我们使用随机化来简化 Raft 的领导者选举算法。

5 Raft 共识算法

Raft 是一种用于管理第 2 节中所述的复制日志的算法。图 2 以简化的形式总结了该算法以便参考，图 3 列出了该算法的关键属性；本节其余部分将逐个讨论这些图中的各个元素。

Raft 通过首先选举出一个独特的领导者，然后将管理复制日志的全部责任赋予该领导者来实现共识。领导者从客户端接收日志条目，将其复制到其他服务器，并告知服务器何时可以将日志条目应用到其状态机中。拥有领导者简化了复制日志的管理。例如，领导者可以独立决定新条目在日志中的位置，而无需咨询其他服务器，数据也以简单的方式从领导者流向其他服务器。领导者可能会失败或与其他服务器断开连接，在这种情况下将选举出新的领导者。

基于领导者机制，Raft 将共识问题分解为三个相对独立的子问题，接下来的子节中将分别讨论：

- 领导者选举：当现有领导者失败时，必须选择一个新的领导者（第 5.2 节）。

- 日志复制：领导者必须接受日志条目 <table><tr><td colspan=”2”> 状态 </td></tr><tr><td colspan=”2”> 所有服务器上的持久状态：(在响应 RPC 之前更新到稳定存储)</td></tr><tr><td>curren
服务器最近看到的最新任期(首次启动时初始化为 0，单调递增)</td></tr><tr><td>votedFor</td><td>
当前任期中获得选票的候选者 ID(如果没有则为 null)</td></tr><tr><td>log[]</td><td>
日志条目；每个条目包含状态机的命令以及领导者接收到该条目时的任期（索引从
1 开始）</td></tr><tr><td> 所有服务器上的易失状态：</td><td> 已知被提交的最高
日志条目索引(初始化为 0，单调递增)</td></tr><tr><td>commitIndex</td><td> 已应
用到状态机的最高日志条目索引(初始化为 0，单调递增)</td></tr><tr><td>lastApplied</td><td></td>
领导者上的易失状态：(选举后重新初始化)</td><td></td></tr><tr><td>nextIndex[]</td><td>
对于每个服务器，下一个要发送给该服务器的日志条目索引（初始化为领导者最后
一条日志的索引 +1）</td></tr><tr><td>matchIndex[]</td><td> 对于每个服务器，已
知在该服务器上复制的最高日志条目索引(初始化为 0，单调递增)</td></tr></table>

AppendEntries RPC

由领导者调用以复制日志条目（$5.3）；也用作心跳（$5.2）。

参数：

term
leaderId
prevLogIndex
prevLogTerm

entries[]

leaderCommit

结果：

term

success

接收方实现：

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

2. 如果日志中不存在 prevLogIndex 处的条目且其任期与 prevLogTerm 不匹配（$5.3），则回复 false

3. 如果已有条目与新条目冲突（索引相同但任期不同），则删除已有条目及其之后的所有条目（$5.3）

4. 追加日志中尚未存在的新条目

5. 如果 leaderCommit > commitIndex，则设置 matchIndex = min(leaderCommit, 最后一个新条目的索引)

RequestVote RPC

由候选者调用以收集选票（$5.2）。

参数：

term

candidatedId

lastLogIndex

lastLogTerm

结果：

term

voteGranted

候选者的任期

请求选票的候选者

候选者最后一条日志条目的索引（$5.4）

候选者最后一条日志条目的任期（$5.4）

当前任期，用于候选者更新自身，true 表示候选者已获得选票

接收方实现：

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

2. 如果 votedFor 为 null 或 candidatedId，且候选者的日志至少与接收方的日志一样新，则授予选票（$5.2, $5.4）

服务器规则

所有服务器:

- 如果 commitIndex > lastApplied：增加 lastApplied，将 log[$lastApplied$] 应用到状态机（$5.3）

- 如果 RPC 请求或响应包含的 term T > currentTerm：设置 currentTerm = T，转换为跟随者（$5.1）

跟随者（$5.2）:

- 响应来自候选者和领导者的 RPC 请求

- 如果在选举超时时间内未收到当前领导者发送的 AppendEntries RPC 或未授予候选者选票：转换为候选者

候选者（$5.2）:

- 转换为候选者时，开始选举:

- 增加 currentTerm

- 自己投票

- 重置选举计时器

- 向所有其他服务器发送 RequestVote RPC

- 如果从多数服务器收到选票：成为领导者

- 如果收到来自新领导者的 AppendEntries RPC：转换为跟随者

- 如果选举超时结束：启动新的选举

领导者:

- 选举后：向每个服务器发送初始的空 AppendEntries RPC（心跳）；在空闲期间重复发送，以防止选举超时（$5.2）

- 如果收到客户端命令：将条目追加到本地日志，待条目应用到状态机后响应（$5.3）

- 如果最后一条日志索引 ≥ 某个跟随者的 nextIndex：向该跟随者发送包含从 nextIndex 开始的日志条目的 AppendEntries RPC

- 如果成功：更新该跟随者的 nextIndex 和 matchIndex（$5.3）

- 如果由于日志不一致导致 AppendEntries 失败：减少 nextIndex 并重试（$5.3）

- 如果存在一个 N，使得 $N >$ commitIndex，多数 matchIndex[i] ≥ N，且 log[N].term = currentTerm：设置 commitIndex = N（$5.3, $5.4）

图 2：Raft 共识算法的简化总结（不包含成员变更和日志压缩）。左上角方框中的服务器行为被描述为一组独立触发且反复执行的规则。如 §5.2 这样的章节编号表示特定功能在何处被讨论。形式化规范 [31] 对算法进行了更精确的描述。选举安全：在一个任期中最多只能选举出一个领导者。§5.2

领导者追加-only：领导者永远不会覆盖或删除其日志中的条目；它只会追加新的条目。§5.3

日志匹配：如果两个日志中包含索引和任期相同的条目，则在该索引之前的所有条目中，两个日志完全相同。§5.3

领导者完整性：如果某个日志条目在某个任期中被提交，则该条目将在所有更高任期的领导者日志中存在。§5.4

状态机安全：如果某服务器已将某个日志条目应用到其状态机中，则其他任何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成立。章节编号表示每个属性在何处被讨论。

从客户端接收请求并将其复制到集群中，迫使其他日志与之保持一致（第 5.3 节）。

- 安全性：Raft 的关键安全性属性是图 3 中的状态机安全性属性：如果任意一个服务器已将某个日志条目应用到其状态机中，则其他任何服务器都不能为相同的日志索引应用不同的命令。第 5.4 节描述了 Raft 如何确保这一属性；解决方案涉及对第 5.2 节中所述的选举机制施加额外限制。

在介绍共识算法之后，本节讨论了可用性问题以及系统中时间因素的作用。

5.1 Raft 基本原理

一个 Raft 集群包含若干服务器；五个是典型数量，这使得系统能够容忍两个故障。在任意时刻，每个服务器处于三种状态之一：领导者、跟随者或候选者。在正常运行中，系统中恰好有一个领导者，其余所有服务器均为跟随者。跟随者是被动的：它们不会主动发出请求，只是响应来自领导者和候选者的请求。领导者处理所有客户端请求（如果客户端联系到跟随者，跟随者会将其重定向至领导者）。第三种状态“候选者”用于选举新领导者，如第 5.2 节所述。图 4 展示了状态及其转换；转换将在下文讨论。

Raft 将时间划分为任意长度的任期，如图 5 所示。任期用连续的整数编号。每个任期开始于一次选举，在选举中，一个或多个候选者试图成为领导者，如第 5.2 节所述。如果候选者赢得选举，则它将作为领导者运行整个任期。在某些情况下，选举可能导致分裂投票。在这种情况下，任期将以没有领导者结束；一个新的任期（伴随新的选举）将很快开始。Raft 确保在一个任期中最多只有一个领导者。

不同的服务器可能在不同时间观察到任期之间的转换，在某些情况下，某个服务器甚至可能未观察到选举或整个任期。任期在 Raft 中充当逻辑时钟 [14]，并允许服务器检测过时信息，例如过时的领导者。每个服务器存储一个当前任期号，该编号随时间单调递增。每当服务器通信时，会交换当前任期号：如果一个服务器的当前任期小于另一个服务器，则它会将其当前任期更新为较大的值。如果候选者或领导者发现自己的任期已过时，它会立即切换回跟随者状态。如果服务器接收到一个任期号过时的请求，它将拒绝该请求。

Raft 服务器通过远程过程调用（RPC）进行通信，基本共识算法仅需要两种类型的 RPC。RequestVote RPC 由候选者在选举期间发起（第 5.2 节），AppendEntries RPC 由领导者发起，用于复制日志条目并提供心跳机制（第 5.3 节）。第 7 节增加了一种用于服务器之间传输快照的第三种 RPC。如果服务器在合理时间内未收到响应，则会重试 RPC，并且为实现最佳性能，服务器会并行发出 RPC 请求。

5.2 领导者选举

Raft 使用心跳机制触发领导者选举。当服务器启动时，它们最初处于跟随者状态。只要服务器接收到有效的来自领导者或候选者的 RPC。领导者会定期向所有跟随者发送心跳（携带无日志条目的 AppendEntries RPC），以维持其权威性。如果跟随者在一段时间（称为选举超时）内未收到任何通信，则会认为没有可用的领导者，并开始选举以选择新的领导者。

