寻找一种易于理解的一致性算法（扩展版）

*本文翻译来自 https://knowledge-sharing.gitbooks.io/raft/content/chapter0.html（第 1,2,3,5 部分）和 https://github.com/maemual/raft-zh_cn（第 4,6,7,8,9,10,11,12 部分）。下划线部分与原文略有不同。（/**/）表示注释和解释，并不是原文和译文中的内容。*

摘要

Raft 是一种用来管理日志复制的一致性算法。它提供了和 Paxos 算法相同的功能和性能，但是它的算法结构和 Paxos 不同，这使得 Raft 比 Paxos 更易于理解，并为建立实际的系统提供了更好的基础。为了提升可理解性，Raft 将一致性算法分解成了几个关键模块，例如领导人选举（*Leader Selection*）、日志复制（*Log Replication*）和安全（*Safety*），并且它通过减少需要考虑的状态数，来加强一致性。通过对用户学习效果的测试表明，Raft 比 Paxos 更容易学习。Raft 还包括了一种动态改变集群成员的新机制，它使用重叠大多数（*Overlapping Majorities*）的方式来保证安全。

1 导言

一致性算法允许一组服务器像一个整体一样工作，该整体能够使它的成员从失败中恢复正常。正因为此，它们在构建可靠的大规模软件系统过程中起着关键作用。过去十多年来，Paxos[15,16]在有关一致性算法的讨论中处于主导地位：大多数一致性算法的实现都基于 Paxos 或者受它影响，并且 Paxos 也成为有关一致性知识教学的主要工具。

不幸的是，Paxos 非常难于理解，尽管在使其更容易上手方面做了大量尝试。而且 Paxos 的架构需要经过复杂的修改才能用于实际的系统。结果，无论是系统构建者还是学生，都饱受折磨。

我们在经受 Paxos 折磨之后，便开始着手寻找一种新的一致性算法，希望它能为系统构建和教学奠定更好的基础。我们的做法不同于以往，因为首要的目标就是可理解性：我

们是否能为实际的系统实现定义一种一致性算法，用一种比 Paxos 更容易学习的方式来描述它？并且，我们希望这种算法能够使开发变得更加容易，这对系统构建者从直觉上来理解是非常关键的。重要的是不仅仅是算法能够运行，而且还要使它的运行原理更易于理解。

我们工作的结果是找到了一种新的一致性算法，叫做 Raft。在设计 Raft 时我们使用了特殊的技巧来提高可理解性，包括分解（Raft 分解为领导人选举、日志复制与安全）和削减状态（相对于 Paxos，Raft 降低了不确定性和服务器之间互相不一致的方式）。在一个包含两所大学的 43 个学生的调研中发现，Raft 比 Paxos 更加容易理解：在学习了两种算法之后，其中 33 个学生解答 Raft 的问题要比 Paxos 好很多。

Raft 算法和现存的一致性算法在很多地方都类似（主要是 Oki 和 Liskov 的 View stamped Replication[29,22]，但 Raft 具有如下几个新特性：

- 强领导人（*Strong Leader*）：相比于其他算法，Raft 使用了更强的领导人形式。比如，日志条目只从领导人发送到其他服务器。这样简化了对日志复制的管理，并使得 Raft 更易于理解。
- 领导人选举（*Leader Selection*）：Raft 使用随机定时器来选举领导人。这种方式只是对心跳机制增加了少量改动，而心跳对所有一致性算法来说都是必需的，但却能够更加简单、快速地解决冲突。
- 成员变更（*Membership Change*）：Raft 使用了一种新的称为联合共识（*Joint Consensus*）的方式，来处理集群中一组服务器的成员变更问题。该方式中，两个不同配置（/*新配置和旧配置*/）中的大多数服务器是重叠的（*Overlap*）。这使得在配置变更过程中，集群仍然能够继续正常运行。

我们认为，在达成教学目标和为实现算法提供基础这两个方面，Raft 要优于 Paxos 及其他算法。Raft 比其他算法更简单，并且更易于理解；Raft 具有更加完整的描述，能够满足构建一个实际系统的需要；Raft 拥有许多开源实现，并且被许多公司所采用；Raft 的安全性已经被正式提出和证明；同时，Raft 在效率方面也可以与其他算法相媲美。

这篇论文的剩余部分会介绍复制状态机（*Replicated State Machine*）问题（第 2 部分），讨论 Paxos 的优缺点（第 3 部分），描述我们在提高可理解性上采取的通用方法（第 4 部分），陈述 Raft 一致性算法（第 5-8 部分），给出对 Raft 算法的评估（第 9 部分），最后讨论了相关工作（第 10 部分）。

2 复制状态机

一致性算法是在复制状态机（*Replicated State Machine*）的背景下提出来的。在这个方法中，一组服务器的状态机计算（/*执行*/）相同状态的相同副本（/*可以理解为指令*/），即使有一部分服务器宕机了，它们仍然能够继续运行。在分布式系统中，复制状态机被用来解决各种容错问题。具有单个集群领导人的大规模系统，例如 GFS[8]、HDFS[38]、RAMCloud[33]，一般使用一个单独的复制状态机来管理领导人选举和存储配置信息，必须能够使领导人从故障（*Crashes*）中恢复。使用复制状态机的例子包括 Chubby[2]和 ZooKeeper[11]。

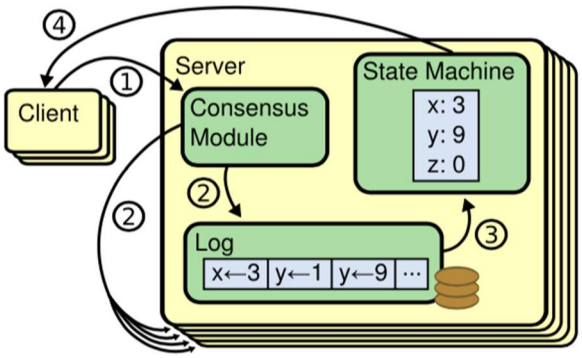

图-1：复制状态机架构。一致性算法管理来自客户端的状态机命令组成的复制日志。状态机处理日志中相同顺序的命令序列，因此会输出相同的结果。

如图-1 所示，一般通过使用复制日志来实现复制状态机。每个服务器存储着一份包含命令序列的日志文件，状态机会按顺序执行这些命令。因为每个日志包含相同的命令，并且顺序也相同，所以每个状态机处理相同的命令序列。由于状态机是确定性的，所以处理相同的状态，得到相同的输出。

保证复制日志的一致性是一致性算法的任务。一个服务器上的一致性模块会接收来自客户端的命令，并把命令添加到它的日志文件中。它同其他服务器上的一致性模块进行通信，确保每一个日志最终包含相同的请求（/*命令*/）且顺序也相同，即使某些服务器故障。一旦这些命令被正确复制，每个服务器的状态机都会按照日志中的顺序去处理，将输出结果返回给客户端。最终，这些服务器看起来就像一个单独的、高可靠的状态机。

对于实际系统，一致性算法一般具有如下特点：

- 安全。满足非拜占庭条件下的安全（从来不会返回错误结果），包括网络延迟、分区、丢包、重复和重排序。
- 高可用。只要集群中的大部分服务器正常运行，且能够互相通信，也可以同客户端通信，这个集群就完全可用。拥有 5 个服务器的集群可以容忍 2 个服务器故障。假设服务器是 fail-stop 故障（/*通过故障检测可以知道服务器崩溃*/）。服务器可以通过读取原来保存在稳定存储中的状态数据恢复，并重新加入集群。
- 不依赖于时序。不依赖时序保证日志的一致性。如果依赖时序，在最坏的情况下可造成可用性问题（如时钟错误、极端情况下的消息延迟）。
- 通常情况下，只要集群中大多数服务器成功响应了某一轮 RPC 调用，一个命令就算完成。少数较慢的服务器不影响整个系统性能。

3 Paxos 存在问题吗

在过去的 10 多年中，Leslie Lamport 的 Paxos 协议[15]几乎成为一致性算法的代名词：它是教学中最常用到的算法，同时，很多一致性算法实现都基于 Paxos。首先，Paxos 定义了能够在单一决策上达成一致的协议，比如单个日志条目。我们把这个子集称为单决策 Paxos。然后合并这个协议的多个实例来实现一系列决策，该决策形成日志文件（多决策 Paxos）。Paxos 能够确保安全性（*Safety*）和活性（*Liveness*）（/*并不能确保活性，参考 FLP*/），并支持集群成员变更。它的正确性已被证明，通常情况下也非常有效。

不幸的是，Paxos 存在两个致命的缺陷。第一个是 Paxos 非常难懂，其完整的解释很少有人能完全理解，只有少数人付出巨大努力才能理解 Paxos。因此，很多人尝试用一些简单的术语来解释 Paxos[16,20,21]。这些解释都集中在单决策子集，但仍是很有挑战的。在 NSDI2012 会议上的一次非正式调查显示，很少有人熟悉 Paxos，即使是一些有经验的研究人员也不例外。我们自己也曾受其折磨，直到我们读过几篇对 Paxos 简化理解的文章，并设计了自己的算法之后，才算理解了整个 Paxos 协议，然而这个过程花费了将近一年的时间。

我们假定 Paxos 的晦涩来源于，它以单决策子集为基础的多决策达成。单决策 Paxos 晦涩难懂：它被分为两个阶段，但没有简单直观的解释，且难以单独去理解。正因为如此，它很难直观地说明为什么单决策协议是有效的。多决策 Paxos 的规则进一步增加了算法的复杂性。我们认为，在多决策上达成一致的问题，可以采用其他更加直接和明显的方式来进行分解。

Paxos 的第二个缺陷是，它难以为实际的系统实现提供良好的基础。原因之一是，业界并没有广泛的对多决策 Paxos 算法进行商讨。Lamport 的描述大部分都是关于单决策的 Paxos，他只是给出了实现多决策 Paxos 的一些可能的方式，缺少很多细节。有许多实现和优化 Paxos 的尝试，比如[26]、[39]和[13]，但它们都互不相同，且与 Lamport 给出的说法也不同。像 Chubby 系统已经实现了类似 Paxos 的算法，但是大多数情况下没有披露更多的细节。

此外，Paxos 架构不易于构建实际的系统，这也是单决策分解带来的另一个后果。例如，独立地挑选出一组日志条目，然后把它们变成一个连续的日志，并没有带来什么好处，反倒增加了复杂性。基于日志来设计系统更加简单而有效，新的日志条目只需按照指定的顺序追加到日志中去。另一个问题是，Paxos 使用对等的 peer-to-peer 方式作为其核心（/*服务器是对等的*/）（尽管最终提出了一种使用弱领导人形式来优化性能的建议）。这种方法只有在做出单个决策的情况下才有意义，但几乎没有实际系统会选择使用这种方式。如果需要做一系列决策，简单高效的做法是：先选出一个领导人，然后由领导人来协调做出决策。

因此，构建实际系统无法从 Paxos 获取更大的益处。每个始于 Paxos 的实现，都会发现很难实现它，然后开发了一个完全不同的架构。这很耗时而且容易出错，理解 Paxos 的难度又加剧了这个问题。证明理论的正确性，Paxos 公式可能是一个很好的选择，但是由于实现上有很大的不同，Paxos 理论证明根本体现不出价值。下面是来自 Chubby 实现的一条有代表性的评论：