要开始选举，跟随者会将其当前任期递增，并转换到候选者状态。随后，它为自己投出一票，并并行向集群中的其他所有服务器发送 RequestVote RPC。候选者会持续处于该状态，直到以下三种情况之一发生：(a) 它赢得了选举，(b) 另一台服务器确立为领导者，或 (c) 一段时间过去后仍未产生胜出者。这些结果将在下面的段落中分别讨论。

如果候选者收到全集群中多数服务器对其在相同任期内的投票，则该候选者赢得选举。每个服务器在一个给定任期中最多只会为一个候选者投票，且遵循先到先得的原则（注意：第 5.4 节对投票增加了额外限制）。多数规则确保在特定任期中至多只有一个候选者能够赢得选举（图 3 中的选举安全属性）。一旦候选者赢得选举，它便成为领导者。随后，它会向所有其他服务器发送心跳消息，以确立其权威并防止新的选举发生。

在等待投票期间，候选者可能收到来自其他服务器的 AppendEntries RPC，该 RPC 声称其为领导者。如果领导者任期（包含在 RPC 中）至少等于候选者的当前

任期，则候选者认可该领导者为合法，并返回到跟随者状态。如果 RPC 中的任期小于候选者的当前任期，则候选者拒绝该 RPC，并继续处于候选者状态。

第三种可能的结果是候选者既未获胜也未失败：如果许多跟随者同时成为候选者，投票可能被分割，导致没有候选者获得多数。当这种情况发生时，每个候选者都将超时，并通过递增其任期并发起新一轮 RequestVote RPC 来启动新的选举。然而，如果没有额外措施，分割投票可能会无限重复。

Raft 通过随机选举超时来确保分割投票罕见且能够快速解决。为了防止分割投票的发生，选举超时从一个固定区间内随机选择（例如，$150 - 300$ ms）。这使得服务器分布更均匀，大多数情况下仅有一个服务器会超时；该服务器赢得选举并发送心跳消息，早于其他服务器超时。相同的机制也用于处理分割投票。每个候选者在选举开始时重新启动其随机选举超时，并在该超时结束前等待，然后再开始下一轮选举；这降低了新选举中再次出现分割投票的可能性。第 9.3 节表明，该方法能够快速选举出领导者。

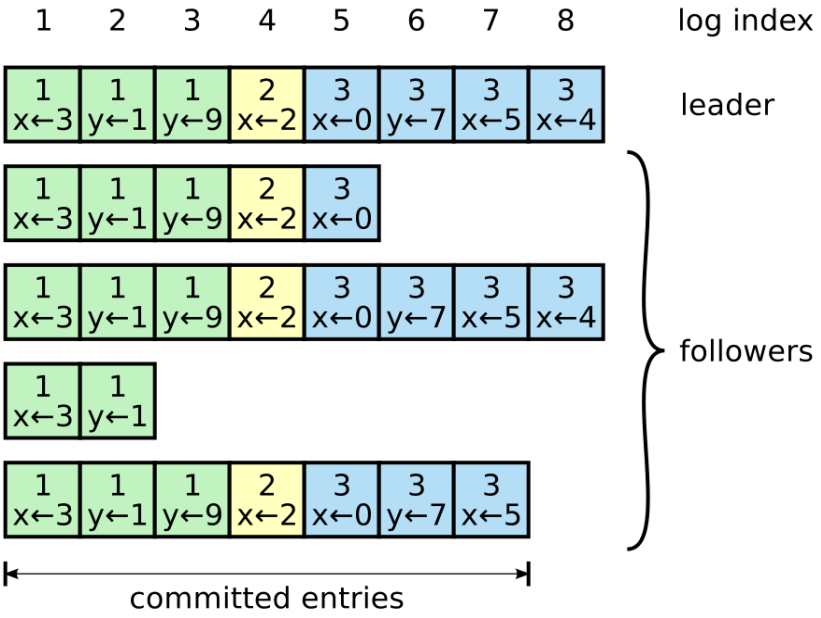

图 6：日志由条目组成，条目按顺序编号。每个条目包含其创建时的任期（每个方框中的数字）以及一个供状态机执行的命令。当一个条目可以安全地应用于状态机时，该条目被视为已提交。

选举是理解性指导我们选择设计替代方案的一个例子。最初我们计划采用排名系统：每个候选者被分配一个唯一的排名，用于在竞争的候选者之间进行选择。如果一个候选者发现另一个排名更高的候选者，它将返回到跟随者状态，以便更高排名的候选者能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引入了微妙的问题（一个排名较低的服务器可能需要超时并再次成为候选者，如果排名更高的服务器失败，但如果它过早这样做，可能会重置向选举领导者取得进展的进度）。我们对算法进行了多次调整，但每次调整后都会出现新的边界情况。最终我们得出结论，随机重试方法更直观且易于理解。

5.3 日志复制

一旦领导者被选举出来，它就开始处理客户端请求。每个客户端请求包含一个要由复制的状态机执行的命令。领导者将该命令作为新条目追加到其日志中，然后并行向其他所有服务器发送 AppendEntries RPC 以复制该条目。当该条目已安全复制（如下所述）后，领导者将其应用到其状态机中，并将该执行结果返回给客户端。如果跟随者崩溃或运行缓慢，或网络数据包丢失，领导者将无限期地重试 AppendEntries RPC（即使已经响应了客户端），直到所有跟随者最终保存所有日志条目。

日志的组织方式如图 6 所示。每个日志条目存储一个状态机命令以及该条目被领导者接收时的任期编号。日志条目中的任期编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。每个日志条目还具有一个整数索引标识符——确认其在日志中的位置。

领导者决定何时可以将日志条目应用到状态机中；这样的条目被称为已提交。Raft 保证已提交的条目是持久的，并且最终会被所有可用的状态机执行。当创建该条目的领导者已将其复制到多数服务器上时，该条目即被提交（例如图 6 中的条目 7）。这同时也提交了领导者日志中所有先前的条目，包括之前领导者创建的条目。第 5.4 节讨论了在领导者变更后应用此规则的一些微妙之处，并说明了这种提交定义是安全的。领导者会跟踪其已知的最高索引已提交条目，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得知。一旦跟随者了解到某条日志条目已被提交，它就会将其应用到本地状态机中（按日志顺序）。

我们设计了 Raft 日志机制，以在不同服务器之间的日志之间保持高水平的一致性。这不仅简化了系统的运行行为并使其更具可预测性，而且是确保安全性的关键组成部分。Raft 保持以下属性，这些属性共同构成了图 3 中的日志匹配属性：

- 如果两个不同日志中的条目具有相同的索引和任期，则它们存储相同的命令。

- 如果两个不同日志中的条目具有相同的索引和任期，则这两个日志在所有先前条目上完全相同。

第一个属性源于领导者在给定任期中为给定日志索引最多创建一个条目，且日志条目永远不会改变其在日志中的位置。第二个属性由 AppendEntries 执行的简单一致性检查保证。当发送一个 AppendEntries RPC 时，领导者会包含其日志中紧接在新条目之前的条目的索引和任期。如果跟随者在其日志中未找到具有相同索引和任期的条目，则它将拒绝这些新条目。该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，且在日志被扩展时，一致性检查会保持日志匹配属性。因此，每当 AppendEntries 成功返回时，领导者就知道跟随者日志在其新条目之前与自身日志完全一致。

在正常运行期间，领导者和跟随者之间的日志保持一致，因此 AppendEntries

的一致性检查永远不会失败。然而，领导者崩溃可能导致日志不一致（旧领导者可能未完全复制其日志中的所有条目）。这些不一致性可能在一系列领导者和跟随者崩溃后不断累积。图 7 说明了跟随者日志可能与新领导者日志不同的方式。一个跟随者可能

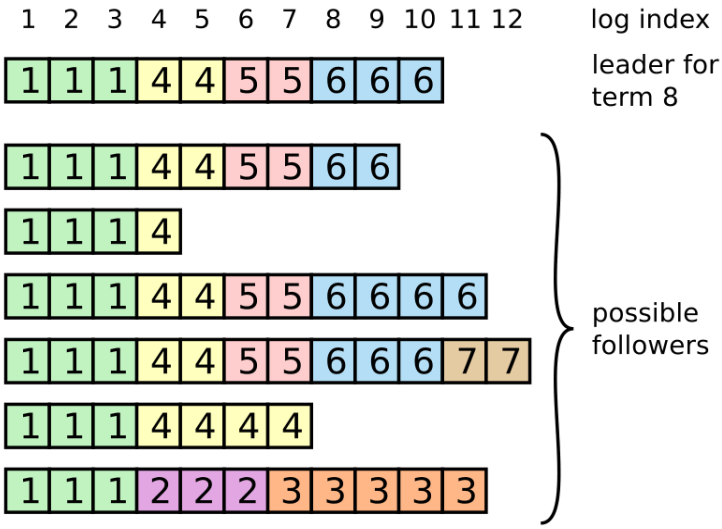

图 7：当顶部的领导者接管时，跟随者日志中可能发生任何一种情况（a-f）。每个方框代表一个日志条目；方框中的数字是其任期。跟随者可能缺少领导者上的条目（a-b），可能包含额外的未提交条目（c-d），或两者兼有（e-f）。例如，情况（f）可能发生在该服务器在任期 2 时是领导者，添加了若干条目到其日志，但在提交任何条目之前崩溃；它迅速重启，成为任期 3 的领导者，并添加了几个新条目；在任期 2 或任期 3 的任何条目被提交之前，服务器再次崩溃并持续停机数个任期。

缺少领导者上存在的条目，也可能包含领导者上不存在的额外条目，或两者皆有。日志中的缺失和多余条目可能跨越多个任期。

在 Raft 中，领导者通过强制跟随者日志与其自身日志同步来处理不一致性。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 节将证明，当结合一个额外限制时，这种操作是安全的。

为了将跟随者日志与其自身日志同步，领导者必须找到两个日志首次达成一致的最新日志条目，删除该点之后跟随者日志中的所有条目，并向跟随者发送该点之后领导者日志中的所有条目。所有这些操作都是在 AppendEntries RPC 执行的一致性检查过程中发生的。领导者为每个跟随者维护一个 nextIndex，即领导者将要发送给该跟随者的下一个日志条目的索引。当领导者首次接管时，它将所有 nextIndex 初始化为日志中最后一个索引之后的索引（图 7 中的 11）。如果跟随者日志与领导者日志不一致，那么在下一次 AppendEntries RPC 中，AppendEntries 的一致性检查将失败。在被拒绝后，领导者会递减 nextIndex 并重试 AppendEntries RPC。最终，nextIndex 将达到领导者和跟随者日志匹配的点。当这种情况发生时，AppendEntries 将成功，从而清除跟随者日志中的任何冲突条目，并追加来自领导

者日志的条目（如有）。一旦 AppendEntries 成功，跟随者日志就与领导者日志保持一致，并在整个任期中保持一致。

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，跟随者可以包含冲突条目的项及其首次存储该条目的索引。通过这些信息，领导者可以将 nextIndex 减少，从而跳过该项的所有冲突条目；每个具有冲突条目的项只需一次 AppendEntries RPC 即可完成复制，而无需为每个条目单独发送一次 RPC。实际上，我们怀疑这种优化是不必要的，因为故障发生频率很低，且不太可能出现大量不一致的条目。

通过该机制，当领导者在电源恢复时，无需采取任何特殊操作来恢复日志一致性。它只需开始正常运行，日志会自动根据副本一致性检查的故障而收敛。领导者永远不会在其自身的 log 中覆盖或删除条目（如图 3 中所述的领导者追加-only 属性）。

这种日志复制机制表现出第 2 节中描述的理想的共识属性：Raft 只要多数服务器处于运行状态，就可以接受、复制并应用新的日志条目；在正常情况下，一个新条目可以通过一次 RPC 轮次复制到集群的多数节点；且单个缓慢的跟随者不会影响性能。

5.4 安全性

前几节描述了 Raft 如何选举领导者并复制日志条目。然而，迄今为止所描述的机制尚不足以确保每个状态机以完全相同的顺序执行完全相同的命令。例如，当领导者提交多个日志条目时，某个跟随者可能暂时不可用，之后该跟随者可能被选举为领导者并用新条目覆盖这些条目；结果可能导致不同状态机执行不同的命令序列。

本节通过为可以被选举为领导者的服务器添加一个限制，来完善 Raft 算法。该限制确保在任一任期中，领导者包含前一任期中已提交的所有条目（如图 3 所示的领导者完整性属性）。在满足选举限制的前提下，我们进一步明确了提交规则。最后，我们给出了领导者完整性属性的证明草图，并说明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已提交的日志条目。在某些共识算法中，例如 Viewstamped Replication [22]，即使领导者最初不包含所有已提交的条目，也可以被选举为领导者。这些算法包含额外机制来识别缺失的条目，并在选举过程中或选举后不久将其传输给新领导者。不幸的是，这导致了大量额外的机制和复杂性。Raft 采用了一种更简单的方案，即保证在新领导者被选举的瞬间，每个新领导者都拥有前一任期中所有已提交条目，无需将这些条

目传输给领导者。这意味着日志条目仅沿一个方向流动——从领导者流向跟随者，领导者永远不会在其日志中覆盖已存在的条目。

Raft 通过投票过程来防止候选者在日志中不包含所有已提交条目时赢得选举。候选者必须与集群的多数节点联系才能被选举，这意味着每个已提交的条目必须至少存在于这些节点中的一个。如果候选者的日志至少与该多数中的任何其他日志一样“最新”(“最新”将在下文精确定义)，则它将包含所有已提交的条目。RequestVote RPC 实现了这一限制：该 RPC 包含候选者日志的信息，投票者如果发现自己的日志比候选者的日志更“最新”，则拒绝投票。

Raft 通过比较两个日志中最后一条条目的索引和任期来判断哪个日志更“最新”。如果两个日志的最后条目任期不同，则任期较晚的日志更“最新”；如果两个日志以相同的任期结束，则较长的日志更“最新”。

5.4.2 提交前一任期的日志条目

如第 5.3 节所述，领导者知道一旦其当前任期中的某个条目被多数服务器存储，该条目即被提交。如果领导者在提交条目之前崩溃，后续领导者将尝试完成该条目的复制。然而，领导者不能立即得出结论，认为一旦某个条目被多数服务器存储，该条目就来自前一任期的已提交条目。图-

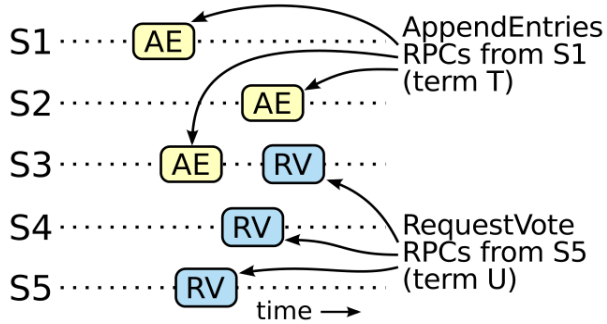

图 9：如果 S1（任期为 T 的领导者）提交了一个来自其任期的新日志条目，而 S5 在之后的任期 U 中被选举为领导者，那么必须至少有一个服务器（S3）既接受了该日志条目，又投票支持了 S5。

图 8 展示了一种情况，即一个旧的日志条目被存储在多数服务器上，但仍可能被未来的领导者覆盖。

为消除图 8 中类似的问题，Raft 从不通过计数副本来提交来自前一个任期的日志条目。只有来自当前领导者任期的日志条目才通过计数副本被提交；一旦通过这种方式提交了一个当前任期的日志条目，那么所有先前的日志条目将由于日志匹配属性而被间接提交。在某些情况下，领导者可以安全地得出一个旧日志条目已被提交的结论（例如，如果该条目被存储在所有服务器上），但 Raft 为了简化，采取了更为保守的方法。

Raft 在提交规则中引入这种额外复杂性，是因为当领导者复制来自前一个任期的日志条目时，这些日志条目会保留其原始的任期编号。在其他共识算法中，如果新领导者复制前一个“任期”的日志条目，必须使用其新的“任期编号”。Raft 的方法使得日志条目的推理更加简单，因为它们在时间上和不同日志之间都保持相同的任期编号。此外，Raft 中的新领导者发送的来自前一个任期的日志条目数量少于其他算法（其他算法必须发送冗余的日志条目以重新编号，才能被提交）。

5.4.3 安全性论证

在完整的 Raft 算法基础上，我们现在可以更精确地论证领导者完整性属性成立（该论证基于安全性证明；见第 9.2 节）。我们假设领导者完整性属性不成立，然后推导出矛盾。假设在任期 T 的领导者（记为 $_{\mathrm{T}}$）提交了一个来自其任期的日志条目，但该条目并未被某个未来任期的领导者存储。考虑最小的任期 $U > T$，其领导者（记为 $_{\mathrm{U}}$）不存储该条目。

1. 该已提交的条目在领导者 $_{\mathrm{U}}$ 选举时一定不存在于其日志中（领导者从不会删除或覆盖条目）。

2. 领导者 $_{\mathrm{T}}$ 已在集群的多数节点上复制了该条目，且领导者 $_{\mathrm{U}}$ 从集群的多数节点获得了选票。因此，至少有一个服务器（“投票者”）既接受了来自领导者 $_{\mathrm{T}}$ 的条目，又投票支持了领导者 $_{\mathrm{U}}$，如图 9 所示。投票者是推导出矛盾的关键。

3. 投票者必须在投票支持领导者 $_{\mathrm{U}}$ 之前就已接受来自领导者 $_{\mathrm{T}}$ 的已提交条目；否则它会拒绝来自领导者 $_{\mathrm{T}}$ 的 AppendEntries 请求（因为其当前任期会高于 T）。

4. 当投票者投票支持领导者 $_{\mathrm{U}}$ 时，它仍然保留了该条目，因为所有中间的领导者都包含该条目（根据假设），领导者从不会删除条目，而跟随者仅在与领导者冲突时才会删除条目。

5. 投票者将选票授予了领导者 $_{\mathrm{U}}$，因此领导者 $_{\mathrm{U}}$ 的日志必须与投票者日志一样最新。这导致了两种矛盾之一。

6. 首先，如果投票者和领导者 $_{\mathrm{T}}$ 拥有相同的最后一个日志任期，那么领导者 $_{\mathrm{U}}$ 的日志必须至少与投票者日志一样长，因此其日志必须包含投票者日志中的所有条目。这与假设矛盾，因为投票者包含了已提交的条目，而领导者 $_{\mathrm{U}}$ 被假设不包含。