Paxos 算法的描述与实际实现之间存在巨大的鸿沟…最终的系统将会基于一个没有被证明的协议而构建起来。

由于这些问题，我们认为 Paxos 没有为系统构建或者教学提供一个良好的基础。鉴于一致性算法对于构建大规模软件系统的重要性，我们决定尝试设计一种比 Paxos 更好的一致性算法，Raft 算法就这样被设计出来。

4 可理解性设计

设计 Raft 算法，我们有几个初衷：它必须提供一个完整的实际系统的实现基础，这样才能大大减少开发者的工作；它必须在任何情况下都是安全的，并且在大多数的情况下都是可用的；并且它的大部分操作必须是高效的。但是我们最重要的目标，也是最大的挑战，是可理解性。它必须保证对于大多数人都可以容易地理解。另外，它必须能够让人形成直观的认识，这样，系统的构建者才能够在实现中进行扩展。

在设计 Raft 算法的时候，有很多方面（*point*）需要我们在各种备选方案中进行选择。在这种情况下，我们评估备选方案基于可理解性原则：解释各个备选方案有多大的难度（例如，Raft 的状态空间有多复杂，是否有晦涩的含义（*implications*））？对于一个读者而言，完全理解这个方案和含义是否容易？

我们意识到这种可理解性分析，具有高度的主观性；因此，我们使用了两种通常适用的技术来解决这个问题。第一个技术就是众所周知的问题分解：只要有可能，我们就将问题分解成几个相对独立的，可解决的、可解释的和可理解的子问题。例如，我们将 Raft 算法分成领导人选举、日志复制、安全和角色改变几个部分。

我们使用的第二个方法是通过减少状态的数量，来简化需要考虑的状态空间，使得系统更加连贯，并且尽可能消除不确定性。特别的，所有的日志是不允许有空洞的，并且 Raft 限制了日志之间变成不一致状态的可能。尽管在大多数情况下我们都试图去消除不确定性，但是也有一些情况下，不确定性可以提升可理解性。尤其是，随机方法增加了不确

定性，但是他们有利于减少状态空间数量，通过处理所有可能选择时使用相似的方法。我们使用随机方法简化 Raft 中领导人选举算法。

5 Raft 一致性算法

Raft 是一种用来管理章节 2 中描述的复制日志的算法。图 2 为了参考之用，总结这个算法的简略版本，图 3 列举了这个算法的一些关键特性。图中的这些元素会在剩下的章节逐一介绍。

Raft 通过选举一个杰出的领导人，然后给予他全部的管理复制日志的责任来实现一致性。领导人从客户端接收日志条目(*Log Entries*)，把日志条目复制到其他服务器上，并且告诉其他的服务器，在保证安全性的情况下，应用日志条目到他们的状态机中。拥有一个领导人大大简化了对复制日志的管理。例如，领导人可以决定新的日志条目需要放在日志中的什么位置，而不需要和其他服务器商议，并且数据都从领导人流向其他服务器。一个领导人可能会宕机，或者和其他服务器失去连接，在这种情况下新的领导人会被选举出来。

通过领导人方式，Raft 将一致性问题分解成了三个相对独立的子问题，这些问题会在接下来的子章节中进行讨论：

- 领导人选举：当旧的领导人宕机的时候，一个新的领导人必须被选举出来（章节 5.2）。
- 日志复制：领导人必须从客户端接收日志条目，然后复制到集群中的其他节点，并且强制要求其他节点的日志保持和领导人相同。
- 安全性：在 Raft 中，安全性的关键是在图 3 中展示的状态机安全：如果有任何的服务器节点已经应用了一个确定的日志条目到它的状态机中，那么其他服务器节点不能在同一个日志索引位置应用一个不同的指令。章节 5.4 阐述了 Raft 算法是如何保证这个特性的；解决方案涉及到一个额外的选举机制（5.2 节）上的限制。

在展示一致性算法之后，这一章节会讨论可用性的一些问题和计时在系统的作用。

状态（*State*）：

所有服务器上的持久性状态(在响应 RPC 请求之前已经更新到了稳定的存储设备上):

参数	解释
currentTerm	服务器已知最新的任期（在服务器首次启动的时候初始化为 0，单调递增）
votedFor	当前任期内收到选票的候选者 id。如果没有投给任何候选者则为空。
log[]	日志条目；每个条目包含了用于状态机的命令，以及领导人接收到该条目时的任期（第一个索引为 1）

所有服务器上的易失性状态:

参数	解释
commitIndex	已知已提交的最高的日志条目的索引（初始值为 0，单调递增）
lastApplied	已经被应用到状态机的最高的日志条目的索引（初始值为 0，单调递增）

领导人（服务器）上的易失性状态(选举后已经重新初始化):

参数	解释
nextIndex[]	对于每一台服务器，发送到该服务器的下一个日志条目的索引（初始值为领导人最后的日志条目的索引+1）

参数	解释
matchIndex[]	对于每一台服务器，已知的已经复制到该服务器的最高日志条目的索引（初始值为 0，单调递增）

追加条目 RPC（*AppendEntries RPC*）：

由领导人调用用于日志条目的复制同时也被当做心跳使用

参数	解释
term	领导人的任期
leaderId	领导人 ID。跟随者可以对客户端进行重定向（译者注：跟随者根据领导人 id 把客户端的请求重定向到领导人，比如有时客户端把请求发给了跟随者而不是领导人）
prevLogIndex	紧邻新日志条目之前的那个日志条目的索引
prevLogTerm	紧邻新日志条目之前的那个日志条目的任期
entries[]	需要被保存的日志条目（被当做心跳使用时，则日志条目内容为空；为了提高效率可能一次性发送多个）
leaderCommit	领导人的已知已提交的最高的日志条目的索引

返回值	解释
term	当前任期，对于领导人而言，它会更新自己的任期

返回值	解释
success	结果为真，如果跟随者所含有的条目和 prevLogIndex 以及 prevLogTerm 匹配上了

接收者的实现：

1) 返回假。如果领导人的任期小于接收者的当前任期（译者注：这里的接收者是指跟随者或者候选者）（5.1 节）。
2) 返回假。如果接收者日志中没有包含这样一个条目：即该条目的任期在 prevLogIndex 上能和 prevLogTerm 匹配上。（译者注：在接收者日志中如果能找到一个和 prevLogIndex 以及 prevLogTerm 一样的索引和任期的日志条目，则继续执行下面的步骤，否则返回假）（5.3 节）。
3) 如果一个已经存在的条目和新条目（译者注：即刚刚接收到的日志条目）发生了冲突（因为索引相同，任期不同），那么就删除这个已经存在的条目以及它之后的所有条目（5.3 节）。
4) 追加日志中尚未存在的任何新条目。
5) 如果领导人的已知已经提交的最高的日志条目的索引 leaderCommit 大于接收者的已知已经提交的最高的日志条目的索引 commitIndex，则把接收者的已知已经提交的最高的日志条目的索引 commitIndex，重置为领导人的已知已经提交的最高的日志条目的索引 leaderCommit，或者是上一个新条目的索引，取两者的最小值。

请求投票 RPC（*RequestVote RPC*）：

由候选者负责调用用来征集选票（5.2 节）

参数	解释
term	候选者的任期号
candidateId	请求选票的候选者的 Id
lastLogIndex	候选者最后日志条目的索引值
lastLogTerm	候选者最后日志条目的任期号

返回值	解释
term	当前任期号，以便于候选者去更新自己的任期号
voteGranted	候选者赢得了此张选票时为真

接收者实现：

1. 如果 term < currentTerm 返回 false（5.2 节）。
2. 如果 votedFor 为空或者为 candidateId，并且候选者的日志至少和自己一样新，那么就投票给他（5.2 节，5.4 节）。

所有服务器需遵守的规则：

所有服务器：

- 如果 commitIndex > lastApplied，那么就 lastApplied 加一，并把 log[lastApplied]应用到状态机中（5.3 节）
- 如果接收到的 RPC 请求或响应中，任期号 T > currentTerm，那么就令 currentTerm 等于 T，并切换状态为跟随者（5.1 节）。

跟随者（5.2 节）：

- 响应来自候选者和领导人的请求
- 如果在超过选举超时时间的情况之前，没有收到当前领导人（即该领导人的任期需与这个跟随者的当前任期相同）的心跳/附加日志，或者是给某个候选者投了票，就自己变成候选者。

候选者（5.2 节）：

- 在转变成候选者后就立即开始选举过程
    - 自增当前的任期号（currentTerm）
    - 给自己投票
    - 重置选举超时计时器
    - 发送 RequestVote RPC 给其他所有服务器
- 如果接收到大多数服务器的选票，那么就变成领导人
- 如果接收到来自新的领导人的 AppendEntries RPC，转变成跟随者
- 如果选举过程超时，再次发起一轮选举

领导人：

- 一旦成为领导人：发送空的 AppendEntries RPC（心跳）给其他所有的服务器；在一定的空余时间之后不停的重复发送，以阻止跟随者超时（5.2 节）
- 如果接收到来自客户端的请求：附加条目到本地日志中，在条目被应用到状态机后响应客户端（5.3 节）
- 如果对于一个跟随者，最后日志条目的索引值大于等于 nextIndex，那么：发送从 nextIndex 开始的所有日志条目：
    - 如果成功：更新相应跟随者的 nextIndex 和 matchIndex

- 如果因为日志不一致而失败，减少 nextIndex 重试

- 假设存在大于 commitIndex 的 N，使得大多数的 matchIndex[i]≥N 成立，且 log[N].term==currentTerm 成立，则令 commitIndex 等于 N（5.3 和 5.4 节）

图 2：一个关于 Raft 一致性算法的浓缩总结（不包括成员变换和日志压缩）。

特性	解释
选举安全特性	对于一个给定的任期号，最多只会有一个领导人被选举出来（5.2 节）
领导人只附加原则	领导人绝对不会删除或者覆盖自己的日志，只会增加（5.3 节）
日志匹配原则	如果两个日志在某一相同索引位置日志条目的任期号相同，那么我们就认为这两个日志从头到该索引位置之间的内容完全一致（5.3 节）
领导人完全特性	如果某个日志条目在某个任期号中已经被提交，那么这个条目必然出现在更大任期号的所有领导人中（5.4 节）
状态机安全特性	如一服务器已将给定索引位置的日志条目应用至其状态机中，则其他任何服务器在该索引位置不会应用不同的日志条目（5.4.3 节）

图 3：Raft 在任何时候都保证以上的各个特性。

5.1 Raft 基础

一个 Raft 集群包括若干服务器。通常是 5 个，这样，集群能够容忍 2 个服务器发生故障。在任意的时间，每个服务器会处于以下三种状态之一：领导人（*Leader*）、候选者

（*Candidate*）、跟随者（*Follower*）。在正常情况下，只有一个服务器是领导人，剩下的服务器是跟随者。跟随者是被动的：它们不会发送任何请求，只是响应来自领导人和候选者的请求。领导人处理来自客户端的所有请求（如果一个客户端与跟随者进行通信，跟随者会将信息发送领导人）。候选者是用来选取一个新的领导人，这一部分会在 5.2 节进行阐释。图-4 阐述了这些状态，以及它们之间的转换；它们的转换会在下面进行讨论。