7. 否则，领导者 $_{\mathrm{U}}$ 的最后一个日志任期必须大于投票者的任期。而且它必须大于 T，因为投票者的最后一个日志任期至少为 T（它包含来自任期 T 的已提交条目）。创建领导者 $_{\mathrm{U}}$ 最后一个日志条目的前一个领导者必须在其日志中包含该已提交条目（根据假设）。然后，根据日志匹配属性，领导者 $_{\mathrm{U}}$ 的日志也必须包含该已提交条目，这又是一个矛盾。

8. 这完成了矛盾的推导。因此，所有任期大于 T 的领导者都必须包含在任期 T 中已提交的所有条目。

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

在领导者完整性属性的基础上，我们可以证明图 3 中的状态机安全性属性，该属性指出：如果一个服务器已将其状态机应用了某个索引的日志条目，则此后没有任何其他服务器会为相同索引应用不同的日志条目。当一个服务器将其日志条目应用到状态机时，其日志必须与领导者日志在该条目之前完全一致，且该条目必须已被提交。现在考虑任意服务器首次应用某个日志索引的最低任期；日志完整性属性保证所有更高任期的领导者都将存储该相同的日志条目，因此在后续任期中应用该索引的服务器将应用相同的值。因此，状态机安全性属性成立。

最后，Raft 要求服务器按照日志索引顺序应用条目。结合状态机安全性属性，这意味着所有服务器将按相同顺序应用完全相同的日志条目集合到其状态机中。

5.5 追随者和候选者的崩溃

到目前为止，我们一直关注领导者故障。追随者和候选者的崩溃比领导者崩溃更容易处理，且它们的处理方式完全相同。如果一个追随者或候选者崩溃，那么之后发送给它的请求投票（RequestVote）和追加条目（AppendEntries）远程过程调用（RPC）将失败。Raft 通过无限重试来处理这些故障；如果崩溃的服务器重启，那么该 RPC 将成功完成。如果一个服务器在完成一个 RPC 但尚未响应之前崩溃，那么它在重启后将再次收到相同的 RPC。Raft 的 RPC 是幂等的，因此这种情况不会造成任何危害。例如，如果一个追随者接收到一个包含其日志中已存在条目的追加条目请求，它将忽略该请求中的这些条目。

5.6 时间和可用性

我们对 Raft 的一个要求是：安全性不应依赖于时间——系统不应因为某些事件发生得比预期快或慢而产生错误结果。然而，可用性（系统能够及时响应客户端的能力）不可避免地依赖于时间。例如，如果消息交换时间超过服务器崩溃之间的典型时间，候选者将无法长时间存活以赢得选举；没有稳定的领导者，Raft 就无法取得进展。

领导者选举是 Raft 中对时间最敏感的部分。只要系统满足以下时间要求，Raft 就能够选举并维持一个稳定的领导者：

$$broadcastTime \ll electionTimeout \ll MTBF$$

在这个不等式中，broadcastTime 是一个服务器并行向集群中所有服务器发送 RPC 并接收响应的平均时间；electionTimeout 是第 5.2 节中描述的选举超时时间；MTBF 是单个服务器故障之间的平均时间。广播时间应比选举超时时间小一个数量级，以便领导者能够可靠地发送心跳消息，防止追随者启动选举；考虑到选举

超时时间采用随机化方法，该不等式也使得出现分裂投票的可能性很小。选举超时时间应比 MTBF 小几个数量级，以便系统能够稳步前进。当领导者崩溃时，系统将大约在选举超时时间内不可用；我们希望这仅占总体时间的很小一部分。

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

广播时间和 MTBF 是底层系统的属性，而选举超时时间是我们必须选择的参数。Raft 的 RPC 通常要求接收方将信息持久化到稳定存储中，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，选举超时时间很可能在 10 毫秒到 500 毫秒之间。典型

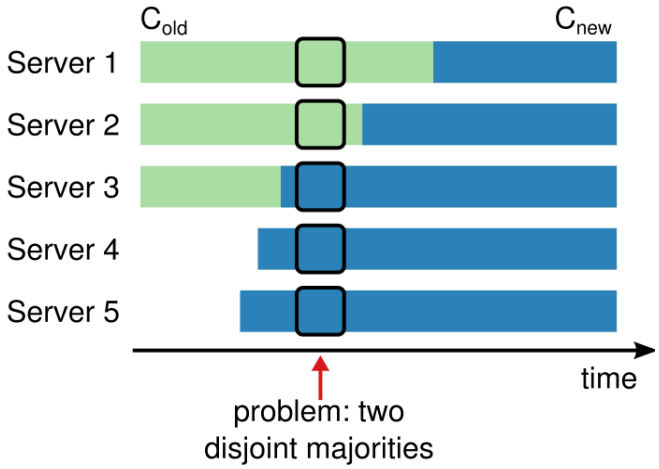

图 10：直接从一个配置切换到另一个配置是不安全的，因为不同服务器会在此时切换。在这个例子中，集群从三个服务器扩展到五个服务器。不幸的是，存在一个时间点，同一任期下可能选举出两个不同的领导者，一个拥有旧配置（C_{old}）的多数，另一个拥有新配置（C_{new}）的多数。

服务器的 MTBF 通常为几个月甚至更长，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。实际上，偶尔需要更改配置，例如当服务器故障时替换它们，或调整复制程度。虽然可以通过将整个集群离线、更新配置文件，然后重启集群来实现配置变更，但这会使集群在变更期间不可用。此外，如果涉及任何手动操作，可能会引发操作员错误。为避免这些问题，我们决定自动化配置变更，并将其集成到 Raft 共识算法中。

为了使配置变更机制安全，必须确保在转换过程中不存在某个时刻，使得同一任期下可能选举出两个领导者。不幸的是，任何直接从旧配置切换到新配置的方法都是不安全的。无法原子地同时切换所有服务器，因此在转换过程中集群可能会分裂为两个独立的多数（见图 10）。

为了确保安全性，配置变更必须采用两阶段方法。实现两阶段的方法有很多。例如，一些系统（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，集群首先切换到一种我们称之为“联合共识”的过渡配置；一旦联合共识被提交，系统则过渡到新配置。联合共识结合了旧配置和新配置：

- 日志条目被复制到两个配置中的所有服务器。

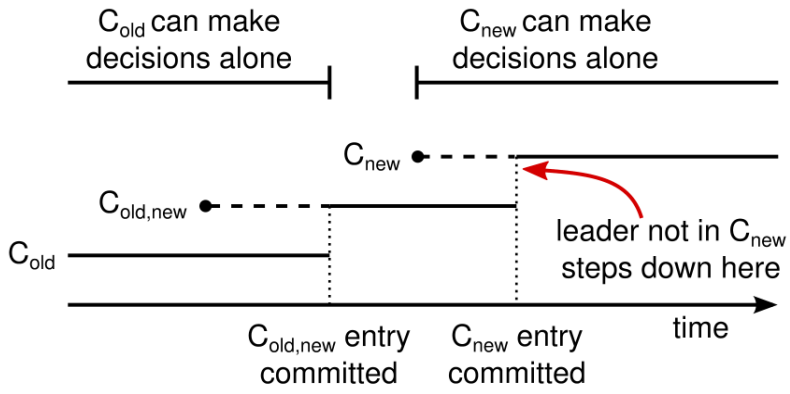

图 11：配置变更的时间线。虚线表示已创建但尚未提交的配置条目，实线表示最新的已提交配置条目。领导者首先在其日志中创建 $C_{\text{old,new}}$ 配置条目并将其提交至 $C_{\text{old,new}}$（即 C_{old} 和 C_{new} 的多数）；然后它创建 C_{new} 条目，并将其提交至 C_{new} 的多数。在任意时刻，C_{old} 和 C_{new} 都不能独立做出决策。

- 任一配置中的任意服务器均可担任领导者。

- 关于选举和条目提交的共识，需要分别获得旧配置和新配置的多数支持。

联合共识使得各个服务器可以在不同时间点过渡到不同配置，而不会损害安全性。此外，联合共识允许集群在配置变更期间持续处理客户端请求。

集群配置通过复制日志中的特殊条目进行存储和传递；图 11 展示了配置变更过程。当领导者接收到从 C_{old} 到 C_{new} 的配置变更请求时，它将配置用于联合共识（图中为 $C_{\text{old,new}}$）作为日志条目，并使用前述机制将其复制到其他节点。一旦某个服务器将其新配置条目添加到本地日志中，它将使用该配置进行所有后续决策（服务器始终使用其日志中最新的配置，无论该条目是否已提交）。这意味着领导者将使用 $C_{\text{old,new}}$ 的规则来判断 $C_{\text{old,new}}$ 日志条目的提交时机。如果领导者崩溃，新领导者可能在 C_{old} 或 $C_{\text{old,new}}$ 中被选举出来，具体取决于获胜候选者是否已收到 $C_{\text{old,new}}$ 条目。无论如何，在此期间 C_{new} 都不能独立做出决策。

一旦 $C_{\text{old,new}}$ 被提交，C_{old} 和 C_{new} 都不能在未经对方批准的情况下做出决策，且领导者完整性属性确保只有拥有 $C_{\text{old,new}}$ 日志条目的服务器才能被选举为领导者。此时领导者可以创建描述 C_{new} 的日志条目并将其复制到集群中。同样，一旦该配置被各服务器看到，它就会立即生效。当新配置在 C_{new} 的规则下被提交后，旧配置将不再相关，不再属于新配置的服务器可以被关闭。如图 11 所示，不存在 C_{old} 和 C_{new} 同时独立做出决策的时刻；这保证了系统的安全性。

还需解决三个关于重新配置的问题。第一个问题是新加入的服务器可能最初没有存储任何日志条目。如果它们以这种状态加入集群，可能需要很长时间才能追上日志，期间可能无法提交新的日志条目。为避免可用性断点，Raft 在配置变更之前引入了一个额外阶段，即新服务器以非投票成员身份加入集群（领导者会将其日志条目复制给它们，但它们不参与多数投票）。一旦新服务器追上了集群的其余部分，就可以按照上述方式继续进行重新配置。

第二个问题是集群领导者可能不在新配置中。在这种情况下，领导者在提交 C_{new} 日志条目后将退出（返回到从属状态）。这意味着在提交 C_{new} 的过程中，领导者将管理一个不包含自身的集群；它会复制日志条目，但自身不计入多数投票。领导者状态的转换发生在 C_{new} 被提交时，因为这是新配置首次能够独立运行的时刻（从 C_{new} 中始终可以选出领导者）。在此之前，可能只有 C_{old} 中的服务器才能被选举为领导者。

第三个问题是被移除的服务器（即不在 C_{new} 中的服务器）可能扰乱集群。这些服务器将不会收到心跳，因此会超时并启动新的选举。它们随后会发送带有新任期号的 RequestVote RPC 请求，这将导致当前领导者退回到从属状态。最终会选举出一个新领导者，但被移除的服务器会再次超时，该过程将重复进行，从而导致可用性较差。

为避免此问题，服务器在认为当前存在领导者时会忽略 RequestVote RPC 请求。具体而言，如果一个服务器在接收到当前领导者心跳的最小选举超时时间内收到 RequestVote RPC 请求，则它不会更新自己的任期或授予其投票权。这不会影响正常的选举过程，因为在正常选举中每个服务器会在至少经过最小选举超时时间后才开始选举。然而，这有助于避免被移除服务器带来的干扰：如果领导者能够持续接收到集群的心跳，则它不会被更高任期号所驱逐。

7 日志压缩

Raft 的日志在正常运行过程中会不断增长以包含更多的客户端请求，但在实际系统中，日志不能无限制增长。随着日志变长，它占用的空间越来越多，重放所需的时间也越长。若没有某种机制来丢弃日志中累积的过时信息，最终将导致可用性问题。

快照是日志压缩最简单的方法。在快照机制中，将当前系统的完整状态写入稳定的存储介质中的快照，然后将日志中到该快照为止的部分进行压缩和清理。

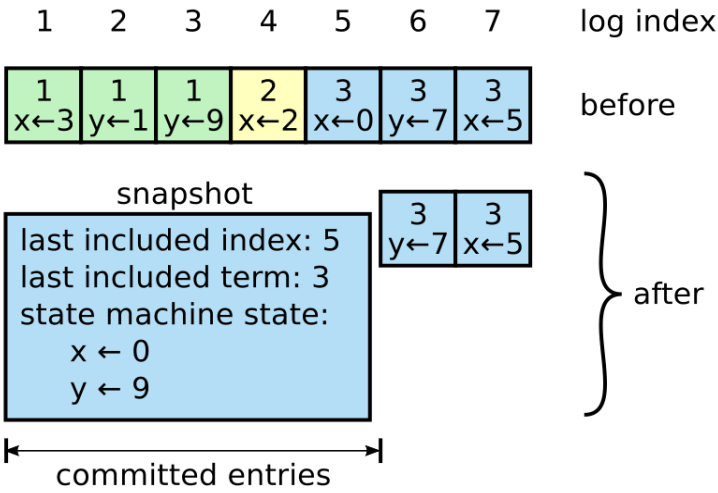

图 12：一个服务器会将其日志（索引 1 到 5）中的已提交条目替换为一个新的快照，该快照仅存储当前状态（本例中的变量 x 和 y）。快照的最后一个包含索引和任期用于将快照定位在日志中第 6 条目之前的正确位置。

该点将被丢弃。快照机制被用于 Chubby 和 ZooKeeper，本节的其余部分描述 Raft 中的快照机制。

可以采用渐进式的压缩方法，例如日志清理 [36] 和日志结构合并树 [30, 5]。这些方法一次只处理数据的一部分，因此在时间上更均匀地分散了压缩的负载。它们首先选择一个积累了大量已删除和被覆盖对象的数据区域，然后以更紧凑的方式重写该区域中的活跃对象，并释放该区域。与快照机制相比，这需要显著更多的额外机制和复杂性，而快照机制通过始终对整个数据集进行操作简化了问题。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 展示了 Raft 中快照的基本思想。每个服务器独立地创建快照，仅覆盖其日志中的已提交条目。大部分工作是状态机将当前状态写入快照。Raft 还在快照中包含少量元数据：最后一个包含索引是快照替换的日志中最后一个条目的索引（即状态机已应用的最后一条条目），最后一个包含任期是该条目的任期。这些信息被保留以支持快照后第一个日志条目的 AppendEntries 一致性检查，因为该条目需要前一个日志索引和任期。为了支持集群成员变更（第 6 节），快照还包括在最后一个包含索引处的日志中最新的配置。一旦服务器完成写入快照，它就可以删除日志中直到最后一个包含索引的所有条目，以及任何先前的快照。

虽然服务器通常独立地创建快照，但领导者必须偶尔向落后于它的从节点发送快照。这发生在领导者已经丢弃了它需要发送给从节点的下一个日志条目时。幸运的是，在正常操作中这种情况很少发生：一个跟上进度的从节点本应已经拥有该条目。然而，一个异常缓慢的从节点或一个新加入集群的服务器（第 6 节）则不会。为了使这样的从节点保持同步，领导者需要通过网络向其发送一个快照。

领导者使用一种名为 InstallSnapshot 的新远程过程调用（RPC）来向严重落后的从节点发送快照；见图 13。当从节点接收到该 RPC 的快照时，它必须决定如何处理其现有的日志条目。通常，快照将包含日志中尚未存在的新信息。在这种情况下，从节点会丢弃其整个日志；因为快照将完全覆盖它，且可能包含与快照冲突的未提交条目。如果从节点接收到的快照描述的是其日志的前缀（由于重传或误操作），那么快照覆盖的日志条目将被删除，但快照之后的日志条目仍然有效，必须保留。

这种快照方法偏离了 Raft 的强领导者原则，因为从节点可以在不知道领导者的情况下创建快照。然而，我们认为这种偏离是合理的。虽然领导者有助于避免在达成共识过程中出现冲突决策，但快照操作已经达成了一致，因此不存在冲突决策。数据仍然只从领导者流向从节点。

InstallSnapshot RPC

由领导者调用，用于将快照的片段发送给从节点。领导者始终按顺序发送片段。

参数:

term 领导者的任期
leaderId 以便从节点可以将客户端重定向
lastIncludedIndex 快照替换日志中从该索引到该索引（含）的所有条目
lastIncludedTerm 最后包含索引的任期
offset 快照文件中片段的字节偏移量
data[] 快照片段的原始字节数据，从偏移量开始
done 如果是最后一个片段则为真
结果:
term 当前任期，用于领导者更新自身
接收方实现:
1. 如果 term < currentTerm，则立即回复
2. 如果是第一个片段（offset 为 0），则创建新的快照文件
3. 在指定偏移量处将数据写入快照文件
4. 如果 done 为假，则回复并等待更多数据片段
5. 保存快照文件，丢弃任何索引更小的现有或部分快照
6. 如果现有日志条目与快照的最后一个包含条目具有相同的索引和任期，则保留该条目之后的日志并回复
7. 丢弃整个日志

8. 使用快照内容重置状态机（并加载快照中的集群配置）

图 13：InstallSnapshot RPC 的概览。快照被分割成多个片段进行传输；每个片段都向从节点发送一个“心跳”信号，使其可以重置选举计时器。

领导者本应已经拥有该条目。然而，一个异常缓慢的从节点或一个新加入集群的服务器（第 6 节）则不会。为了使这样的从节点保持同步，领导者需要通过网络向其发送一个快照。降低了，现在只有跟随者可以重新组织他们的数据。

我们考虑了一种基于领导者的方法，其中只有领导者会创建一个快照，然后将该快照发送给每个跟随者。然而，这种方法有两个缺点。首先，将快照发送给每个跟随者会浪费网络带宽，并减慢快照生成的过程。每个跟随者已经拥有生成自身快照所需的信息，而且通常对于服务器来说，从本地状态生成一个快照比通过网络发送和接收一个快照要便宜得多。其次，领导者的实现会更加复杂。例如，领导者需要在复制新的日志条目到跟随者的同时，并行地将快照发送给跟随者，以避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进行快照。如果服务器快照过于频繁，会浪费磁盘带宽和能量；如果快照过于稀疏，则可能耗尽其存储容量，并增加重启时重放日志所需的时间。一个简单的策略是在日志达到固定字节数时进行快照。如果这个大小显著大于预期的快照大小，那么快照过程的磁盘带宽开销将很小。