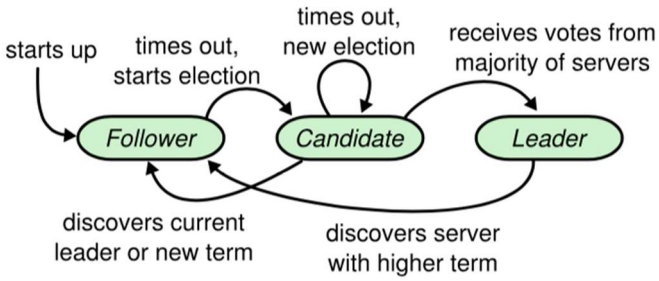

图-4：服务器的状态。跟随者只响应其他服务器的请求。如果跟随者没有收到任何消息，它会成为一个候选者并且开始一次选举。收到大多数服务器投票的候选者，会成为新的领导人。领导人在它们宕机之前会一直保持领导人的状态。

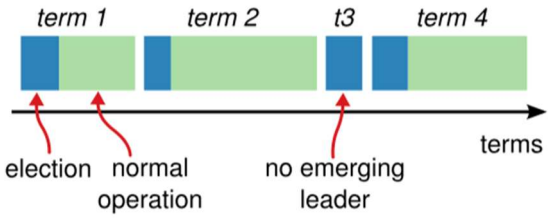

图-5：时间被分为一个个的任期（Term），每一个任期的开始都是领导人选举。在成功选举之后，一个领导人会在任期内管理整个集群。如果选举失败，该任期就会因为没有领导人而结束。这个转变会在不同的时间的不同服务器上观察到。

如图-5 所示，Raft 算法将时间划分成为任意不同长度的任期（*Term*）。任期用连续的数字进行表示。每一个任期的开始都是一次选举，就像 5.2 节所描述的那样，一个或多个候选者试图成为领导人。如果一个候选者赢得了选举，它就会在该任期的剩余时间担任领

导人。在某些情况下，选票会被瓜分，有可能没有选出领导人，那么，将会开始另一个任期，并且立刻开始下一次选举。Raft 算法保证在给定的一个任期最少要有一个领导人。

不同的服务器可能会在任期内观察到多次不同的状态转换，在某些情况下，一台服务器可能看不到一次选举或者一个完整的任期。任期，在 Raft 中充当逻辑时钟的角色，并且它们允许服务器检测过期的信息，比如旧领导人。每一台服务器都存储着一个当前任期的数字，这个数字会单调的增加。当服务器之间进行通信时，会互相交换当前任期号；如果一台服务器的当前任期号比其他服务器的小，则更新为较大的任期号。如果一个候选者或者领导人意识到它的任期号过时了，它会立刻转换为跟随者状态。如果一台服务器收到的请求的任期号是过时的，那么它会拒绝此次请求。

Raft 中的服务器通过远程过程调用（RPC，Remote Procedure Call）来通信，基本的 Raft 一致性算法仅需要 2 种 RPC。RequestVote RPC 是候选者在选举过程中触发的（5.2 节），AppendEntries RPC 是领导人触发的，为的是复制日志条目和提供一种心跳（*Heartbeat*）机制（5.3 节）。第 7 章加入了第三种 RPC 来在各个服务器之间传输快照（*Snapshot*）。如果服务器没有及时收到 RPC 的响应，它们会重试，并且它们能够并行的发出 RPC 来获得最好的性能。

5.2 领导人选举

Raft 使用一种心跳机制来触发领导人的选举。当服务器启动时，它们会初始化为跟随者。一台服务器会一直保持跟随者的状态，只要它们能够收到来自领导人或者候选者的有效 RPC。领导人会向所有跟随者周期性发送心跳（不带有任何日志条目的 AppendEntries RPC）来保证它们的领导人地位。如果一个跟随者在一个周期内没有收到心跳信息，就叫做选举超时，然后它就会认为没有可用的领导人，并且开始一次选举以选出一个新的领导人。

为了开始选举，一个跟随者会自增它的当前任期并且转换状态为候选者。然后，它会给自己投票并且给集群中的其他服务器发送 RequestVote RPC。一个候选者会一直处于该状态，直到下列三种情形之一发生：

- 它赢得了选举；
- 另一台服务器赢得了选举；
- 一段时间后没有任何一台服务器赢得了选举。

这些情形会在下面的章节中分别讨论。

如果一个候选者在一个任期内收到了来自集群中大多数服务器的投票，就会赢得选举。在一个任期内，一台服务器最多能给一个候选者投票，按照先到先服务原则（注意：在 5.4 节针对投票添加了一个额外的限制）。大多数原则使得在一个任期内最多有一个候选者能赢得选举（表-3 中提到的选举安全原则）。一旦有一个候选者赢得了选举，它就会成为领导人。然后，它会向其他服务器发送心跳信息来建立自己的领导人地位，并且阻止新的选举。

当一个候选者等待别人的选票时，它有可能会收到来自其他服务器发来的声明其为领导人的 AppendEntries RPC。如果这个领导人的任期（包含在它的 RPC 中）比当前候选者的当前任期要大，则候选者认为该领导人合法，并且转换自己的状态为跟随者。如果在这个 RPC 中的任期小于候选者的当前任期，则候选者会拒绝此次 RPC，继续保持候选者状态。

第三种情形是，一个候选者既没有赢得选举，也没有输掉选举：如果许多跟随者在同一时刻都成为了候选者，选票会被分散，可能没有候选者能获得大多数的选票。当这种情形发生时，每一个候选者都将会超时，并且通过自增任期号和发起另一轮 RequestVote

RPC 来开始新的选举。然而，如果没有其他手段来分配选票的话，这种情形可能会无限的重复下去。

Raft 使用随机的选举超时时间来确保第三种情形很少发生，并且能够快速解决。为了防止在一开始是选票就被瓜分，选举超时时间是在一个固定的间隔内随机选出来的（例如 150-300ms）。这种机制使得各个服务器能够分散开来，在大多数情况下只有一个服务器会率先超时；它会在其他服务器超时之前赢得选举，并且向其他服务器发送心跳信息。同样的机制被用于选票被瓜分的情况。每一个候选者在开始一次选举的时候会重置一个随机的选举超时时间，等待直到超时后，再进行下一次选举。这能够减小在新的选举中一开始选票就被瓜分的可能性。9.3 节展示了这种方法能够快速的选出一个领导人。

选举，是一个引导我们设计替代算法具备可理解性的例子。最开始时，我们计划使用一种排名系统：给每一个候选者分配一个唯一的排名，用于在竞争的候选者之中选出领导人。如果一个候选者发现了另一个比它排名高的候选者，那么它会回到跟随者的状态，这样排名高的候选者会很容易地赢得选举。但是我们发现这种方法在可用性方面有一点问题（一个低排名的服务器在高排名的服务器宕机后，需要等待超时才能再次成为候选者，但是如果它这么做的太快，它能重置选举领导人的过程）。我们对这个算法做了多次调整，但是每次调整后都会出现一些新的问题。最终我们认为随机重试的方法是更明确的，并且更易于理解。

5.3 日志复制

一旦选出了领导人，它就开始接收客户端的请求。每一个客户端请求都包含一条需要被复制状态机（*Replicated State Machine*）执行的命令。领导人把这条命令作为新的日志条

目加入到它的日志中去，然后并行的向其他服务器发起 AppendEntries RPC，要求其他服务器复制这个条目。当这个条目被安全的复制之后（下面的部分会详细阐述），领导人会将这个条目应用到它的状态机中并且会向客户端返回执行结果。如果跟随者崩溃了或者运行缓慢或者是网络丢包了，领导人会无限的重试 AppendEntries RPC（甚至在它向客户端响应之后），直到有的跟随者最终存储了所有的日志条目。

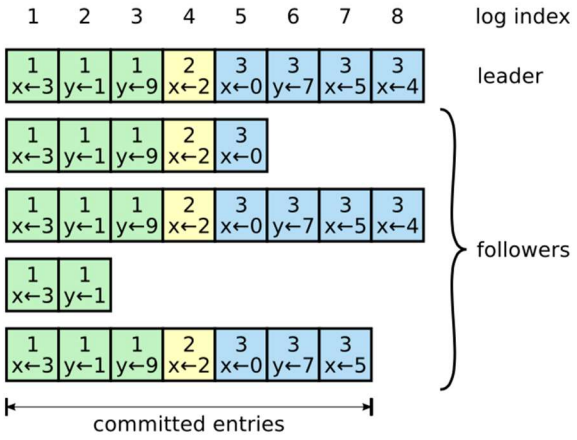

图-6：日志由有序编号的日志条目组成。每个日志条目包含它被创建时的任期号（每个方块中的数字），并且包含用于状态机执行的命令。如果一个条目能够被状态机安全执行，就被认为可以提交了。

日志就像图-6 所示那样组织的。每个日志条目存储着一条被状态机执行的命令，以及当这条日志条目被领导人接收时的任期号。日志条目中的任期号用来检测在不同服务器上日志的不一致性，并且能确保图-3 中的一些特性。每个日志条目也包含一个整数索引来表示它在日志中的位置。

领导人决定什么时候将日志条目应用到状态机是安全的；这种条目被称为是已提交的（*Committed*）。Raft 保证已提交的日志条目是持久化的，并且最终会被所有可用的状态机执行。一旦被领导人创建的条目已经复制到了大多数的服务器上，这个条目就称为已提交的（例如，图-6 中的 7 号条目）。领导人日志中之前的条目都是已提交的，包括由之前的

领导人创建的条目。5.4 节将会讨论，当领导人更替之后应用这条规则的微妙之处，并且也会讨论这种提交（*Commitment*）的定义是安全的。领导人跟踪记录它所知道的已提交的条目的最大索引值，并且这个索引值会包含在之后的 AppendEntries RPC 中（包括心跳中），为的是让其他服务器都知道这个条目已经提交。一旦一个跟随者知道了一个日志条目已经是已提交的，它会将该条目应用至本地的状态机（按照日志顺序）。

我们设计了 Raft 日志机制来保证不同服务器上日志的一致性。这样做不仅简化了系统的行为，使得它更可预测，并且也是保证安全性不可或缺的一部分。Raft 保证以下特性，并且也保证了表-3 中的日志匹配原则（*Log Matching Property*）：

- 如果在不同日志中的两个条目有着相同的索引和任期号，则它们所存储的命令是相同的。
- 如果在不同日志中的两个条目有着相同的索引和任期号，则它们之间的所有条目都是完全一样的。

第一条特性源于领导人在一个任期里，在给定的一个日志索引位置最多创建一条日志条目，同时该条目在日志中的位置也从来不会改变。第二条特性源于追加日志（*AppendEntries*）的一个简单的一致性检查。当发送一个 AppendEntries RPC 时，领导人会把紧邻新日志条目之前的日志条目的索引和任期号都包含在里面。如果跟随者没有在它的日志中找到相同索引和任期号的日志，它就会拒绝新日志条目。这个一致性检查就像一个归纳步骤：一开始空的日志的状态一定是满足日志匹配原则的，一致性检查保证了当日志添加时的日志匹配原则。因此，只要追加日志（*AppendEntries*）返回成功的时候，领导人就知道跟随者的日志和它的是一致的了。