第二个性能问题是写入快照可能需要相当长的时间，我们不希望这会延迟正常的操作。解决方案是使用“写时复制”（copy-on-write）技术，以便在不干扰正在写入的快照的情况下接受新的更新。例如，使用函数式数据结构构建的状态机天然支持这种机制。或者，可以利用操作系统的“写时复制”支持（例如 Linux 中的 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 如何支持线性可读语义 [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 的客户端会将所有请求发送给领导者。当客户端首次启动时，它会连接到一个随机选择的服务器。如果该客户端的第一个选择不是领导者，那么该服务器将拒绝客户端的请求，并提供关于最近听到的领导者的信息（AppendEntries 请求中包含领导者的网络地址）。如果领导者崩溃，客户端请求将超时；随后客户端会尝试连接到随机选择的其他服务器。

我们为 Raft 设定的目标是实现线性可读语义（每个操作似乎在调用和响应之间某个时刻以瞬时、精确一次的方式执行）。然而，如前所述，Raft 可能会多次执行命令：例如，如果领导者在提交日志条目后、但在响应客户端之前崩溃，客户端

将使用新的领导者重新提交该命令，从而导致该命令被再次执行。解决方案是让客户端为每个命令分配唯一的序列号。然后，状态机跟踪每个客户端已处理的最新序列号及其相关的响应。如果它接收到一个序列号已被执行的命令，它将立即响应，而不会重新执行该请求。

只读操作可以在日志中不写入任何内容的情况下处理。然而，如果不采取额外措施，这可能会返回过时的数据，因为响应请求的领导者可能已被一个它不知道的更新的领导者所取代。线性可读的读取操作绝不能返回过时的数据，Raft 需要两个额外的预防措施来确保这一点，而无需使用日志。首先，领导者必须拥有最新的已提交条目信息。领导者完整性属性保证领导者拥有所有已提交的条目，但在其任期开始时，它可能不知道哪些条目是已提交的。为了获取这些信息，它需要提交一个来自其任期的条目。Raft 通过让每个领导者在其任期开始时在日志中提交一个空白的无操作（no-op）条目来处理这个问题。其次，领导者在处理只读请求之前必须检查自己是否已被撤销（如果选举了更近期的领导者，其信息可能已过时）。Raft 通过在响应只读请求之前，让领导者与集群中的多数节点交换心跳消息来处理这个问题。或者，领导者可以依赖心跳机制来提供一种租约形式 [9]，但这依赖于时间来保证安全性（它假设时钟偏差是有界的）。

9 实现与评估

我们已将 Raft 作为存储 RAMCloud [33] 配置信息的复制状态机的一部分，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。源代码免费公开 [23]。此外，还有大约 25 个独立的第三方开源实现 [34]，这些实现基于本文的草稿，处于不同开发阶段。此外，多家公司正在部署基于 Raft 的系统 [34]。

本节的其余部分从三个方面评估 Raft：可理解性、正确性和性能。

9.1 可理解性

为了衡量 Raft 相对于 Paxos 的可理解性，我们在斯坦福大学高级操作系统课程以及加州大学伯克利分校分布式计算课程中，对大三和研究生学生进行了实验研究。我们录制了 Raft 和 Paxos 的视频讲座，并创建了相应的测验。Raft 讲座涵盖了本文的内容，除了日志压缩部分；Paxos

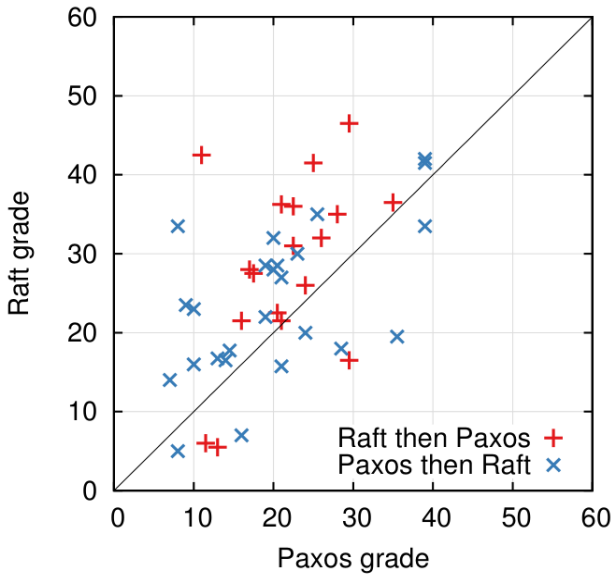

图 14：比较 43 名参与者在 Raft 和 Paxos 测验中的表现的散点图。对角线以上的点（33 个）表示在 Raft 测验中得分更高的参与者。

本次课程覆盖了足够多的内容，使学生能够构建等效的复制状态机，包括单决议 Paxos、多决议 Paxos、重新配置，以及实践中需要的一些优化（例如领导者选举）。测验测试了学生对算法的基本理解，同时也要求学生分析边界情况。每位学生观看一个视频，完成相应的测验，再观看第二个视频，完成第二个测验。大约一半的参与者先完成 Paxos 部分，另一半则先完成 Raft 部分，以考虑个体表现差异以及从研究第一部分中获得的经验。我们比较了参与者在每次测验中的得分，以判断参与者是否对 Raft 有更深入的理解。

我们尽力使 Paxos 与 Raft 之间的比较尽可能公平。实验在两个方面偏向于 Paxos：43 名参与者中有 15 人报告有 Paxos 的先前经验，且 Paxos 视频比 Raft 视频长 14

平均而言，参与者在 Raft 测验中的得分比在 Paxos 测验中高出 4.9 分（满分 60 分，Raft 的平均得分为 25.7，Paxos 的平均得分为 20.8）；图 14 展示了他们的个人得分。配对的 t 检验表明，以 95

我们还构建了一个线性回归模型，该模型基于三个因素预测新学生的测验得分：他们参加的是哪一测验、他们先前接触 Paxos 的经验程度，以及

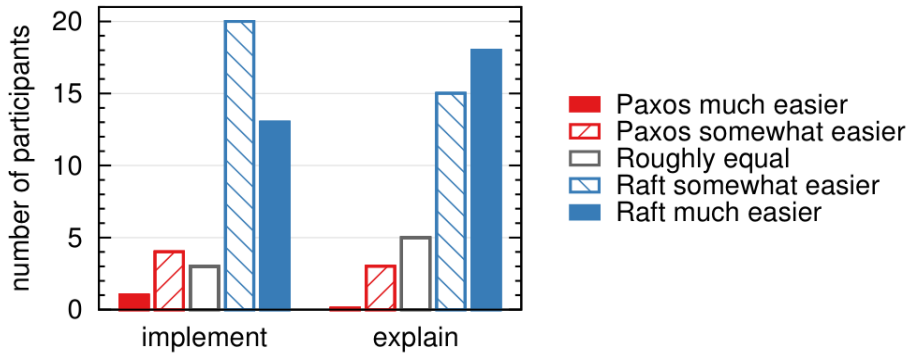

图 15：参与者使用 5 分制（左）表示他们认为哪个算法更容易在功能正确且高效的系统中实现，以及（右）哪个算法更容易向计算机科学研究生解释。

他们学习算法的顺序。该模型预测，测验选择会使得 Raft 得分领先 12.5 分。这一数值显著高于观察到的 4.9 分差异，因为许多实际学生具有先前的 Paxos 经验，这大大有利于 Paxos，而对 Raft 的提升则较小。有趣的是，该模型还预测，对于已经参加过 Paxos 测验的人，Raft 的得分会低 6.3 分；尽管我们不清楚原因，但这一结果在统计上是显著的。

我们还在测验后对参与者进行了调查，了解他们认为哪个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认为 Raft 更容易实现和解释（每个问题中 33 人/41 人）。然而，这些自我报告的感受可能不如参与者在测验中的得分可靠，且参与者可能受到我们假设 Raft 更容易理解这一前提的影响。

Raft 用户研究的详细讨论可参见 [31]。

9.2 正确性

我们为第 5 节描述的共识机制开发了一套形式化规范和安全性证明。形式化规范 [31] 使用 TLA+ 规范语言 [17] 将图 2 中总结的信息完全精确化。该规范约 400 行，是证明的主体。它本身也对任何实现 Raft 的开发者具有参考价值。我们使用 TLA 证明系统 [7] 机械地证明了日志完整性属性。然而，这一证明依赖于尚未机械验证的不变量（例如，我们尚未证明规范的类型安全性）。此外，我们撰写了一篇非正式的、完整的状态机安全性证明 [31]，该证明仅依赖规范本身，且无需额外假设。<table><tr><td> 担忧 </td><td> 缓解偏见的措施 </td><td> 可供审查的材料 [28, 31]</td></tr><tr><td> 课程质量一致 </td><td> 由同一讲师讲授。Paxos 部分基于并改进了多个大学已使用的材料。Paxos 部分视频比 Raft 部分长 14