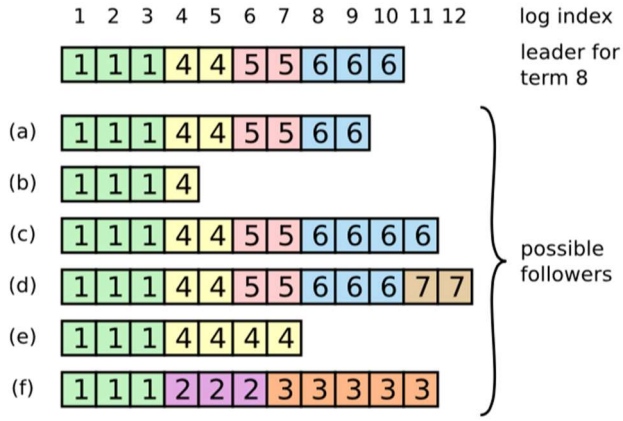

图-7：当一个领导人成功当选时，跟随者日志可能有以下情况（a~f）。一个格子表示一个日志条目；格子中的数字是它的任期。一个跟随者可能会丢失一些条目（a,b）；可能多出来一些未提交的条目（c,d）；或者两种情况都有（e,f）。例如，场景 f 在如下情况下就会发生：如果一台服务器在任期 2 时是领导人，并且向它的日志中添加了一些条目，然后在将它们提交之前就宕机了，之后它很快重启了，成为了任期 3 的领导人，又向它的日志中添加了一些条目，然后在任期 2 和任期 3 中的条目提交之前，它又宕机了，并且几个任期内都一直处于宕机状态。

在一般情况下，领导人和跟随者的日志保持一致，因此追加日志（*AppendEntries*）一致性检查通常不会失败。然而，领导人的崩溃会导致日志不一致（旧的领导人可能没有完全复制完日志中的所有条目）。这些不一致会导致一系列领导人和跟随者崩溃。图-7 阐述了一些跟随者可能和新的领导人日志不同的情况。一个跟随者可能会丢失掉领导人上的一些条目，也有可能包含领导人没有的一些条目，也有可能两者都会发生。丢失的或者多出来的条目可能会持续多个任期。

在 Raft 算法中，领导人通过强制跟随者复制它的日志来处理日志的不一致。这就意味着，跟随者的冲突日志会被领导人的日志覆盖。5.4 节会说明当添加了一个额外的限制之后这是安全的。

为了使得跟随者的日志和自己的一致，领导人需要找到跟随者与它的日志一致的地方，删除跟随者在该位置之后的条目，然后将自己在该位置之后的条目发送给跟随者。这些操作都在 AppendEntries RPC 进行一致性检查时完成。领导人给每一个跟随者维护了一个 nextIndex，它表示领导人将要发送给该追随者的下一条日志条目的索引。当一个领导人开始掌权时，它会将 nextIndex 初始化为它的最新的日志条目索引数+1（图-7 中的 11）。如果一个跟随者的日志和领导人的不一致，追加日志（*AppendEntries*）一致性检查会在下一次 AppendEntries RPC 时返回失败。在失败之后，领导人会将 nextIndex 递减然后重试 AppendEntries RPC。最终 nextIndex 会达到一个领导人和跟随者日志一致的地方。这时，AppendEntries 会返回成功，跟随者的冲突日志条目都被移除，并且添加所缺少的领导人日志条目。一旦追加日志（*AppendEntries*）返回成功，跟随者和领导人的日志就一致了，这样的状态会保持到该任期结束。

如果需要的话，算法还可以进行优化来减少 AppendEntries RPC 失败的次数。例如，当拒绝一个 AppendEntries 请求时，跟随者可以记录下冲突日志条目的任期号和自己存储那个任期的最早的索引。通过这些信息，领导人能够直接递减 nextIndex 跨过那个任期内所有的冲突条目；这样的话，一个冲突的任期需要一次 AppendEntries RPC，而不是每一个冲突条目需要一次 AppendEntries RPC。在实践中，我们怀疑这种优化是否是必要的，因为 AppendEntries 一致性检查很少失败并且也不太可能出现大量的日志条目不一致的情况。

通过这种机制，一个领导人在掌权时不需要采取另外特殊的方式来恢复日志的一致性。它只需要使用一些常规的操作，通过响应追加日志（*AppendEntries*）一致性检查的失败，能使得日志自动的趋于一致。一个领导人从来不会覆盖或者删除自己的日志（表-3 中的领导人只增加原则）。

这个日志复制机制展示了在第 2 章中阐述的所希望的一致性特性：只要大多数的服务器是正确的，Raft 就能够接受、复制并应用新的日志条目。在通常情况下，一个新的日志条目可以在一轮 RPC 内，在大多数服务器上完成复制，并且一个速度很慢的跟随者并不会影响整体的性能。

5.4 安全性

前面的章节里描述了 Raft 算法是如何选举和复制日志的。然而，到目前为止，描述的机制并不能充分的保证每一个状态机会按照相同的顺序执行相同的指令。例如，一个跟随者可能会进入不可用状态，同时领导人已经提交了若干的日志条目，然后这个跟随者可能会被选举为领导人，并且覆盖这些日志条目。因此，不同的状态机可能会执行不同的指令序列。

这一节通过在领导选举的时候增加一些限制来完善 Raft 算法。这一限制保证了任何的领导人对于给定的任期号，都拥有了之前任期的所有被提交的日志条目（图 3 中的领导人完整特性）。增加这一选举时的限制，我们对于提交时的规则也更加清晰。最终，我们将展示对于领导人完整特性的简要证明，并且说明领导人完整性特性是如何引导复制状态机做出正确行为的。

5.4.1 选举限制

在所有以领导人为基础的一致性算法中，领导人最终必须要存储全部已经提交的日志条目。在一些一致性算法中，例如：View stamped Replication，即使一开始没有包含全部已提交的条目，也可以被选为领导人。这些算法都有一些另外的机制来保证找到丢失的条目，并将它们传输给新的领导人，这个过程要么在选举过程中完成，要么在选举之后立即开始。不幸的是，这种方式大大增加了复杂性。Raft 使用了一种更简单的方式来保证在新的领导人开始选举时，在之前任期的所有已提交的（*Committed*）日志条目都会出现在新领

导人的日志中，而不需要将这些条目传送给领导人。这就意味着日志条目只有一个流向：从领导人流向跟随者。领导人永远不会覆盖已经存在的日志条目。

Raft 使用投票的方式来处理，避免一个没有包含全部日志条目的候选者赢得选举，除非该候选者的日志包含了所有已提交的条目。一个候选者为了赢得选举，必须要同集群中的大多数服务器进行通信，这就意味着，每一个已提交的日志条目至少在其中一个服务器上出现。如果候选者的日志至少和大多数服务器上的日志一样新（*up-to-date*，会在后面经确定义），那么它将包含有所有已经提交的日志条目。RequestVote RPC 实现了这个限制：这个 RPC 中包含候选者的日志信息，如果它自己的日志比其他候选者的日志要新，那么它会拒绝其他候选者的投票请求。

Raft 通过比较日志中最后一个条目的索引和任期号，来决定两个日志哪一个更新。如果两个日志的任期号不同，任期号大的更新；如果任期号相同，更长的日志更新。

5.4.2 提交之前任期的日志条目

正如 5.3 节中描述的那样，只要一个日志条目被存储在了大多数的服务器上，领导人就知道属于当前任期的条目已经提交了。如果领导人在提交之前就崩溃了，将来的领导人会尝试着继续完成复制日志。然而，新领导人并不能马上确定，一个来自前一任期的条目已经被提交，虽然该条目已经存储到了大多数服务器上。图-8 说明了一种情况，一个存储在了大多数服务器上的日志条目仍然被新上任的领导人覆盖了。

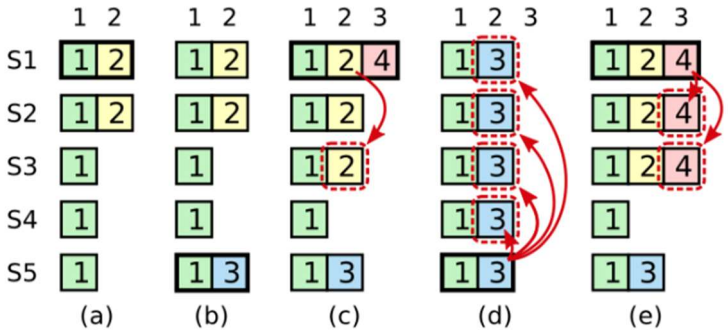

图-8：如图的时间序列说明了为什么领导人不能通过之前任期的日志条目判断它的提交状态。在(a)中，S1 是领导人并且部分复制了索引 2 上的日志条目。在(b)中 S1 崩溃了；S5 通过 S3、S4 和自己的投票赢得了选举，并且在索引 2 上接收了另一个日志条目。在(c)中 S5 崩溃了，S1 重启了，通过 S2、S3 和自己的投票赢得了选举，并且继续索引 2 处的复制，这时任期 2 的日志条目已经在大部分服务器上完成了复制，但是还并没有提交。在(d)中，如果 S1 崩溃了，S5 会通过 S2、S3、S4 的投票成为领导人，然后用它自己在任期 3 的日志条目覆盖掉其他服务器的日志条目。然而，如果 S1 在崩溃之前，它当前任期在大多数服务器上复制了一个日志条目，就像在(e)中那样，那么这个条目就会被提交（S5 就不会赢得选举）。在这时，之前的日志条目就会正常被提交。

为了消除图-8 中描述的问题，Raft 从来不会通过计算复制的日志条目数，来提交之前任期的日志条目。只有领导人当前任期的日志条目才能通过计算数目来进行提交。一旦当前任期的日志条目以这种方式被提交，那么由于日志匹配原则（*Log Matching Property*），之前的日志条目也都会被间接的提交。在某些情况下，领导人可以安全的知道一个旧的日志条目是否已经被提交（例如，通过观察该条目是否存储到所有服务器上），但 Raft 使用了一种更加保守的方法来简化问题。

因为当领导人从之前任期复制日志条目时，日志条目保留了它们上一个任期的编号，这使得 Raft 在提交规则中增加了额外的复杂性。在其他的一致性算法中，如果一个新的领导人要复制之前任期中的日志条目，它必须要使用新的任期编号。Raft 采用的方式，使得判断日志条目更加容易，因为它们在整个日志文件中一直维护着同样的任期编号。另外，和其他的一致性算法相比，Raft 算法中的新领导人会发送更少的之前任期的日志条目（其他算法在提交之前，必须要发送冗余的日志条目以便记住它们）。

5.4.3 安全性论证

给出完整的 Raft 算法，现在我们能够更精确地论证领导人完备性原则（这个论证基于安全性证明；详见 9.2 节）。我们假定领导人完备性原则是不成立的，然后推导出矛盾。假定领导人（leaderT）在任期 T 提交了一个日志条目，但是这个日志条目并没有存储到下一个任期的领导人日志中。假设最小的任期 U>T，领导人（leaderU）没有存储这个日志条目。

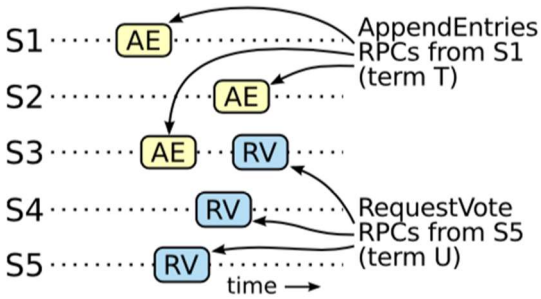

图-9：如果 S1（任期 T 的领导人）在它的任期提交了一个日志条目，并且 S5 在之后的任期 U 成为领导人，那么最少会有一个服务器（S3）接收了这个日志条目，并且会投票给 S5。

1. 在 leaderU 选举时，一定没有那个被提交的日志条目（领导人从来不会删除或者覆盖日志条目）。
2. leaderT 在集群的大多数服务器上复制了这个条目。因此，至少有一个服务器（投票者）既接收了来自 leaderT 的日志条目，又投票给 leaderU，如图-9 所示。这个投票者是产生矛盾的关键。
3. 投票者必须在给 leaderU 投票之前接收来自 leaderT 的已提交日志条目；否则它会拒绝来自 leaderT 的追加日志（AppendEntries）请求（它的当前任期会比 T 要大）。

4. 投票者会在它给 leaderU 投票时存储那个条目，因为任何中间的领导人都保有该条目（基于假设），领导人从来不会移除这个条目，并且跟随者也只会在和领导人冲突时才会移除日志条目。
5. 投票者给 leaderU 投票了，所以 leaderU 的日志必须和投票者的一样新。这就导致了一个矛盾。
6. 首先，如果投票者和 leaderU 最后一条日志条目的任期编号相同，那么 leaderU 的日志一定和投票者的一样长，因此它的日志包含全部投票者的日志条目。这是矛盾的，因为在假设中投票者和 leaderU 包含的已提交条目是不同的。
7. 除此之外，leaderU 的最后一条日志的任期编号一定比投票者的大。另外，它也比 T 要大，因为投票者的最后一条日志条目的任期编号最小也要是 T（它包含了所有任期 T 提交的日志条目）。创建 leaderU 最后一条日志条目的上一任领导人必须包含已经提交的日志条目（基于假设）。那么，根据日志匹配原则（*Log Matching*），leaderU 也一定包含那条提交的日志条目，这也是矛盾的。
8. 这时就完成了矛盾推导。因此，所有比任期 T 大的领导人一定包含所有在任期 T 提交的日志条目。
9. 日志匹配原则（*Log Matching*）保证了未来的领导人也会包含被间接提交的日志条目，就像图-8 中(d)时刻索引为 2 的条目。

通过给出了领导人完备性原则（*leader Completeness*），我们能够证明表-3 中的状态机安全原则（*State Machine Safety*）。状态机安全原则讲的是，如果一个服务器将给定索引上的日志条目应用到了它自己的状态机上，其他服务器的同一索引位置不可能应用的是其他条目。在一个服务器应用一条日志条目到它自己的状态机中时，它的日志必须和领导人的日志在该条目和之前的条目上相同，并且已经被提交。现在我们来考虑在任何一个服务器

应用一个指定索引位置的日志的最小任期；日志完整性特性保证拥有更高任期编号的领导人会存储相同的日志条目，所以之后的任期里应用某个索引位置的日志条目也会是相同的值。因此，状态机安全特性是成立的。

最后，Raft 算法需要服务器按照日志中索引位置顺序应用日志条目。和状态机安全特性结合起来看，这就意味着所有的服务器会应用相同的日志序列到自己的状态机中，并且是按照相同的顺序。

5.5 跟随者与候选者崩溃

截止到目前，我们只讨论了领导人崩溃的问题。解决跟随者和候选者崩溃的问题，比领导人崩溃要简单得多，这两者崩溃的处理方式是一样的。如果一个跟随者或者候选者崩溃了，那么之后发送给它的 RequestVote RPC 和 AppendEntries RPC 会失败。Raft 通过无限的重试来处理这些失败；如果崩溃的服务器重启了，RPC 就会成功完成。如果一个服务器在收到了 RPC 之后但在响应之前崩溃了，那么它会在重启之后再次收到同一个 RPC。因为 Raft 中的 RPC 都是幂等的，因此不会有什么问题。例如，如果一个跟随者收到了一个已经包含在它的日志中的追加日志（*AppendEntries*）请求，它会忽视这个新的请求。

5.6 时序与可用性

我们对于 Raft 的要求之一就是安全性不依赖于时序（*Timing*）：系统不能仅仅因为一些事件发生的比预想的快一些或慢一些就产生错误。然而，可用性（系统可以及时响应客户端的特性）不可避免的要依赖时序。例如，如果消息交换在服务器崩溃时花费更多的时间，候选者不会等待太长的时间来赢得选举；没有一个稳定的领导人，Raft 将无法工作。

领导人选举是 Raft 中对时序要求最关键的地方。Raft 会选出并保持一个稳定的领导人，只有系统满足下列时序要求（*Timing Requirement*）：

broadcastTime << electionTimeout << MTBF

在这个不等式中，broadcastTime 是指，一个服务器并行的向集群中的其他服务器发送 RPC，并收到这些服务器的响应的平均时间；electionTimeout 指的就是在 5.2 节描述的选举超时时间；MTBF 指的是单个服务器发生故障的间隔时间的平均数。broadcastTime 应该比 electionTimeout 小一个数量级，为的是使领导人能够持续发送心跳信息（*Heartbeat*）来阻止跟随者开始选举；根据已经给出的随机选举超时时间方法，这个不等式也使得瓜分选票的情况变成不可能。electionTimeout 也要比 MTBF 小几个数量级，为的是使得系统稳定运行。当领导人崩溃时，大约会在整个 electionTimeout 的时间内不可用；我们希望这种情况仅占全部时间的很小一部分。

broadcastTime 和 MTBF 是由系统决定的属性，但 electionTimeout 是我们必须做出选择的。Raft 的 RPC 需要接收方将信息持久化保存到稳定的存储中，所以广播时间大约是 0.5 毫秒到 20 毫秒，这取决于存储技术（/*持久化所需时间与存储技术相关*/）。因此，electionTimeout 一般在 10ms 到 500ms 之间。大多数服务器的 MTBF 都在几个月甚至更长，很容易满足这个时序需求。

6 集群成员变化

到目前为止，我们都假设集群的配置（加入到一致性算法的服务器集合）是固定不变的。但是在实践中，偶尔是会改变集群配置的。例如，替换那些宕机的机器或者改变副本数量。尽管可以通过暂停集群，更新所有配置，然后再重启集群的方式来实现。但是在更改的时候，集群将不可用。另外，如果存在手工操作步骤，那么也会有操作失误的风险。为了避免这样的问题，我们决定自动化改变配置，并且将其纳入到 Raft 一致性算法中。

为了让配置修改机制安全，那么在转换的过程中，不能够存在任何时间点，两个领导人同时被选举成功，即避免同一任期有两个领导人。不幸的是，任何集群从旧的配置直接转换到新的配置都是不安全的。一次性原子地转换所有服务器是不可能的，所以在转换期间，集群存在划分成两个独立的大多数群体的可能性（见图 10）。

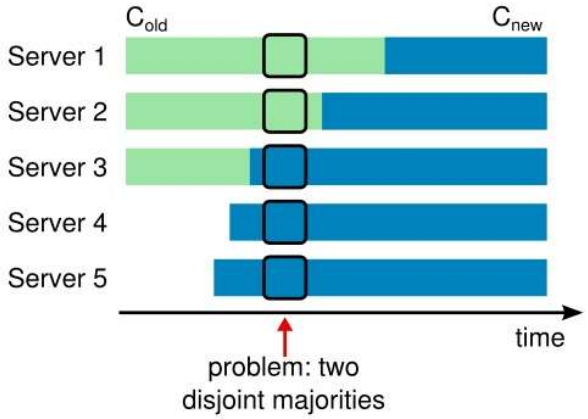

图 10：直接从旧配置转到新配置是不安全的，因为各个机器可能在任何的时候进行转换。在这个例子中，集群配额从 3 台机器变成了 5 台。不幸的是，存在这样的一个时间点，两个不同的领导人在同一个任期里都可以被选举成功。一个是通过旧配置当选，一个通过新配置当选。

为了保证安全性，配置更改使用两阶段方法。目前有很多种两阶段的实现。例如，有些系统在第一阶段停掉旧配置，所以集群就不能处理客户端请求；然后在第二阶段启用新配置。在 Raft 中，集群先切换到一个过渡的配置，我们称之为联合共识（*Joint Consensus*）；一旦联合共识被提交，那么系统就切换到新的配置上。联合共识是旧配置和新配置的结合：

- 日志条目被复制给集群中新、旧配置的所有服务器。
- 新、旧配置的服务器都可以成为领导人。

- 达成一致（针对选举和提交）需要分别在两种配置上获得大多数的支持。

联合共识允许独立的服务器在不影响安全性的前提下，在不同的时间进行配置转换。此外，联合共识可以让集群在配置转换的过程中，依然响应客户端的请求。

集群配置在复制日志中以特殊的日志条目来存储和通信；图 11 展示了配置的转换过程。当一个领导人接收到一个改变配置从 C-old 到 C-new 的请求，他会以前面描述的日志条目和副本的形式，存储联合共识配置（图中的 C-old,new）。一旦一个服务器将新配置日志条目增加到他的日志中，他就会用这个配置，来做出未来所有的决定（服务器总是使用最新的配置，无论他是否已经被提交）。这意味着领导人要使用 C-old,new 的规则，来决定日志条目 C-old,new 什么时候需要提交。如果领导人崩溃了，被选出来的新领导人可能是使用 C-old 配置，也可能是 C-old,new 配置，这取决于赢得选举的候选者，是否已经接收到了 C-old,new 配置。在任何情况下，C-new 配置在这一时期都不会单方面的做出决定。

一旦 C-old,new 被提交，那么无论是 C-old 还是 C-new，在没有经过他人批准的情况下都不可能做出决定。并且领导人完全特性，保证了只有拥有 C-old,new 日志条目的服务器才有可能被选举为领导人。这个时候，领导人创建一条关于 C-new 配置的日志条目，并复制给集群就是安全的了。再者，每个服务器在见到新的配置的时候就会立即生效。当新的配置在 C-new 的规则下被提交，旧的配置就变得无关紧要，同时不使用新的配置的服务器就可以关闭了。如图 11，C-old 和 C-new 没有任何机会同时做出单方面的决定；这保证了安全性。

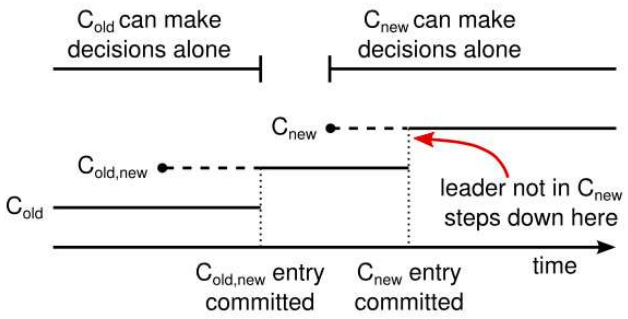

图 11：一个配置切换的时间线。虚线表示已经被创建，但是还没有被提交的配置日志条目，实线表示最后被提交的配置日志条目。领导人首先在自己的日志中创建 C-old,new 配置条目，并提交到 C-old,new 中（C-old 的大多数和 C-new 的大多数）。然后他创建 C-new 条目并提交到 C-new 中的大多数。这样就不存在 C-new 和 C-old 可以同时做出决定的时间点。