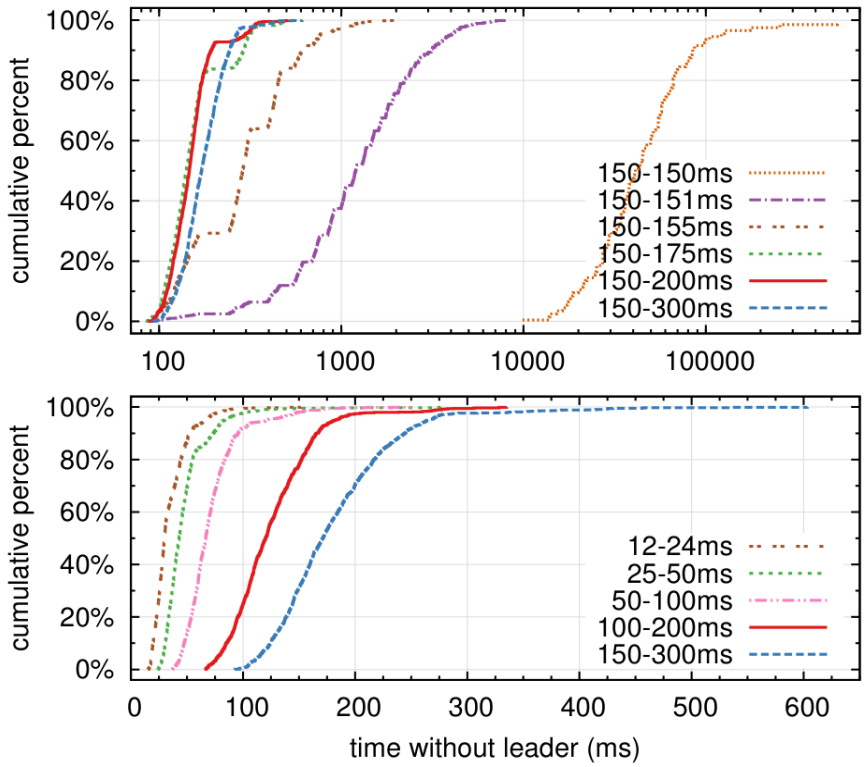

图 16：检测并替换故障领导者所需的时间。上图改变了选举超时中的随机性程度，下图则调整了最小选举超时。每条线代表 1000 次试验（“150-150 ms”例外为 100 次试验），对应于特定的选举超时设置；例如，“150-155 ms”表示选举超时在 150 毫秒到 155 毫秒之间随机且均匀地选取。测量是在一个包含五台服务器、广播时间约为 15 毫秒的集群上进行的。九台服务器集群的结果类似。

相对精确（约 3500 字）。

9.3 性能

Raft 的性能与其他共识算法（如 Paxos）类似。性能最重要的场景是当已建立的领导者正在复制新的日志条目时。Raft 通过最少的消息数量（领导者到集群一半节点的一次往返）实现了这一目标。此外，Raft 的性能还可以进一步提升。例如，它很容易支持请求的批处理和流水线处理，以提高吞吐量并降低延迟。文献中提出了许多针对其他算法的优化方法；其中许多方法也可应用于 Raft，但我们将此留待未来工作。

我们使用自己的 Raft 实现来测量 Raft 领导者选举算法的性能，并回答两个问题：第一，选举过程是否能快速收敛？第二，领导者崩溃后能够实现的最小停机时间是多少？

为了测量领导者选举过程，我们反复对五台服务器组成的集群的领导者进行崩溃操作，并记录检测到崩溃并选举出新领导者所需的时间（见图 16）。为了生成

最坏情况，每次试验中各服务器的日志长度不同，因此一些候选者不具备成为领导者资格。此外，为了鼓励出现分裂投票，我们的测试脚本在终止进程前，会触发领导者同步广播心跳 RPC 请求（这模拟了领导者在崩溃前复制新日志条目时的行为）。领导者在心跳间隔内被随机均匀地崩溃，该心跳间隔为所有测试中最小选举超时的一半。因此，最小可能的停机时间约为最小选举超时的一半。

图 16 的上图显示，选举超时中加入少量随机性就足以避免选举中的分裂投票。在没有随机性的情况下，由于存在大量分裂投票，我们的测试中领导者选举时间始终超过 10 秒。加入仅 5 毫秒的随机性可显著改善情况，使中位停机时间降至 287 毫秒。使用更多随机性可以改善最坏情况表现：当随机性为 50 毫秒时，1000 次试验中的最坏情况完成时间（最长耗时）为 513 毫秒。

图 16 的下图显示，通过减小选举超时可以降低停机时间。当选举超时为 12-24 毫秒时，平均选举时间仅需 35 毫秒（最长试验耗时为 152 毫秒）。然而，将超时进一步降低会违反 Raft 的时间要求：领导者在其他服务器开始新选举前难以广播心跳。这可能导致不必要的领导者变更，从而降低系统的整体可用性。我们建议使用保守的选举超时，例如 150-300 毫秒；这类超时不太可能导致不必要的领导者变更，同时仍能提供良好的可用性。

10 相关工作

关于共识算法的研究成果众多，其中许多属于以下几类之一：

- Lamport 对 Paxos 的原始描述 [15]，以及为更清晰地解释该算法所做的尝试 [16, 20, 21]。

- 对 Paxos 的扩展，补充了缺失的细节并修改算法以提供更坚实的基础用于实现 [26, 39, 13]。

- 实现共识算法的系统，例如 Chubby [2, 4]、ZooKeeper [11, 12] 和 Spanner [6]。Chubby 和 Spanner 的算法尚未详细发表，尽管两者均声称基于 Paxos。ZooKeeper 的算法已详细发表，但与 Paxos 差异较大。

- 可应用于 Paxos 的性能优化方法 [18, 19, 3, 25, 1, 27]。

- Oki 和 Liskov 提出的视图加时间复制（VR），这是一种与 Paxos 同时发展起来的共识替代方案。原始描述 [29] 与分布式事务协议交织在一起，但最近的更新中已将核心共识协议分离出来 [22]。VR 采用基于领导者的机制，与 Raft 有许多相似之处。

Raft 与 Paxos 之间最大的区别在于 Raft 的强领导者机制：Raft 将领导者选举作为共识协议中的一个核心组成部分，且它集中于领导者的作用，从而确保了系统的稳定性和可预测性。尽可能在领导者中实现全部功能。这种方法导致了一个更简单的算法，更容易理解。例如，在 Paxos 中，领导者选举与基本共识协议是相互独立的：它仅作为性能优化手段，而非实现共识所必需的。然而，这导致了额

外的机制：Paxos 包含了用于基本共识的两阶段协议以及一个独立的领导者选举机制。相比之下，Raft 将领导者选举直接集成到共识算法中，并将其作为共识两阶段中的第一阶段。这使得 Raft 的机制比 Paxos 更少。

与 Raft 类似，VR 和 ZooKeeper 都是基于领导者的，因此共享了 Raft 相对于 Paxos 的许多优势。然而，Raft 的机制比 VR 或 ZooKeeper 更少，因为它最小化了非领导者中的功能。例如，在 Raft 中，日志条目仅单向流动：从领导者向外通过 AppendEntries RPC 传递。而在 VR 中，日志条目双向流动（领导者在选举过程中可以接收 log 条目）；这导致了额外的机制和复杂性。ZooKeeper 的公开描述也实现了日志条目与领导者之间的双向传输，但其实现方式似乎更接近 Raft [35]。

Raft 比我们所知的任何基于共识的日志复制算法都具有更少的消息类型。例如，我们统计了 VR 和 ZooKeeper 在基本共识和成员变更中使用的消息类型（不包括日志压缩和客户端交互，因为这些与算法几乎无关）。VR 和 ZooKeeper 各自定义了 10 种不同的消息类型，而 Raft 仅有 4 种消息类型（两种 RPC 请求及其响应）。Raft 的消息类型比其他算法稍密集一些，但整体上更简单。此外，VR 和 ZooKeeper 在描述领导者变更时涉及传输整个日志，因此需要额外的消息类型来优化这些机制，使其具有实用性。

Raft 的强领导者方法简化了算法，但排除了一些性能优化。例如，在某些条件下，无领导者的 Egalitarian Paxos（EPaxos）可以通过无领导的方式实现更高的性能 [27]。EPaxos 利用状态机命令的可交换性。只要其他并发提出的命令与该命令可交换，任何服务器只需一轮通信即可提交该命令。然而，如果并发提出的命令彼此不可交换，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，EPaxos 在服务器之间很好地平衡了负载，并能在广域网（WAN）环境下实现比 Raft 更低的延迟。然而，这为 Paxos 增加了显著的复杂性。

其他工作提出了或实现了多种集群成员变更方法，包括 Lamport 的原始提案 [15]、VR [22] 和 SMART [24]。我们选择 Raft 采用联合共识方法，因为这种方法利用了共识协议的其余部分，因此成员变更所需的额外机制非常少。Lamport 的基于 α 的方法不适合 Raft，因为它假设在没有领导者的情况下也能达成共识。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影响正常请求处理的情况下进行；相比之下，VR 在配置变更期间会停止所有正常处理，而 SMART 则对未完成的请求数量施加类似于 α 的限制。Raft 的方法在机制上也比 VR 或 SMART 更简单。

11 结论

算法通常以正确性、效率和/或简洁性为主要设计目标。尽管这些目标都值得追求，但我们认为可理解性同样重要。在开发者将算法转化为实际实现之前，无法实现其他任何目标，而实际实现不可避免地会偏离并扩展原始发表形式。除非开