关于重新配置还有三个问题。第一个问题是，新的服务器可能初始化没有存储任何的日志条目。当这些服务器以这种状态加入到集群中，那么他们需要一段时间来更新追赶，这时还不能提交新的日志条目。为了避免这种可用性的间隔时间，Raft 在配置更新之前使用了一种额外的阶段，在这个阶段，新的服务器以没有投票权身份加入到集群中来（领导人复制日志给他们，但是不考虑他们是大多数）。一旦新的服务器追赶上了集群中的其他机器，重新配置可以像上面描述的一样处理。

第二个问题是，集群的领导人可能不是新配置的一员。在这种情况下，领导人就会在提交了 C-new 日志之后退位（回到跟随者状态）。这意味着有这样的一段时间，领导人管理着集群，但是不包括他自己；他复制日志但是不把他自己算作是大多数之一。当 C-new 被提交时，会发生领导人过渡，因为这时是最早新的配置可以独立工作的时间点（将总是能够在 C-new 配置下选出新的领导人）。在此之前，可能只能从 C-old 中选出领导人。

第三个问题是，移除不在 C-new 中的服务器可能会扰乱集群。这些服务器将不会再接收到心跳，所以当选举超时，他们就会进行新的选举过程。他们会发送拥有新的任期号的 RequestVote RPCs，这样会导致当前的领导人回退成跟随者状态。新的领导人最终会被选出来，但是被移除的服务器将会再次超时，然后这个过程会再次重复，导致整体可用性大幅降低。

为了避免这个问题，当服务器确认当前领导人存在时，服务器会忽略 RequestVote RPCs。特别的，当服务器在当前最小选举超时时间内收到一个 RequestVote RPC，他不会更新当前的任期号或者投出选票。这不会影响正常的选举，每个服务器在开始一次选举之前，至少等待一个最小选举超时时间。然而，这有利于避免被移除的服务器扰乱：如果领导人能够发送心跳给集群，那么他就不会被更大的任期号废黜。

7 日志压缩

Raft 的日志在正常操作中不断的增长，但是在实际的系统中，日志不能无限制的增长。随着日志不断增长，它会占用越来越多的空间，花费越来越多的时间来重置。如果没有一定的机制去清除日志里积累的陈旧信息，那么会带来可用性问题。

快照是最简单的压缩方法。在快照系统中，整个系统的状态都以快照的形式写入到稳定的持久化存储中，然后在那个时间点之前的日志全部丢弃。快照技术被用在 Chubby 和 ZooKeeper 中，接下来的章节介绍 Raft 的快照技术。

增量压缩的方法，例如日志清理[36]或者日志结构合并树[30,5]，都是可行的。这些方法每次只对一小部分数据进行操作，这样就分散了压缩的负载压力。首先，他们先选择一个已经积累的大量已经被删除或者被覆盖对象的区域，然后重写那个区域还活跃的对象，之后释放那个区域。和简单操作整个数据集合的快照相比，需要增加复杂的机制来实现。状态机可以使用和快照相同的接口实现 LSMtree，但是日志清除就需要修改 Raft 了。

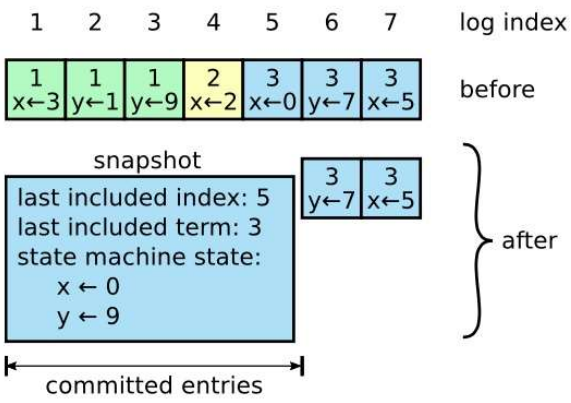

图 12：一个服务器用新的快照替换了从 1 到 5 的条目，快照值存储了当前的状态。快照中包含了最后的索引位置和任期号。

图 12 展示了 Raft 的快照思想。每个服务器独立的创建快照，快照仅包括已经被提交的日志。快照的主要工作包括将状态机的状态写入到快照中。Raft 也包含一些少量的元数据到快照中：最后被包含索引（*last included index*）指的是被快照取代的最后的条目在日志中的索引值（状态机最后应用的日志），最后被包含的任期（*last included term*）指的是该条目的任期号。保留这些数据是为了支持快照后紧接着的第一个条目的追加日志（AppendEntries）请求时的一致性检查，因为这个条目需要前一日志条目的索引值和任期号。为了支持集群成员更新（第 6 节），快照中也将最后的一次配置作为最后一个条目存下来。一旦服务器完成一次快照，它就可以删除最后索引位置之前的所有日志和快照了。

尽管服务器都是独立的创建快照，但是偶尔需要领导人发送快照给一些缓慢的跟随者。这通常发生在当领导人已经丢弃了下一条需要发送给跟随者的日志条目的时候。幸运的是，这种情况不是常规操作：一个与领导人保持同步的跟随者通常都会有这个条目。然而一个运行非常缓慢的跟随者或者新加入集群的服务器（第 6 节）将不会有这个条目。这时让这个跟随者更新到最新的状态的方式就是通过网络把快照发送给他们。

安装快照 RPC（*InstallSnapshot RPC*）：

安装快照 RPC 由领导人调用，用以将快照分块发送给跟随者。领导人总是按顺序发送快照分块。

参数	解释
term	领导人的任期号
leaderId	领导人的 Id，以便于跟随者重定向请求
lastIncludedIndex	快照中包含的最后日志条目的索引值
lastIncludedTerm	快照中包含的最后日志条目的任期号
offset	分块在快照中的字节偏移量
data[]	从偏移量开始的快照分块的原始字节
done	如果这是最后一个分块则为 true

返回结果	解释
term	当前任期号（currentTerm），便于领导人更新自己

接收者实现:

1) 如果 term<currentTerm 就立即回复

2) 如果是第一个分块（offset 为 0）就创建一个新的快照

3) 在指定偏移量写入数据

4) 如果 done 是 false，则继续等待更多的数据

5) 保存快照文件，丢弃具有较小索引的任何现有或部分快照

6) 如果现存的日志条目与快照中最后包含的日志条目具有相同的索引值和任期号，则保留其后的日志条目并进行回复

7) 丢弃整个日志

8) 使用快照重置状态机（并加载快照的集群配置）

图 13：一个关于安装快照的简要概述。为了便于传输，快照都是被分成分块的；每个分块都向跟随者显示了领导人的生命迹象，所以跟随者可以重置选举超时计时器。

领导人使用一种叫做安装快照的 RPC（*InstallSnapshot RPC*）来发送快照给太落后的跟随者；见图 13。当跟随者通过这种 RPC 接收到快照时，它必须自己决定对于已经存在的日志该如何处理。通常快照包含没有在接收者日志中存在的信息。在这种情况下，跟随者丢弃其整个日志；它全部被快照取代，并且可能包含与快照冲突的未提交条目。如果接收到的快照是自己日志的前面部分（由于网络重传或者错误），那么被快照包含的条目将会被全部删除，但是快照后面的条目仍然有效，必须保留。

这种快照的方式背离了 Raft 的强领导人原则，因为跟随者可以在不知道领导人情况下创建快照。但是我们认为这种背离是值得的。领导人的存在，是为了解决在达成一致性的时候的冲突，但是在创建快照的时候，一致性已经达成，这时不存在冲突了，所以没有领

导人也是可以的。数据依然是从领导人传给跟随者，只是跟随者可以重新组织他们的数据。

我们考虑过一种替代的基于领导人的快照方案，即只有领导人创建快照，然后发送给所有的跟随者。但是这样做有两个缺点。第一，发送快照会浪费网络带宽并且延缓了快照处理的时间。每个跟随者都已经拥有了所有产生快照需要的信息，而且很显然，自己从本地的状态中创建快照比通过网络接收别人发来的要经济。第二，领导人的实现会更加复杂。例如，领导人需要发送快照的同时，并行将新的日志条目发送给跟随者，这样才不会阻塞新的客户端请求。

还有两个问题影响了快照的性能。首先，服务器必须决定什么时候应该创建快照。如果快照创建的过于频繁，那么就会浪费大量的磁盘带宽和其他资源；如果创建快照频率太低，他就要承受耗尽存储容量的风险，同时也增加了从日志重建的时间。一个简单的策略就是当日志大小达到一个固定大小的时候就创建一次快照。如果这个阈值设置的显著大于期望的快照的大小，那么快照对磁盘压力的影响就会很小了。

第二个影响性能的问题就是写入快照需要花费一段时间，并且我们还不希望影响到正常操作。解决方案是通过写时复制的技术，这样新的更新就可以被接收而不影响到快照。例如，具有函数式数据结构的状态机天然支持这样的功能。另外，操作系统的写时复制技术的支持（如 Linux 上的 fork）可以被用来创建完整的状态机的内存快照（我们的实现就是这样的）。

8 客户端交互

这一节将介绍客户端是如何和 Raft 进行交互的，包括客户端如何发现领导人和 Raft 是如何支持线性化语义的[10]。这些问题对于所有基于一致性的系统都存在，并且 Raft 的解决方案和其他的也差不多。

Raft 中的客户端发送所有请求给领导人。当客户端启动的时候，它会随机挑选一个服务器进行通信。如果客户端第一次挑选的服务器不是领导人，那么那个服务器会拒绝客户端的请求并且提供他最近接收到的领导人的信息（附加条目请求包含了领导人的网络地址）。如果领导人已经崩溃了，那么客户端的请求就会超时；客户端之后会再次重试随机挑选服务器的过程。

我们 Raft 的目标是要实现线性化语义（每一次操作立即执行，只执行一次，在他调用和收到回复之间）。但是，如上述，Raft 是可以执行同一条命令多次的：例如，如果领导人在提交了这条日志之后，但是在响应客户端之前崩溃了，那么客户端会和新的领导人重试这条指令，导致这条命令就被再次执行了。解决方案就是客户端对于每一条指令都赋予一个唯一的序列号。然后，状态机跟踪每条指令最新的序列号和相应的响应。如果接收到一条指令，它的序列号已经被执行了，那么就立即返回结果，而不重新执行指令。

只读的操作可以直接处理而不需要记录日志。但是，在不增加任何限制的情况下，这么做可能会冒着返回脏数据的风险，因为领导人响应客户端请求时可能已经被新的领导人废黜了，但是它还不知道。线性化的读操作必须不能返回脏数据，Raft 需要使用两个额外的措施在不使用日志的情况下保证这一点。首先，领导人必须有关于被提交日志的最新信息。领导人完全特性保证了领导人一定拥有所有已经被提交的日志条目，但是在它任期开

始的时候，它可能不知道哪些是已经被提交的。为了知道这些信息，它需要在它的任期里提交一条日志条目。Raft 中通过领导人在任期开始的时候，提交一个空白的日志条目来实现。第二，领导人在处理只读的请求之前，必须检查自己是否已经被废黜了（如果一个更新的领导人被选举出来，自己的信息已经变脏了）。Raft 中通过让领导人在响应只读请求之前，先和集群中的大多数节点交换一次心跳信息来处理这个问题。可选的，领导人可以依赖心跳机制来实现一种租约的机制[9]，但是这种方法依赖时间来保证安全性（假设时间误差是有界的）。

9 算法实现和评估

我们已经为 RAMCloud[33]实现了 Raft 算法，作为存储配置信息的复制状态机的一部分，并且帮助 RAMCloud 协调故障转移。这个 Raft 实现包含大约 2000 行 C++代码，其中不包括测试、注释和空行。这些代码是开源的[23]。同时也有大约 25 个其他独立的第三方的基于这篇论文草稿的开源实现[34]，针对不同的开发场景。同时，很多公司已经部署了基于 Raft 的系统。

这一节会从三个方面来评估 Raft 算法：可理解性、正确性和性能。

9.1 可理解性

为了和 Paxos 比较 Raft 算法的可理解能力，我们针对高年级的本科生和研究生，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上，进行了一次学习的实验。我们分别拍了针对 Raft 和 Paxos 的视频课程，并准备了相应的小测验。Raft 的视频讲课，除了日志压缩，覆盖了这篇论文的所有内容；Paxos 讲课包含了足够的资料来创建一个等价的复制状态机，包括单决策 Paxos，多决策 Paxos，重新配置和一些实际系统需要的性能优化（例如领导人选举）。小测验测试一些对算法的基本理解和解释一些边角的示例。每个学生都是看完第一个视频，回答相应的测试，再看第二个视频，回答相应的

测试。大约有一半的学生先进行 Paxos 部分，然后另一半先进行 Raft 部分，这是为了说明两者从第一部分的算法学习中获得的表现和经验的差异。我们计算参加人员的每一个小测验的得分，来看参与者是否在 Raft 算法上更加容易理解。

我们尽可能的使得 Paxos 和 Raft 的比较更加公平。这个实验偏爱 Paxos 表现在两个方面：43 个参加者中有 15 个人在之前有一些 Paxos 的经验，并且 Paxos 的视频要长 14%。如表格 1 总结的那样，我们采取了一些措施来减轻这种潜在的偏见。我们所有的材料都可供审查[28,31]。

关心	缓和偏见采取的手段	可供查看的材料
相同的讲课质量	两者使用同一个讲师。Paxos 使用的是现在很多大学里经常使用的。Paxos 会长 14%。	视频
相同的测验难度	问题以难度分组，在两个测验里成对出现。	小测验
公平评分	使用评价量规。随机顺序打分，两个测验交替进行。	评价量规（rubric）

表 1：考虑到可能会存在的偏见，对于每种情况的解决方法，和相应的材料。

参加者平均在 Raft 的测验中比 Paxos 高 4.9 分（总分 60，那么 Raft 的平均得分是 25.7，而 Paxos 是 20.8）；图 14 展示了每个参与者的得分。配置 t-检验（又称 student‘st-test）表明，在 95%的可信度下，真实的 Raft 分数分布至少比 Paxos 高 2.5 分。

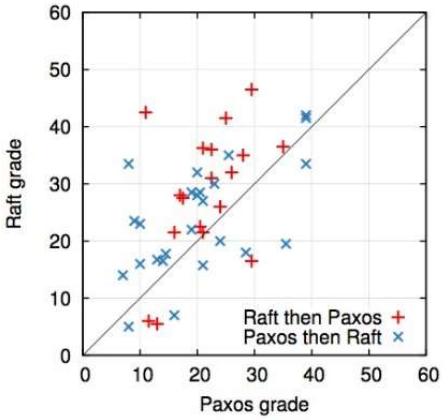

图 14：一个散点图表示了 43 个学生在 Paxos 和 Raft 的小测验中的成绩。在对角线之上的点表示在 Raft 获得了更高分数的学生。

我们也建立了一个线性回归模型来预测一个新的学生的测验成绩，基于以下三个因素：他们使用的是哪个小测验，之前对 Paxos 的经验，和学习算法的顺序。模型预测，对小测验的选择会产生 12.5 分的差别。这显著的高于之前的 4.9 分，因为很多学生在之前都已经有了对于 Paxos 的经验，这相当明显的帮助 Paxos，对 Raft 就没什么太大影响了。但是奇怪的是，模型预测对于先进行 Paxos 小测验的人而言，Raft 的得分低了 6.3 分，虽然我们不知道为什么，这似乎在统计上是有意义的。

我们同时也在测验之后调查了参与者，他们认为哪个算法更加容易实现和解释；这个的结果在图 15 上。压倒性的结果表明 Raft 算法更加容易实现和解释（41 人中的 33 个）。但是，这种自己报告的结果不如参与者的成绩更加可信，并且参与者可能因为我们的 Raft 更加易于理解的假说而产生偏见。

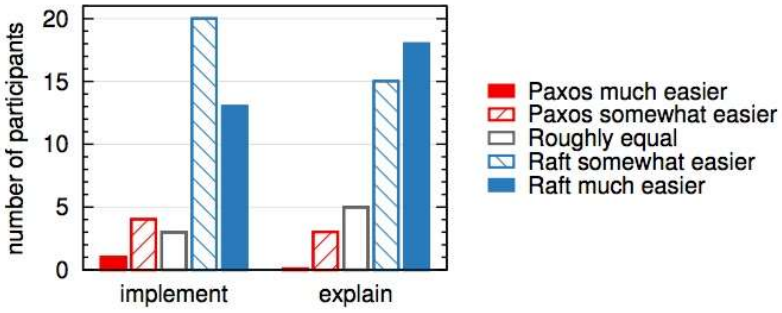

图 15：通过一个 5 分制的问题，参与者（左边）被问哪个算法他们觉得在一个高效正确的系统里更容易实现，右边被问哪个更容易向学生解释。

关于 Raft 用户学习有一个更加详细的讨论[31]。

9.2 正确性

在第 5 节，我们已经制定了正式的规范，和对一致性机制的安全性证明。这个正式规范[31]使用 TLA+规范语言使图 2 中总结的信息非常清晰[17]。它长约 400 行，并作为证明的主题。同时对于任何想实现 Raft 的人也是十分有用的。我们通过 TLA 证明系统[7]非常机械的证明了日志完全特性。然而，这个证明依赖的约束前提还没有被机械证明（例如，我们还没有证明规范的类型安全）。而且，我们已经写了一个非正式的证明关于状态机安全性是完备的，并且是相当清晰的（大约 3500 个词）。

9.3 性能

Raft 和其他一致性算法例如 Paxos 有着差不多的性能。在性能方面，最重要的关注点是，当领导人被选举成功时，什么时候复制新的日志条目。Raft 通过很少数量的消息包（一轮从领导人到集群大多数机器的消息）就达成了这个目的。同时，进一步提升 Raft 的性能也是可行的。例如，很容易通过支持批量操作和管道操作来提高吞吐量和降低延迟。对于其他一致性算法已经提出过很多性能优化方案，其中有很多也可以应用到 Raft 中来，但是我们暂时把这个问题放到未来的工作中去。

我们使用 Raft 实现来衡量 Raft 领导人选举的性能并且回答两个问题。首先，领导人选举的过程收敛是否快速？第二，在领导人宕机之后，最小的系统宕机时间是多久？

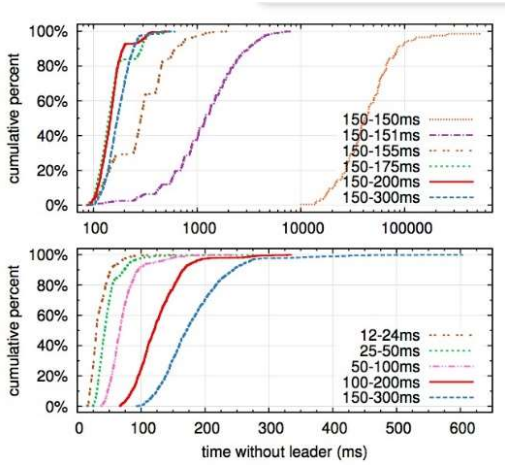

图 16：发现并替换一个已经崩溃的领导人的时间。上面的图考察了在选举超时时间上的随机化程度，下面的图考察了最小选举超时时间。每条线代表了 1000 次实验（除了 150-150 毫秒只试了 100 次），和相应的确定的选举超时时间。例如，150-155 毫秒意思是，选举超时时间从这个区间范围内随机选择并确定下来。这个实验在一个拥有 5 个节点的集群上进行，其广播时延大约是 15 毫秒。对于 9 个节点的集群，结果也差不多。

为了衡量领导人选举，我们反复的使一个拥有 5 个节点的服务器集群的领导人宕机，并计算需要多久才能发现领导人宕机，并选出一个新的领导人（见图 16）。为了构建一个最坏的场景，在每一次的尝试里，服务器都有不同长度的日志，意味着有些候选者是没有成为领导人的资格的。另外，为了促成选票瓜分的情况，我们的测试脚本在终止领导人之前，同步发送了一次心跳广播（这大约和领导人在崩溃前复制一个新的日志给其他机器很像）。领导人均匀的随机的在心跳间隔里宕机，也就是最小选举超时时间的一半。因此，最小宕机时间大约就是最小选举超时时间的一半。

图 16 中上面的图表明，只需要在选举超时时间上，使用很少的随机化就可以大大避免选票被瓜分的情况。在没有随机化的情况下，选举过程往往都需要花费超过 10 秒钟。仅仅增加 5 毫秒的随机化时间，就大大的改善了选举过程，现在平均的宕机时间只有 287 毫秒。增加更多的随机化时间可以大大改善最坏情况：通过增加 50 毫秒的随机化时间，最坏的完成情况（1000 次尝试）只要 513 毫秒。

图 16 中下面的图显示，通过减少选举超时时间，可以减少系统的宕机时间。在选举超时时间为 12-24 毫秒的情况下，只需要平均 35 毫秒就可以选举出新的领导人（最长的一次花费了 152 毫秒）。然而，进一步降低选举超时时间的话，就会违反 Raft 的时间不等式需求：在选举新领导人之前，领导人就很难发送完心跳包。这会导致没有意义的领导人改变并降低了系统整体的可用性。我们建议使用更为保守的选举超时时间，比如 150-300 毫秒；这样的时间不大可能导致没有意义的领导人改变，而且依然提供不错的可用性。

10 相关工作

已经有很多关于一致性算法的工作被发表出来，其中很多都可以归到下面的类别中：

- Lamport 关于 Paxos 的原始描述，和尝试描述的更清晰。
- 关于 Paxos 的更详尽的描述，补充遗漏的细节并修改算法，使得可以提供更加容易的实现基础。
- 实现一致性算法的系统，例如 Chubby，ZooKeeper 和 Spanner。对于 Chubby 和 Spanner 的算法并没有公开发表其技术细节，尽管他们都声称是基于 Paxos 的。ZooKeeper 的算法细节已经发表，但是和 Paxos 着实有着很大的差别。
- Paxos 可以应用的性能优化。
- Oki 和 Liskov 的 ViewstampedReplication（VR），一种和 Paxos 差不多的替代算法。原始的算法描述和分布式传输协议耦合在了一起，但是核心的一致性算法在