发者对算法有深刻理解，并能对其形成直观认识，否则他们很难在实现中保留其理想特性。

在本文中，我们解决了分布式共识问题，即一个被广泛接受但难以理解的算法——Paxos，多年来一直困扰着学生和开发者。我们提出了一种新的算法——Raft，并已证明其比 Paxos 更易于理解。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设计 Raft 的方式；随着设计的推进，我们反复使用了一些技术，例如将问题分解和简化状态空间。这些技术不仅提高了 Raft 的可理解性，也使我们更容易确信其正确性。

12 致谢

没有 Ali Ghodsi、David Mazieres 以及伯克利 CS 294-91 和斯坦福 CS 240 学生的支持，用户研究将不可能完成。Scott Klemmer 帮助我们设计了用户研究，Nelson Ray 指导我们进行统计分析。用户研究中的 Paxos 幻灯片大量借鉴了 Lorenzo Alvisi 最初创建的幻灯片。特别感谢 David Mazières 和 Ezra Hoch 在发现 Raft 中的细微错误方面提供的帮助。许多人士对论文和用户研究材料提供了有益的反馈，包括 Ed Bugnion、Michael Chan、Hugues Evrard 等。丹尼尔 · 吉芬，阿琼 · 戈帕兰，乔恩 · 何沃，维马拉库马尔 · 杰亚库马尔，安基塔 · 凯杰里瓦尔，亚历山大 · 克拉库恩，阿米特 · 莱维，乔尔 · 马丁，佐藤硕，奥列格 · 佩索克，大卫 · 拉莫斯，罗伯特 · 范 · 雷内斯，梅德勒 · 罗森布鲁姆，尼古拉斯 · 施皮尔，德伊安 · 斯特凡，安德鲁 · 斯通，瑞安 · 斯图斯曼，大卫 · 特雷伊，史蒂芬 · 杨，马特伊 · 扎哈里亚，24 位匿名会议评审员（含重复项），以及我们的导师爱德 die 科勒。文纳 · 沃格尔在推特上分享了一个早期版本的链接，为 Raft 带来了显著的关注。本工作得到了巨规模系统研究中心和多尺度系统中心的支持，这两个中心是六家在聚焦中心研究计划下获得资助的研究中心之一，该计划由半导体研究公司（Semiconductor Research Corporation）发起，STARnet 项目由半导体研究公司资助，由 MARCO 和 DARPA 支持，同时获得了国家科学基金会（Grant No. 0963859）的资助，以及来自 Facebook、Google、Mellanox、NEC、NetApp、SAP 和三星的资助。Diego Ongaro 由 Junglee 公司斯坦福研究生奖学金支持。

参考文献

[1] Bolosky, W. J., Bradshaw, D., Haagens, R. B., Kusters, N. P., 和 Li, P. 基于 Paxos 复制状态机的高性能数据存储。在 Proc. NSDI'11, 网络化系统设计与实现 USENIX 会议（2011），USENIX，第 141-154 页。

[2] Burrows, M. 为松散耦合分布式系统设计的 Chubby 锁服务。在 Proc. OSDI'06, 操作系统设计与实现研讨会（2006），USENIX，第 335-350 页。

[3] Camargos, L. J., Schmidt, R. M., 和 Pedone, F. 多协调 Paxos。在 Proc. PODC'07, ACM 分布式计算原理研讨会（2007），ACM，第 316-317 页。

[4] Chandra, T. D., Griesemer, R., 和 Redstone, J. Paxos 的实现：一个工程视角。在 Proc. PODC'07, ACM 分布式计算原理研讨会（2007），ACM，第 398-407 页。

[5] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., 和 Gruber, R. E. Bigtable：一种用于结构化数据的分布式存储系统。在 Proc. OSDI'06, 操作系统设计与实现 USENIX 研讨会（2006），USENIX，第 205-218 页。

[6] Corbett, J. C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J. J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P., Hsieh, W., Kanthak, S., Kogan, E., Li, H., Lloyd, A., Melnik, S., Mwacura, D., Nagle, D., Quinlan, S., Rao, R., Rolig, L., Saito, Y., Szymaniak, M., Taylor, C., Wang, R., 和 Woodford, D. Spanner：谷歌的全球分布式数据库。在 Proc. OSDI'12, 操作系统设计与实现 USENIX 会议（2012），USENIX，第 251-264 页。

[7] Cousineau, D., Doligez, D., Lamport, L., Merz, S., Ricketts, D., 和 Vanzetto, H. TLA+ 证明。在 Proc. FM'12, 正式方法研讨会（2012），D. Giannakopoulou 和 D. Méry 编辑，Lecture Notes in Computer Science 卷 7436，Springer，第 147-154 页。

[8] Ghemawat, S., Gobioff, H., 和 Leung, S.-T. Google 文件系统。在 Proc. SOSP'03, 操作系统原理 ACM 研讨会（2003），ACM，第 29-43 页。

[9] Gray, C., 和 Cheriton, D. 租约：分布式文件缓存一致性的高效容错机制。在第 12 届 ACM 操作系统原理研讨会论文集（1989），第 202-210 页。

[10] Herlihy, M. P., 和 Wing, J. M. 线性化：并发对象的正确性条件。ACM 编程语言与系统汇刊 12（1990 年 7 月），第 463-492 页。

[11] Hunt, P., Konar, M., Junqueira, F. P., 和 Reed, B. ZooKeeper：互联网规模系统中的无阻塞协调。在 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：主备系统中的高性能广播。在 Proc. DSN'11, IEEE/IFIP 可靠系统与网络国际会议（2011），IEEE 计算机学会，第 245-256 页。

[13] Kirsch, J., 和 Amir, Y. 为系统构建者设计的 Paxos。技术报告 CNDS-2008-2，约翰霍普金斯大学，2008 年。

[14] Lamport, L. 时间、时钟和分布式系统中的事件排序。Communications of the ACM 21, 7（1978 年 7 月），第 558-565 页。

[15] Lamport, L. 部分时间议会。ACM 计算机系统汇刊 16, 2（1998 年 5 月），第 133-169 页。

[16] Lamport, L. Paxos 的简化版。ACM SIGACT 新闻 32, 4（2001 年 12 月），第 18-25 页。

[17] Lamport, L. 系统规范：TLA+ 语言及硬件和软件工程师的工具。Addison-Wesley，2002 年。

[18] Lamport, L. 一般共识与 Paxos。技术报告 MSR-TR-2005-33，微软研究院，2005 年。

[19] Lamport, L. 快速 Paxos。分布式计算 19, 2（2006 年），第 79-103 页。

[20] Lampson, B. W. 如何使用共识构建高可用系统。在分布式算法，O. Babaoglu 和 K. Marzullo 编辑，Springer-Verlag，1996 年，第 1-17 页。

[21] Lampson, B. W. Paxos 的 ABCD。在 Proc. PODC'01, ACM 分布式计算原理研讨会（2001），ACM，第 13-13 页。

[22] Liskov, B., 和 Cowling, J. 视图标记复制的再审视。技术报告 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., 和 Howell, J. 复制状态服务的 SMART 迁移方法。在 Proc. EuroSys'06, ACM SIGOPS/EuroSys 计算机系统欧洲会议（2006），ACM，第 103-115 页。

[25] Mao, Y., Junqueira, F. P., 和 Marzullo, K. Mencius：为广域网构建高效的复制状态机。在 Proc. OSDI'08, USENIX 操作系统设计与实现会议（2008），USENIX，第 369-384 页。

[26] Mazieres, D. Paxos 实用化。http: //www.scs.stanford.edu/ dm/home/ papers/paxos.pdf 2007 年 1 月。

[27] Moraru, I., Andersen, D. G., 和 Kaminsky, M. 在平等议会中存在更多的共识。在 Proc. SOSP'13, ACM 操作系统原理研讨会（2013），ACM。

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

[29] Oki, B. M., 和 Liskov, B. H. 视图复制：一种支持高可用分布式系统的主副本方法。在 Proc. PODC'88, ACM 分布式计算原理研讨会（1988），ACM，第 8-17 页。

[30] O'Neil, P., Cheng, E., Gawlick, D., 和 O'Neil, E. 日志结构合并树（LSM-tree）。Acta Informatica 33, 4（1996），351-385。

[31] Ongaro, D. 共识：连接理论与实践。斯坦福大学博士论文，2014 年（工作进行中）。

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

[32] Ongaro, D., 和 Ousterhout, J. 寻找一个易于理解的共识算法。在 Proc ATC'14, USENIX 年度技术会议（2014），USENIX。

[33] Ousterhout, J., Agrawal, P., Erickson, D., Kozyrakis, C., Leverich, J., Mazières, D., Mitra, S., Narayanan, A., Ongaro, D., Parulkar, G., Rosenblum, M., Rumble, S. M., Stratmann, E., 和 Stutsman, R. RAMCloud 的必要性。Communications of the ACM 54（2011 年 7 月），121-130。

[34] Raft 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., 和 Ousterhout, J. K. 一种日志结构文件系统的設計與實現。ACM Trans. Comput. Syst. 10（1992 年 2 月），26-52。

[37] Schneider, F. B. 使用状态机方法实现容错服务：一个教程。ACM 计算机综述 22, 4（1990 年 12 月），299-319。

[38] Shvachko, K., Kuang, H., Radia, S., 和 Chansler, R. Hadoop 分布式文件系统。在 Proc. MSST'10, 质量存储系统与技术研讨会（2010），IEEE 计算机学会，第 1-10 页。

[39] van Renesse, R. Paxos 的适度复杂化。技术报告，康奈尔大学，2012 年。