最近的更新里被分离了出来。VR 使用了一种基于领导人的方法，和 Raft 有很多相似之处。

Raft 和 Paxos 最大的不同之处就在于 Raft 的强领导特性：Raft 使用领导人选举作为一致性协议里必不可少的部分，并且将尽可能多的功能集中到了领导人身上。这样就可以使得算法更加容易理解。例如，在 Paxos 中，领导人选举和基本的一致性协议是正交的：领导人选举仅仅是性能优化的手段，而且不是一致性所必须要求的。但是，这样就增加了多余的机制：Paxos 同时包含了针对基本一致性要求的两阶段提交协议和针对领导人选举的独立的机制。相比较而言，Raft 就直接将领导人选举纳入到一致性算法中，并作为两阶段一致性的第一步。相对 Paxos，Raft 这样就减少了很多机制。

像 Raft 一样，VR 和 ZooKeeper 也是基于领导人的，因此他们也拥有一些 Raft 的优点。但是，Raft 比 VR 和 ZooKeeper 拥有更少的机制，因为 Raft 尽可能的减少了非领导人的功能。例如，Raft 中日志条目都遵循着从领导人发送给其他人这一个方向：AppendEntries RPC 是向外发送的。在 VR 中，日志条目的流动是双向的（领导人可以在选举过程中接收日志）；这就导致了额外的机制和复杂性。根据 ZooKeeper 公开的资料看，它的日志条目也是双向传输的，但是它的实现更像 Raft。

和上述提及的其他基于一致性的日志复制算法相比，Raft 的消息类型更少。例如，我们数了一下 VR 和 ZooKeeper 使用的用来基本一致性需要和成员改变的消息数（排除了日志压缩和客户端交互，因为这些都比较独立且和算法关系不大）。VR 和 ZooKeeper 都分别定义了 10 种不同的消息类型，相对的，Raft 只有 4 种消息类型（两种 RPC 请求和对应的响应）。Raft 的消息稍微比其他算法的信息量要大，但是都很简单。另外，VR 和 ZooKeeper 都在领导人改变时传输了整个日志，为了能够实践中使用，所以额外的消息类型就很必要了。

Raft 的强领导人模型简化了整个算法，但是同时也排斥了一些性能优化的方法。例如，平等主义 Paxos（EPaxos）在某些没有领导人的情况下可以达到很高的性能。平等主义 Paxos 充分发挥了在状态机指令中的交换性。任何服务器都可以在一轮通信下就提交指令，除非其他指令同时被提出了。然而，如果指令都是并发的被提出，并且互相之间不通信沟通，那么 EPaxos 就需要额外的一轮通信。因为任何服务器都可以提交指令，所以 EPaxos 在服务器之间的负载均衡做的很好，并且很容易在 WAN 网络环境下获得很低的延迟。但是，在 Paxos 上增加了非常明显的复杂性。

一些集群成员变换的方法已经被提出或者在其他的工作中被实现，包括 Lamport 的原始的讨论，VR 和 SMART。我们选择使用联合共识的方法因为对一致性协议的其他部分影响很小，这样只需要很少的一些机制就可以实现成员变换。Lamport 的基于 α 的方法之所以没有被 Raft 选择，是因为它假设在没有领导人的情况下也可以达到一致性。和 VR、SMART 相比较，Raft 的重新配置算法可以在不限制正常请求处理的情况下进行；相比较的，VR 需要停止所有的处理过程，SMART 引入了一个和 α 类似的方法，限制了请求处理的数量。和 VR、SMART 比较而言，Raft 需要更少的额外机制来实现。

11 结论

算法的设计通常会把正确性，效率或者简洁作为主要的目标。尽管这些都是很有意义的目标，但是我们相信，可理解性也是一样的重要。在开发者把算法应用到实际的系统中之前，这些目标没有一个会被实现，这些都会必然的偏离发表时的形式。除非开发人员对这个算法有着很深的理解并且有着直观的感觉，否则将会对他们而言很难在实现的时候保持原有期望的特性。

在这篇论文中，我们尝试解决分布式一致性问题，但是一个广为接受但是十分令人费解的算法 Paxos 已经困扰了无数学生和开发者很多年了。我们创造了一种新的算法 Raft，

显而易见的比 Paxos 要容易理解。我们同时也相信，Raft 也可以为实际的实现提供坚实的基础。把可理解性作为设计的目标改变了我们设计 Raft 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，比如分解问题和简化状态空间。这些技术不仅提升了 Raft 的可理解性，同时也使我们坚信其正确性。

12 感谢

这项研究必须感谢以下人员的支持：AliGhodsi，DavidMazie`res，和伯克利 CS294-91 课程、斯坦福 CS240 课程的学生。ScottKlemmer 帮我们设计了用户调查，NelsonRay 建议我们进行统计学的分析。在用户调查时使用的关于 Paxos 的幻灯片很大一部分是从 LorenzoAlvisi 的幻灯片上借鉴过来的。特别的，非常感谢 DavidMazieres 和 EzraHoch，他们找到了 Raft 中一些难以发现的漏洞。许多人提供了关于这篇论文十分有用的反馈和用户调查材料，包括 EdBugnion，MichaelChan，HuguesEvrard，DanielGiffin，ArjunGopalan，JonHowell，VimalkumarJeyakumar，AnkitaKejriwal，AleksandarKracun，AmitLevy，JoelMartin，SatoshiMatsushita，OlegPesok，DavidRamos，RobbertvanRenesse，MendelRosenblum，NicolasSchiper，DeianStefan，AndrewStone，RyanStutsman，DavidTerei，StephenYang，MateiZaharia 以及 24 位匿名的会议审查人员（可能有重复），并且特别感谢我们的领导人 EddieKohler。WernerVogels 发了一条早期草稿链接的推特，给 Raft 带来了极大的关注。我们的工作由 Gigascale 系统研究中心和 Multiscale 系统研究中心给予支持，这两个研究中心由关注中心研究程序资金支持，一个是半导体研究公司的程序，由 STARnet 支持，一个半导体研究公司的程序由 MARCO 和 DARPA 支持，在国家科学基金会的 0963859 号批准，并且获得了来自 Facebook，Google，Mellanox，NEC，NetApp，SAP 和 Samsung 的支持。DiegoOngaro 由 Junglee 公司，斯坦福毕业团体支持。

参考

[1] BOLOSKY, W. J., BRADSHAW, D., HAAGENS, R. B.,KUSTERS, N. P., AND LI, P. Paxos replicated state machines as the basis of a high-performance data store.In Proc. NSDI'11, USENIX Conference on Networked Systems Design and Implementation (2011), USENIX,pp. 141–154.

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

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

[4] CHANDRA, T. D., GRIESEMER, R., AND REDSTONE, J.Paxos made live: an engineering perspective. In Proc.PODC'07, ACM Symposium on Principles of Distributed Computing (2007), ACM, pp. 398–407.

[5] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C.,WALLACH, D. A., BURROWS, M., CHANDRA, T.,FIKES, A., AND GRUBER, R. E. Bigtable: a distributed storage system for structured data. In Proc. OSDI'06,USENIX Symposium on Operating Systems Design and Implementation (2006), USENIX, pp. 205–218.

[6] CORBETT, J. C., etc. Spanner: Google's globally-distributed database. In Proc. OSDI'12, USENIX Conference on Operating Systems Design and Implementation (2012), USENIX, pp. 251–264.

[7] COUSINEAU, D., DOLIGEZ, D., LAMPORT, L., MERZ,S., RICKETTS, D., AND VANZETTO, H. TLA+ proofs.In Proc. FM'12, Symposium on Formal Methods (2012),D. Giannakopoulou and D. M´ery, Eds., vol. 7436 of Lecture Notes in Computer Science, Springer, pp. 147–154.

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

[9] GRAY, C., AND CHERITON, D. Leases: An efficient fault tolerant mechanism for distributed file cache consistency.In Proceedings of the 12th ACMS symposium on Operating Systems Principles (1989), pp. 202–210.

[10] HERLIHY, M. P., AND WING, J. M. Linearizability: a correctness condition for concurrent objects. ACM Transactions on Programming Languages and Systems 12 (July1990), 463–492.

[11] HUNT, P., KONAR, M., JUNQUEIRA, F. P., AND REED,B. ZooKeeper: wait-free coordination for internet-scale systems. In Proc ATC'10, USENIX Annual Technical Conference (2010), USENIX, pp. 145–158.

[12] JUNQUEIRA, F. P., REED, B. C., AND SERAFINI, M.Zab: High-performance broadcast for primary-backup systems. In Proc. DSN'11, IEEE/IFIP Int'l Conf. on Dependable Systems & Networks (2011), IEEE Computer Society, pp. 245–256.

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

2008.

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

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

[16] LAMPORT, L. Paxos made simple. ACM SIGACT News32, 4 (Dec. 2001), 18–25.

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

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

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

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

[21] LAMPSON, B. W. The ABCD's of Paxos. In Proc.PODC'01, ACM Symposium on Principles of Distributed Computing (2001), ACM, pp. 13–13.

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

[23] LogCabin source code. http://github.com/logcabin/logcabin.LORCH, J. R., ADYA, A., BOLOSKY, W. J., CHAIKEN, R., DOUCEUR, J. R., AND HOWELL, J. The SMART way to migrate replicated stateful services. In Proc. EuroSys'06, ACMSIGOPS/EuroSys European Conference on Computer Systems (2006), ACM, pp. 103–115.

[25] MAO, Y., JUNQUEIRA, F. P., AND MARZULLO, K.Mencius: building efficient replicated state machines for WANs. In Proc. OSDI'08, USENIX Conference on Operating Systems Design and Implementation (2008),USENIX, pp. 369–384.

[26] MAZI`E RES, D. Paxos made practical. http://www.scs.stanford.edu/~dm/home/papers/paxos.pdf, Jan. 2007.

[27] MORARU, I., ANDERSEN, D. G., AND KAMINSKY, M.There is more consensus in egalitarian parliaments. In Proc. SOSP'13, ACM Symposium on Operating System Principles (2013), ACM.

[28] Raft user study. http://ramcloud.stanford.edu/~ongaro/userstudy/.

[29] OKI, B. M., AND LISKOV, B. H. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In Proc. PODC'88,ACM Symposium on Principles of Distributed Computing (1988), ACM, pp. 8–17.

[30] O'NEIL, P., CHENG, E., GAWLICK, D., AND ONEIL, E.The log-structured merge-tree (LSM-tree). Acta

Informatica 33, 4 (1996), 351–385.

[31] ONGARO, D. Consensus: Bridging Theory and Practice.PhD thesis, Stanford University, 2014 (work in progress).http://ramcloud.stanford.edu/~ongaro/thesis.pdf.

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

[33] OUSTERHOUT, J., AGRAWAL, P., ERICKSON, D.,KOZYRAKIS, C., LEVERICH, J., MAZI`ERES, D., MITRA,S., NARAYANAN, A., ONGARO, D., PARULKAR,G., ROSENBLUM, M., RUMBLE, S. M., STRATMANN,E., AND STUTSMAN, R. The case for RAMCloud. Communications of the ACM 54 (July 2011), 121–130.

[34] Raft consensus algorithm website. http://raftconsensus.github.io.

[35] REED, B. Personal communications, May 17, 2013.

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

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

[38] SHVACHKO, K., KUANG, H., RADIA, S., AND CHANSLER, R. The Hadoop distributed file system.In Proc. MSST'10, Symposium on Mass Storage Systems and Technologies (2010), IEEE Computer Society,pp. 1–10.

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